SIFFEA: Scalable Integrated Framework for Finite Element Analysis

Xiangmin Jiao, Xiang-yang Li, and Xiaosong Ma

Department of Computer Science
University of Illinois, Urbana, IL 61801, USA

Abstract. SIFFEA is an automated system for parallel finite element method (PFEM) with unstructured meshes on distributed memory machines. It synthesizes mesh generator, mesh partitioner, linear system assembler and solver, error estimator, and adaptive mesh refiner. SIFFEA is an implicit parallel environment: The user need only specify the application model in serial semantics; all internal computations and communications are transparent to the user. SIFFEA is designed based on the object-oriented philosophy, which enables easy extensibility, and a clear and simple user interface for PFEM.

1 Introduction

The parallel finite element method (PFEM) is widely used in computational solution of system simulations in various engineering problems. Generally, there are six steps in the finite element analysis: mathematical modeling, geometric modeling, mesh generation, linear system formulation, numerical solution, and adaptive refinement. Many software packages are available on parallel machines for each individual component of FEM [6, 11, 12]. However, there is a lack of automated integrated systems for PFEM. In the sequel, the users must do a lot of application coding in integrating softwares from various sources.

To address this problem, we are developing SIFFEA, an integrated framework for FEM on scalable distributed memory machines, by synthesizing these widely used components. SIFFEA includes a mesh generator; a mesh distributor; a linear system assembler; an error estimator and adaptive mesh refiner; and interfaces with mesh partitioners and linear system solvers. It is designed based on object-oriented philosophy, and written in C++. The communications are carried out using MPI for portability. Owing to object-oriented design, each component of SIFFEA is highly encapsulated, and the components interact with each other through well-defined interfaces, so that the implementation of one component can be changed without affecting the others. The structure of SIFFEA is diagrammed in Fig. 1.

SIFFEA provides a clear and simplified user interface. Up to date, the design of SIFFEA has been focusing on linear elliptic problems. For this kind of problems, the user need only provide geometry specification, and serial subroutines for computing element matrices and vectors, and boundary conditions. All internal computations and communications of SIFFEA are done transparently to the

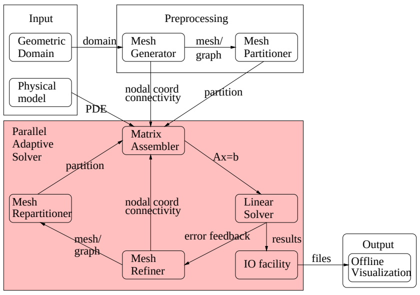

Fig. 1. The structure of SIFFEA.

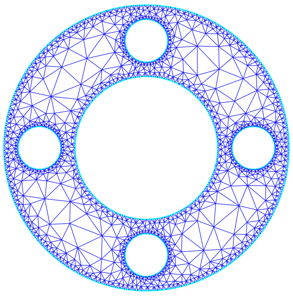

Fig. 2. A sample mesh.

user. We are looking into extending SIFFEA for time-dependent and nonlinear problems, while preserving this simple user interface.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts about mesh generation and refinement, and their sequential and parallel algorithms. In Sec. 3, we describe the interface with the mesh partitioner, and the algorithms for distributing the mesh onto processors after partitioning. Section 4 addresses the issues about parallel linear system assembler and solvers. In Sec. 5, we discuss the basic concepts behind the error estimator and the adaptive mesh refiner. Section 6 presents some preliminary experimental results. In Sec. 7, we compare SIFFEA with some other PFEM systems. Section 8 concludes the paper with the discussion of future implementation and research directions.

2 Mesh Generation and Delaunay Refinement

Mesh generation [14, 15, 18] is a fundamental problem in FEM. In particular, the *unstructured mesh* is widely used in FEM solvers, due to its advantages of varying local topology and spacing in reducing the problem size, and adapting to complex geometries and rapid changing solutions.

It is well-known that the numerical errors in FEM depend on the *quality* of the mesh. In particular, it is desirable that the angles of each element are not too small [3, 22]. The Delaunay triangulation and refinement algorithms generate high quality meshes satisfying this criterion. Specifically, Delaunay triangulation maximizes the minimum angle among all elements [21]; Delaunay refinement allows one to add extra vertices, called *Steiner points*, in order to further improve the quality of the mesh [18, 19].

