

# FINITE ELEMENT ANALYSIS

ARDEN RASMUSSEN

APRIL 28, 2019

ABSTRACT. Finite Element Analysis (FEA) is the computational process prescribed by Finite Element Methods (FEM) in order to solve for numerical solutions to boundary value problems for partial differential equations. Finite Element Analysis is used in many specializations of mechanical engineering, and is used frequently in that discipline, and occasionally in other disciplines as well. This paper provides an introduction to the mathematics of Finite Element Methods, and why they are useful. Then, it provides and explains the simplistic implementation of Finite Element Analysis. The implementation provided with this paper is written in C++, and is commented for better readability of the code.

Spatial Discretization Rewrite

Temporal Discretization

Construction of the Approximation

Temporal Iterations

Figures

Appendix

Finite element methods is a computational process used for the determination of an approximate solution to a given partial differential equation, with given forcing term and boundary conditions. Finite element analysis is the practical implementation of finite element methods computationally on a physical problem in order to ascertain more insight into the physical system in question. We will first explain finite element methods, and provide a thorough introduction into the mathematics behind the methods. Then we will also provide an explanation of finite element analysis, demonstrating the mathematics of FEM with respect to an example problem, and develop a computational program that we will use to solve the partial differential equation using FEM.

Finite element methods is very useful, due to the fact that there are only a few situations where a solution to a partial differential equation can be found analytically, so we need methods to find a close approximation of the actual solution. Currently there are a handful of methods that can be used for this purpose, however finite element methods are currently the best methods that we know of. Meaning that the error between the analytical solution and the solution constructed by finite element analysis converges to zero faster than other methods.

The process that we utilize for constructing the finite element methods is called the Galerkin method, and is presented by [KH15]. We will closely follow the method

outlined by their paper. For this introduction we will limit the scope to two dimensions, but the mathematics can be easily generalized to higher dimensions, and we will make note of the sections that would have to change for higher dimension problems.

We divide this paper into two major parts. Part 1 explains the process of FEM, and demonstrates the mathematics and the concepts behind the method, and how it can be used to solve the partial differential equations. Part 2 explains the process of FEA, and discusses the algorithms, data structures, and optimizations that can be done to construct a program to efficiently compute the solution utilizing the previously discussed FEM.

---

## Part 1. Mathematical Theory

---

### 1. PROBLEM STATEMENT

Throughout this paper and our sample code, we implement finite element methods using the general form of the heat equation as our differential equation to consider. We define the domain to be  $\Omega$  such that  $\Omega \in \mathbb{R}^2$  is a bounded 2D domain with boundary  $\Gamma$ .

The general form of the heat equation takes the form

$$\frac{\partial u}{\partial t} = Lu + f,$$

where  $L$  is an operator of the form

$$L = \sum_{\alpha, \beta=1}^2 A^{\alpha\beta} \frac{\partial}{\partial x^\alpha} \frac{\partial}{\partial x^\beta} + \sum_{\alpha=1}^2 B^\alpha \frac{\partial}{\partial x^\alpha} + C$$

and  $f$  is some forcing function. We will also enforce that  $A$  is a positive definite matrix. And where  $u$  is a function that takes the position and the time and returns a real  $u(\vec{x}, t) : \mathbb{R}^2 \times \mathbb{R} \rightarrow \mathbb{R}$ . Note that  $\vec{x} \in \Omega$ , and  $t \in \mathbb{R}$ .

Rewriting this expression into a single equation we can express our problem statement as

$$(1.1) \quad \frac{\partial u}{\partial t} = \sum_{\alpha, \beta=1}^2 A^{ij} \frac{\partial}{\partial x^\alpha} \left[ \frac{\partial u}{\partial x^\beta} \right] + \sum_{\gamma=1}^2 B^\gamma \frac{\partial u}{\partial x^\gamma} + Cu + f$$

Note that for this process to function, we must also require some conditions. We first need the boundary conditions in order to determine any solution. Therefore we require that  $u(\partial\Omega, t)$  is provided. Note that this could mean that the boundary is split into several sections, each one with different conditions, but the value of  $u$  at any point along the boundary must be known.

If considering the more general time dependent form of the equation we are also required to know initial conditions. That is to say that  $u(\Omega, 0)$  must also be provided. Using the initial conditions and the boundary conditions, we are able to construct an approximation through the use of finite element methods.

Given  $A, B, C, f$ , and appropriate conditions, we must construct a solution for  $u$ . This is the key purpose of finite element methods, as for most circumstances there does not exist an analytical solution to this differential equation. Thus we must utilize finite element analysis in order to construct the best possible approximation of the solution to the differential equation, and then it is possible to use that approximation for further analysis.

### 2. CONDITIONS

For our proposed problem statement, we will require some conditions in order to make it possible to determine a solution to the problem. The conditions that we will require are boundary conditions and if we are considering a time dependent system, we will also need an initial condition.

**2.1. Boundary Conditions.** We must prescribe boundary conditions on our problem statement, to ensure that there is a single solution. For now we will only proceed utilizing Dirichlet boundary conditions. It should be noted that other conditions are possible with this method, such as Neumann, and Newton boundary conditions. These other boundary conditions require 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.

For the Dirichlet boundary conditions we must define the value of our solution at every point on the boundary. We define this as

$$\partial u(\vec{x}, t) \quad \vec{x} \in \partial\Omega.$$

We will use this notation to denote the function defining the values of the solution to the problem along the boundary at time  $t$ . This can be written as

$$u(\partial\Omega, t) \equiv \partial u(\partial\Omega, t).$$

It is important to note that there is no restrictions to the form of  $\partial u$ . The user defined function can take any form, and thus requiring the solution to also fit any form of boundary conditions.

**2.2. Initial Conditions.** If the problem being considered were to be time dependent, then it is also necessary to define an initial condition to the problem. That is to say that it is required to provide some function that defines the value of  $u$  at  $t = 0$ . We define this function as

$$u_0(\vec{x}) \quad \vec{x} \in \Omega.$$

We use this notation to denote the function defining the values of the solution to the problem, that is to say

$$u(\Omega, t = 0) \equiv u_0(\Omega).$$

Note that this function also has no restrictions other than that it must be defined on the entirety of the domain  $\Omega$ , although it need not be continuous on the domain.

If the problem being considered is not time dependent, then the initial condition can be ignored, and only the boundary conditions must be specified.

Using these prescribed conditions, both the boundary and the initial conditions, it is now possible to utilize finite element methods to attain an approximation of the solution to the problem statement.

