Effective Algorithm for Building and Solving Linear Systems

Sebastian Pena Serna
Fraunhofer-IGD
sebastian.pena.serna@igd.fraunhofer.de

João Silva
University of Minho / Fraunhofer-IGD
jsilva@igd.fraunhofer.de

Andre Stork
Fraunhofer-IGD / TU-Darmstadt
andre.stork@igd.fraunhofer.de

Adérito Fernandes Marcos
Portuguese Open University Lisbon
marcos@univ-ab.pt

Abstract

Several mesh-based techniques in computer graphics such as shape deformation, mesh editing, animation and simulation, build and solve linear systems. The most common method to build a linear system consists in traversing the topology (connectivity) of the mesh, producing in general a representation of the set of equations in form of a sparse matrix. Similarly, the solution of the system is achieved, by means of iterating over the set of equations in the default sequence of the vertices (unknowns). This paper presents a new algorithm, which optimizes the build of the linear system and its storage, and which allows the iteration over the set of equations in any arbitrary order. Additionally, our algorithm enables rapid modifications to the linear system, avoiding a complete rebuild.

Keywords

Effective memory handling, rapid simulation, solver acceleration, dynamic linear systems, mesh-based applications

1. INTRODUCTION

Several computer graphics algorithms successfully use techniques from other disciplines such as physics, numerical analysis, linear algebra, among others, in order to make graphics and interaction more realistic and powerful for the needs of the application. Nevertheless, some of the adopted techniques are applied without major changes, which restricts or neglects its proper applicability and its versatility for computer graphics applications. There are several mesh-based methods such as shape deformation, mesh editing, animation and simulation, which require the build and the solution of a linear system, however the processing time and the memory consumption are very high. These high consumptions are caused because of the assembly of the linear system, which requires the computation of the set of equations for the unknowns.

Although the computation of the equations needs in any case to be performed, it is possible to minimize the consumption of resources, by means of using the neighboring information of the mesh. This neighboring information can help us avoiding the assembly of the linear system and therefore reducing the processing time and the memory consumption. On the other hand, it also generates flexibility for handling the equations, enabling their iteration and solution in any arbitrary order. This flexibility can also benefit other techniques, where the topology of the mesh is constantly changed. In this case, the linear system does not need to be recomputed every time the topology is changed, since these changes can independently be applied to the corresponding elements without affecting the rest of the system, which also minimizes the computation time for every iteration.

In order to find a solution, which can accelerate the employment of linear systems and minimize its needed resources, we first studied different applications within the computer graphics community, which could provide us some hints regarding their utilization of this kind of systems. Afterwards, we started analyzing how the discretization methods for physical problems work, as well as different solvers, aiming at understanding their requirements (Section 2). This information helped us developing a methodology, which can effectively improve the build and solution of linear systems and which can also reduce the consumption of resources (Section 3). We also started with the implementation of the algorithm, in order to evaluate and compare its performance, however we still need to advance forward in this direction (Section 4). Finally, we summarized our contribution and we outlooked our future work in this field.

2. RELATED WORK

As we stated in the previous section, we started studying different techniques in computer graphics such as deformable models ([Nealen 06]), shape modeling ([Alexa 06]), animation ([Müller 08]) and simulation ([Bridson 06]), in order to understand how linear systems are used within this community. The study of this techniques revealed us, that linear systems are used in a clas-

sical way and there is no optimization procedures to reduce its build time. Hence, we decided to analyze partial differential equations ([Langtangen 03]) and discretization techniques, particularly the Finite Element Method ([Hughes 00] and [Smith 04]), in order to analyze the requirements for building linear systems.

Additionally, we wanted to understand how the solution of a linear system is computed, hence we revised the literature regarding iterative solvers ([Saad 00]) and specially the conjugate gradient method ([Shewchuk 94]). Based on this information, we realized that we need to concentrate in a physical problem, since the generated systems could present different properties concerning symmetry, definiteness, among others, and depending on this properties, there are solvers which are better suited than others. Hence, we decided to concentrate our effort in the solid mechanics problem [Bower 09] and to investigate the build and the solution of the linear system for this kind of physics.