SIFFEA contains a two-dimensional mesh generator, which can generate high-quality meshes, including exact Delaunay triangulations, constrained Delaunay triangulations, and conforming Delaunay triangulations, on complex domains. The conforming Delaunay triangulations guarantee no small angles, and are thus suitable for finite element analysis. An example of a generated mesh is illustrated in Fig. 2. The mesh generator also supports adaptive mesh refinement based on *a posterior* error estimation, which will be discussed in Sec. 5.

The input to the mesh generator is a PSLG [14] description of the geometric domain, which can contain points and line segments. These line segments are called the *constrained edges*. The input domain can also contain holes, as shown in Fig. 2. A clockwise or counterclockwise sequence of edges of a polygon defines a hole. In particular, a clockwise sequence defines a hole inside the polygon, and a counterclockwise sequence defines a hole outside the polygon. Every hole has labels, which specify whether the hole belongs to the mesh, and the element size requirement inside the hole. Boundary conditions may be associated with the constrained edges. To achieve this goal, the user labels each constrained edge with an integer, which will be inherited by the Steiner points on it; then he/she defines a subroutine for computing the boundary condition based on the coordinates and the mark of a point. This is further discussed in Sec. 4.1. By convention, the user marks the uninterested edges with 0.

The mesh generation is designed using the object-oriented technology. We separate the mesh representation from the algorithms. Internally, the Mesh class uses the efficient quad-edge [9] data structure to represent the mesh. The algorithms, including Delaunay triangulation, Delaunay refinement, and adaptive refinement, are designed using the high-level interfaces of the Mesh class. Each algorithm is encapsulated in a separate class.

The current mesh generator in SIFFEA is sequential, which is observed the performance bottleneck of PFEM. We are currently designing and implementing a parallel mesh generation algorithm. The basic idea is as follows. Assume that there are p processors. We first apply some domain decomposition techniques, such as the point separator [16] or the k-d tree [7], to divide the input domain into p subdomains. Each processor is then assigned a subdomain, and performs the Delaunay triangulation and refinement on it. Note that the Delaunay refinement may add Steiner points on the separator edges between subdomains. Therefore, after each processor finishes triangulation and refinement locally, it must update its point set, by merging with the Steiner points on its adjoining separator edges from other processors; then it resumes the Delaunay refinement. This process repeats until no new points are added. To reduce the number of Steiner points added on the separator edges (and hence reduce communication), we must carefully choose the separator edges, and make the initial points on the separator edges dense enough during domain decomposition.

3 Mesh Partition and Distribution

After mesh generation, the mesh must be distributed across processors, so that in the subsequent steps, the linear system can be assembled and solved in parallel. In this section, we describe our approaches for mesh partition and distribution.

3.1 Mesh Partition

Mesh partition is to decompose the mesh into roughly equal size of subdomains while minimizing the cut-edges between the subdomains. Its purpose is to achieve better load balance and reduce communication overhead during linear system assembly and solve. Mesh partition is typically done using a graph partitioning algorithm. In our current implementation, we use ParMetis [12], which is a fast parallel graph partitioning algorithm, to carry out the work.

The ParMetis subroutine for graph partitioning requires the input of a distribute graph. In particular, the mesh nodal graph must be distributed across processors before calling ParMetis. If a sequential mesh generator is used, we accomplish this by assigning vertices in blocks to processors. That is, if n is the number of vertices, and p is the number of processors, then vertex i is assigned to processor $i/\lceil n/p \rceil$. Note that ParMetis only requires vertex adjacency list. To collocate the mesh and utilize memory efficiently, we distribute the vertex coordinates and the element connectivities along with the vertex adjacency list. The initial element partition is determined as follows: We assign an element to the owner of its second largest vertex. This simple heuristic has the advantage of not requiring extra communication to determine the element partition, and it yields fairly even distribution of elements in practice.

