NASA/CR-95- 206808

NAG3-1637

SENSITIVITY ANALYSIS AND OPTIMIZATION OF ENCLOSURE RADIATION WITH APPLICATIONS TO CRYSTAL GROWTH

BY

MICHAEL M. TILLER

B. M. Eng., University of Minnesota, 1991
M. S., University of Illinois, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

AUGUST 1995

WE HEREBY RECOMMEND THAT THE THESIS BY

MICHAEL M. TILLER

ENTITLED SENSITIVITY ANALYSIS AND OPTIMIZATION OF ENCLOSURE RADIATION WITH APPLICATIONS TO CRYSTAL GROWTH

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

[illegible]
Director of Thesis Research

A. L. Addy
Head of Department

Committee on Final Examination†

[illegible]
Chairperson

[illegible]

[illegible]

D A Tortorelli

[illegible]

† Required for doctor's degree but not for master's.

O-517

Abstract

In engineering, simulation software is often used as a convenient means for carrying out experiments to evaluate physical systems. The benefit of using simulations as "numerical" experiments is that the experimental conditions can be easily modified and repeated at much lower cost than the comparable physical experiment. The goal of these experiments is to "improve" the process or result of the experiment. In most cases, the computational experiments employ the same trial and error approach as their physical counterparts. When using this approach for complex systems, the cause and effect relationship of the system may never be fully understood and efficient strategies for improvement never utilized. However, it is possible when running simulations to accurately and efficiently determine the sensitivity of the system results with respect to simulation parameters (*e.g.,* initial conditions, boundary conditions, and material properties) by manipulating the underlying computations. This results in a better understanding of the system dynamics and gives us efficient means to improve processing conditions.

We begin by discussing the steps involved in performing simulations. Then we consider how sensitivity information about simulation results can be obtained and ways this information may be used to improve the process or result of the experiment. Next, we discuss optimization and the efficient algorithms which use sensitivity information. We draw on all this information to propose a generalized approach for integrating simulation and optimization, with an emphasis on software programming issues.

After discussing our approach to simulation and optimization we consider an application involving crystal growth. This application is interesting because it includes radiative heat transfer. We discuss the computation of radiative view factors and the impact this mode of heat transfer has on our approach. Finally, we will demonstrate the results of our optimization.

Acknowledgements

In getting to this point, I have of course been helped by many people along the way. Most importantly, I would like to thank my parents for taking the time to show me how much fun learning can be. If I was curious about something while growing up, they always had the time to help me unravel the subject and contribute their own experiences.

I would also like to thank my wife Deepa for her support during our time in graduate school. In particular, I have often relied on her to help me organize my thoughts when trying to formalize ideas. I also depend on her to brighten my day when I'm feeling overburdened.

I'm grateful to my advisor Professor Jonathan Dantzig for providing me this opportunity. Throughout my graduate studies, Jon has kept an open mind about my ideas, has allowed me to pursue them and shown a great deal of patience in the process. In addition, Professor Daniel Tortorelli has made a substantial contribution to my graduate studies.

Last, but not least, are my fellow graduate students, past and present, who have helped me along the way. First, Thomas O'Connor and Nagendre Palle for welcoming me to the lab when I first started graduate school. Next, Tim Morthland and Paul Byrne for their contributions in making the lab such a pleasant place to work. Along the way, Mark Rhodes from Argonne National Labs joined the cast of 351 MEB and made a significant contribution to the spirit of the lab. If it were not for Sepehr Ebrahimi motivating me to complete my thesis, I would have had to complete my Ph.D. after starting work at Ford Motor Company. Finally, I'd like to thank Robert McDavid for sharing with me his knowledge of so many things I know nothing about.

Table Of Contents

Chapter 1

Simulation

1.1 Overview

In engineering, simulation software is used as a convenient means of investigating processes and judging product performance. The benefit of using simulations is that experimental conditions can be easily modified and repeated at relatively low cost. In general, the goal of these numerical experiments is to in some way "improve" the process or result of the experiment. In most cases, the computational experiments employ the same trial-and-error approach used in physical experiments. The only way to understand the cause and effect relationships in the system is by observing them and examining trends in results. Using this approach results in important aspects of the system never being fully understood, and efficient strategies for improvement never being utilized.

In this chapter, we first discuss where the simulations system of equations come from. Then, we present a general form for nonlinear systems of equations and discuss how they are solved. Finally, we derive some of the important terms used to solve these systems for both steady-state and transient simulations.

1.2 Systems of Nonlinear Equations

Simulations are discretized computer representations of physical systems. One method of forming a discrete representation is the finite element method (FEM) which divides the domain into small pieces called elements. In FEM formulations, the continuous governing equations, which usually apply at every point in the physical domain, are only enforced in an "average" sense over the elements. The degrees of freedom are represented at the vertices of the elements and interpolated over the element using basis functions. The equation for each degree of freedom is written in terms of "neighboring" degrees of freedom. The result is a system of equations with very few non-zero terms, such a system is termed *sparse.*

If the underlying phenomena of the physical system are nonlinear, the systems of equations that arise from discretization will also be nonlinear. We assume that such a system can be written in the following general form:

$$\mathbf{R}(\mathbf{u}(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b}) = \mathbf{0} \tag{1.1}$$

where $\mathbf{R}$ is called the residual vector, $\mathbf{u}$ is the solution to the nonlinear system of equations, $\mathbf{b}$ is a vector of *design variables* or *simulation parameters*, and $\mathbf{v}$ represents other quantities which are functions of $\mathbf{b}$.

1.3 Solution Algorithms

1.3.1 Newton-Raphson

If the tangent matrix $\frac{\partial \mathbf{R}}{\partial \mathbf{u}}$ can be calculated accurately and is non-singular, then the Newton-Raphson method is a very effective method for solving the residual equations for $\mathbf{u}$. Each iteration of the Newton-Raphson method solves:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}}(\mathbf{u}_i(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b})\Delta\mathbf{u}_i = -\mathbf{R}(\mathbf{u}_i(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b}) \tag{1.2}$$

Note that Equation (1.2) is a linearization of the residual function around the point $\mathbf{u}_i$. This linearization results in a linear system of equations which are used to solve for the increment $\Delta\mathbf{u}_i$. The quantity $\Delta\mathbf{u}_i$ is used to determine $\mathbf{u}_{i+1}$ by using the updating relation:

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta\mathbf{u}_i \tag{1.3}$$

1.3.2 Linear Systems of Equations

There are two classes of methods for solving linear systems like the one in Equation (1.2), they are direct methods and iterative (or indirect) methods. A good example of a direct methods is LU decomposition which works by factoring the tangent matrix into lower (L) and upper (U) triangular matrices, and then solving the system by back-substitution. Indirect methods, like Krylov subspace methods, work by multiplying vectors by the tangent matrix (and sometimes its transpose).

There are many interesting contrasts between these two methods. First, iterative methods terminate when the answer is deemed "good enough", while direct methods will provide only their "best" answer. Another important difference is that direct methods require the portions of the matrix to be stored in memory and decomposition can result in additional non-zero terms being created, while iterative methods only require matrix-vector products which can be done without even forming the matrix.

