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

ARDEN RASMUSSEN

Abstract. Finite Element Methods is a process that is utilized to numerically approximate solutions to other wise impossible partial differential equations provided with arbatrairy boundaries. This paper serves as a brief explination and introduction to the methods of finite element analysis. This paper provides a complete explanation of the every process involved, a reader should be able to implement all the necessary steps required in the analysis.

1. Introduction

Finite Element Methods (FEM) is a process that is utilized for numerically computing an approximate solution to a partial differential equations, on some specified domain. There are only a few cases where this process is analytically solvable, but for most situations this approximation is as close as technology currently allows us to achieve.

The uses of finite element methods in computation and analysis are often called Finite Element Analysis (FEA), while the mathematics of the process is called FEM.

We will utilize the process closely set forth by [KH15]. This method is constructed around the Galerkin method for generating the approximation. The scope of this introduction is limited to two dimensions, but all of the mathematics is easily generalized to higher or lower dimensions. The only major part that would require rework is Mesh Generation (Section 7). The process can also be implemented for time-dependent systems, but we limit ourselves to time independence for our introduction.

In the process of FEM that we analyze, there are two major sections, **Theoretical**, and **Numerical**. The theoretical parts are required to be completed prior to any numerical computation. These stages of the process must be done manually in order to determine the program to implement. These stages are extremely dependent on the general form of the problem at hand and is where much of the mathematics is required. Then there are the numerical computational stages. These steps are frequently independent of the problem at hand, and thus much of the work can be automated. A single implementation of the computational steps can be utilized for a general class of problems. Aswell as many of the individual steps can be used for any problem, and are completely independent of the rest of the process.

2. Problem Statement

For our explanation, we will consider the 2D steady state heat equations. This is a simplification of the unsteady multidimensional heat transfer model. Since we are focusing on the time-independent analysis, there is no need for an initial condition. We define our domain to be Ω such that $\Omega \subset \mathbb{R}^2$ is a bounded 2D domain with boundary Γ.

We write the general expression for the time-dependent general heat equation as

$$\rho c_p \frac{\partial T}{\partial t} + \rho c_p \vec{u} \cdot \nabla T - \nabla \cdot (\bar{\bar{x}} \nabla T) = f \quad \text{in } \Omega.$$

Where $T(\vec{x}, t)$ is the temperature at point $\vec{x} \in \Omega$ at time $t \in (0, \tau)$. ρ is the density, and c_p denotes the specific heat capacity. For our case we assume that they are constant and 1. The forcing function f is given by $f = \rho h$ where h is the rate of heating per unit mass. $\vec{u}$ is the velocity, and $\bar{\bar{x}}$ is the thermal conductivity is a 2×2 matrix.

Using many simplifications and our assumption that there is no change in time, we find the problem statement to be

$$\rho c_p \vec{u} \cdot \nabla T - \nabla \cdot (k \nabla T) = f \quad \text{in } \Omega. \tag{2.1}$$

Using the definition of gradient and divergence operators, we find that this simplified equation can be written as

$$\sum_{i=1}^{2} u_i \frac{\partial T}{\partial x_i} - \sum_{i=1}^{2} \sum_{j=1}^{2} \frac{\partial}{\partial x_i} \left(k \frac{\partial T}{\partial x_j} \right) = f \quad \text{in } \Omega.$$

Note that the solution of this equations is defined with some constant, meaning that if T is a solution

Date: December 17, 2018.

to this equation, then $T + C$ for $C \in \mathbb{R}$ is also a solution. In order to avoid this, we must impose some boundary conditions.

3. Boundary Conditions

We must prescribe boundary conditions on our problem statement(Eq 2.1), to ensure that there is a single solution. For this initial introduction, we only present the process with Dirichlet boundary conditions

$$T = T_0 \quad \text{on } \Gamma,$$

where $T_0(\vec{x})$ is some function defining the temperature along each of the edges of the boundary. Note that this function can be of any form, including a piecewise function.

It is important to note that other boundary conditions, such as Neumann, and Newton boundary conditions can be used. These other boundary conditions add more changes to the formulation of the problem later on and require extra modifications to the implementation. Because of the extra alterations that they would necessitate, we will only focus on the Dirichlet boundary conditions.

If the time-dependence of the problem had been retained, we would also require some initial condition

$$T(\vec{x}, 0) = T^0(\vec{x}).$$

that would be defined on our domain Ω. However, we are able to neglect the initial condition as we are only looking for the stationary boundary value problem solution.

4. Variational Formulation

We first define the concept of a residual. The residual is the process of constructing a helper function $R(f)$, such that when f is the exact solution $R(f) = 0$. This simplifies the process of solving this partial differential equation, into minimizing the value of the residual. This concept is used multiple times throughout the process of FEA.

We construct the residual of equation 2.1 as

$$R(\vec{x}) = \rho c_p \vec{u} \cdot \nabla T - \nabla \cdot (k \nabla T) - f.$$

Clearly $R(\vec{x}) = 0$ if and only if T is the exact solution. Multiplying the residual by some test function w and integrating, we obtain the weighted residual

$$\int_\Omega w(\vec{x}) R(\vec{x}) d\vec{x}.$$

For the exact solution, this will always be zero because $R(\vec{x}) = 0$. Thus, we can select any function for the test function w, and this expression can still be satisfiable. However, we will restrict our test functions to ones that are zero along the boundaries, this simplifies the computations later. We can expand the weighted residual to obtain

$$\int_\Omega \rho c_p w \vec{u} \cdot \nabla T d\vec{x} - \int_\Omega w \nabla \cdot (k \nabla T) d\vec{x} = \int_\Omega w f d\vec{x}. \tag{4.1}$$

Applying Green's theorem on the second integral we are able to express it as

$$-\int_\Omega w \nabla \cdot (k \nabla T) d\vec{x} = \int_\Omega k \nabla w \cdot \nabla T d\vec{x} - \int_\Gamma k w \nabla T \cdot \vec{n} ds.$$

The integral over Γ goes to zero because we require our test function to be zero on the boundary. Thus we obtain the representation of the continuous variational problem

$$\int_\Omega \rho c_p w \vec{u} \cdot \nabla T d\vec{x} + \int_\Omega k \nabla w \cdot \nabla T d\vec{x} = \int_\Omega w f d\vec{x}. \tag{4.2}$$

5. Discretization

We define our approximate solution to be in the Sobolev space $H^1(\Omega)$. The Sobolev space is a space of continuous functions that are infinitely differentiable. The span of the Sobolev space is infinite, so we will consider a finite subspace $V_h \subset H^1(\Omega)$, and we will construct our approximation in this subspace. We define a set of basis functions for the subspace V_h to be $\{\varphi_j\}$. We leave these basis functions as arbitrary functions for now and will show the process for the constructions of the basis functions in a later section.