### 3. VARIATIONAL FORMULATION

In order to construct the variational formulation of our problem statement, we must first introduce the concept of a residual. The residual is the construction of a function  $R$  such that when our approximation is the exact solution then  $R = 0$ . Using the residual we now want to construct a function that minimizes the value of the residual. We will construct the residual of equation 1.1 as

$$R(\vec{x}, t) = \frac{\partial u}{\partial t} - Lu - f$$

Clearly by the construction of  $R$ ,  $R(\vec{x}, t) = 0$  if and only if  $u$  is the exact solution to the partial differential equation. Now we multiply the residual by some arbitrary

test function, and integrate over our domain

$$\int_{\Omega} w(\vec{x})R(\vec{x},t)dA = 0.$$

For the exact solution  $u$  this will always be zero, because  $R(\vec{x},t) = 0$  for all  $\vec{x} \in \Omega$ . Because of this we are able to select any test function  $w$ , and the weighted residual will still be zero, given the exact solution.

We now expand the weighted residual to obtain

$$\int_{\Omega} w \frac{\partial u}{\partial t} dA - \int_{\Omega} w L u dA - \int_{\Omega} w f dA = 0.$$

Rearranging this expression we clearly see that

$$\int_{\Omega} w \frac{\partial u}{\partial t} dA = \int_{\Omega} w L u dA + \int_{\Omega} w f dA.$$

Expanding our the  $L$  operator on  $u$  and splitting the integral over the sum, we find

$$\begin{aligned} \int_{\Omega} w \frac{\partial u}{\partial t} dA &= \sum_{\alpha, \beta=1}^2 \int_{\Omega} w A^{\alpha\beta} \frac{\partial}{\partial x^{\alpha}} \left[ \frac{\partial u}{\partial x^{\beta}} \right] dA + \sum_{\gamma=1}^2 \int_{\Omega} w B^{\gamma} \frac{\partial u}{\partial x^{\gamma}} dA \\ &\quad + \int_{\Omega} w C u dA + \int_{\Omega} w f dA. \end{aligned}$$

We notice that  $w$  is always independent of time so that the first integral can be rewritten as

$$\int_{\Omega} w \frac{\partial u}{\partial t} dA = \frac{\partial}{\partial t} \int_{\Omega} w u dA.$$

We can now apply integration by parts on the second integral, and rewrite this as

$$\int_{\Omega} w A^{\alpha\beta} \frac{\partial}{\partial x^{\alpha}} \left[ \frac{\partial u}{\partial x^{\beta}} \right] dA = \int_{\partial\Omega} A^{\alpha\beta} w \frac{\partial u}{\partial x^{\beta}} dA - \int_{\Omega} \frac{\partial A^{\alpha\beta} w}{\partial x^{\alpha}} \frac{\partial u}{\partial x^{\beta}} dA.$$

Since our restriction on the selection of of test functions  $w$ , we know that  $w(\vec{x}) = 0 \forall \vec{x} \in \partial\Omega$ . Thus the integral over the boundary must be zero. Using this simplification, we obtain the representation of the continuous variational formulation of the problem to be

$$\begin{aligned} \frac{\partial}{\partial t} \int_{\Omega} w u dA &= - \sum_{\alpha, \beta=1}^2 \int_{\Omega} \frac{\partial A^{\alpha\beta} w}{\partial x^{\alpha}} \frac{\partial u}{\partial x^{\beta}} dA + \sum_{\gamma=1}^2 \int_{\Omega} B^{\gamma} w \frac{\partial u}{\partial x^{\gamma}} dA \\ (3.1) \quad &\quad + \int_{\Omega} C w u dA + \int_{\Omega} w f dA. \end{aligned}$$

#### 4. SPACIAL DISCRETIZATION

Now that we have a variational formulation of the partial differential equation, we wish to discretize the equation, so that it is possible to apply the method of finite elements.

In order to discretize the problem, we must have an understanding of the space that the solution resides in. It is clear that the solution to the equation will be in the Sobolev space  $W^{1,2}(\Omega)$ . The Sobolev space is the set of functions functions that are infinitely differentiable. In this case we can tell that our solution resides in the specific Sobolev space  $W^{1,2}(\Omega)$ . It is also important to mention that this

Rewrite this section to specify that this is only the spacial discretization.

can be written as  $H^1(\Omega)$ , because this Sobolev space is also a Hilbert space. A Hilbert space is one that possess the structure for inner product. Thus we are able to conclude that our solution exists in a space that supports inner product, and is infinitely differentiable.

Since the process of finite element methods is inherently intended for computation, problems arise when infinities come into play. Since the span of the Sobolev space  $H^1(\Omega)$  is infinite, we are required to consider a subspace of it. This will be the subspace of the Sobolev space that our approximation of the solution lives in, we will call this subspace  $\tilde{V} \subset H^1(\Omega)$ .

Specify that the approximation without bc only exists in  $H_0^1(\Omega)$ , or the values that are 0 on the boundaries.

We will construct a finite set of basis functions for the subspace  $\tilde{V}$  to be  $\{\varphi_j\}$ . For now we will leave these basis functions as arbitrary functions, but the process for the construction of the basis functions will be presented in a later section.

The approximation of  $u$  we define as  $u$ , since the approximation should converge to the exact solution, utilizing the same notation does not cause any issues. We construct this approximation such that it is an element of the subspace of the Sobolev space as we have defined. That is to say

$$u \in \tilde{V} \subset H^1(\Omega)$$

Since our approximation is constructed in  $\tilde{V}$ , then it must be expressible as a linear combination of the basis function  $\varphi_j$  of the finite subspace  $\tilde{V}$  that  $\tilde{u}$  is an element of. Thus we can express  $\tilde{u}$  as

$$(4.1) \quad u = \sum_{j=1}^N u_j(t) \varphi_j(\vec{x}),$$

where  $u_j(t)$  is a function of time that once it is found, it is possible to construct the final approximation. Using this expression for our approximation the goal of the method is to determine the functions  $u_j(t)$ .

We can now substitute our expression for  $u$  into the continuous variational formulation of the problem (3.1).

$$(4.2) \quad \begin{aligned} \frac{\partial}{\partial t} \int_{\Omega} w \left[ \sum_{j=1}^N u_j \varphi_j \right] dA = & - \sum_{\alpha, \beta=1}^2 \int_{\Omega} \frac{\partial A^{\alpha\beta} w}{\partial x^{\alpha}} \frac{\partial}{\partial x^{\beta}} \left[ \sum_{j=1}^N u_j \varphi_j \right] dA \\ & + \sum_{\gamma=1}^2 \int_{\Omega} B^{\gamma} w \frac{\partial}{\partial x^{\gamma}} \left[ \sum_{j=1}^N u_j \varphi_j \right] dA \\ & + \int_{\Omega} C w \left[ \sum_{j=1}^N u_j \varphi_j \right] dA + \int_{\Omega} w f dA. \end{aligned}$$

It is possible to rewrite this expression, with several simplifications. First is to note that since  $w$  and  $\varphi_j$  are independent of  $t$ , and  $u$  is independent of  $\vec{x}$ . Using this it is possible to rewrite the first term, by moving constants with respect to the integral ( $u_j$ ) out of the integral, and then to move constants with respect to the derivative (the integral) out of the derivative. Thus we can write the first term as

$$\frac{\partial}{\partial t} \int_{\Omega} w \left[ \sum_{j=1}^N u_j \varphi_j \right] dx \vec{x} = \sum_{j=1}^N \left[ \frac{\partial u_j}{\partial t} \int_{\Omega} w \varphi_j dA \right].$$