Given the general form for a set of simultaneous linear equations:

$$\mathbf{Ax} = \mathbf{b} \tag{1.4}$$

it is worth noting that the Krylov subspace methods require only the matrix-vector products $\mathbf{Az}$ and $\mathbf{z}^T\mathbf{A}$ for arbitrary vectors $\mathbf{z}$. Such matrix vector products can be quickly evaluated for sparse matrices. The computational cost of a matrix-vector product is primarily a function of the sparseness of the matrix, and not of its structure. On the other hand, the efficiency of factorization methods such as LU decomposition are directly related to matrix structure. The reason for this is that the envelope of the $\mathbf{L}$ and $\mathbf{U}$ matrices will be the same as the envelope for the matrix to be factored. The closer the non-zero elements are to the diagonal, the smaller the envelope of $\mathbf{A}$ and the fewer operations which must be performed in the factorization (see Watkins [1]).

The convergence rate of Krylov subspace methods is related to the condition number of the matrix $\mathbf{A}$. For the conjugate gradient algorithm, operating on a symmetric positivie definite system, the number of iterations required to converge to a given tolerance is proportional to the square root of the condition number. For many algorithms quantitative expressions for convergence are not available. As a general rule, the smaller the condition number of a matrix, the faster Krylov methods will converge (see Golub and Van Loan [2] for details).

In many cases it is possible to construct an approximation of $\mathbf{A}$, called $\tilde{\mathbf{A}}$, with a structure such that it is straightforward and efficient to calculate $\tilde{\mathbf{A}}^{-1}$. We then apply the Krylov subspace method to the modified system of equations:

$$[\tilde{\mathbf{A}}^{-1}\mathbf{A}]\mathbf{x} = \tilde{\mathbf{A}}^{-1}\mathbf{b} \tag{1.5}$$

Any vector $\mathbf{x}$ which solves Equation (1.5) is a solution to Equation (1.4). The difference is that the condition number of the matrix $[\tilde{\mathbf{A}}^{-1}\mathbf{A}]$ should be significantly reduced, depending on how well $\tilde{\mathbf{A}}$ approximates $\mathbf{A}$.

Using $\tilde{\mathbf{A}}$ to enhance the convergence of Krylov subspace methods is called preconditioning (see Barret, *et al.* [3] or Golub and Van Loan [2]). One simple way to construct $\tilde{\mathbf{A}}$ is to make it a diagonal matrix, comprising the diagonal elements of $\mathbf{A}$. This approach makes inversion of $\tilde{\mathbf{A}}$ trivial. Another approach is to extract the diagonal and N bands above and below the diagonal from $\mathbf{A}$ and place them in $\tilde{\mathbf{A}}$. In this case, LU decomposition can be used to decompose $\tilde{\mathbf{A}}$. Yet another approach is to perform LU decomposition on $\mathbf{A}$ but not allow any fill (zero elements becoming non-zero elements). This is called partial or incomplete LU factorization. The implementation of preconditioned methods is such that it is not actually necessary to form $\tilde{\mathbf{A}}^{-1}$ but rather to compute $\tilde{\mathbf{A}}^{-1}\mathbf{z}$ and $\mathbf{z}^T\tilde{\mathbf{A}}^{-1}$ for any arbitrary $\mathbf{z}$. The best preconditioning method depends on the properties of $\mathbf{A}$ itself.

1.4 Steady State Simulations

In steady state problems, the governing equations are of the form:

$$\mathbf{K}(\mathbf{u}(\mathbf{b}),\mathbf{v}(\mathbf{b}),\mathbf{b})\mathbf{u}(\mathbf{b}) = \mathbf{f}(\mathbf{u}(\mathbf{b}),\mathbf{v}(\mathbf{b}),\mathbf{b}) \tag{1.6}$$

where $\mathbf{K}$ is (historically) called the secant stiffness matrix and $\mathbf{f}$ is the force vector. To solve the system we must find the $\mathbf{u}$ which satisfies Equation (1.1), with $\mathbf{R}$ defined as:

$$\mathbf{R}(\mathbf{u}(\mathbf{b}),\mathbf{v}(\mathbf{b}),\mathbf{b}) = \mathbf{K}(\mathbf{u}(\mathbf{b}),\mathbf{v}(\mathbf{b}),\mathbf{b})\mathbf{u}(\mathbf{b}) - \mathbf{f}(\mathbf{u}(\mathbf{b}),\mathbf{v}(\mathbf{b}),\mathbf{b}) \tag{1.7}$$

Recall that the Newton-Raphson algorithm in Section 1.3 required knowledge of the tangent matrix, $\frac{\partial \mathbf{R}}{\partial \mathbf{u}}$. For this system of equations, the tangent matrix would be:

$$\frac{\partial R_i}{\partial u_k} = \frac{\partial K_{ij}}{\partial u_k} u_j + K_{ik} - \frac{\partial f_i}{\partial u_k} \tag{1.8}$$

The use of indicial notation in some expressions is necessary to clarify exactly what operations are to be performed.

1.5 Transient Simulations

Next, we consider a continuous first-order system of differential equations,

$$\mathbf{C}(\mathbf{u},\mathbf{b})\dot{\mathbf{u}} + \mathbf{K}(\mathbf{u},\mathbf{b})\mathbf{u} = \mathbf{f}(\mathbf{u},\mathbf{b}) \tag{1.9}$$

We solve the system by integrating the differential equation over discrete quantities of time called *time steps.* To discretize the system, we use the following relation:

$$\theta \dot{\mathbf{u}}^{n+1} + (1-\theta)\dot{\mathbf{u}}^{n} = \frac{\mathbf{u}^{n+1} - \mathbf{u}^{n}}{\Delta t_n} \tag{1.10}$$

where the superscripts indicate which time step the vectors are associated with. Dropping the explicit dependencies and using Equation (1.10) to integrate Equation (1.9) we get the algebraic system of equations:

$$\left[\mathbf{C}^{n+1} + \theta \Delta t_n \mathbf{K}^{n+1}\right] \mathbf{u}^{n+1} = \left[\mathbf{C}^{n} - (1-\theta)\Delta t_n \mathbf{K}^{n}\right] \mathbf{u}^{n} + \Delta t_n \left[\theta \mathbf{f}^{n+1} + (1-\theta)\mathbf{f}^{n}\right] \tag{1.11}$$

where $\mathbf{u}^n$ is the solution at the current time step, $\mathbf{u}^{n+1}$ is the solution at the next time step and Δt_n is the time step. The residual function for a transient system can then be written as:

$$\begin{aligned} \mathbf{R}(\mathbf{u}^{n+1}, \mathbf{u}^{n}, \mathbf{b}) &= \left[\mathbf{C}^{n+1} + \theta \Delta t_n \mathbf{K}^{n+1}\right] \mathbf{u}^{n+1} - \left[\mathbf{C}^{n} - (1-\theta)\Delta t_n \mathbf{K}^{n}\right] \mathbf{u}^{n} - \\ & \quad \Delta t_n \left[\theta \mathbf{f}^{n+1} + (1-\theta)\mathbf{f}^{n}\right] \end{aligned} \tag{1.12}$$

Note that for a transient system, the quantity $\mathbf{v}$ in Equation (1.2) is represented by the solution $\mathbf{u}^n$.