The approximation of T we define as T_h. It is possible to express this approximation as a linear combination of the basis functions φ_j for the finite subspace V_h that T_h is within. We write this expression as

$$T_h = \sum_{j=1}^{N} T_j \varphi_j(\vec{x}), \tag{5.1}$$

where T_j is a constant that we solve for later to achieve our final apprximation. We can now substitute our expression for T_h for T in the continuous variational problem (4.2). Since the continuous variational problem was satisfied for any test function, we make the decision to use φ_j as the test functions. These two substitutions provide us with the expression

$$\int_\Omega \rho c_p \varphi_i \vec{u} \cdot \nabla \left[\sum_{j=1}^{N} T_j \varphi_j \right] d\vec{x} + \int_\Omega k \nabla \varphi_i \cdot \nabla \left[\sum_{j=1}^{N} T_j \varphi_j \right] d\vec{x} = \int_\Omega \varphi_i f d\vec{x}.$$

Which we then rewrite as

$$\sum_{j=1}^{N}\left[T_j\int_\Omega \rho c_p\varphi_i\vec{u}\cdot\nabla\varphi_j d\vec{x}\right] + \sum_{j=1}^{N}\left[T_j\int_\Omega k\nabla\varphi_i\cdot\nabla\varphi_j d\vec{x}\right] = \int_\Omega \varphi_i f d\vec{x}.$$

This provides us with a system of equations for $i = 1, 2, \ldots, N$. Thus we express this system in matrix notation as

$$(C+K)T = F. \tag{5.2}$$

Where T is a vector in $\mathbb{R}^N$ of the T_j constants, and the elements of the matrices C, K, and F are defined as

$$c_{ij} = \rho c_p\int_\Omega \varphi_i\vec{u}\cdot\nabla\varphi_j d\vec{x} \qquad i,j = 1,\ldots,N$$
$$k_{ij} = k\int_\Omega \nabla\varphi_i\cdot\nabla\varphi_j d\vec{x} \qquad i,j = 1,\ldots,N$$
$$f_i = \int_\Omega \varphi_i f d\vec{x} \qquad i = 1,\ldots,N.$$

We note that it is often simpler to consider a matrix A such that $A = C + K$, then our expression for the system of equations becomes

$$AT = F.$$

This expression tends to be simpler to implement, and so we will continue by using this expression. If other (non-Dirichlet) boundary conditions had been implemented, this expression for the system of equations would be different. As the boundary conditions integrals are not zero, and so they will add an additional term to this system of equations.

6. Localization

While it is possible to find the elements of the matrices in (5.2) on the global domain, this proves to be exceedingly difficult, and so we will use a more generalizable method. Since we are able to select our basis function φ_j we can carefully construct a set of basis functions such that it is possible to split the domain into discrete elements. Where each element is a mutually exclusive subset of the domain. We can then implement a generalization to construct the system of equations for each element. Then using an algorithm we combine each system of equations for each element to construct the global matrices, which can then be solved for our approximation.

We discretize the domain Ω into N triangles. This process of constructing the triangular decomposition of the domain is detailed in the numerical computation section 7, Mesh Generation. For now, we assume that we have constructed a mesh of N triangles, and we denote the each of these triangles E_e, where $e = 1,\ldots,N$. We then also define the triangle vertices to be labeled $E_e^{(i)}$ where $i = 1, 2, 3$.

In 1D this step is fairly trivial as all elements have the exact same dimensions, but in our 2D implementation, each element can have variable shapes and sizes. In order to resolve this, we construct a master space.

6.1. Master Space.

The master space is a domain that we construct in order to simplify the construction of our local approximations. For our situation with a triangular element, we construct our global space such that the vertices of a triangle lie on the points $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. Then for each element in our mesh, we construct the transformation matrix T to transform between master space and element space. We define our master element to be $\hat{E}$. An image of this master element can be seen in figure 6.1.

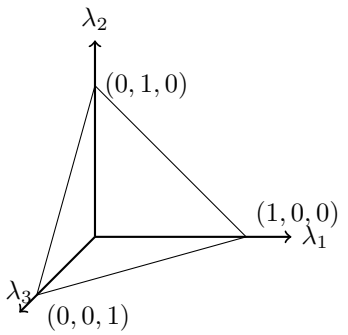


FIGURE 6.1. Master element $\hat{E}$, in barycentric coordinate system, that is used over the variable local elements.

This master element is in what is called a Barycentric coordinate system. Conversion from our trilinear coordinates to the Barycentric coordinates can be done by using the following equations

$$\lambda_1 = \frac{(Y_2-Y_3)(x-X_3)+(X_3-X_2)(y-Y_3)}{(Y_2-Y_3)(X_1-X_3)+(X_3-X_2)(Y_1-Y_3)}$$
$$\lambda_2 = \frac{(Y_3-Y_1)(x-X_3)+(X_1-X_3)(y-Y_3)}{(Y_2-Y_3)(X_1-X_3)+(X_3-X_2)(Y_1-Y_3)}$$
$$\lambda_3 = 1-\lambda_1-\lambda_2.$$

Where X_i is the x component of the ith vertex of the element, the same applies for the y component. x, and y is the x and y component of the point in the triangle that we want to convert to the Barycentric coordinates.

Converting from Barycentric coordinates to trilinear coordinates uses the following equations

$$x = \lambda_1 X_1 + \lambda_2 X_2 + \lambda_3 X_3$$
$$y = \lambda_1 Y_1 + \lambda_2 Y_2 + \lambda_3 Y_3$$

6.2. **Local Basis.** We want to construct three local basis functions, that are defined only for the current element. We aim to construct three local basis functions functions $\varphi_1^{(e)}$, $\varphi_2^{(e)}$, and $\varphi_3^{(e)}$. Each shape function should be linear, such that at its corresponding vertex the local basis function is 1 and at the other vertices the function should evaluate to 0.