Similar process can be applied to the other terms in equation 4.2. By pulling the constants out of the integrals, and splitting the integral over the summation, we are able to rewrite the expression as

$$(4.3) \quad \sum_{j=1}^N \left[ \frac{\partial u_j}{\partial t} \int_{\Omega} w \varphi_j dA \right] = - \sum_{j=1}^N \left[ u_j \sum_{\alpha, \beta=1}^2 \int_{\Omega} \frac{\partial A^{\alpha\beta} w}{\partial x^{\alpha}} \frac{\partial \varphi_j}{\partial x^{\beta}} dA \right] \\ + \sum_{j=1}^N \left[ u_j \sum_{\gamma=1}^2 \int_{\Omega} B^{\gamma} w \frac{\partial \varphi_j}{\partial x^{\gamma}} dA \right] \\ + \sum_{j=1}^N \left[ u_j \int_{\Omega} C w \varphi_j dA \right] + \int_{\Omega} w f dA.$$

Now, since we have not placed any restrictions on  $w$ , and it can be any function, we will chose the basis functions  $\varphi_i$  as our test functions. Substituting that into the expression we obtain the discretized version of the variational formulation of the problem.

$$(4.4) \quad \sum_{j=1}^N \left[ \frac{\partial u_j}{\partial t} \int_{\Omega} \varphi_i \varphi_j dA \right] = - \sum_{j=1}^N \left[ u_j \sum_{\alpha, \beta=1}^2 \int_{\Omega} \frac{\partial A^{\alpha\beta} \varphi_i}{\partial x^{\alpha}} \frac{\partial \varphi_j}{\partial x^{\beta}} dA \right] \\ + \sum_{j=1}^N \left[ u_j \sum_{\gamma=1}^2 \int_{\Omega} B^{\gamma} \varphi_i \frac{\partial \varphi_j}{\partial x^{\gamma}} dA \right] \\ + \sum_{j=1}^N \left[ u_j \int_{\Omega} C \varphi_i \varphi_j dA \right] + \int_{\Omega} \varphi_i f dA.$$

It can be useful to examine this equation using the inner products as they act upon functions. That is to say, that we use the  $L^2$  inner product defined as

$$\langle f, g \rangle_{L^2} = \int_{\Omega} f g dA.$$

Using this notation, the expression can be rewritten to be

$$\sum_{j=1}^N \langle \varphi_i, \varphi_j \rangle \frac{\partial u_j}{\partial t} = - \sum_{j=1}^N \sum_{\alpha, \beta=1}^2 \left\langle \frac{\partial}{\partial x^{\alpha}} [A^{\alpha\beta} \varphi_i], \frac{\partial \varphi_j}{\partial x^{\beta}} \right\rangle u_j \\ + \sum_{j=1}^N \sum_{\gamma=1}^2 \left\langle B^{\gamma} \varphi_i, \frac{\partial \varphi_j}{\partial x^{\gamma}} \right\rangle u_j + \sum_{j=1}^N \langle C \varphi_i, \varphi_j \rangle u_j + \langle \varphi_i, f \rangle.$$

It is now possible to factor out the  $u_j$  from the right hand side summations. This will provide the equation

$$\sum_{j=1}^N \langle \varphi_i, \varphi_j \rangle \frac{\partial u_j}{\partial t} = \left[ - \sum_{j=1}^N \sum_{\alpha, \beta=1}^2 \left\langle \frac{\partial}{\partial x^{\alpha}} [A^{\alpha\beta} \varphi_i], \frac{\partial \varphi_j}{\partial x^{\beta}} \right\rangle \right. \\ \left. + \sum_{j=1}^N \sum_{\gamma=1}^2 \left\langle B^{\gamma} \varphi_i, \frac{\partial \varphi_j}{\partial x^{\gamma}} \right\rangle + \sum_{j=1}^N \langle C \varphi_i, \varphi_j \rangle \right] u_j + \langle \varphi_i, f \rangle.$$

Since it is possible to replace  $w$  with *any* of the basis functions, that means that  $i = 1, 2, \dots, N$ . This provides us with a system of equations, which is then expressible in terms of matrices. The matrix representation of the system is

$$(4.5) \quad G\partial_t U = (\bar{A} + \bar{B} + \bar{C})U + F.$$

Where  $U$  is a vector in  $\mathbb{R}^N$  composed of the  $u_j$  functions. We use the notation  $\partial_t$  to denote the operator of taking the partial derivative of the  $U$  vector with respect to time, element wise. The elements of  $G$ ,  $\bar{A}$ ,  $\bar{B}$ ,  $\bar{C}$ , and  $F$  are defined as

$$\begin{aligned} G_{ij} &= \langle \varphi_i, \varphi_j \rangle & i, j &= 1, \dots, N \\ \bar{A}^{ij} &= - \sum_{\alpha, \beta=1}^2 \left\langle \frac{\partial}{\partial x^\alpha} [A^{\alpha\beta} \varphi_i], \frac{\partial \varphi_j}{\partial x^\beta} \right\rangle & i, j &= 1, \dots, N \\ \bar{B}^{ij} &= \sum_{\gamma=1}^2 \left\langle B^\gamma \varphi_i, \frac{\partial \varphi_j}{\partial x^\gamma} \right\rangle & i, j &= 1, \dots, N \\ \bar{C}^{ij} &= \langle C \varphi_i, \varphi_j \rangle & i, j &= 1, \dots, N \\ F &= \langle \varphi_i, f \rangle & i, j &= 1, \dots, N. \end{aligned}$$

It is often simpler to consider a matrix  $M$  such that  $M = \bar{A} + \bar{B} + \bar{C}$ , then our expression for the matrix form of the system of equations becomes

$$(4.6) \quad G\partial_t U = MU + F.$$

This matrix representation is how we will continue to consider the problem. For most cases this is the system of equations that will be found. However, this expression is only sufficient for Dirichlet boundary conditions, as the integral over the boundary for other boundary conditions would not be zero. Thus there would be additional terms in the system of equations. This is not an issue with the Dirichlet boundary conditions, so we do not need to consider any additional terms in the matrix representation.

If the problem that is being considered were to be time independent, then the matrix formulation of the system of equations would be

$$MU = F.$$

We will continue to consider both the time dependent and the time independent problems, as both are useful in different situations. However, they are solved in different ways, some simplifications can be made with one but not the other.

## 5. TEMPORAL DISCRETIZATION



---

## Part 2. Mathematical Implementation

---

### 6. LOCALIZATION

While it is occasionally possible to find the elements of the matrices in equation 4.6 on the global domain, this proves to be exceedingly difficult in most if not all cases. To remediate this issue, we implement a generalizable method that will work for any domain and functions that are being considered. Since we have the ability to select any arbitrary basis functions as per our definition of the approximation to the solution, then it is possible to select a basis such that it becomes possible to split the domain into a finite number of subdomains.