Augarde et al. [Augarde 06] explained that in the linear elasticity problem, the Galerkin method causes the stiffness matrix to be symmetric and positive definite. This fact makes the Conjugate Gradient Method a suitable solver for the linear system of equations yielded in the linear elasticity problem. Saad and Vorst presented, in [Saad 00], an in-depth historical perspective of iterative solutions to linear systems. They attribute the origin of iterative solution of linear systems to the work of Gauss in the early nineteenth century and show how the main contributions over the years led to the iterative solvers we have nowadays.

The studied related work suggested us, that the Conjugate Gradient method is currently the most appropriate solver for computing the solution of a linear system of equations in an iterative form and without using a hierarchy of discretizations or adaptivity methods. Hence, we decided to implement the Conjugate Gradient Method based on [Shewchuk 94]. He presented a practical explanation on how the Conjugate Gradient works and explained the building blocks, i.e. the method of Steepest Descent, the method of Conjugate Directions and finally their relations within the Conjugate Gradient method.

There are some techniques, dealing with the simulation of changing meshes ([Klingner 06]). These techniques usually remesh the whole model after every iteration (or mesh change), hence they also rebuild the linear system. We believe, that the algorithm proposed in this paper, will also contribute to the rapid simulation of dynamic meshes. To the best of our knowledge, there are no investigations in the same direction as the proposed in this paper. There have been made several efforts regarding the improvement of solvers, either with new methods or with parallelization techniques on the CPU or the GPU, but there is not enough information about the intelligent handling and processing of linear systems.

3. METHODOLOGY

The algorithm, which we are proposing, aims at effectively building and solving linear systems, by means of reducing the processing time and of minimizing the memory consumption. A linear system represented in the matrix form is:

$$Ax = b \tag{1}$$

where A is the matrix of coefficients, x is the vector of unknowns and b is the vector of solutions. Based on the study of the related work, we have understood the objective of this system in getting the solution of the problem. The matrix of coefficients aims at collecting the information for the equations regarding the connectivity of the vertices (edge topology) and of the unknowns (vertices without boundary conditions) themselves. The process for building the matrix of coefficients is normally based on traversing the given mesh (element by element), in order to identify the neighboring elements, which need to contribute to an edge (non diagonal positions) or to a vertex (diagonal positions). This process is expensive and when the geometry and the topology of mesh is changed, the matrix of coefficients needs to be also changed.

Hence, we realized, that if we could have the information concerning the elements around an edge, we could easily collect and build the information of the non diagonal positions of the linear system. Moreover, if we store the information for every edge (elements around the edge) within a small *edge matrix*, we will have enough flexibility to modify or recompute only the incident edges to a vertex, when the vertex changes, without affecting the rest of the linear system. Analogically, we can perform the same strategy for storing the information of the diagonal positions of the linear system, computing a small *diagonal matrix* for every unknown. These two sets of matrices are an equivalent representation of the matrix of coefficients, which we will refer to as the *equivalent matrix*.

Structure wise, we no longer use the traditional sparse matrix to store the matrix of coefficients. Instead, we have divided it into two vectors (see Figure 1 for a visual representation). In one vector we store the *edge matrices* and on the other we store the *diagonal matrices*. Every element of both vectors is a nxn matrix where n is the dimension (1D, 2D or 3D) of the problem.

The two main characteristics of a matrix of coefficients, sparsity and symmetry, are used by the *equivalent matrix* to minimize memory consumption: only the non-zero values are stored and only one instance of every edge is stored for every pair of connected vertices. In order to explain how our algorithm uses these characteristics for building the *equivalent matrix* and for the sake of clarity, we have subdivided the process into three steps:

1. Constructing the needed neighboring information
2. Computing the set of *edge matrices*
3. Computing the set of *diagonal matrices*

These three simple steps enable the minimization of the space in memory and the reduction of the processing time.