Differentiating the transient system show in Equation (1.12) by $\mathbf{u}^{n+1}$ we get the following definitions for the tangent matrix:

$$\begin{aligned}\frac{\partial R_i}{\partial u_k^{n+1}} &= \left[\frac{\partial C_{ij}^{n+1}}{\partial u_k^{n+1}} + \theta\Delta t_n \frac{\partial K_{ij}^{n+1}}{\partial u_k^{n+1}}\right] u_j^{n+1} + \\ & \left[C_{ik}^{n+1} + \theta\Delta t_n K_{ik}^{n+1}\right] - \Delta t_n \theta \frac{\partial f_i^{n+1}}{\partial u_k^{n+1}} \end{aligned} \tag{1.13}$$

1.6 Parallelization

Given a system whose tangent matrix is sparse (*e.g.,* finite element analysis), there are some interesting opportunities for exploiting the inherent parallelism by performing domain decomposition. Details about efficiency and implementation can be found in Sharma, *et al.* [4]. Much of the speed-up achieved by domain decomposition is the result of the algorithms used for solving the linearized system (see Section 1.3). For now, we discuss how such problems are posed in terms of quantities that we have already discussed.

For simplicity we consider a system decomposed into just two separate domains, although the same analysis can be used for an arbitrary number of decomposed domains. In addition, we only consider the solution and design vectors. The solutions for the two domains are called $\mathbf{u}_1$ and $\mathbf{u}_2$. Now let us assume that we have a residual function for each domain, and we refer to them as $\mathbf{R}_1(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b})$ and $\mathbf{R}_2(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b})$. Our goal is to find the zeroes (roots) for both residual functions. To this end, we construct a global residual function $\mathbf{R}(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b})$ and solution vector as follows:

$$\mathbf{R}(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b}) = \begin{bmatrix} \mathbf{R}_1(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b}), \mathbf{b}) \\ \mathbf{R}_2(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b}) \end{bmatrix} \tag{1.14}$$

$$\mathbf{u}(\mathbf{u}_1(\mathbf{b}), \mathbf{u}_2(\mathbf{b}), \mathbf{b}) = \begin{bmatrix} \mathbf{u}_1(\mathbf{b}) \\ \mathbf{u}_2(\mathbf{b}) \end{bmatrix} \tag{1.15}$$

Next, we use the Newton-Raphson method described in section 1.3 to find the roots of the global residual. Recall that for each Newton-Raphson iteration we solve the system:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}}(\mathbf{u}_i(\mathbf{b}), \mathbf{b})\Delta \mathbf{u}_i = -\mathbf{R}(\mathbf{u}_i(\mathbf{b}), \mathbf{b}) \tag{1.16}$$

which in the case of our decomposed domain becomes:

$$\begin{bmatrix} \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_1}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) & \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_2}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \\ \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_1}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) & \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_2}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \end{bmatrix} \begin{Bmatrix} \Delta \mathbf{u}_{1_i} \\ \Delta \mathbf{u}_{2_i} \end{Bmatrix} = \begin{Bmatrix} -\mathbf{R}_1(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \\ -\mathbf{R}_2(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \end{Bmatrix} \tag{1.17}$$

Specifically note that the tangent matrix has become:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}}(\mathbf{u}_i(\mathbf{b}), \mathbf{b}) = \begin{bmatrix} \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_1}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) & \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_2}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \\ \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_1}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) & \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_2}(\mathbf{u}_{1_i}(\mathbf{b}), \mathbf{u}_{2_i}(\mathbf{b}), \mathbf{b}) \end{bmatrix} \tag{1.18}$$

and recall that the Krylov subspace method requires the matrix-vector product:

$$\begin{bmatrix} \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_1} & \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_2} \\ \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_1} & \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_2} \end{bmatrix} \begin{Bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{Bmatrix} = \begin{Bmatrix} \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_1}\mathbf{z}_1 + \frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_2}\mathbf{z}_2 \\ \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_1}\mathbf{z}_1 + \frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_2}\mathbf{z}_2 \end{Bmatrix} \tag{1.19}$$

which can be done in parallel, each processor using a different segment of the tangent matrix to operate on.

In addition to speeding up solution times by parallelizing computations, domain decomposition has applications in what is commonly called concurrent engineering. Examples of systems for concurrent engineering can be found in Choi [5] and Tortorelli [6]. It is sometimes necessary to solve different aspects of the same problem using different simulation packages. For example it might be necessary, in a single application, to solve for temperatures using one simulation package (*e.g.,* FIDAP), and perform stress analysis with another (*e.g.,* ANSYS). Unfortunately, in cases where coupling between the two results exist (*i.e.,* the temperature solution depends on the stress solution and the stress solution depends on the temperature solution) it may be impossible to express this coupling in either package (*i.e.,* it may not be possible to compute $\frac{\partial \mathbf{R}_i}{\partial \mathbf{u}_j}$ for $i \neq j$).

Michaleris, *et al.* [7] describe how to compute sensitivities with multiple simulation packages by alternating between the systems, simultaneously converging on solutions for both. While it is possible to obtain a converged solution for these cases, it requires access to a complete tangent matrix, although not necessarily in core all at once. So, drawing from our example above, the terms $\frac{\partial \mathbf{R}_1}{\partial \mathbf{u}_2}$ and $\frac{\partial \mathbf{R}_2}{\partial \mathbf{u}_1}$ shown in Equation (1.18) must be supplied outside the simulation packages. For this reason it is generally easier to have these computations done from within the same simulation package if possible.

In the next chapter, we use our knowledge of how systems of equations are formed and solved to demonstrate how sensitivity information can be derived for simulation results. We will find that the terms used by the Newton-Raphson algorithm will be very useful in computing sensitivity information.

Chapter 2

Sensitivity Analysis

2.1 Overview

Accurate and efficient methods will be demonstrated for determining the sensitivity of the system response to changes in the simulation parameters (*e.g.,* initial conditions, boundary conditions, material properties) by manipulating the underlying computations. In this chapter, we will discuss how such sensitivity information can be obtained, and in the next chapter we discuss how this sensitivity data can be used to improve the process or result of a simulation.

We consider a program which performs simulations (the *simulator*) as a device which transforms "input data", such as nodal coordinates, boundary conditions, and material properties, *into* some kind of result, such as nodal temperatures and nodal velocities. Likewise, a simulator which performs sensitivity calculations transforms sensitivity information *about* input data into sensitivity information *about* the result. In this chapter we will discuss in detail how such sensitivity information can be computed.

Efficient calculation of sensitivity information requires an accurate tangent matrix, $\frac{\partial \mathbf{R}}{\partial \mathbf{u}}$. Sensitivity analysis has been applied for some time in structural engineering applications (see Venkayya [8], Haug *et al.* [9] or Austin *et al.* [10]). For structural systems with linear response, computing sensitivities is straightforward because the already computed stiffness matrix will be equal to the tangent matrix. However for nonlinear problems, computing sensitivities is made more complicated because the tangent matrix is different than the stiffness matrix. Fortunately, since Newton-Raphson is a common solver which requires $\frac{\partial \mathbf{R}}{\partial \mathbf{u}}$, the tangent matrix is often available. However, for complicated phenomena such as radiation and solidification, some terms are sometimes neglected from the tangent matrix, which will lead to erroneous sensitivities. When the tangent matrix is not formed correctly, other less efficient algorithms, such as finite different approximations can be used. An excellent overview, for both linear and nonlinear systems, can be found in Haftka and Gürdal [11].