We first discretize the domain  $\Omega$  into  $N$  mutually exclusive subdomains. Each of these subdomains is called an *element*. For the purposes of this paper, each element will be defined to be a triangle subdomain of  $\Omega$ . The set of all elements is called the triangular decomposition or the mesh of the domain, the process for constructing the triangular decomposition is discussed further in section 7. For the localization process, the specifics of the triangular decomposition is not necessary, so we will not go in depth on the structure of the triangular decomposition in this section.

For each element of the mesh, we denote the element as  $E^{(e)}$ , where  $e = 1, \dots, N$ . Then each vertex of the triangle is labeled as  $E_i^{(e)}$  where  $i = 1, 2, 3$ . The notation for the  $x$  and  $y$  values of each vertex we denote as  $X_i^{(e)}$  and  $Y_i^{(e)}$  respectively, where  $i = 1, 2, 3$ .

In 1D the process is trivial as all elements have the same dimensions, but in higher dimensional implementation, each element can have variable shapes and sizes. This becomes more difficult because integrating over each element would require entirely different process each time to account for the differences in shape and size. To resolve this issue, we construct a master space, which is possible to transition to from any element, and back to each element. Thus we only need determine a method to integrate over our master element.

**6.1. Master Space.** The master space is a domain that is constructed in order to simplify the construction of the local approximations. For the situation with triangular elements, we construct the master space such that the vertices of a triangle lie on the points  $(1, 0, 0)$ ,  $(0, 1, 0)$ ,  $(0, 0, 1)$ . Then for each element in the mesh, we construct a transformation between the master space and the element space.

We define the master element to be  $\hat{E}$ . An image of the master element can be seen in figure ??.

The master element utilizes what is called a Barycentric coordinate system. The barycentric coordinates are demonstrated by each vertex being its own dimension, so if the element were to be quadrangles, then the barycentric coordinates would be in  $\mathbb{R}^4$  instead of  $\mathbb{R}^3$  as they are for triangles. The conversion between the global coordinates, and the Barycentric coordinates can be done using the following

equations

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

Where  $X_i$  and  $Y_i$  represent the  $x$  and  $y$  coordinates of the  $i$ th vertex of the element.  $x, y$  are the coordinates of the point in the triangle that is to be converted to the Barycentric coordinates. Converting from Barycentric coordinates to the global coordinates uses the following equations

$$\begin{aligned}x &= \lambda_1 X_1 + \lambda_2 X_2 + \lambda_3 X_3 \\ Y &= \lambda_1 Y_1 + \lambda_2 Y_2 + \lambda_3 Y_3\end{aligned}$$

**6.2. Local Basis.** The first step is to construct a local basis function, that are only defined for the current element under consideration. There should be a basis element for every vertex in the element, so for triangular element, there will be three local basis functions. The notation for the local basis functions is  $\varphi_1^{(e)}$ ,  $\varphi_2^{(e)}$ , and  $\varphi_3^{(e)}$ . To make computation simple, the construct the basis function such that they are linear, with the requirement that at the corresponding vertex, the local basis function is 1 and at all other vertices of the element the basis function will be 0.

The conversion from global space to the bilinear coordinates does exactly this task. Thus the definition of the local basis functions are

$$\begin{aligned}\varphi_1^{(e)} &= \frac{(Y_2^{(e)} - Y_3^{(e)})(x - X_3^{(e)}) + (X_3^{(e)} - X_2^{(e)})(y - Y_3^{(e)})}{(Y_2^{(e)} - Y_3^{(e)})(X_1^{(e)} - X_3^{(e)}) + (X_3^{(e)} - X_2^{(e)})(Y_1^{(e)} - Y_3^{(e)})} \\ \varphi_2^{(e)} &= \frac{(Y_3^{(e)} - Y_1^{(e)})(x - X_3^{(e)}) + (X_1^{(e)} - X_3^{(e)})(y - Y_3^{(e)})}{(Y_2^{(e)} - Y_3^{(e)})(X_1^{(e)} - X_3^{(e)}) + (X_3^{(e)} - X_2^{(e)})(Y_1^{(e)} - Y_3^{(e)})} \\ \varphi_3^{(e)} &= 1 - \varphi_1^{(e)} - \varphi_2^{(e)}.\end{aligned}\tag{6.1}$$

Thus for every element there exists three local basis functions. A plot of these local basis functions is shown in figure ?? on an arbitrary triangular element.

**6.3. Global Basis.** The next step is to construct the global basis functions. Similarly to the strategy for the construction of the local basis functions, each global basis will be associated with a single vertex of the mesh. Thus for a mesh with  $N$  elements, then there will be some number of global basis functions less than  $3N$ . The method to define the global basis function is to define it to be 1 at the associated vertex, and 0 at all other vertices in the mesh.

In order to achieve this, the global basis function can be constructed as a piecewise combination of  $k$  different local basis functions, where  $k$  is the number of element that share a given vertex. The exact formulation of the global basis functions is dependent on the triangular mesh, and the current vertex index. An example of what a global basis function could be like is depicted in figure ?. One can intuitively think of the global basis functions as  $k$  sided pyramids this their peak directly above the associated vertex.

**6.4. Local System.** Using the definition of local basis functions that were constructed in section 6.2, the construction of an element-specific system of equations is preformed. The system of equations is identical to the global system, but is simply specified to a single element. Thus the local system of equations in matrix form is

$$G^{(e)} \partial_t U^{(e)} = \left( \overline{A}^{(e)} + \overline{B}^{(e)} + \overline{C}^{(e)} \right) U^{(e)} + F^{(e)}.$$

In this system of equations, the elements of each matrix are only determined by the local basis function on the element  $e$ . The size of the matrix is equivalent to the number of vertices in the element, thus for the triangular element case, the matrices have a size of three. The element of these matrices can be found through the following equations

$$\begin{aligned}
 G_{ij}^{(e)} &= \int_{E_e} \varphi_i^{(e)} \varphi_j^{(e)} dA & i, j = 1, 2, 3 \\
 \overline{A}_{ij}^{(e)} &= - \sum_{\alpha, \beta=1}^2 \int_{E_e} \frac{\partial \varphi_j^{(e)}}{\partial x^\beta} \frac{\partial}{\partial x^\alpha} \left( A_{\alpha\beta} \varphi_i^{(e)} \right) dA & i, j = 1, 2, 3 \\
 \overline{B}_{ij}^{(e)} &= \sum_{\gamma=1}^2 \int_{E_e} B^\gamma \varphi_i^{(e)} \frac{\partial \varphi_j^{(e)}}{\partial x^\gamma} dA & i, j = 1, 2, 3 \\
 \overline{C}_{ij}^{(e)} &= \int_{E_e} C \varphi_i^{(e)} \varphi_j^{(e)} dA & i, j = 1, 2, 3 \\
 F_i^{(e)} &= \int_{E_e} \varphi_i^{(e)} f dA & i = 1, 2, 3.
 \end{aligned}
 \tag{6.2}$$

Using only the local matrix elements, it is possible to construct the global system, utilizing some specific algorithms that are discussed in section 8.