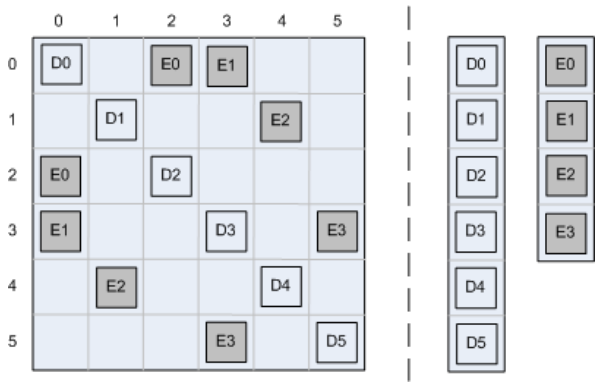

Figure 1. Graphic representation of a matrix of coefficients (left). The same matrix of coefficient represented as the equivalent matrix (right).

In addition, the new representation of the matrix of coefficients allows the flexible handling and modification of individual vertices or edges, without affecting the rest of the matrix.

3.1. Build of the Equivalent Matrix

The *equivalent matrix* replaces the matrix of coefficients by a set of small matrices, which can be computed faster and which requires less space in memory. In order to avoid traversing the whole mesh, when computing the matrix of coefficients, we precompute the neighboring information. We also precompute the *element matrices*, which are the basis for computing the edge and diagonal matrices.

3.1.1. Constructing the neighboring information

We need to precompute two kinds of neighboring information: i) elements around an edge and ii) elements around an unknown. This information is computed during the initialization process and it is updated, if some changes to the topology of the mesh are made. The neighboring information allows us computing the non diagonal and the diagonal positions of the matrix of coefficients independently. We are using a mesh data structure, which automatically constructs the neighboring information, however we will explain this process for the sake of completeness. During the loading process of the mesh, we initialize two double arrays (db), where we store the needed information for the edges ($dbEdg$) and for the unknowns ($dbUkn$). When we read an element, we append its index to its six corresponding edges in $dbEdg$ and to its four corresponding vertices (unknowns) in $dbUkn$. By the end of the loading process, the neighboring information is also ready.

The example mesh shown in Figure 2 will be used in association with Figure 3 and Figure 4 to accompain the explanations given in sections 3.1.2 and 3.1.3 respectively.

3.1.2. Computing the edge matrices

Given the neighboring information of the elements around an edge and the element matrices, we can easily compute

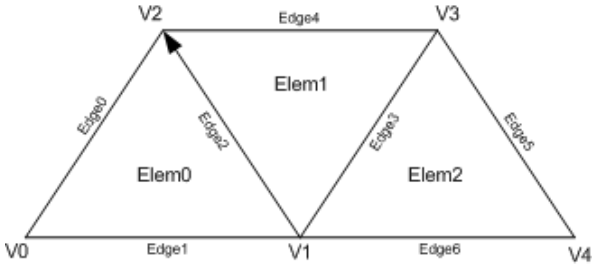

Figure 2. A 2D mesh composed of three elements.

the non diagonal positions, by means of traversing the elements around the edge and adding the contribution of the corresponding element. On Figure 3 it is shown the computations involved in the build of Edge2. Since both elements Elem0 and Elem1 share Edge2, the components of both elements regarding this edge (colored in gray) are added to the Edge2's matrix. Note that each element has two contributions to every used edge and since one is the transposed of the other, only one is added to the edge matrix. The used contribution is chosen based on the direction of the edge.

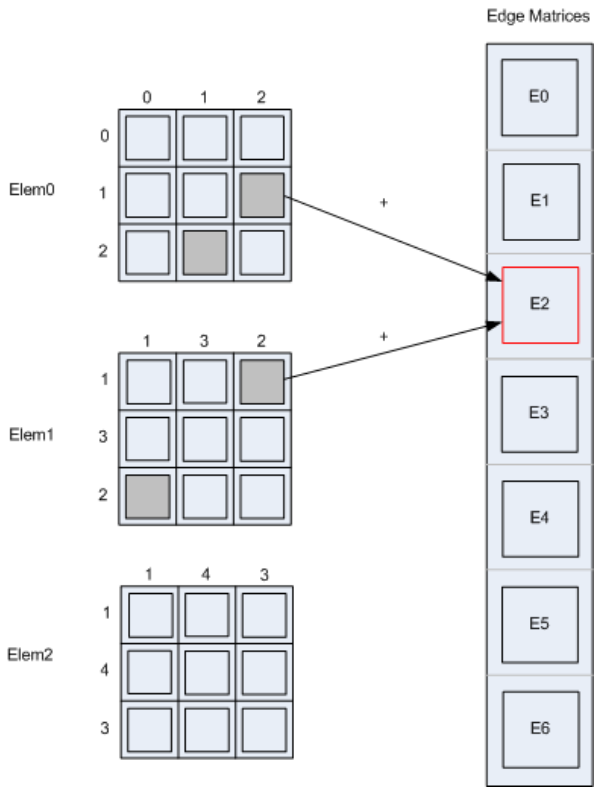

Figure 3. Element contributions to the build of the third edge matrix (E2).