Unfortunately, even if the source code is available, most existing simulator's data structures make no provision for sensitivity information, and experience has shown that considerable specialized changes are necessary to adapt these simulators to process sensitivity information for nonlinear problems. Even if one makes this effort, it is still very cumbersome to inform the simulator of sensitivity information about the input data, because conventional simulators lack a sensitivity "vocabulary" in their user interfaces. To illustrate this point, consider the geometric definition of the model. The only geometric data normally required by a FEA simulator consists of nodal coordinates and element connectivity. However, it is more natural for a designer to want sensitivity information with respect to some aggregation of nodal data, such as a dimension of the computational domain. This calculation requires knowledge of the sensitivity of each node's coordinates with respect to that dimension. How do we obtain this sensitivity, and how should it be communicated to the simulator? Similar observations may be made about any other simulation parameter. This is not to say that the task is impossible, rather that it must be done differently for every simulator. For this reason, we have developed a simulator designed for both efficient solution and efficient sensitivity analysis.

2.2 Integrated Sensitivity Analysis

The application of optimization has been slower for nonlinear problems, such as solidification, radiative heat transfer and fluid flow, because efficient schemes to compute sensitivities are difficult to implement. Many of the techniques described by Tortorelli [12] are of value for these problems, because they provide a formal way to determine the sensitivity of the simulated response with respect to simulation parameters for almost any problem where the system response is determined by use of computer simulation. Below we discuss the specifics of these techniques, how they have been applied to specific engineering problems, and later show how the techniques can be formalized to construct a framework for simulation and sensitivity analysis.

To implement these methods effectively, the entire approach to communication with the simulator must be revised. Efficient calculation of sensitivity information requires that, for each computation in the simulation package, an analogous computation involving sensitivity information must also be available. Modifying existing simulation codes thus requires adding these analogous computations, as well as the logic to utilize them. The approach we have taken here has been to formalize the computations involved in the simulation and to express them in terms of abstract C++ class specifications. Thus, we build into the code the ability to accept and manipulate sensitivity information.

As we shall see in the next chapter, much of the sensitivity analysis we perform is motivated by optimization problems. In these problems we define a function $G(\mathbf{u}(\mathbf{b}), \mathbf{b})$ where $\mathbf{u}$ represents the vector of simulation results and $\mathbf{b}$ represents the vector of simulation parameters which are design variables. We seek to minimize the function G, called our *objective* function, with respect to $\mathbf{b}$. The most efficient optimization schemes require accurate calculation of $\frac{dG}{d\mathbf{b}}$. The more accurate the value for $\frac{dG}{d\mathbf{b}}$ the faster the optimization will converge. One way to express the gradient of G is by using the chain rule as follows:

$$\frac{dG}{d\mathbf{b}} = \frac{\partial G}{\partial \mathbf{b}} + \frac{\partial G}{\partial \mathbf{u}} \frac{d\mathbf{u}}{d\mathbf{b}} \tag{2.1}$$

2.3 Direct Differentiation Method

In this section we discuss how sensitivity information influences the relationship between optimization and simulation. Specifically, we demonstrate how to compute the sensitivity of simulation results with respect to simulation parameters, also known as the response sensitivity $\frac{d\mathbf{u}}{d\mathbf{b}}$. Response sensitivities represent the "cause and effect" relationship between the simulation result and simulation parameter variations. We begin by differentiating Equation

(1.1) with respect to $\mathbf{b}$, giving us:

$$\frac{d\mathbf{R}}{d\mathbf{b}} = \frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{d\mathbf{b}} + \frac{\partial \mathbf{R}}{\partial \mathbf{v}}\frac{d\mathbf{v}}{d\mathbf{b}} + \frac{\partial \mathbf{R}}{\partial \mathbf{b}} = \mathbf{0} \tag{2.2}$$

where we use the fact that Equation (1.1) holds for all $\mathbf{b}$. Rearranging terms and considering a single design variable b_k yields:

$$\frac{d\mathbf{u}}{db_k} = -\left[\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right]^{-1}\left(\frac{\partial \mathbf{R}}{\partial b_k} + \frac{\partial \mathbf{R}}{\partial \mathbf{v}}\frac{d\mathbf{v}}{db_k}\right) \tag{2.3}$$

Note that $\frac{\partial \mathbf{R}}{\partial \mathbf{v}}$ represents the tangent of the residual function with respect to a quantity whose value is known *a priori*, and for this reason it appears on the right hand side of Equation (2.3).

2.3.1 Steady State Simulations

Taking the governing equations in Equation (1.6) and differentiating with respect to b_k gives:

$$\frac{d\mathbf{K}}{db_k}\mathbf{u} + \mathbf{K}\frac{d\mathbf{u}}{db_k} = \frac{d\mathbf{f}}{db_k} \tag{2.4}$$

Expanding the equation further and including partial derivatives,

$$\left(\frac{\partial \mathbf{K}}{\partial b_k} + \frac{\partial \mathbf{K}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{db_k}\right)\mathbf{u} + \mathbf{K}\frac{d\mathbf{u}}{db_k} = \frac{\partial \mathbf{f}}{\partial b_k} + \frac{\partial \mathbf{f}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{db_k} \tag{2.5}$$

and then regrouping terms gives:

$$\left(K_{il} + \frac{\partial K_{ij}}{\partial u_l} u_j - \frac{\partial f_i}{\partial u_l}\right) \frac{du_l}{db_k} = \frac{\partial f_i}{\partial b_k} - \frac{\partial K_{il}}{\partial b_k} u_l \tag{2.6}$$

Notice we obtain the same result by using Equation (2.3) with $\mathbf{v} = \mathbf{0}$,

$$\frac{d\mathbf{u}}{db_k} = -\left[\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right]^{-1} \frac{\partial \mathbf{R}}{\partial \mathbf{b_k}} \tag{2.7}$$

where

$$\frac{\partial R_i}{\partial u_k} = \left(K_{ij}\delta_{jk} + \frac{\partial K_{ij}}{\partial u_k} u_j - \frac{\partial f_i}{\partial u_k}\right) \tag{2.8}$$

$$\frac{\partial R_i}{\partial b_k} = \frac{\partial K_{ij}}{\partial b_k} u_j - \frac{\partial f_i}{\partial b_k} \tag{2.9}$$

For this reason, we can avoid the steps shown in Equations (2.4)-(2.6) by using the definition of the residual system directly.