## 7. MESH GENERATION

For the implementation of finite element methods it is required to first discretize the global domain into a set of finite subdomains. Each subdomain is called an element. It is then possible to compute the values of the local system of equations with respect to a given element, then to combine these values to construct the global system. The first step is to discretize the global domain into a set of finite elements.

These elements can be of the form of an  $N$  sided polygon, not that it will not be regular each side will have different lengths. This paper will utilize triangular meshes, but many implementations also implement quadrangle, and hexagonal meshes. Since the construction of meshes with elements with more than three sides is considerably more difficult, we will limit this paper to only discussing the triangular mesh construction.

Constructing the triangular mesh is the most generalized method, as any polygon with more edges can be constructed by several triangles. Thus the mesh of polygons with more than three sides, can be derived from the triangular mesh, without significant difficulty.

The process that will be used for the construction of the triangular mesh is known as Constrained Delaunay Triangulation. Then to improve the mesh and

consequently improve the accuracy of the approximation, a mesh refinement algorithm is utilized to ensure that the elements of the mesh are of an acceptable quality.

**7.1. Delaunay Triangulation.** Delaunay triangulation is the straight line dual of the Voronoi diagram. The Delaunay triangulation of a domain is commonly used in many different situations, including for the purpose of mesh generation for finite element analysis.

The key property of Delaunay triangulation is that no point may lie within the circumcircle of any other triangle.

**Definition 7.1** (Circumcircle). The circumcircle of a triangle, is the circle is defined by having all three vertices of the triangle on the circle. For any given triangle there is only one such circle that satisfies this. In figure ??, a depiction of the circumcircle for a provided triangle is given.

**Definition 7.2** (Delaunay Triangulation). Let  $S$  be a set of points in the plane  $\mathbb{R}^2$ . 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) all of the vertices of the triangle  $t$  are on the boundary of circle  $C$ ,

$$p \in \partial C \quad \forall p \in t$$

- (2) and no other vertex of  $S$  is in the interior of  $C$ .

$$p \notin S \quad \forall p \in \text{Int}(C)$$

By this definition of the Delaunay triangulation, the triangulation is unique for most set of points  $S$ . The only exception to this rule is when a set of four points construct the square. For the square there are two equally valid triangulations, and dependent on the implementation one or the other must be selected.

A major advantage of the Delaunay triangulation for mesh generation is that it inherently avoids triangles with small included angles, as the circumcircles of these triangles would be very large, and would likely include other vertices. This avoidance is extremely useful in finite element analysis, and will produce more accurate approximations of the solution to the partial differential equation.

There are many different algorithms that can be implemented for the construction of the Delaunay triangulation. The notable methods include; Divide and Conquer, Incremental, Sweep Line, and Edge Flipping. The specifics of these algorithms will be discussed further in section ??.

With the Delaunay triangulation constructed, one must next apply the boundaries of the domain to the mesh. The process is called the construction of the Constrained Delaunay Triangulation.

**7.2. Constrained Delaunay Triangulation.** The Constrained Delaunay Triangulation, is a modification of the Delaunay Triangulation such that specified edges and vertices are forced to exist in the triangulation. Because of the forcing of edges into the triangulation, the mesh may not strictly satisfy the definition of Delaunay triangulation. Thus we define the constrained triangulation as the closest triangulation to the Delaunay triangulation that includes all of the required edges.

This sense of “closeness” to the Delaunay mesh is such that we use as few modifications from a Delaunay triangulation in order to enforce the constrained edges to construct the constrained Delaunay triangulation.

The constrained edges are most commonly utilized for the purpose of defining boundaries of the domain, or enforcing a location of a medium change. Since the result of the unconstrained Delaunay triangulation will have a strictly convex shape. This makes it necessary to apply the edge constraints for all domains that do not have a strictly convex outer hull.

There are a number of methods for the construction of the constrained Delaunay triangulation. A few of these implementations enforce the constraints during the initial construction of the triangulation. However, more commonly the constraints are enforced upon an already constructed triangulation. The details on the implementation of these algorithms is discussed in section ??.

**7.3. Mesh Refinement.** The direct output of the constrained Delaunay triangulation algorithms may be sufficient in a few situations, but for most cases, it would result in suboptimal meshes. For generalizability, we discuss the methods that are utilized to refine the generated mesh, thus improving upon the quality of the meshes.

Mesh refinement is almost always required, because along constrained edges, there may only be a single element. This element is forced to have one of its edges be the entity of the constrained edge. This tends to lead to skinny triangles, ones who's minimum interior angle is very small. There are frequently undesirable in the final mesh. We denote that these triangles are not *well-shaped*.

**Definition 7.3** (Well-Shaped). A triangle is well-shaped if all its interior angles are greater than or equal to some defined constant angle  $\alpha$  (commonly  $\alpha \approx 30^\circ$ ).

It is also possible that all the elements in the mesh are well-shaped, but they could all be very large. A finer mesh will always result in more accurate approximations, as will be made clear later. Thus we may also want to define a notion of *well-sized* triangles, so that it is possible to refine the element size in the mesh.

**Definition 7.4** (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 for determining if a triangle satisfies it, as long as there exists a value  $\delta > 0$  such that any well-shaped triangle that fits within a circle of radius  $\delta$  would satisfy the grading function.

This definition of the well-sized triangles, describes that any function can be used in order to define when a triangle is well-sized. This is very important in constructing meshes that are very fine around specified features, and the coarser in the unimportant regions. The one restriction of the grading function is that there must be some lower bound, such that all triangles that fit within the lower bound will be accepted. This enforces that the grading function may not require triangles to become infinitely small.

We will now define what it means for a triangle to be “nice”.

**Definition 7.5** (nice). A triangle is said to be *nice*, if it is both well-shaped (Def 7.3) and well-sized (Def 7.4).

Using these definitions for how to classify the elements of a mesh, it becomes possible to implement algorithms in order to refine any mesh. The technicalities of the mesh refinement algorithms is discussed in 12.2.

## 8. ASSEMBLY OF THE GLOBAL SYSTEM

Although it is possible to directly compute the elements of the global matrices, it is ineffective and not practical for any non trivial problems. Thus we will not consider the explanation of how to construct the global matrices directly.

The method that we utilize is an incremental construction of the global matrix, through the addition of elements from each of the local matrices. We begin by initializing the matrix  $G$ ,  $M = \overline{A} + \overline{B} + \overline{C}$  and  $F$  of zeros.  $G$  and  $M$  are  $N \times N$  matrices, and  $F$  is a  $N \times 1$  matrix, where  $N$  is the number of vertices in the triangulation.