Conveniently this is exactly what the barycentric coordinates of the master element provides us with, thus we define our local basis functions to be

(6.1)
$$\varphi_1^{(e)} = \frac{\left(Y_2^{(e)} - Y_3^{(e)}\right)\left(x - X_3^{(e)}\right) + \left(X_3^{(e)} - X_2^{(e)}\right)\left(y - Y_3^{(e)}\right)}{\left(Y_2^{(e)} - Y_3^{(e)}\right)\left(X_1^{(e)} - X_3^{(e)}\right) + \left(X_3^{(e)} - X_2^{(e)}\right)\left(Y_1^{(e)} - Y_3^{(e)}\right)}$$

(6.2)
$$\varphi_2^{(e)} = \frac{\left(Y_3^{(e)} - Y_1^{(e)}\right)\left(x - X_3^{(e)}\right) + \left(X_1^{(e)} - X_3^{(e)}\right)\left(y - Y_3^{(e)}\right)}{\left(Y_2^{(e)} - Y_3^{(e)}\right)\left(X_1^{(e)} - X_3^{(e)}\right) + \left(X_3^{(e)} - X_2^{(e)}\right)\left(Y_1^{(e)} - Y_3^{(e)}\right)}$$

(6.3)
$$\varphi_3^{(e)} = 1 - \varphi_1^{(e)} - \varphi_2^{(e)}$$

Thus for every element, there are three local basis functions. A plot of these local basis functions is shown in figure 6.2.

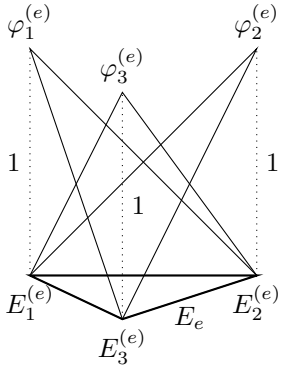


FIGURE 6.2. Plot of local basis function $\varphi_1^{(e)}$, $\varphi_2^{(e)}$, and $\varphi_3^{(e)}$, on some arbatrairy element E_e.

6.3. **Global Basis.** We now construct the global basis functions. We chose the global basis functions to be associated with a vertex of the mesh, thus for N mesh vertices are N global basis functions. We want the global basis functions to be 1 at their associated vertex, and at all other mesh vertices, the global basis function should be 0. In order to achieve this we construct the global basis functions as a piecewise combination of n different local basis functions, where n is the number of elements that share a given vertex. There is not a simple mathematical expression for the global basis function, as it greatly depends on the triangular mesh and the current vertex index. However, an example of a global basis function is shown in figure 6.3.

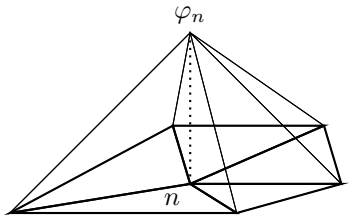


FIGURE 6.3. Gloabal basis function φ_n, demonstrating a potential shape of the global basis functions on a limited portion of a triangular mesh.

6.4. **Local System.** We use our definition of the local basis functions to apply in the construction of an element-specific system of equations from (5.2). Thus we construct our local systems of equations in matrix form as

$$\left(C^{(e)} + K^{(e)}\right) T^{(e)} = F^{(e)}.$$

Here the elements of each matrix are only determined by the local basis function of the element e, and the size of the matrix is equivalent to the number of local degrees fo freedom, which in this case is two (One for each vertex of the element). The elements of these matrices can be found through the following equations

(6.4)
$$c_{ij}^{(e)} = \rho c_p \int_{E_e} \varphi_i^{(e)} \vec{u} \cdot \nabla \varphi_j^{(e)} d\vec{x} \qquad i, j = 1, 2, 3$$

(6.5)
$$k_{ij}^{(e)} = \int_{E_e} \varphi_i^{(e)} \cdot \nabla \varphi_j^{(e)} d\vec{x} \qquad i, j = 1, 2, 3$$

(6.6)
$$f_i^{(e)} = \int_{E_e} \varphi_i^{(e)} f d\vec{x} \qquad i = 1, 2, 3.$$

Thus we only need to compute these local matrix elements, then construct the global system utilizing all of the local matricies.

6.5. **Numerical Integration.** As it can be seen in equation (6.4) we need to be able to integrate over the element E_e. In order to achieve this integration over a triangular element, we implement the Barycentric coordinate systems, and a method of numerical integration called Gaussian Quadrature.

By using the Barycentric coordinates, our integrals become significantly easier to evaluate.

$$\begin{aligned} I &= \int_{E_e} f(x,y)da \\ &= \int_0^1 \int_0^1 f\left(\bar{x},\bar{y}\right) d\lambda_1 d\lambda_2, \end{aligned}$$

where $\bar{x}$ and $\bar{y}$ are the x and y coordinates that are achived from the coordinate transfer from Barycentric to trilinear.

Using this formulation of the integral, any numerical integration methods should suffice. We will implement 2D Gaussian Quadrature, defined on the triangle.

Gaussian Quadrature is a method that approximates the integral such that

$$\int f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i),$$

where w_i, and x_i are specified weights and positions respectivly. For the 2D alternative, we can express this in the form

$$\int_\gamma f(x,y)dxdy \approx \frac{1}{2}\sum_{i=1}^{n} w_i f(x_i,y_i).$$

Where γ is the triangle with vertices $(0,0)$, $(1,0)$, and $(0,1)$. This is the exact vertices that are provided by the triangle over λ_1 and λ_2. Thus we use the Gaussian quadrature implementation in the Barycentric coordinate system, and convert the coordinates into trilinear coordinates in order to evaluate whatever function at the point.

The methods for constructing the weights and positions of sample points for this form of integration is out of the scope of this paper, and so will be ignored. We utilize the weights and points constructed by [HKA12].

7. 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".

7.1. 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.

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 7.1 (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.

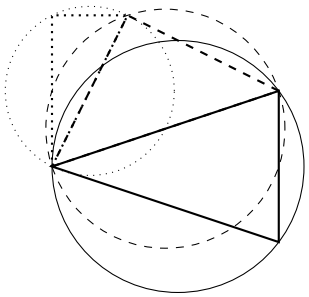


Figure 7.4. Demonstration of the Circumcircle definition of Delaunay. This shows how the circumcircle of each triangle does not contain any vertex from any other triangles.

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.

7.2. **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.

This sense of closeness is such that it is a few modifications from an actual Delaunay triangulation, that still enforces the constrained edges.