As pointed out previously, for linear problems we find the simplified result,

$$\frac{d\mathbf{u}}{db_k} = \mathbf{K}^{-1}\left(\frac{\partial \mathbf{f}}{\partial b_k} - \frac{\partial \mathbf{K}}{\partial b_k}\mathbf{u}\right) \tag{2.10}$$

Since the system response is solved as:

$$\mathbf{u} = \mathbf{K}^{-1}\mathbf{f} \tag{2.11}$$

the sensitivities of a linear system may be found by applying a "pseudo-load", $\frac{\partial \mathbf{f}}{\partial b_k} - \frac{\partial \mathbf{K}}{\partial b_k}\mathbf{u}$ so that:

$$\frac{d\mathbf{u}}{db_k} = \mathbf{K}^{-1}\left\{\frac{\partial \mathbf{f}}{\partial b_k} - \frac{\partial \mathbf{K}}{\partial b_k}\mathbf{u}\right\} \tag{2.12}$$

2.3.2 Transient Simulations

Using the definition of the residual function given in Equation (1.12) and the definition of the tangent matrix found in Equation (1.13), to apply Equation (2.3) we need to compute $\frac{\partial \mathbf{R}}{\partial \mathbf{u}^n}$ and $\frac{\partial \mathbf{R}}{\partial b_k}$.

Differentiating Equation (1.12) with respect to $\mathbf{u}^n$ gives:

$$\begin{aligned}\frac{\partial R_i}{\partial u_k^n} &= \left[\frac{\partial C_{ij}^{n+1}}{\partial u_k^n} + \theta \Delta t_n \frac{\partial K_{ij}^{n+1}}{\partial u_k^n}\right] u_j^{n+1} - [C_{ik}^n - (1-\theta)\Delta t_n K_{ik}^n] \\ &\quad - \left[\frac{\partial C_{ij}^n}{\partial u_k^n} - (1-\theta)\Delta t_n \frac{\partial K_{ij}^n}{\partial u_k^n}\right] u_j^n - \Delta t_n \left[\theta \frac{\partial f_i^{n+1}}{\partial u_k^n} + (1-\theta)\frac{\partial f_i^n}{\partial u_k^n}\right]\end{aligned} \tag{2.13}$$

Likewise, differentiating instead by b_k gives us:

$$\begin{aligned}\frac{\partial \mathbf{R}}{\partial b_k} &= \left[\frac{\partial \mathbf{C}^{n+1}}{\partial b_k} + \theta \Delta t_n \frac{\partial \mathbf{K}^{n+1}}{\partial b_k}\right] \mathbf{u}^{n+1} - \left[\frac{\partial \mathbf{C}^n}{\partial b_k} - (1-\theta)\Delta t_n \frac{\partial \mathbf{K}^n}{\partial b_k}\right] \mathbf{u}^n \\ &\quad - \Delta t_n \left[\theta \frac{\partial \mathbf{f}^{n+1}}{\partial b_k} + (1-\theta)\frac{\partial \mathbf{f}^n}{\partial b_k}\right]\end{aligned} \tag{2.14}$$

2.4 Adjoint Method

The adjoint method (see Haftka and Gürdal [11] or Haug *et al.* [9]) is an alternative to the direct differentiation method. To calculate sensitivities using this technique, a new objective function is formed by augmenting the original objective function as follows:

$$\hat{G} = G + \boldsymbol{\lambda}^T \mathbf{R} \tag{2.15}$$

where $\boldsymbol{\lambda}$ is a vector, as yet unknown which is independent of $\mathbf{r}$ and $\mathbf{b}$. Assuming that a valid solution $\mathbf{u}$ has already been calculated such that $\mathbf{R}(\mathbf{u}(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b}) = \mathbf{0}$, then there is no difference between $\hat{G}$ and G given *any arbitrary value* for $\boldsymbol{\lambda}$. The importance of defining $\hat{G}$ is seen when we differentiate with respect to a design variable, which gives:

$$\frac{dG}{db_j} = \frac{d\hat{G}}{db_j} = \frac{\partial G}{\partial b_j} + \frac{\partial G}{\partial \mathbf{u}}\frac{d\mathbf{u}}{db_j} + \boldsymbol{\lambda}^T \left(\frac{\partial \mathbf{R}}{\partial b_j} + \frac{\partial \mathbf{R}}{\partial \mathbf{u}}\frac{d\mathbf{u}}{db_j} \right) \tag{2.16}$$

and then rearrange terms to find:

$$\frac{d\hat{G}}{db_j} = \frac{\partial G}{\partial b_j} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{R}}{\partial b_j} + \left(\frac{\partial G}{\partial \mathbf{u}} + \frac{\partial \mathbf{R}}{\partial \mathbf{u}}^T \boldsymbol{\lambda} \right) \cdot \frac{d\mathbf{u}}{db_j} \tag{2.17}$$

It is now convenient to choose $\boldsymbol{\lambda}$ to make the final term on the right hand side of Equation (2.17) vanish, *i.e.*,

$$\frac{\partial G}{\partial \mathbf{u}} + \frac{\partial \mathbf{R}}{\partial \mathbf{u}}^T \boldsymbol{\lambda} = \mathbf{0} \tag{2.18}$$

After solving this equation for $\boldsymbol{\lambda}$, the sensitivity of the objective function is obtained as:

$$\frac{d\hat{G}}{db_j} = \frac{\partial G}{\partial b_j} + \boldsymbol{\lambda}^T \frac{\partial \mathbf{R}}{\partial b_j} \tag{2.19}$$

Note that we do not require $\frac{\partial u}{\partial b}$, thus we may avoid solving the series of equations defined by Equation (2.3)

For steady state problems, the main preference for the direct differentiation or adjoint method depends on the ratio of the number of design variables to the number of constraints. The direct differentiation method requires us to solve Equation (2.3) for every design variable, whereas the adjoint method requires us to solve Equation (2.18) for every constraint/objective function. For this reason the adjoint method is preferred in situations where there are more design variables then constraint/objective functions. For transient problems, the direct differentiation method is more straightforward to implement and is often prefered over the adjoint method for this reason (see Haftka and Gürdal [11]). For brevity, we do not consider the formulation of the adjoint method for transient problems.

2.5 Solving Linear Sensitivity Equations

Note that Equation (2.3) must be solved for the right hand side associated with each design variable, using the same matrix. The need to solve for multiple right hand sides has some influence over the algorithm used. From Section 1.3.2, two common methods for solving linear systems of equations are LU decomposition and the Krylov subspace family of methods. If LU decomposition is used during each Newton-Raphson iteration, the already decomposed matrix can be used to solve the different right hand sides of Equation (2.3). Since the matrix is already decomposed, each additional right hand side requires only a back-substitution operation (see Watkins [1] for detail). When using the adjoint method, using incomplete

LU decomposition as a preconditioner for the Krylov methods gave the best results for our example problem (see Chapter 6 for detailed statistics).

Chapter 3

Optimization

3.1 Overview

The goal of optimization is to determine values for design variables (*i.e.,* a *design*) that satisfy some objective as closely as possible. For example, in structural applications, the objective may be to minimize the structure weight. In addition to an objective, there may also be a number of constraints which must be satisfied for the design to be considered feasible. Constraints for structural problems might require stresses and strains in the structure to be below certain safe values. In our applications, we will consider the design variables to be continuous, and we will also place upper and lower limits on the design variable values.