Every element matrix  $G^{(e)}$  and  $M^{(e)}$  are  $3 \times 3$  matrices, and each element matrix  $F^{(e)}$  is a  $3 \times 1$  matrix. We need to construct a method that will allow us to associate the elements of the local matrix to the elements of the global matrix. The first part of this is determining a method to associate the element vertex to the global vertex. We define a function *node* to do just this

$$\begin{aligned} node : \mathbb{R} \times \{1, 2, 3\} &\rightarrow \mathbb{R} \\ node(e, I) &= i. \end{aligned}$$

This construction of the *node* function, is such that given a element index, and the local index of a vertex in the specified element, it will return the global vertex number.

Using this function, it is possible to construct the algorithm to assemble the global system of equations. The general method is to add each local matrix element to the associated global matrix element, where the association is done using the *node* function. This can be viewed as

$$A_{node(e,I)node(e,J)} = A_{node(e,I)node(e,J)} + A_{IJ}^{(e)} \quad I, J = 1, 2, 3.$$

This is done for all of the elements in the mesh, and all of the local matrix elements for each mesh element.

More specifics of the algorithm for the global matrix construction are discussed in section ??.

For the time independent problem statements, this is the extent of what needs to be done for the assembly of the global system of equations. It provides us with an expresion of the form

$$MU = F.$$

So far in the construction we have neglected the boundary basis functions, and that causes this matrix to be unsolvable. Fixing this is the next step of the process.

## 9. LINEAR SYSTEMS

Using finite element methods, constructs a system of equations on the order of  $N \times N$ , where  $N$  is the number of verticies in the triangulation. With finer meshes will produce a finer approximation of the solution to the partial differential equation. Because of this, most implementation will have highly refined meshes, ones that would have  $\sim 10^4$  verticies. Because of this one must develop a more efficient method for computing solutions to systems of linear equations. Most is this process is discussed further in the section ??. However we will introduce some mathematical concepts that are heavily used in the computational methods.

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

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

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 vector in the Krylov subspace are not necessarily orthogonal, nor normal.

**9.1.1. Reasoning.** We provide a sort 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 invers of a matrix-like by geometric formula

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

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

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

Finally we conclude that

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

Using this in our expression for  $x$  we find

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

Using this we extract the basis that composes  $x$  to be

$$\{b, Ab, A^2b, A^3b, \dots\}$$

The Krylov subspaces takes this infinite basis and cuts it off at some given  $r$  values, thus providing us with

$$\{b, Ab, A^2b, \dots, A^{r-1}b\}.$$

Which the the Krylov subspaces.

Note that this is not a full proof of the construction of the Krylov subspace, but it should be a sufficient explanation for the uses in the numeric approximation of systems of linear equations.

## 10. CONSTRUCTION OF THE APPROXIMATION

### 11. TEMPORAL ITERATIONS



---

## Part 3. Computational

---

### 12. MESH GENERATION

The mathematical theory of the mesh generation was discussed in section 7. This section will describe the specifics of the implementation, and the algorithms that are utilized for the construction of the Delaunay triangulation of an arbitrary domain.

**12.1. 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 constructs the unconstrained Delaunay triangulation then enforces the required edges into the triangulation. So, we begin with the process for construction the unconstrained triangulations.

**12.1.1. 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

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

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.

12.1.2. *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.
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.

**12.2. 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.

Using the definitions from section 7.3 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 7.5).
2. If all triangles pass then Halt. Otherwise select the largest triangle that fails  $\Delta$ , and determine its circumcenter  $c$ .
3. Traverse the triangulation from any vertex of  $\Delta$  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 12.1. 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 Ruppert’s algorithm, any minimum angle below  $20.7^\circ$  is guaranteed to halt. While Chew’s second algorithm is guaranteed to halt for angles below  $28.6^\circ$  and will often succeed with angles below  $34^\circ$ . Because of this improvement on the guaranteed minimum angle is why our focus is on Chew’s algorithm for mesh refinement.

**12.3. Implementation.** There are many possible 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. Triangle implements several algorithms, we chose to utilize the divide and conquer algorithm, developed by Chew.

### 13. MATRIX REPRESENTATION

There are issues that arise due to 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. Currently this is the first issue as without sufficient memory, it is impossible to run the program.

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.

**13.0.1. 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.

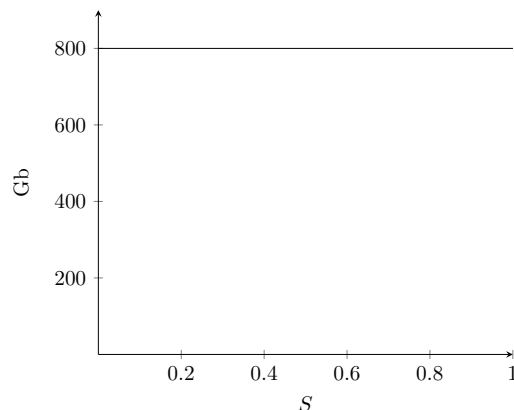


FIGURE 13.1. Memory usage for full matrix storage at different percentages of sparsity.

13.0.2. *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 13.0.1.

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

$$SN^2 \cdot 24 = N^2 \cdot 8$$

$$S = \frac{8}{24}$$

$$S = \frac{1}{3}.$$

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.

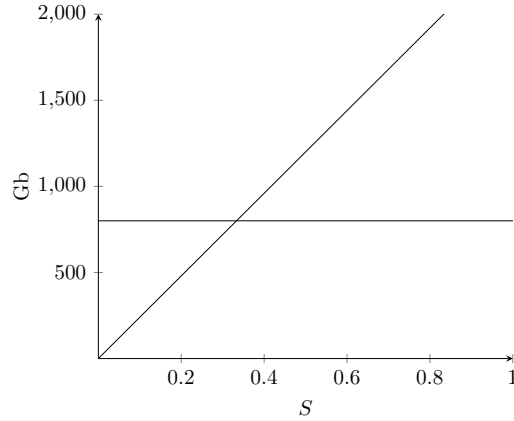


FIGURE 13.2. Memory usage for DOK at different percentages of sparsity.

13.0.3. *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

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

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.

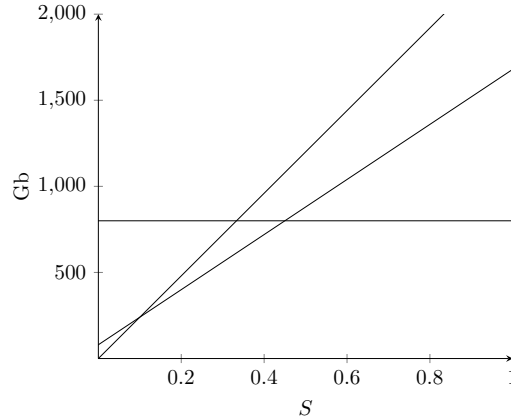


FIGURE 13.3. Memory usage for CRS and DOK at different percentages of sparsity.