After the initial distribution, the ParMetis subroutine ParMetisPartKWay() is then called, which returns the new processor assignment of the locally owned vertices on each processor. This assignment will determine the partition of the global stiffness matrix, as discussed in Sec. 4.2. Note that an element partition is also needed by the linear system assembler. Again, the element partition can be derived from the vertex partition as above.

3.2 Data Redistribution

After determining vertex and element partitions, we must ship vertices and elements to their assigned processors. We refer to this stage as *data redistribution*. In particular, four types of data must be communicated at this stage, and they all have all-to-all communication patterns.

First of all, each processor must gather the vertices assigned to it, which includes gathering the global indices, the coordinates, and the adjacency list of each vertex. Note that the adjacency list will be used by memory management of the global matrix. Since a processor is generally assigned nonconsecutive vertices, to facilitate the subsequent steps, we compute a vertex renumbering such that the vertices on each processor are consecutive in the new number system.

Secondly, we ship the element connectivities to the assigned processors. Since the element assignments were based on the old number system, the connectivities are mapped to the new number system after shipped to the owner processors.

One or more vertices of a straddling element on the cut-edges are not local to the element's owner processor. These nonlocal vertices are called the *ghost points,* which are needed by the element matrix and vector computation. In the third step, each processor determines the ghost points, and gathers their coordinates.

Finally the boundary vertices and their marks are shipped to the processors. A boundary vertex must be sent to a processor, if it is owned or is a ghost point of that processor. Since the number of boundary vertices is generally small, a broadcast is used instead of vector scatters, to simplify communication pattern. After the broadcasting, each processor deletes the unneeded boundary vertices.

4 Linear System Assembly and Solution

The ultimate step of computational solution of FEM is to assemble a system of linear equations, and then solve it using a sparse linear solver. Matrix-based sparse linear solvers are most popular in the FE community, partially due to the availability of many high-quality sparse matrix solvers [8]. In our current implementation, we also adopted a matrix-based approach. In this section, we address the issues related to linear solution.

4.1 Mathematical Model Specification and PDE Template

Traditionally, linear system assembly is left to the user application code, since the linear system comes from the application model. To reduce the burden of the user, SIFFEA instead only requires the user to specify the application model, and all the details of assembly are hidden from the user. The application model is encapsulated in a user-defined class, which provides four serial member functions: getDof(), formESM(), formELV(), and formBC(). getDof() returns the degree of freedom of each vertex. formESM() and formELV() compute the element stiffness matrix and load vector respectively based on vertex coordinates. formBC() computes the boundary condition of a give point based on its coordinates and mark. We unify the interface of formBC() for Dirichlet, Neumann and mixed boundary conditions, by asking the user to treat all of them as mixed conditions, i.e., the weighted sum of the Dirichlet and Neumann conditions, where the weight could be zero. SIFFEA automatically distinguishes the Dirichlet and Neumann conditions at runtime by checking the weights when necessary.

Separating model specification from assembly implementation greatly reduces the user's burden. However, it can still be tedious to compute element matrices and vectors. We observe that finite element models usually share a small number of fundamental equations. This is because the models are typically derived from a few fundamental laws. Based on this observation, we are creating a *PDE template*, which is a collection of the most commonly used equations and their element matrices and vectors. These equations are called *template equations*, and each template equation is organized the same as a user-defined model. In the template, however, the functions formESM() and formELV() are overloaded functions to support different types of elements. The standard element library can be used in defining these functions.

A template equation can be used as is, or be inherited when the user equation is almost identical to the template equation. Several template equations can also be aggregated to model more complex problems. Furthermore, the user can perform basic arithmetic operations on the equations to create new equations. The PDE template can also be extended by adding classes into it.

4.2 Global Matrix and Vector Assembly

Recall that in the global stiffness matrix, each row and column corresponds to a vertex in the mesh, and there is a nonzero at row i and column j if vertex i and j are adjacent in the mesh. It then follows that the global matrix is symmetric, and its nonzero pattern can be determined statically. SIFFEA employs the vertex adjacency information to preallocate memory space for the global matrix, to minimize the number of system calls for memory allocations.