We consider only unconstrained optimization, where a continuous function is to be minimized with respect to design variables. The theory for this class of problems is discussed extensively by Vanderplaats [13]. Several software packages exist for solving this class of problems [14–16].

The most efficient optimization techniques require precise knowledge of how the function to be minimized responds to changes in its arguments (*i.e.,* the sensitivity) [11,13]. Although algorithms exist for performing the optimization without using this information directly, this often requires many more function evaluations. In many cases, if the user does not supply the sensitivities, they must be computed by a finite difference approximation.

Recently, there has been considerable effort to apply optimization techniques to various nonlinear systems. Examples of how nonlinear systems were adapted to provide sensitivity information can be found in applications of thermoelastic, transient heat conduction systems, by Tortorelli *et al.* [12,17], elastic-viscoplasticity, by Zhang and Mukherjee [18], and airfoil design, by Joh [19], Sorensen [20] and Burgreen *et al.* [21].

3.2 Design Optimization

As mentioned in Chapter 2, for optimization problems we define a function $G(\mathbf{u}(\mathbf{b}), \mathbf{b})$ which measures the quality of potential designs. We may, for example, define G to measure the difference between the simulated solution and some prescribed desired solution, so that $G(\mathbf{u}(\mathbf{b}), \mathbf{b}) = 0$ represents the ideal.

Let us consider a simple optimization example involving one-dimensional steady state heat conduction. The governing equation for this system is [22]:

$$\frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right) = 0 \tag{3.1}$$

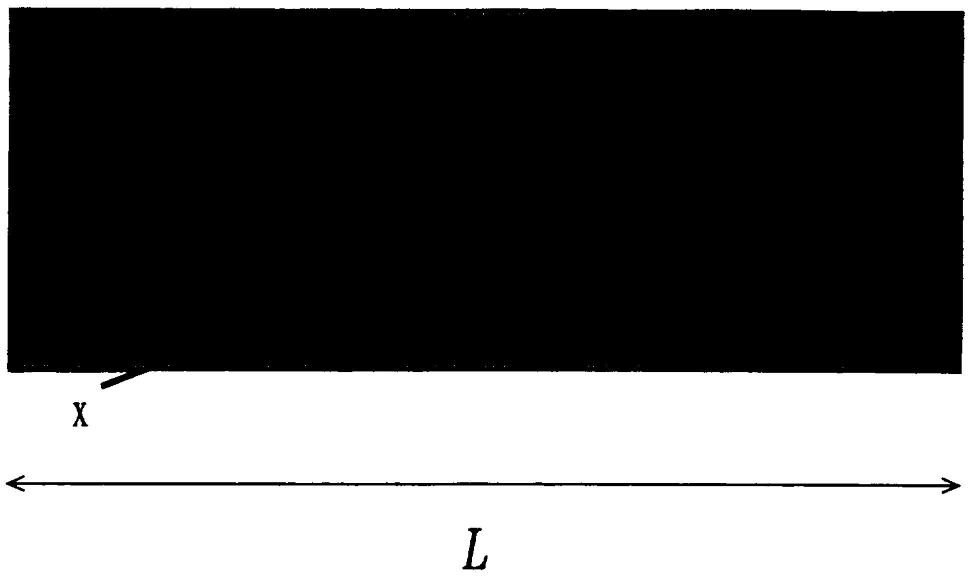


Figure 3.1: Steady State Heat Conduction

For this problem the temperature T takes the place of our generic unknown u in previous sections. Figure 3.1 shows the physical domain. The material has uniform conductivity k and length L. The boundary conditions are:

$$\begin{aligned} T(x=0) &= 1.0 && (3.2) \\ k\left.\frac{\partial T}{\partial x}\right|_{x=L} &= -h(T(L)-T_\infty) && (3.3) \end{aligned}$$

where h is the convection coefficient and T_∞ is the ambient temperature. The analytical solution for the temperature distribution in the slab is:

$$T(x) = 1.0 - \frac{h(1-T_\infty)}{1+Lh}x \tag{3.4}$$

Suppose our objective is to determine values for h, T_∞ and L. such that $T(L/2) = 0.7$. We define the design vector $\mathbf{b}$ such that:

$$\mathbf{b} = \left\{ \begin{array}{c} h \\ T_\infty \\ L \end{array} \right\} \tag{3.5}$$

It is important to recognize at this point that if the values for h, T_∞ and L are considered design parameters then it is more appropriate to refer to the temperature solution as $T(x, \mathbf{b})$.

We quantify the objective as:

$$G(T(x,\mathbf{b}),\mathbf{b}) = \sqrt{(T(L/2,\mathbf{b}) - .7)^2} \tag{3.6}$$

In order to apply a gradient based optimization algorithm, it is necessary to calculate the sensitivities which we write as:

$$\frac{dG}{d\mathbf{b}} = \frac{\partial G}{\partial \mathbf{b}} + \frac{\partial G}{\partial T}\frac{\partial T}{\partial \mathbf{b}} \tag{3.7}$$

For the objective function in Equation (3.6) there is no implicit dependency on the design variables so $\frac{\partial G}{\partial \mathbf{b}} = 0$. The quantity $\frac{\partial T}{\partial \mathbf{b}}$ can be computed from Equation (3.4) as:

$$\frac{\partial T}{\partial h} = \frac{hL(1 - T_\infty)x}{(1 + hL)^2} - \frac{(1.0 - T_\infty)\,x}{1 + hL} \tag{3.8}$$

$$\frac{\partial T}{\partial T_\infty} = \frac{hx}{1 + hL} \tag{3.9}$$

$$\frac{\partial T}{\partial L} = \frac{h^2(1 - T_\infty)\,x}{(1 + hL)^2} \tag{3.10}$$

There is no unique solution to this optimization problem, as there are an infinite number of **b** vectors which give $G(T(x,\mathbf{b}),\mathbf{b}) = 0$. For example, the following values for design variables all satisfy our objective:

$$\mathbf{b} = \left\{ \begin{array}{c} .1 \\ 0.0 \\ 15.0 \end{array} \right\} \tag{3.11}$$

$$\mathbf{b} = \left\{ \begin{array}{c} .15 \\ 0.0 \\ 10.0 \end{array} \right\} \tag{3.12}$$

$$\mathbf{b} = \left\{ \begin{array}{c} .3 \\ .15 \\ 8.0 \end{array} \right\} \tag{3.13}$$

$$ \tag{3.14}$$

3.3 Inverse Problems

Inverse problems are an interesting special case of design optimization problems. In an inverse problem, some aspect of the solution **u** is known *a priori* (*e.g.,* from experimental data) and the intent is to determine values for **b** such that the simulation results match the known results. Our previous example problem could be considered an inverse problem if experimental results (*e.g.,* measuring the temperature with a thermocouple) had determined that the temperature in the middle of the slab was 0.7 degrees Celsius. In this case we would like to determine the experimental conditions (h or T_∞) which lead to this result. As we demonstrated, this problem had multiple solutions because we were trying to determine the value of two parameters given a single data point.