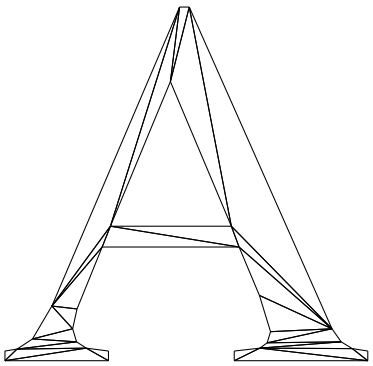


FIGURE 7.5. Constraied Delaunay triangulation. Of the letter A. This mesh has been constructed without any quality or refinement.

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.

7.3. **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.

7.3.1. *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.

7.3.2. *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.

7.4. **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 7.2 (Well-shaped)**.** A triangle is well-shaped if all its angles are greater than or equal to some angle α (commonly $\alpha = 30°$).

Definition 7.3 (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 7.2) triangle

that fits within a circle of radius δ would satisfy the grading function.

Definition 7.4 (*Nice* Triangle)**.** A triangle is *nice* if it is both well-shaped(Def 7.2) and well-sized(Def 7.3).

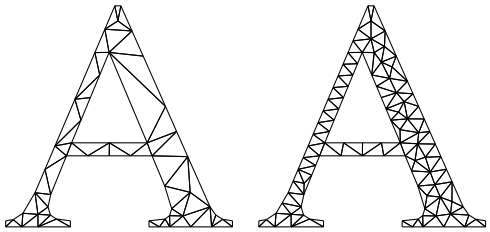
The use of a non-constant grading function is to allow dynamically sized meshes. Such that in areas of interest there can be significantly more refinement, and areas of less interest can be generalized. The restrictions on the function simply stated is that there must be some size of a triangle that 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 7.4).
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 7.3. 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.



(a) (b)

FIGURE 7.6. These two images demonstrate the progressive refinement of a mesh depending of the refinement restrictions of angle and area. (a) 60 triangles, (b) 132 triangles.

7.5. **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++. However, for mesh refinement, we utilized the open source software Triangle[She96], which implements the Delaunay triangulation construction and mesh refinement.

8. Assembly of the Global System

It is possible to directly construct the global matrix, however, this method is ineffective for practical computation and so in this paper, we will ignore the direct computation.

The method that we will use is an incremental construction of the global matrix, through the addition of elements from the local matrices. We begin by initializing the matrix $A = C + K$ and F of zeros. A is a $N \times N$ matrix, and F is a $N \times 1$ matrix, where N is the number of vertices in the triangulation. Then the corresponding contributions of the element matrices $A^{(e)}$ and $F^{(e)}$ are added in a loop over all elements.

Algorithm 8.1 element-by-element assembly

Let A by a $N \times N$ matrix of zeros.
Let F by a $N \times 1$ matrix of zeros.
for all E_e **do**
  Let $A^{(e)}$ be a 3×3 matrix, such that
  $a_{IJ}^{(e)} = c_{IJ}^{(e)} + k_{IJ}^{(e)} \quad I, J = 1, 2, 3$
  $i = node(e, I) \quad j = node(e, J) \quad I, J = 1, 2, 3$
  $A_{ij} = A_{ij} + A_{IJ}^{(e)}$
  $F_i = F_i + F_I^{(e)}$
end for
return A and F.

Where $node(e, I)$ provides the relation between global and local node numbers. Such that given an element and element vertex number, it returns the global vertex number. With the triangular mesh, there is no nice mathematical formula for this expression, but it is simple for any implementation.

9. Dirichlet Boundary Conditions

Now that the global system of equations has been constructed, we are able to impose the Dirichlet boundary conditions to the system.

With our global system of equations in the general form of

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_N \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_N \end{pmatrix}. \tag{9.1}$$

This system of equations is not well behaved, as we have neglected to prescribe the boundary conditions until now. This means that our matrix is overdetermined and cannot be solved. In order to fix this, we must apply the boundary conditions. By the Dirichlet boundary conditions, we know that for any vertex in the mesh that is on the boundary $\vec{x_i} \in \Gamma$,

$$T_i = T_0(\vec{x_i}).$$

We modify the system of equations by overwriting the equations associated with the T_i value, to satisfy the conditions. This can be done like so

$$\begin{aligned} a_{ij} &= \delta_{ij} \quad j = 1, \ldots, N \\ F_i &= T_0(\vec{x_i}), \end{aligned}$$

where δ is the Kronecker delta function

$$\delta_{ij} = \begin{cases} 0 & j \neq i \\ 1 & j = i \end{cases}.$$

This modification must be performed for each vertex of the triangulation that lies on the boundary. This process will present a matrix vaguely of the form

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1(N-1)} & a_{1N} \\ 0 & 1 & \cdots & 0 & 0 \\ a_{31} & a_{32} & \cdots & a_{3(N-1)} & a_{3N} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ a_{N1} & a_{N2} & \cdots & a_{N(N-1)} & a_{NN} \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_{N-1} \\ T_N \end{pmatrix} = \begin{pmatrix} F_1 \\ T_2 \\ F_3 \\ \vdots \\ T_{N-1} \\ F_N \end{pmatrix}.$$

This example shows a matrix where the 1, and $N-1$ vertices lie on the boundary, and so those equations are rewritten.

10. Linear Systems

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 straightforward 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.

10.1. Data Structures.

The first issue that we need to solve is the issue of memory requirements for the large matrices. 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 imply memory efficiency, and the memory usage required to store the matrix.

We assume that the indices 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 dimension 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.

10.1.1. *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 sparsity 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 baseline to compare to, to determine the efficiency of our later methods.

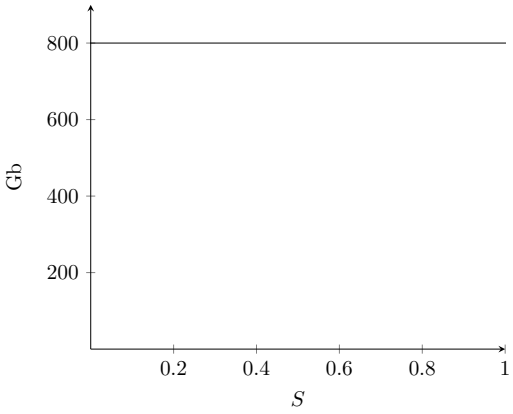


FIGURE 10.7. Memory usage for full matrix storage at different percentages of sparicty.