The global matrix is partitioned row-oriented onto processors. In particular, each processor owns the rows whose corresponding vertices are assigned to it. Linear system assembly works completely under the new vertex number system, so that each processor owns some consecutive rows of the global matrix. The global vector is also partitioned similarly.

To achieve the best scalability, the global matrix and vector are assembled on all processors concurrently. Each processor is in charge of constructing the element matrices and vectors of its locally owned elements. Since all coordinates needed are already cached in the local memory, computing element matrices does not introduce extra communication. The element matrices and vectors are then accumulated into the global matrix and vector with moderate amount of communication.

4.3 Boundary Condition Adjustment

After the element matrices are assembled into the global matrix, the global matrix is singular, and boundary conditions must be applied. In the implementation, boundary condition adjustment is taken care of during the element matrices are accumulated into the global matrix for the best performance. However, for the sake of presentation, we discuss the boundary condition separately here.

We handle the Neumann and mixed conditions in the standard ways [23]. There are a variety of ways to apply the Dirichlet conditions [23]. We adopted a simple approach, in which each row corresponding to a Dirichlet boundary vertex is replaced by the boundary condition of that vertex. To preserve symmetry, the off-diagonal nonzeros in the column corresponding to that vertex is also eliminated and moved to the right-hand side. This method has the advantages of not changing the number of equations, and hence simplifies the implementation. However, it suffers from a tiny amount of redundant computation.

4.4 Linear Solver

We currently use PETSc for the solution of the sparse linear system. PETSc supports a variety of iterative solvers, and provides a rich set of functionality such as matrix and vector management [6]. Many other high-quality parallel sparse linear solvers are also available in the public domain, such as Aztec [11], PSPASES [10], and SPOOLES [2] etc.. We plan to provide interfaces with other linear solvers in the near future, and make it flexible to switch between the solvers. In particular, the multigrid and direct solvers are planned to be supported.

5 Error Estimation and Adaptive Mesh Refinement

After solving the linear system in parallel, we obtain an initial numerical solution u. The initial mesh M does not guarantee a good solution, and hence it must be refined properly [4, 5]. In this section, we consider the issues regarding to adaptive mesh refinement, and the algorithms for it.

A spacing function such as el_M and lfs_M [15, 18] specifies how fine a mesh should be at a particular region. This function can be derived from the previous numerical results or from the local geometry feature. The spacing function for a well-shaped mesh should be smooth in the sense that it changes slowly as a function of distance, i.e., it satisfies the α-*Lipschitz* condition [13].

Our spacing function is based on Li *et al.* [13]. First, a refine-coarsening factor δ_p of each point p is obtained from *a posterior* error analysis, by comparing the error vector e of the initial solution u with a scalar global error measurement ϵ. For example, if we choose e to be the magnitude of residual, and ϵ the norm of e, then δ_p can be defined as: $(1)(e_p/\epsilon)^{-1/k}$, if $l < e_p/\epsilon < L$; $(2)1/l^{1/k}$, if $e_p/\epsilon \leq l$; $(3)1/L^{1/k}$, if $e_p/\epsilon \geq L$; where k is the accuracy order of the elements, and $0 < l \leq 1 \leq L$. The spacing function h is then obtained from δ_p [13].

The mesh is then adaptively refined according to h. We would like to use the structure of the current mesh M as much and as efficiently as possible. The first step of our algorithm is to compute a maximum spacing function f that satisfies the new spacing requirement h. Then, we generate a β-sphere-packing of the domain with respect to f using the following procedure [13].

Let $S_1 = \{B(p, f(p)/2) | p \in M\}$. Then sample points in every triangle of the mesh and let S_2 be the set of these spheres defined by them. The spheres in $S_1 \cup S_2$ are ordered as the following: First the spheres centered on the boundary; then, all other spheres in S_1 in increasing order of radii; then followed by all spheres in S_2 in increasing order of radii. The intersection relation of spheres defines a *Conflict Graph* (CG). Let S be the set of spheres which form the *Lexical-First Maximal Independent Set* ($LFMIS$) [13] of CG. M' is then the Delaunay triangulation of the centers of the spheres in S. The generated mesh is well shaped and the size is within a constant factor of the optimal mesh [13].