3.1.3. Computing the diagonal matrices

In a similar way, we compute the diagonal positions of the linear system. In this case, we use the neighboring infor-

mation regarding the elements around an unknown and we consider the contribution of every involved element to the diagonal.

Consider Figure 4, where the build of the diagonal of vertex V1 is being performed. From Figure 2 it is clear that vertex V1 is shared by the three elements. Therefore, D1 is calculated by adding up the three contributions to V1 of the three elements.

The union of the *edge matrices* and the *diagonal matrices* is equivalent to a matrix of coefficients. Although, we need to consider these two sets of matrices, in order to solve the system, we can arbitrary decide the order in which we want to solve it. This flexibility could be advantageous for a rapid convergency, since it is equivalent to having a linear system with an optimized ordering of the unknowns, leading to an improvement of the performance of the solver ([Oliker 02]). Moreover, since the matrices within the two sets are independent, we can change or update them without a major effort, because we would only need to recompute a small set of matrices, avoiding the computation of the whole set of equations.

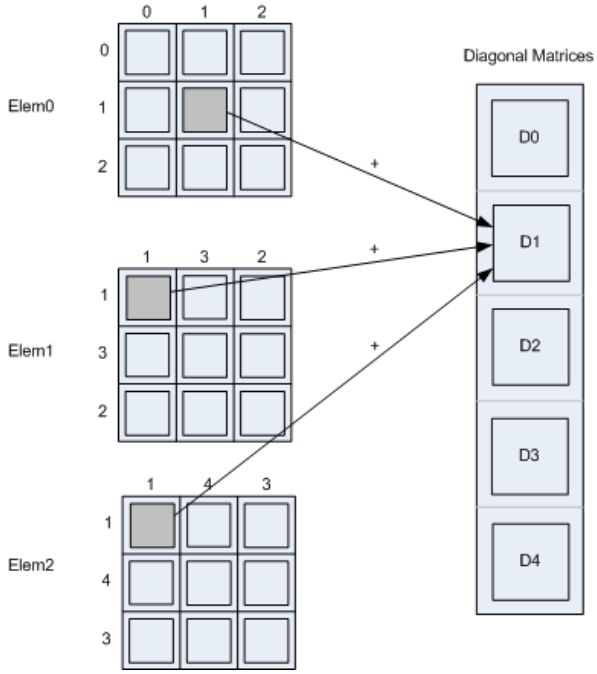

Figure 4. Element contributions to the build of the second diagonal matrix (D1).

3.2. Solution of the Equivalent System

The implemented algorithm to solve the linear system of equations is the Conjugate Gradient as it was proposed in [Shewchuk 94]. The only noticeable change done so far is the way we multiply the equivalent matrix with a vector.

3.2.1. Multiplication of the equivalent matrix with a vector

Having the matrix of coefficients stored in the *equivalent matrix* form requires a special method for its multiplication with a vector.

```
1  foreach rst in resultVector do
2  |  resultVector[rst] = 0
3  end
4  foreach edge in edgeVector do
5  |  edgeStartVertex = start vertex of this edge
6  |  edgeEndVertex = end vertex of this edge
7  |  resultVector += (edgeVector[edge] * multiVector)
8  |  resultVector += (edgeVector[edge]^T * multiVector)
9  end
10 foreach diag in diagVector do
11 |  resultVector += (diagVector[diag] * multiVector)
12 end
```

Algorithm 1: Multiplication of the equivalent matrix with the vector multiVector and store the result in resultVector.