10.1.2. *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* indices 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 10.1.1.

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.

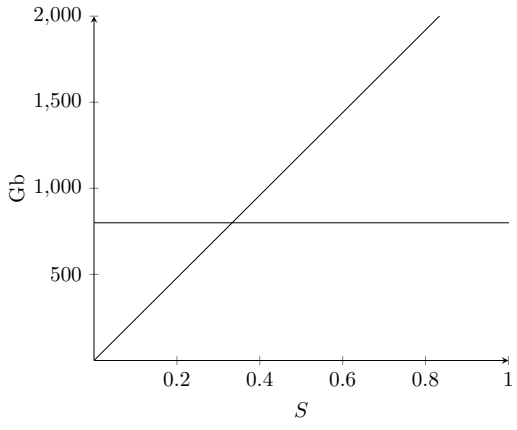


FIGURE 10.8. Memory usage for DOK at different percentages of sparicty.

10.1.3. *Compressed Row Storage.* This method takes 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 exist 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 computational 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 interesting 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 efficient 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 efficient.

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

It is interesting 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 in which this method should be utilized.

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

$$0 < S < 0.5$$

Because of this asymptotic approach of this range of sparsity for maximum efficiency, we use this method of sparse matrix storage in our implementation.

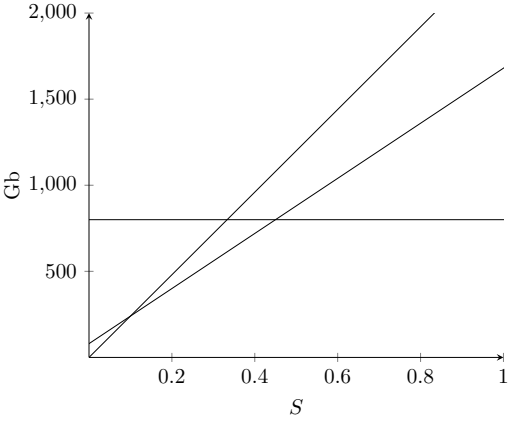


FIGURE 10.9. Memory usage for CRS and DOK at different percentages of sparicty.

10.1.4. *Vectors.* We will similarly have large vectors and can implement a similar method for sparse vector storage, but this would be counterproductive, as our large vectors are not likely to be sparse. Thus using a sparse representation of a vector would actually consume more memory than the full vector storage method.

10.2. **Krylov Subspace.** An extremely useful tool for the numerical approximation of large systems of equations is *Krylov Subspaces.* Krylov subspaces are defined by Krylov[Kry31] as

$$\mathcal{K}_r(A, b) = \text{span}\left\{b, Ab, A^2b, \ldots, A^{r-1}b\right\}$$

where b is some vector. This basis assists in the approximation for linear systems of the form $Ax = b$. The subscript r simply defines the point at which we limit our approximation, if r is infinite then the Krylov subspace is the full space, and solutions in that Krylov subspace will be exact.

Note that the basis vectors in the Krylov subspace are not necessarily orthogonal, nor normal.

10.2.1. *Reasoning.* We provide a short example of the reasoning for the construction of the Krylov subspace. We consider the example where we want to solve a system of linear equations of the form $Ax = b$. Where $A = I + \varepsilon$.

$$\begin{aligned} Ax &= b \\ x &= A^{-1}b \\ x &= (I+\varepsilon)^{-1}b. \end{aligned}$$

We construct an approximation for the inverse of a matrix-like by geometric formula

$$\begin{aligned} I + \varepsilon + \varepsilon^2 + \cdots + \varepsilon^n &= S \\ \varepsilon + \varepsilon^2 + \cdots + \varepsilon^{n+1} &= S\varepsilon \\ S - I + \varepsilon^{n+1} &= S\varepsilon \\ S - S\varepsilon &= I - \varepsilon^{n+1} \\ S(I-\varepsilon) &= I - \varepsilon^{n+1} \\ S &= \left(I - \varepsilon^{n+1}\right)(I-\varepsilon)^{-1}. \end{aligned}$$

When $n \to \infty$ and ε is small enough ($\|\varepsilon\| < 1$), then $\varepsilon^{n+1} \to 0$. Thus

$$\begin{aligned} S &= \left(I - \varepsilon^{n+1}\right)(I-\varepsilon)^{-1} \\ S &= (I-\varepsilon)^{-1}. \end{aligned}$$

Finaly we can conclude that

$$A^{-1} = (I+\varepsilon)^{-1} = I - \varepsilon - \varepsilon^2 - \cdots.$$

Using this in our expression for x we find

$$\begin{aligned} x = A^{-1}b &= b - \varepsilon b - \varepsilon^2 b - \cdots \\ &= b - (A-I)b - (A-I)^2 b - \cdots \\ &= b - Ab + b - A^2b + 2Ab - b + \cdots. \end{aligned}$$

Using this we extract the basis that composes x to be

$$\left\{b, Ab, A^2b, A^3b, \ldots\right\}$$

The Krylov subspace takes this infinite basis and cuts it off at some given r value, thus providing us with

$$\left\{b, Ab, A^2b, \ldots, A^{r-1}b\right\}$$

Which is our Krylov subspace? This is not a full proof of the construction of the Krylov subspace, but it should be a sufficient explanation for our purposes.

10.3. **Stationary Iterative Methods.** All iterative methods can be expressed in the form

$$x^{(k)} = Bx^{(k-1)} + c,$$

when B nor c are dependent on the iteration k, then this is a stationary iterative method. We comment on two stationary iterative methods *Jacobi*, and *Gauss-Seidel* methods. There are several other stationary iterative methods, but we limit ourselves to examining only these two methods.

10.3.1. *Jacobi Method.* The Jacobi method is a very simple method for solving linear systems. It is very easy to understand and implement, but the convergence of the approximation can be very slow.

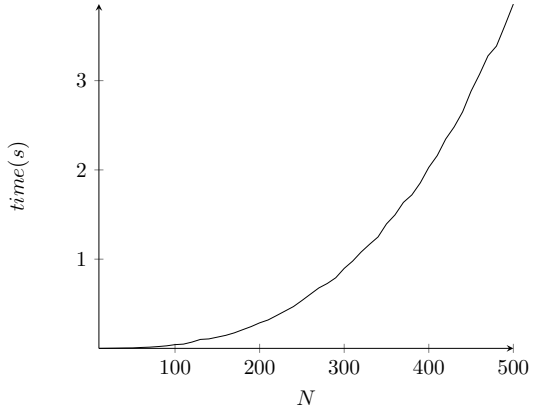


FIGURE 10.10. Time complexity of Jacobi method.