We are currently developing a parallel adaptive mesh refinement algorithm, in which the same idea as the parallel mesh generation is to be used.

6 Experimental Study

The input geometry of our testing application is the mesh as shown in Fig. 2. The mesh contains 11736 vertices, 29220 edges, and 17480 triangles, with minimum angle of 20.1°. The model being solved is a Laplace equation. The boundary conditions are set as follows. The innermost circle has no displacement, and the outermost circle has unit displacement in their normal directions. All the other boundaries have zero natural boundary condition. The computational solution of this sample problem is visualized in Fig. 3 and Fig. 4, in which Fig. 3 plots the displacement vectors of all the mesh vertices, and Fig. 4 shows the displacements in the x direction of the vertices in a three dimensional plot.

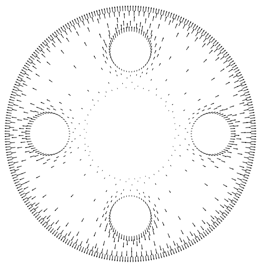

Fig. 3. Displacement vectors.

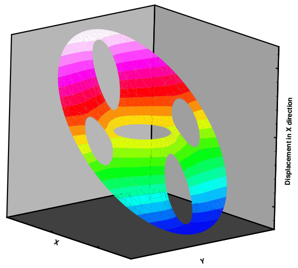

Fig. 4. Displacements in x direction.

We conducted some preliminary experiments on a 32-processor Origin 2000 at NCSA, which is a distributed shared memory machine. Each processor is a 195 MHz R10000. The native implementation of MPI on Origin was used during the test. The sparse linear solver used was Conjugate Gradient method, where the absolute and relative convergence tolerances were both set at 10^{-8}. The preprocessing step is done in sequential, and takes about 29.0 seconds, including both mesh generation and refinement.

We present the execution times of data distribution, mesh partitioning, matrix assembly, and linear solver in Table 1. The results show that the execution times of data distributions, which involve intensive communication, do not change much with the number of processors. The mesh partitioner has no speedup at all, maybe due to the following reasons: (1) The initial vertex distribution is essentially random, which causes large amount of communication on large number of processors; (2) the complexity of the partitioning problem itself increases with the number of the processors. This also shows the importance of parallel mesh generation: After parallel mesh generation, each processor will contain a mesh of a subdomain, which already gives a fairly good partition; then a cheaper partition refinement algorithm can be used. The matrix assembly code achieves good speedup up to 16 processors. The linear solver delivers reasonable speedup up to 8 processors, noting that in the experiment, we observed that the number of iterations tends to increase with the number of processors. We expect to achieve more speedups on larger number of processors with larger applications.

7 Related Work

A lot of work has been done on PFEM in the past few years. Various approaches have been used by researchers. In this section, we summarize some features of related work and highlight their differences from SIFFEA.

The parallel FEM on Now [17], developed at the RPI, replicates the mesh topological information and geometric data on all processors to allow concurrent processing. It currently only supports the multigrid method, and achieves

Table 1. Performance results on Origin 2000 at NCSA.

Time in seconds	1	2	4	8	16
Initial distribution	-	0.023	0.024	0.025	0.025
ParMetis	-	0.114	0.112	0.124	0.315
Data redistribution	-	0.032	0.026	0.024	0.025
Matrix assembly	0.195	0.101	0.061	0.045	0.030
Linear solver	2.18	1.28	0.80	0.51	0.64

speedups only up to 3 processors. Replication of data limits its usage for larger problems. It does not support adaptive refinement either.

Archimedes [20], developed at the CMU, is an integrated tool set for solving PDE problems on parallel computers. It employs algorithms for sequential mesh generation, partitioning, and unstructured parallel computation as a base for numerical solution of FEM. The central tool is Author, a data parallelizing compiler for unstructured finite element problems. The user writes a complete sequential FEM solver, including linear system assembly and solve, and then Author automatically parallelizes the solver. Archimedes is written in C.

The Prometheus library [1], is a parallel multigrid-based linear system solver for unstructured finite element problems developed at Berkeley. It takes a mesh distributed across multiple processors, and automatically generates all the coarse representations and operators for standard multigrid methods. Prometheus does not integrate mesh generation and adaptive mesh refinement.