**13.0.4. 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.

## 14. NUMERIC APPROXIMATIONS

For the method of finite elements, we require the computation of a number of approximations. Most of these approximations are inherent in the method, and cannot be avoided. However, there are some computations that are possible to do by hand, but for the structure of the program, we chose to implement these computationally.

These computations primarily include the integration of some function over a triangular element, and the partial differentiation of some functions. These computations are required in the stage to construct the matrix. This process is demonstrated in section 8, and the equations that require the computations are shown in equation 6.2.



**14.1. Local Basis Differentiation.** For the local basis function, it is possible to construct the partial derivatives analytically. This is because the structure of the local basis functions is constant, and known. Because of this we determined the partial derivatives of the local basis functions to be

$$(14.1) \quad \begin{aligned} \frac{\partial \varphi_1^{(e)}}{\partial x} &= \frac{(Y_2^{(e)} - Y_3^{(e)})}{(Y_2^{(e)} - Y_3^{(e)}) (X_1^{(e)} - X_3^{(e)}) + (X_3^{(e)} - X_2^{(e)}) (Y_1^{(e)} - Y_3^{(e)})} \\ \frac{\partial \varphi_1^{(e)}}{\partial y} &= \frac{(X_3^{(e)} - X_2^{(e)})}{(Y_2^{(e)} - Y_3^{(e)}) (X_1^{(e)} - X_3^{(e)}) + (X_3^{(e)} - X_2^{(e)}) (Y_1^{(e)} - Y_3^{(e)})}. \end{aligned}$$

Because it is possible to only utilize the first local basis functions, this is all that we need to compute. This is because, we are always able to rotate the local basis functions. The actual number are unimportant, only the order of the sequence matters.

Using this, we are able to use the actual value of the partial derivatives of the local basis functions in the computation. Thus eliminating one of the necessary approximations.

**14.2. Differentiation.** Unfortunately in the computation of  $\bar{A}_{ij}^{(e)}$  we are required to take the derivative of some user defined function. For the sake of not requiring the user to manually compute the partial derivatives, and define those functions as well, we chose to numerically approximation the partial derivative of this arbitrary function.

Consider a function  $f(x, y)$ . The definition of a derivative is

$$\frac{\partial f}{\partial x} \equiv \lim_{h \rightarrow 0} \frac{f(x+h, y) - f(x, y)}{h}.$$

this equation is a simple two point estimation, where taking two points near by, and then the difference should be the approximation of the derivative. However, this approximation is crude, and does not provide much accuracy. One with minor improvements is the symmetric difference quotient

$$\frac{f(x+h, y) - f(x-h, y)}{2h}.$$

This expression improves slightly on the accuracy of the derivative.

For our purposes, we want the numerical approximations of the derivatives to be as accurate as possible. This is why we chose to use finite differences to approximation the derivatives. These methods are the same as the two expressions provided, just at a larger scale. The two expressions previously presented are two point approximations of the derivative, but it is possible to make a  $k$ -point approximation of the derivative. As  $k$  increases the accuracy of the derivative also increases.

For our implementation we chose to utilize an 8-point approximation of the derivative. The coefficients of the approximation utilized were determined by [For88]. The approximation that we implemented takes the form

$$\left. \frac{\partial f}{\partial x} \right|_{(x,y)} \approx \frac{1}{280} f(x-4h, y) - \frac{4}{105} f(x-3h, y) + \frac{1}{5} f(x-2h, y) - \frac{4}{5} f(x-h, y) + \frac{4}{5} f(x+h, y) - \frac{1}{5} f(x+2h, y) + \frac{4}{105} f(x+3h, y) - \frac{1}{280} f(x+4h, y) + O(h^8).$$

Using this approximation, we achieve an error of  $O(h^8)$ . For many cases this should be an acceptable amount of error.

HELP

Since each derivative is only done with respect to a single element, then we are able to construct a more specific approximation. Currently one must minimize the value of  $h$  in order to compute more accurate approximations, but for practical purposes, we must determine a value of  $h$  to evaluate at. It is not possible to evaluate this approximation with  $h = 0$ . In order to determine the value of  $h$  to utilize, we base it off of the size of the element.

We denote the area or the measure of an element as  $\mu(E_e)$ . This can be computed by the use of the following expression

$$\mu(E_e) = \frac{1}{2} \left| X_1^{(e)} (Y_2^{(e)} - Y_3^{(e)}) + X_2^{(e)} (Y_3^{(e)} - Y_1^{(e)}) + X_3^{(e)} (Y_1^{(e)} - Y_2^{(e)}) \right|.$$

We decided to use  $h = \frac{1}{8}\mu(E_e)$  as the value of  $h$  in the approximation of the partial derivatives. If for any reason this is not sufficient for any problem, then all that needs to be done is to refine the mesh in that region of the domain. This mesh refinement can be controlled by the user through the mesh grading function (Def 7.4).

**14.3. Integration.** The final unavoidable approximation is the integration of arbitrary functions over a triangular element. Due to the triangular nature of the elements, it is not possible to utilize a straightforward implementation of Simpson's method, or a trapezoidal rule. In order to integrate on an arbitrary triangular element we must do several steps. Consider a function  $f$  to be integrated over the element  $E_e$ . Thus we wish to construct an approximation for

$$\int_{E_e} f dA.$$

The first step is to convert the arbitrary triangle into a master triangle element. We define the master element, to be a triangular element such that the vertices are located at  $(0,0)$ ,  $(1,0)$ , and  $(0,1)$ , we denote this master element as  $\Delta$ . This conversion induces a scale factor of the area of the triangular element. Thus our expression becomes

$$\int_{\Delta} f \mu(E_e) dA = \mu(E_e) \int_{\Delta} f dA.$$

At this stage it would be possible to utilize Simpson's method or a trapezoidal method for integration, changing the bounds of the  $y$  integration based upon the  $x$  value, and the expression would be of the form

$$\mu(E_e) \int_{x=0}^{x=1} \int_{y=0}^{y=1-x} f dy dx.$$

However, this method is extremely computationally intensive and does not provide very accurate results for the computational complexity that it requires. To compute this integral, we will utilize a different method of numerical integration called Gaussian Quadrature.

The Gaussian Quadrature rule is a quadrature rule that has been constructed such that a  $n$ -point Gaussian quadrature will yield an exact result for polynomials of the degree  $2n - 1$ . A quadrature rule is most methods of numeric integration, for example trapezoidal rule is a quadrature rule. The premise of the Gaussian quadrature rule is that the sum of a function evaluated at a set of selected points multiplies by selected weights is able to compute the approximation of the function.