The concept for the Jacobi method is to solve for each variable x_i one at a time, with the assumption that all other variables are constant. We solve each equation in isolation, and then the iteration provides us with our method for converging to the actual solution. We consider the $i$he equation in the system,

$$\sum_{j=1}^{n} a_{i,j}x_j = b_i.$$

We then solve for the value of x_i assuming all other values of x are fixed. We determine that

$$x_i = \frac{b_i - \sum_{j \neq i} a_{i,j}x_j}{a_{i,i}}.$$

We then implement the iteration to find the equation

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{i,j}x_j^{(k-1)}}{a_{i,i}}.$$

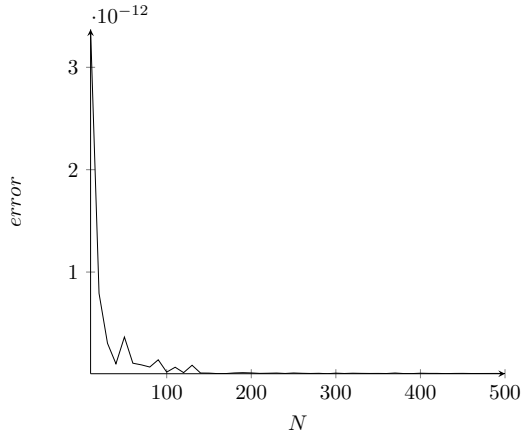


FIGURE 10.11. Error of Jacobi method.

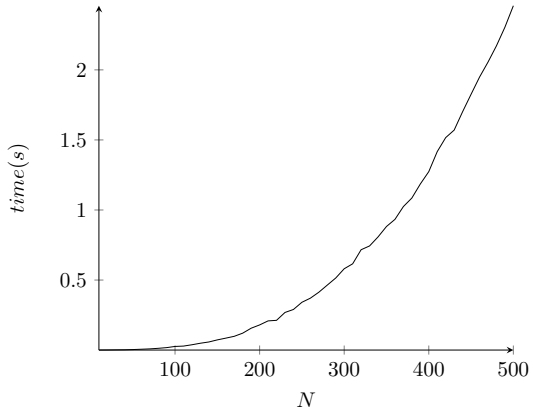


FIGURE 10.12. Time complexity of Gauss-Seidel method.

Using this formula for the computation of the new approximation vector x

Algorithm 10.2 Jacobi Method

```
x = 0
for k = 1, 2, ... do
    x̄ = 0
    for i = 1, 2, ..., n do
        for j = 1, 2, ..., i − 1, i + 1, ..., n do
            x̄_i += a_{i,j} x_j
        end for
        x̄_i = (b_i − x̄_i) / a_{i,i}
    end for
    x = x̄
    if ||Ax − b|| ≤ 10^{−10} then
        return x
    end if
end for
return x
```

Pseudocode for the Jacobi method is shown in Algorithm 10.2. Note the necessity for a temporary $\bar{x}$ term, as the values of x are needed for the computation of the next iteration. We stop the iteration either when a maximum iteration count is reached, or when our approximation is within 10^{-10} of the actual solution. It is possible to alter this tolerance to achieve either more precise approximation or to accelerate convergence at the cost of accuracy.

10.3.2. *Gauss-Seidel Method.* The Gauss-Seidel method is an extension of the Jacobi method, with one optimization. Instead of using the old values of x, the algorithm uses the already computed values of the current iteration.

This method does not improve the rate of convergence significantly, but it is a simple to implement improvement over the Jacobi method. The mathematics and the pseudocode are almost identical to that of the Jacobi method, with the only difference being that instead of using $\bar{x}$ all values are taken from x.

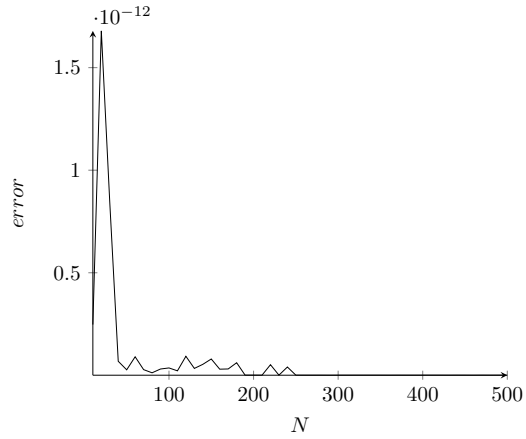


FIGURE 10.13. Error of Gauss-Seidel method.

Because of this similarity, we do not explain in detail the mathematics, but the pseudocode is provided in Algorithm 10.3.

Algorithm 10.3 Gauss-Seidel Method

```
x = 0
for k = 1, 2, ... do
    for i = 1, 2, ..., n do
        x_i = 0
        for j = 1, 2, ..., i − 1, i + 1, ..., n do
            x_i += a_{i,j} x_j
        end for
        x_i = (b_i − x_i) / a_{i,i}
    end for
    if ‖Ax − b‖ ≤ 10^{−10} then
        return x
    end if
end for
return x
```

10.4. **Nonstationary Iterative Methods.** Nonstationary methods differ from their stationary counterparts because the computation involves information that is altered for every iteration. E.g. the values of B and c can depend on the iteration count k. We discuss the nonstationary iterative method called the *Conjugate Gradient* method. All of the nonstationary iterative methods utilize Krylov subspaces, with the exception of a few which we do not discuss.

10.4.1. *Conjugate Gradient Method.* This method is extremely effective for symmetric positive definite systems, and it is one of the most researched nonstationary methods. This method is based on the construction of a sequence of approximations. Each new approximation is generated based on the direction required to minimize the residual.

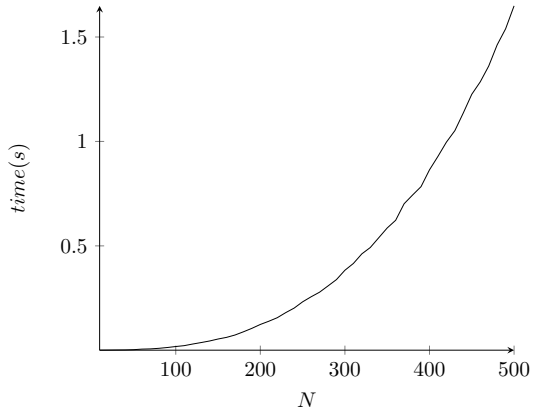


FIGURE 10.14. Time complexity of Conjugate Gradient method.