Algorithm 1 shows the main steps we perform to multiply the equivalent matrix with a vector. We start by setting the $resultVector$ to zero so that the results of the multiplications performed over the matrix can be added to it. For every edge, we find the vertices that form that edge (edgeStartVertex and edgeEndVertex) by consulting the neighborhood information. This is done to know which is this edge position on the matrix. Doing so it is known with which position of $multiVector$ this edge should be multiplied and in what position of the $resultVector$ it should be stored.

Also notice that for every edge two multiplications are done. This is due to the symmetry of the matrix. To provide a better understanding of this procedure consider the example shown in Figure 5. Assuming that we are iterating on edge E1 and that this edge has vertex 0 as a starting vertex and vertex 3 as an ending vertex. In line 7 of the algorithm we would multiply E1 by V3 and store its result in R0 (single underlined boxes). And since E1 also connect vertex 3 to vertex 0 with the same value, in line 8 we store in R3 the multiplication of the transposed E1 with V0 (double underlined boxes).

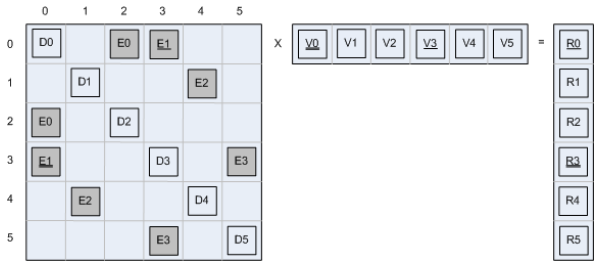

Figure 5. Multiplication of the equivalent matrix (represented as a normal matrix for simplification) with a vector. The single and double underlined boxes indicate the used values when the multiplication iterates on edge E1.

After processing all the edges, we iterate on the diagonals. The process is similar but simpler for each diagonal relates

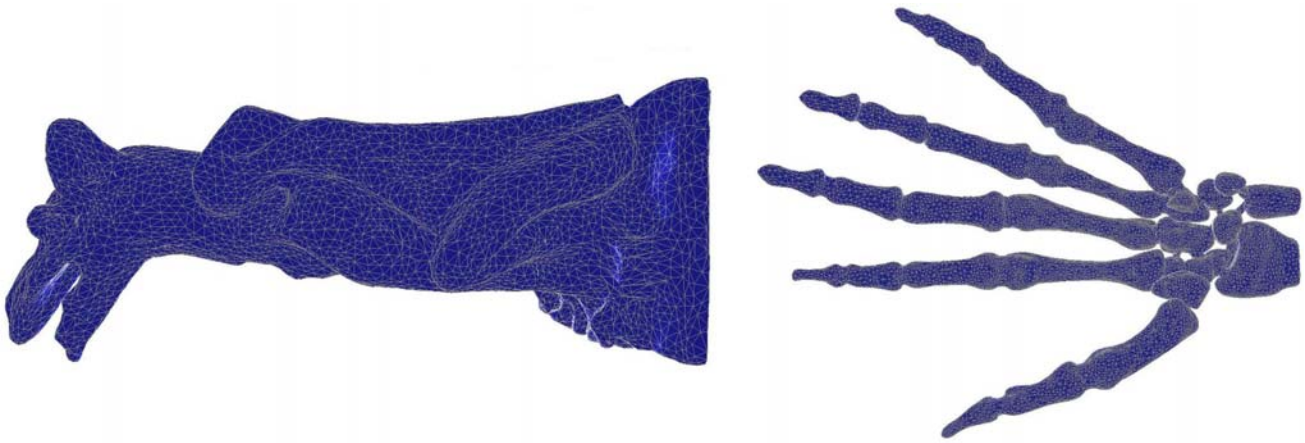

Figure 6. Mesh models for the measurement.

a vertex to itself.

4. RESULTS

We have implemented the build of the *equivalent matrix* (DEM) and its multiplication with a vector, as proposed in the previous section. This implementation is not complex and it can easily be reproduced following the given indications. In order to compare the performance of our algorithm, we have also implemented the classical form to represent the matrix of coefficients, the sparse matrix. We made two implementations of the sparse matrix: i) without the symmetric characteristic (CSM) and ii) with the symmetric characteristic (SSM). Since, the implementation of the sparse matrix was only for the sake of comparison, we decided to make a simple implementation based on an array of linked lists. In the same way, the implementation of the multiplication with a vector for CSM and SSM was made.