This is the same concept as the trapezoidal rule, which is expressed as

$$\int_a^b f dx \approx \frac{\Delta x}{2} \sum_{k=1}^N (f(x_{k-1}) + f(x_k)).$$

This method is clearly a sum of the function as specified points  $x_k$ , multiplied by a specified weight  $\frac{\Delta x}{2}$ .

Gaussian quadrature rules have been developed for the master triangle  $\Delta$ , so we will utilize the computed weights and points from [HKA12]. This means that the approximation of the integral becomes

$$\mu(E_e) \int_{\Delta} f dA \approx \mu(E_e) \sum_{k=1}^N w_k f(x_k, y_k).$$

For our purposes we chose  $N$  to be 128, as that will be sufficiently large that most error should be negligible. If the error is not significantly negligible, then the mesh should be refined thus reducing  $\mu(E_e)$ , and consequentially reducing the scale of the error. The values from [HKA12] have been provided in the appendix, both of the weights and the  $(x, y)$  pairs.

Add weights and points in the appendix.

## 15. SOLVING LINEAR SYSTEMS OF EQUATIONS

A second major issue arises with the time complexity of computing the solution to systems of linear equations. Trivial methods require  $\mathcal{O}(n^3)$  time when scaled to the magnitude of the matrices constructed, the time requirement would become unacceptable. Thus we will examine a number of methods that can be utilized in order to approximate the solution to systems of linear equations.

**15.1. 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.

**15.1.1. 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.

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  $i$ th 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

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

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

---

**Algorithm 15.1** Jacobi Method

---

```

 $x = 0$ 
for  $k = 1, 2, \dots$  do
   $\bar{x} = 0$ 
  for  $i = 1, 2, \dots, n$  do
    for  $j = 1, 2, \dots, i-1, i+1, \dots, n$  do
       $\bar{x}_i += a_{i,j} x_j$ 
    end for
     $\bar{x}_i = \frac{b_i - \bar{x}_i}{a_{i,i}}$ 
  end for
   $x = \bar{x}$ 
  if  $\|Ax - b\| \leq 10^{-10}$  then
    return  $x$ 
  end if
end for
return  $x$ 

```

---

Pseudocode for the Jacobi method is shown in Algorithm 15.1. 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.

**15.1.2. 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  $x$ .

Because of this similarity, we do not explain in detail the mathematics, but the pseudocode is provided in Algorithm 15.2.

**Algorithm 15.2** Gauss-Seidel Method

---

```

 $x = 0$ 
for  $k = 1, 2, \dots$  do
  for  $i = 1, 2, \dots, n$  do
     $x_i = 0$ 
    for  $j = 1, 2, \dots, i-1, i+1, \dots, n$  do
       $x_i \leftarrow x_i + a_{i,j}x_j$ 
    end for
     $x_i = \frac{b_i - x_i}{a_{i,i}}$ 
  end for
  if  $\|Ax - b\| \leq 10^{-10}$  then
    return  $x$ 
  end if
end for
return  $x$ 

```

---

**15.2. 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.

**15.2.1. 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.

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

We construct the iterative approximation in each iteration by a multiple of  $\alpha_i$  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

$$\begin{aligned} q^{(i)} &= Ap^{(i)} \\ r^{(i)} &= r^{(i-1)} - \alpha_i q^{(i)} \end{aligned}$$

We construct  $\alpha_i$  in order to minimize  $r^{(i)T} A^{-1} r^{(i)}$ ,

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

Then to updated the search direcitons using the residual

$$\beta_i = \frac{r^{(i)T} r^{(i)}}{r^{(i-1)T} r^{(i-1)}}$$

$$p^{(i)} = r^{(i)} + \beta_{i-1} p^{(i-1)}$$

where the choice of  $\beta_i$  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 15.3.

---

**Algorithm 15.3** ConjugateGradient

---

```

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

```

---

**15.3. 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.

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.

$$\begin{aligned} Ly &= b && \text{by forward substitution} \\ L^T x &= y && \text{by back substitution.} \end{aligned}$$

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

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

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

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

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

15.3.2. *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

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

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 to the inefficient Gaussian elimination.

We provide pseudocode of this method in Algorithm 15.4.

**Algorithm 15.4** Cholesky method

---

```

for  $j = 0, 1, 2, \dots, N$  do
     $L_{j,j} = 0$ 
    for  $k = 0, 1, 2, \dots, j$  do
         $L_{j,j} += L_{j,k}^2$ 
    end for
     $L_{j,j} = \sqrt{A_{j,j} - L_{j,j}}$ 
    for  $i = j, j+1, j+2, \dots, N$  do
         $L_{i,j} = 0$ 
        for  $k = 0, 1, 2, \dots, j$  do
             $L_{i,j} += L_{i,k}L_{j,k}$ 
        end for
         $L_{i,j} = \frac{A_{i,j} - L_{i,j}}{L_{j,j}}$ 
    end for
end for

```

▷ Calculate  $A = LL^T$  decomposition.

```

 $y = 0$ 
for  $i = 1, 2, \dots, N$  do
     $y_i = 0$ 
    for  $j = 1, 2, \dots, i$  do
         $y_i += L_{i,j}y_j$ 
    end for
     $y_i = \frac{b_i - y_i}{L_{i,i}}$ 
end for

```

▷ Calculate  $Ly = b$  by forward substitution.

```

 $x = 0$ 
for  $i = N, N-1, \dots, 1$  do
     $x_i = 0$ 
    for  $j = i+1, i+2, \dots, N$  do
         $x_i += L_{i,j}^T x_j$ 
    end for
     $x_i = \frac{b_i - x_i}{L_{i,i}^T}$ 
end for
return  $x$ 

```

---

**15.4. 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.

**15.4.1. Positive 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.

15.4.2. *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.

## 16. PLOTTING

For the plotting of the approximations generated by the finite element methods, we utilize the global basis functions. The method that we utilize, is to iterate through every pixel of the desired image, and then determine the appropriate color. The first step is to determine which triangular element the point is within. Using the element and the point, we are able to evaluate the approximation at that point.

Once all of the values for every point in the grid have been stored, we then determine the minimum and the maximum value. Then the values are mapped  $[min, max] \rightarrow [0, 255]$ . Using the mapped values we select a color from an array of colors. The selected color is set as the color of the pixel at the point.

We then utilize libpng[PNG19], to save the image buffer data to a file. This process can be slow, so we implement the images saving in a separate thread. This is important so that more computations can be done on the main thread.

For the time dependent methods, we save multiple images into a folder, and use FFMPEG[FFM19] to construct an animation from the sequence of images.

## REFERENCES

- [BBC<sup>+</sup>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. Constrained delaunay triangulations. *Algorithmica*, 4:92–108, 1989.
- [Che93] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. *9th Annual Symposium on Computational 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.
- [FFM19] Ffmpeg, 2019.
- [For88] Bengt Fornberg. Generation of finite difference formulas on arbitrarily spaced grids. *Mathematics of Computation*, 51(184):699–706, 1988.
- [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.
- [PNG19] libpng, 2019.
- [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.