In general, let us assume that experiments have determined the solution to a steady state heat transfer problem. If the experimental results are designated $\bar{\mathbf{u}}$, then the objective function for the problem could be defined as:

$$G(\mathbf{u}(\mathbf{b}), \mathbf{b}) = \sqrt{\frac{\sum_{i=1}^{N} (u_i - \bar{u}_i)^2}{N}} \tag{3.15}$$

where N is the dimension of the vectors $\mathbf{u}$ and $\bar{\mathbf{u}}$.

3.4 Software

As mentioned previously, there are many optimization packages available for solving unconstrained problems like the example in this chapter. In this section we present some statistics about solving our example problem using DOT [15].

Case	Constants	Design Variables	Initial Values	Optimal Design and Objective	Func. Evals	Grad. Evals
1	$L = 10$, $T_\infty = 0$	h	$h = .05$	$h = .15$ $G = 6.2 * 10^{-16}$	14	4
2	$L = 10$, $T_\infty = 0$	h	$h = 1$	$h = .15$ $G = 3.2 * 10^{-17}$	25	7
3	$L = 10$, $T_\infty = 0$	h	$h = -1$	$h = -1001$ $G = .04$	11	2
4	$L = 10$	T_∞, h	$h = 1$, $T_\infty = 1$	$h = 1$ $T_\infty = .34$ $G = 4.5 * 10^{-23}$	6	2
5	$L = 10$	T_∞, h	$h = 1$, $T_\infty = 0$	$h = .939$ $T_\infty = .336$ $G = 6.1 * 10^{-14}$	8	3
6	$L = 10$	T_∞, h	$h = 0$, $T_\infty = 0$	$h = 0.148$ $T_\infty = -.006$ $G = 5.5 * 10^{-19}$	6	2
7	$L = 10$, $T_\infty = 0$	h	$h = .05$	$h = .15$ $G = 1.7 * 10^{-16}$	20	0
8	$L = 10$, $T_\infty = 0$	h	$h = 1$	$h = .15$ $G = 3.4 * 10^{-25}$	34	0
9	$L = 10$, $T_\infty = 0$	h	$h = -1$	$h = -1001$ $G = .04$	13	0

Figure 3.2: Sample Problem Optimization Results Using DOT

Rather than use the analytic solution presented in this chapter, in this section we employ a finite element model of the example. Table 3.2 shows the results of several optimization runs. As we can see from cases 1 and 2, the finite element model gives the same results as the analytical solution. In addition, these two cases also demonstrate that using only h as a design variable will yield a unique solution for a valid initial guess. Case 3 shows how the optimization software can get lost if given a bad initial guess for a parameter. In this case, h is a convection coefficient and would never, in practice, be negative. In fact, if this were a constrained optimization problem, $h > 0$ would be a good candidate for a constraint. Next, cases 4 through 6 show that having two design variables instead of one can yield multiple optimal solutions. This is because there are two variables, h and T_∞, with which to achieve a single data point, $T(L/2) = .7$. Finally, cases 7-9 show that even for a problem with a single design variable the number of function evaluations necessary, in the absence of gradient information, goes up significantly.

Let us now extend this example by allowing the conductivity of the material to be temperature dependent (*i.e.,* $k(T) = 1.0 - .1T$), making the problem nonlinear. It will be necessary, then, to perform a computer simulation to solve for the temperature distribution in the slab. Table 3.3 shows the results using a nonlinear material property. As we can see, the optimal solution is different but the number of iterations used to solve the nonlinear problem has not changed.

Case	Constants	Design Variables	Initial Values	Optimal Design and Objective	Func. Evals	Grad. Evals
1	$L = 10$, $T_\infty = 0$	h	$h = .05$	$h = .1341$ $G = 6.99 * 10^{-17}$	14	4
2	$L = 10$, $T_\infty = 0$	h	$h = 1$	$h = .1341$ $G = 3.6 * 10^{-19}$	25	7
3	$L = 10$, $T_\infty = 0$	h	$h = -1$	$h = -1001$ $G = .045$	11	2
4	$L = 10$	T_∞, h	$h = 1$, $T_\infty = 1$	$h = 1$ $T_\infty = .34$ $G = 1.6 * 10^{-19}$	6	2
5	$L = 10$	T_∞, h	$h = 1$, $T_\infty = 0$	$h = .939$ $T_\infty = .336$ $G = 1.9 * 10^{-13}$	8	3
6	$L = 10$	T_∞, h	$h = 0$, $T_\infty = 0$	$h = 0.148$ $T_\infty = -.006$ $G = 3.1 * 10^{-21}$	6	2
7	$L = 10$, $T_\infty = 0$	h	$h = .05$	$h = .13406$ $G = 1.7 * 10^{-9}$	19	0
8	$L = 10$, $T_\infty = 0$	h	$h = 1$	$h = .1341$ $G = 4.8 * 10^{-16}$	34	0
9	$L = 10$, $T_\infty = 0$	h	$h = -1$	$h = -1001$ $G = .0045$	13	0

Figure 3.3: Nonlinear Problem Optimization Results Using DOT

Chapter 4

Software Design

Analysis of the optimization and simulation process has resulted in insight about how these two processes can be easily coordinated. We have developed ways of improving interoperability between different software packages by abstracting the behavior of the various components in the system. In addition, we have considered the operations carried out during simulation and formalized a framework for performing these tasks. In this chapter, the computer implementation of this formalization is described.

4.1 Object-Oriented Design

Before describing the approach taken when writing the software , we present some background information about the techniques used. Our intention when writing this software was to use object-oriented design techniques to create a system of reusable components called a *framework*.

In object-oriented programming, a program is constructed from a collection of interacting *objects*. Each object has a set of operations which can be performed on it and this set defines its *interface*. What operations the object can perform is determined by what *class* (or classes) it belongs to.

To demonstrate some of these principles, we include some simple C++ code fragments. Consider the class for geometric shapes shown in Figure 4.1. The permitted operations, or *methods*, of an object which belongs to this class are `Perimeter`, `Area` and `Describe`. The word `public` in this case indicates that the methods are part of its public interface, *i.e.,* those seen by all other objects. Finally, the `= 0` at the end of a method definition indicates that there is no "default" way of performing the operation.

```
class Shape
{
    public:
        virtual double Perimeter() = 0;
        virtual double Area() = 0;
        virtual void Describe() = 0;
};
```

Figure 4.1: Shape Class Definition

Many common shapes, such as circles, fit within this class. To differentiate them, we next define a *subclass*, *i.e.* a specialization, of `Shape` called `Circle`. The definition of `Circle` can be found in Figure 4.2. There are some very important differences between a `Circle` and a `Shape`. First, all objects of the `Circle` class have data, called `members`, associated with them. In this case, every `Circle` object has a `radius` associated with it. This `radius` is *private*, which means it is not visible to other objects, only to the `Circle`.

The next difference is that a `Circle` knows how to compute its `Perimeter` and `Area` as well as how to `Describe` itself. The definition of **how** an operation is performed is called its *implementation* and a `Circle` object has an implementation for every method in its interface. A class which defines an interface without completely implementing all methods, such as `Shape`, is called an *abstract* class. A class with a complete implementation, such as `Circle`, is called a *concrete* class.