In order to measure the performance of the three implementation, we have used two different tetrahedral meshes: i) a gargoyle with 50,000 elements and ii) a hand with 100,000 elements. The Table 1 presents the relevant topological information of the meshes.

Table 1. Topological information of the meshes for the measurement.

Mesh	Vertices	Edges	Elements
Gargoyle	13,044	71,873	49,996
Hand	26,649	144,669	99,995

We have made two kinds of measurements, one for the build process and one for the multiplication process. We have considered 20 multiplications, in order to be able to measure the time, since the measurement for a single multiplication is not accurate. Table 2 shows the results in milliseconds for the mesh models and the two processes.

The results show, that our algorithm is faster than the other two implementations. Our algorithm is much faster than the CSM implementation, since the latter requires almost two times the space in memory than ours. Although for the build process, the SSM implementation is similar to our algorithm, we are still more or less 5% faster. For the multiplication process, our algorithm performs in average 2.5 times faster than the SSM implementation.

Table 2. Measurements for the build and multiplication processes (in milliseconds).

Process	Mesh	CSM	SSM	DEM
Build	Gargoyle	213	125	120
	Hand	459	292	271
Multiplication	Gargoyle	224	141	58
	Hand	526	380	140

In terms of memory consumption, the CSM implementation uses more memory than the SSM and the DEM implementations, since it stores the total non zero values entries of the matrix of coefficients. The SSM and the DEM implementations have the same memory consumption, because both use the symmetric characteristic of the matrix of coefficients and therefore, they only need to store either the upper or the lower part of the matrix.

5. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm, which uses the neighboring information of the mesh to effectively build and solve linear systems. Our algorithm avoids the assembly of the matrix of coefficients and it also reduces the processing time and the memory consumption. The proposed method also enables the handling of the set of equations with more flexibility, allowing their iteration and solution in any arbitrary order. This flexibility is a step forward towards the simulation of meshes with dynamic topology or dynamic meshes. We will further continue with the implementation of our algorithm and we also plan to develop a framework, where the modification of meshes and its real time simulation will be feasible, aiming at integrating design and

analysis of mechanical objects within the same environment. We will also investigate how our could fit within a parallelization scheme.

6. ACKNOWLEDGEMENTS

This work is partially supported by the European projects 3D-COFORM (FP7-ICT-2007.4.3-231809) and FOKUS K3D (FP7-ICT-2007-214993).

References

[Alexa 06] Marc Alexa, editor. *Interactive shape editing*. ACM SIGGRAPH 2006 Courses, 2006.

[Augarde 06] CE Augarde, A. Ramage, and J. Staudacher. An element-based displacement preconditioner for linear elasticity problems. *Computers and structures*, 84(31-32):2306–2315, 2006.

[Bower 09] A.F. Bower. *Applied Mechanics of Solids*. Taylor and Francis, 1st edition, August 2009.

[Bridson 06] Robert Bridson, Ronald Fedkiw, and Matthias Muller-Fischer. Fluid simulation: Siggraph 2006 course notes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 1–87, New York, NY, USA, 2006. ACM.

[Hughes 00] T.J.R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Publications, 2000.

[Klingner 06] B. M. Klingner, B. E. Feldman, N. Chentanez, and J. F. O'Brien. Fluid Animation with Dynamic Meshes. In *International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2006.

[Langtangen 03] H.P. Langtangen. *Computational partial differential equations: numerical methods and diffpack programming*. Springer Berlin, 2nd edition, 2003.

[Müller 08] Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics: class notes. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–90, New York, NY, USA, 2008. ACM.

[Nealen 06] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlon. Physically Based Deformable Models in Computer Graphics. *Computer Graphics Forum*, 25(4):809–836, 2006.

[Oliker 02] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

[Saad 00] Y. Saad and H.A. Van Der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33, 2000.

[Shewchuk 94] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Computer Science Tech. Report*, pages 94–125, 1994.

[Smith 04] I.M. Smith and DV Griffiths. *Programming the finite element method*. Wiley, 4th edition, 2004.