This is done by determining which "direction" must be used in order to minimize the residual, then the approximation is moved by some α amount in that direction. After repeating this process for some number of iterations, the generated approximation should be converging to the actual solution.

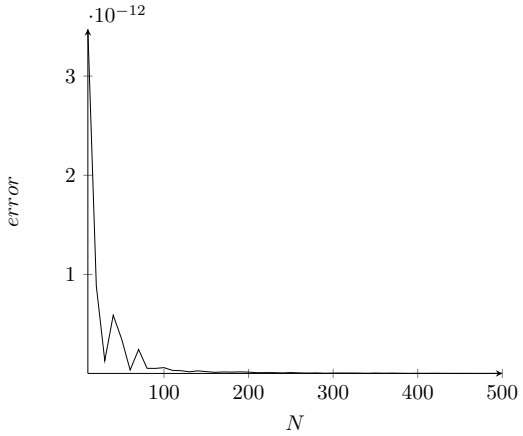


FIGURE 10.15. Error of Conjugate Gradient method.

We construct the iterative approximation in each iteration by a multiple of α_i of the search direction $p^{(i)}$,

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}.$$

We then update the residual $r^{(i)} = b - Ax^{(i)}$ as

$$q^{(i)} = Ap^{(i)}$$
$$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$$

We construct α_i inorder to minimize ${r^{(i)}}^T A^{-1} r^{(i)}$,

$$\alpha_i = \frac{{r^{(i-1)}}^T r^{(i-1)}}{{p^{(i)}}^T A p^{(i)}}.$$

Then to updated the search direcitons using the residual

$$\beta_i = \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i-1)}}^T r^{(i-1)}}$$
$$p^{(i)} = r^{9)} + \beta_{i-1} p^{(i-1)}$$

where the choice of β_i ensures that $r^{(i)}$ and $r^{(i-1)}$ are orthogonal.

This process ties together the construction of the Krylov subspace, and the construction of the approximation based off of the most recently computed krylov basis vector. This means that if we allow our iterations to proceed until N iterations, then we should have constructed the complete Krylov subspace, and thus our approximation should be exact.

Using this approximation we can construct our pseudocode of the algorithm. Which is presented in Algorithm 10.4.

Algorithm 10.4 ConjugateGradient

$x = 0$
$r = b - Ax$
for $i = 1, 2, \ldots, n$ **do**
  $\rho^{(i)} = \|r\|^2$
  if $i = 0$ **then**
    $p = r$
  else
    $\beta = \frac{\rho^{(i)}}{\rho^{(i-1)}}$
    $p = r$
  end if
  $q = Ap$
  $\alpha = \frac{\rho^{(i)}}{\langle p,q \rangle}$
  $x \mathrel{+}= \alpha p$
  $r \mathrel{-}= \alpha q$
  if $\|r\| \leq 10^{-10}$ **then**
    return x
  end if
end for
return x

10.5. **Cholesky Method.** The Cholesky method is a method for solving systems of linear equations based upon the Cholesky decomposition of a matrix. This is the only direct method that we consider. A direct method does not depend on the convergence of a sequence, but instead directly solves the system of equations. This method is useful for the purpose of time prediction. The convergence of iterative methods greatly depends on the system being considered, using this method the time required is directly related to the number of elements. This means that each system of the same size will require the same time to solve.

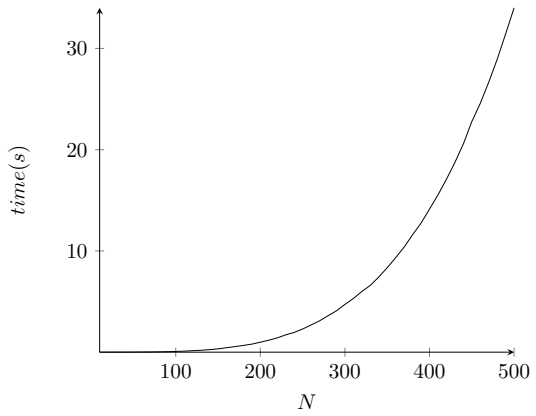


FIGURE 10.16. Time complexity of Cholesky method.

The Cholesky decomposition of a matrix provides a lower triangular matrix L of the form

$$A = LL^T$$

We use this decomposition of A to solve two systems of linear equations, using forward and back substitution, which is an efficient method for finding these solutions.

$$Ly = b \quad \text{by forward substitution}$$
$$L^T x = y \quad \text{by back substitution.}$$

This provides the solution x to the system of linear equations.

We present a plot of the time requirements for the computation in Figure 10.16.

10.5.1. *Cholesky Decomposition.* We utilize the Cholesky-Banachiewicz algorithm for the construction of the lower triangular matrix L. This algorithm states that

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}$$
$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \text{ for } i > j.$$

This construction of the Cholesky decomposition is very easy, and fairly efficient. Thus allowing us to preform this decomposition easily.

10.5.2. *Forward/Back Substitution.* The method of forward and back substitution is an efficient method for solving systems of linear equations of the form $Lx = b$ where L is a lower triangular matrix. Note that the process is almost identical for an upper triangular matrix, just with the order of iteration flipped. We will only provide an explanation for forward substitution.

The first stage is to write the system of equations like so

$$\begin{array}{llll} L_{1,1}x_1 & & & = b_1 \\ L_{2,1}x_1 & +L_{2,2}x_2 & & = b_2 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n,1}x_1 & +L_{n,2}x_2 & \cdots \quad L_{n,n}x_n & = b_2 \end{array}$$

We can clearly solve for x_1 directly, and find

$$x_1 = \frac{b_1}{L_{1,1}}.$$

It is possible to substitute this into the second equation. This will cause the second equation to only have one unknown x_2, and so we can directly solve

for x_2. This process is repeated for all x. Providing the general expression

$$x_m = \frac{b_m - \sum_{i=1}^{m-1} L_{m,i} x_i}{L_{m,m}}.$$

This expression can then be used to compute the solution to the system of linear equations.

Using the Cholesky decomposition and forward and back substitution, we are able to compute the exact solution to the system of equations, without resorting the inefficient Gaussian elimination.

We provide pseudocode of this method in Algorithm 10.5.