8 Conclusion and Future Work

We have described SIFFEA, an integrated framework for PFEM, which synthesizes several widely used components in FEM: mesh generator, mesh partitioner, linear system assembler and solver, error estimator, and adaptive mesh refiner. SIFFEA is easily extensible, and provides a simple user interface, due to its object-oriented design. We are underway of designing and implementing algorithms for parallel mesh generation and refinement, to achieve better scalability. Another direction of the mesh generation is to support the higher order and heterogeneous elements. We also plan to extend the linear system assembler to time-dependent and nonlinear problems. The future plan for linear system solvers is to provide interfaces with multigrid and direct solvers.

References

1. M. Adams, J. W. Demmel, and R. L. Taylor. Prometheus home page. http://www.cs.berkeley.edu/~madams/Prometheus-1.0, 1999.
2. C. Ashcraft and R. Grimes. SPOOLES: An object-oriented sparse matrix library. In *Proc. of the 9th SIAM Conf. on Parallel Processing for Sci. Comp.*, 1999.

3. I. Babuška and A. K. Aziz. On the angle condition in the finite element method. *SIAM J. Numer. Anal.*, 13(2):214–226, 1976.
4. I. Babuška, J. E. Flaherty, W. D. Henshaw, J. E. Hopcroft, J. E. Oliger, and T. Tezduyar, editors. *Modeling, Mesh Generation and Adaptive Numerical Methods for Partial Differential Equations*, volume 75 of *The IMA Volumes in Mathematics and Its Applications*. Springer-Verlag, 1995.
5. I. Babuška, O. C. Zienkiewicz, J. Gago, and E. R. de A. Oliveira, editors. *Accuracy Estimates and Adaptive Refinement in Finite Element Computations*. Wiley Series in Numerical Methods in Engineering. John Wiley, 1986.
6. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, 1998.
7. J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9), Sep. 1975.
8. J. Dongarra. Freely avaliable software for linear algegra on the web. http://www.netlib.org/utk/people/JackDongarra/la-sw.htm, 1999.
9. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graphics*, 4(2):74–123, April 1985.
10. A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. PSPASES: An efficient and scalable parallel sparse direct solver. To appear, 1999.
11. S. A. Hutchinson, L. V. Prevost, J. N. Shadid, C. H. Tong, and R. S. Tuminaro. Aztec user's guide: Version 2.0. Technical report, Sandia National Laborotories, Albuquerque, NM 87185, September 1998.
12. G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. TR 96-036, Computer Science Department, University of Minnesota, Minneapolis, MN 55454, May 1996.
13. X. Y. Li, S. H. Teng, and A. Üngör. Simultaneous refinement and coarsening: adaptive meshing with moving boundaries. In *7th International Meshing Roundtable*, pages 201–210, Dearborn, Mich., 1998.
14. X. Y. Li, S. H. Teng, and A. Üngör. Biting: advancing front meets sphere packing. In *2nd Symposium on trends in unstructured mesh generation*, Boulder, Colorado., 1999.
15. G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proc. 27^{th} Annu. ACM Sympos. Theory Comput.*, pages 683–692, 1995.
16. G. L. Miller, S. H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite element meshes. *SIAM J. Scientific Computing*, 1998.
17. M. V. Nibhanupudi and B. K. Szymanski. Parallel FEM on NoW home page. http://www.cs.rpi.edu/projects/pse/fea_nows/fea_nows.html, 1999.
18. J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, May 1995.
19. J. R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *14th Annual ACM Symposium on Computational Geometry*, pages 86–95, 1998.
20. J. R. Shewchuk and D. R. O'Hallaron. Archimedes home page. http://www.cs.cmu.edu/~quake/arch.html, 1999.
21. R. Sibson. Locally equiangular triangulations. In *Computer Journal*, volume 21, pages 243–245, 1978.
22. G. Strang and G. J. Fix. *An Analysis of the Finite Element Method*. Prentice-Hall, 1973.
23. O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*, volume 1–2. McGraw-Hill, London, fourth edition, 1991.