Algorithm 10.5 Cholesky method

```
                    ▷ Calculate A = LL^T decomposition.
for j = 0, 1, 2, ..., N do
    L_{j,j} = 0
    for k = 0, 1, 2, ..., j do
        L_{j,j} += L_{j,k}^2
    end for
    L_{j,j} = sqrt(A_{j,j} - L_{j,j})
    for i = j, j+1, j+2, ..., N do
        L_{i,j} = 0
        for k = 0, 1, 2, ..., j do
            L_{i,j} += L_{i,k} L_{j,k}
        end for
        L_{i,j} = (A_{i,j} - L_{i,j}) / L_{j,j}
    end for
end for
          ▷ Calculate Ly = b by forward substitution.
y = 0
for i = 1, 2, ..., N do
    y_i = 0
    for j = 1, 2, ..., i do
        y_i += L_{i,j} y_j
    end for
    y_i = (b_i - y_i) / L_{i,i}
end for
           ▷ Calculate L^T x = y by back substitution.
x = 0
for i = N, N-1, ..., 1 do
    x_i = 0
    for j = i+1, i+2, ..., N do
        x_i += L^T_{i,j} x_j
    end for
    x_i = (b_i - x_i) / L^T_{i,i}
end for
return x
```

10.6. **Issues.** We note that these algorithms have requirements on the form of the matrix A. For many of these, the algorithm requires a symmetric positive definite matrix. We comment on some of these restrictions here.

We note that future work must be done to determine if the global matrix that is constructed by the Finite Element Method will consistently satisfy these restraints. If it does then it is possible to provide extra optimizations based on the form of the global matrix. If a general form of the matrix cannot be guaranteed, then we will be forced to utilize significantly slower methods of computing the solution to the linear system.

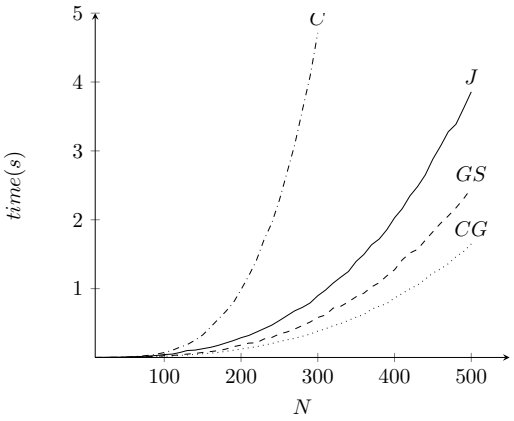


FIGURE 10.17. Plots of the different time requirements for solving the linear systems.

10.6.1. *Possitive Definite.* All of these algorithms require a positive definite matrix. This is required to guarantee that there is a unique solution to the system. This can be considered to ensure that there is a minimum in some N dimensional space, instead of a saddle, or maximum. Only with a minimum can we minimize our residual and use gradient descent to find our equilibrium solution.

10.6.2. *Symmetric.* Most of these algorithms, with the exception of the Cholesky method, require the matrix to be symmetric. This requirement is mostly for computational efficiency and simplicity of the algorithm. There are some methods that can be adapted to function for nonsymmetric matrices, but those become significantly more complex. For most purposes, we should be able to make use of our implementations of these algorithms for finding the solution to the linear system.

11. Solving the Algebraic System

Using the algorithmic method for constructing an approximate solution to the linear system, we can now construct our final approximation of T. By substituting the values of T_j into the equation (5.1), we

find that our approximation is

$$T_h = \sum_{j=1}^{N} T_j \varphi_j. \tag{11.1}$$

where T_j was solved for from the system of equations, and φ_j is the global basis function corresponding to the jth vertex of the element mesh.

12. Further Iterations

Because we chose to only examine the steady state solution, and thus we neglected the time derivative, this indicates that our approximation T_h is the final solution for the problem at hand.

13. Postprocessing

The post-processing step is greatly dependent on the purpose of the FEA. Generating graphs of the solution, or calculating errors are done in this stage. Our implementation has a method for plotting the final solution on the mesh, and an example of this is shown in figure 13.18. Not that that plot is not of an actual solution to this problem, but is intended to only demonstrate the intention of this step.

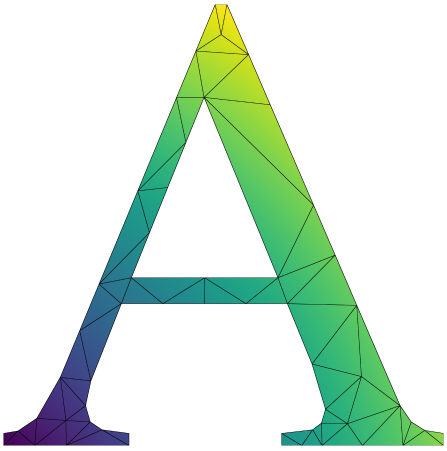


Figure 13.18. Plot of the function $f(x, y) = \sqrt{x^2 + y^2}$ on the mesh of the letter A.

14. Conclusion

The process of Finite Element Analysis is extremely useful in practical implementations for numerically solving partial differential equations on some provided domain. The main stages of the process are to first determine the continuous variational formulation of the partial differential equation, then use that expression to construct the system of equations. We then generate a mesh, and determine the system of equations for each individual element, and use the element systems to modify the global system of equations. Then utilizing an algorithm for approximating the solution to the system of linear equations, we are then able to determine the final approximation of the system.

An advantage of this method is the amount of the process which can be automated. The only need for human computation is for changing the differential equation, the dimension of the problem (1D, 2D, or 3D), and for changing the conditions at the boundaries (Dirichlet, Neuman, Newton, etc.). The rest of the process can be completely handled by a computer. Because of this, many variables such as the shape of the geometry, and the values at the boundaries can be altered without any need for a new program.

Finite Element Methods is one of the industry leading methods for the approximation of partial differential equations, and is frequently used in many applications. This basic introduction to the process should provide an the introductory level of knowledge on the mathematics, and the algorithms that are utilized to implement finite element methods.

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.

[HKA12] Farzana Hussain, M. S. Karim, and Razwan Ahamad. Appropriate gaussian quadrature formulae for triangle. *International Journal of Applied Mathematics and Computation*, 4(1):24–38, 2012.

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

[Kry31] A. N. Krylov. On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *News of Academy of Sciences of the USSR*, 7(4):491–539, 1931.

[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.