Finally, it should be pointed out that for a `Circle` an additional method has been defined called a *constructor*. A constructor is a method called whenever an object of a class is created (or *instantiated*).

```
class Circle : public Shape
{
    private:
        double  radius;
    public:
        Circle(double r) { radius = r; }
        double Perimeter() { return 2.0*M_PI*radius; }
        double Area() { return M_PI*radius*radius; }
        void Describe() { cout << "Circle of radius " << radius << endl; }
};
```

Figure 4.2: Circle Class Definition

In a similar fashion, Figure 4.3 shows how we could define two addition concrete classes, `Rectangle` and `Square`. Note that `Rectangle` is a subclass of `Shape` and `Square` is a subclass of `Rectangle`. The first interesting thing to note about a `Square` is that it only requires one dimension for its constructor as opposed to `Rectangle`'s two. In addition, it can reuse the implementation of rectangle with the exception of its `Describe` method.

```
class Rectangle : public Shape
{
    protected:
        double  width, height;
    public:
        Rectangle(double w, double h) { width = w; height = h; }
        double Perimeter() { return 2.0*(width+height); }
        double Area() { return width*height; }
        void Describe() { cout<<width<<" by "<<height<<" Rectangle"<<endl; }
};

class Square : public Rectangle
{
    public:
        Square(double side) : Rectangle(side, side) { }
        void Describe() { cout<<"Square of size "<<width<<endl; }
};
```

Figure 4.3: Rectangle and Square Class Definitions

Figure 4.4 shows a simple program which makes use of the objects we have defined. The first line of our subroutine creates a `Circle` called `c` with a radius of 5. Next, we create a 2x3 `Rectangle` and a 7x7 `Square`. Then, we call the subroutine `Sum`, shown in Figure 4.5, with various combinations of the objects we have created. The subroutine `Sum` only knows that the objects being passed to it are `Shapes` and prints out information about the area of the two shapes. The output to this program can be found in Figure 4.6.

In summary, each object has both an interface, seen by other objects, and an implementation, known only to itself. Supporting code is written which uses the operations in the objects interface without knowledge of how those operations are implemented.

```
main()
{
    Circle      c(5.0);
    Rectangle   r(2.0,3.0);
    Square      s(7.0);

    Sum(c, r);
    Sum(s, r);
    Sum(c, s);
}
```

Figure 4.4: Main Program

```
Sum(Shape &s1, Shape &s2)
{
    cout << "When I add the area of a ";
    s1.Describe();
    cout << "with the area of a ";
    s2.Describe();
    cout << "I get " << s1.Area()+s2.Area() << endl << endl;
}
```

Figure 4.5: Sum Subroutine

```
When I add the area of a Circle of radius 5
with the area of a 2 by 3 Rectangle
I get 84.5398

When I add the area of a Square of size 7
with the area of a 2 by 3 Rectangle
I get 55

When I add the area of a Circle of radius 5
with the area of a Square of size 7
I get 127.54
```

Figure 4.6: Program Output

This chapter centers around two main objects, the simulation software and the optimization software. The simulation software provides "results" and the sensitivity of those results with the respect to the design variables, while the optimization package requires an "objective" to optimize. Between the simulation and optimization software is the user's specification of the objective, in terms of simulation results. Likewise, the sensitivity of the user's objective to design variables is computed in terms of the simulation results sensitivity.

We have developed a set of abstract classes in C++ to model this relationship. There are several benefits to using abstract classes for such calculations. First, they give us great flexibility in the types of objects which can be used in the simulation. In addition, each object typically builds on the sensitivity information provided by other lower level objects. For sensitivity calculations in particular, these abstract interfaces allow us to do some important consistency checking using the interface definition, independent of the object implementation. These benefits are important because they greatly reduce the difficulty of verifying sensitivity calculations.

4.2 Previous Work

Some very recent approaches to computer simulation have employed aspects of object-oriented design. The notion of object-orientation has its origins in simulation, and is considered to have started with a programming language called Simula [23]. The compilers of object oriented languages have recently become efficient enough that their use in simulation has been steadily increasing. The most notable language of this type is C++, a full description of which can be found in Stroustrup [24]. Another language which has great potential for simulation is Sather, which is described in the Sather language specification by Omohundro [25]. Discussion of object-oriented methods for large simulation projects can be found in Yoon [26]. Examples of simulation projects which used object oriented languages include Ptolemy [27] for system simulation, FEC [28] for finite element analysis, MOBILE [29] for mechatronic systems and ICSIM [30] for neural networks.

In addition there have been a few attempts to combine object oriented design with computer simulation software and sensitivity analysis. Calhoun and Lewandowski used an object oriented approach to model a variety of dynamic systems [31,32]. The approach uses operator overloading to track sensitivity information through the calculations. The major drawback of this approach is that it does not allow for possible algorithmic improvements because the formulation does not go beyond the system's differential equations.

There have also been several examples of object-oriented approaches to finite element analysis. Raphael [33] represents an early attempt at an object-oriented approach. A more advanced model was presented by Dubois-Pèlerin and Zimmerman in [34]. This initial work was done in Smalltalk, but for efficiency reasons the project was migrated to C++ [35]. Most of the initial attempts to represent finite element analysis in C++ tend to favor reuse through inheritance rather than compositional mechanisms.

An example of a generic simulation framework, with an emphasis on finite element analysis, can be found in Tiller [36]. While the approaches described by Calhoun and Lewandowski [31,32] are concerned only with determining sensitivity information, the Tiller's approach exploits this information for the benefit of optimization. This object-oriented approach relies more heavily on composition than previous efforts. This system achieves better reuse by allowing the user to compose objects from abstract types rather than exhaustively define all possible combinations of behavior through inheritance.

4.3 Optimization Abstractions

To solve optimization problems, it is necessary to relay information about the problem specification to the optimization software. Although different optimization packages have different ways of handling this information, the user typically supplies a procedure which provides values and gradient values for the objective and constraint functions. Because each package has a different scheme for relaying information, it is useful to develop a standard "interface" for communicating information.

Our goal is to develop an interface which is as independent of the simulation software package as possible. Developing such an interface involves determining the common features of different problem specifications. The ideal interface should be broad enough to be useful, but narrow enough so that all applicable optimization packages can be made to conform to it. We adopt the previously described notions about object-oriented design in this implementation.

Figure 4.7 shows the *interface* used for generalizing the optimization software. Each item in Figure 4.7 is a *method* which can be called for an object of the `Optimizer` class. The methods themselves define the interface which describes the minimum information that is required for an `Optimizer`. While objects may have additional methods, any reliance on that information will then be specialized to that software package and will represent and would be defined as a subclass. The `Initialize` method allows the optimization software to initialize any internal data. The `Iterate` and `Optimize` methods direct the optimization software to generate improved values for the design variables. Finally, the `Add-Update` method instructs the optimization software of other procedures that should be called whenever a new design is generated.

Optimization Software Interface

`Initialize`

Input: Problem Description
Description: Use information provided in the problem description (discussed later) to initialize arrays.
Output: None

`Iterate`

Input: Number of iterations to perform
Description: Perform some number of optimization iterations (design improvements)
Output: Number of iterations performed and whether the optimization is completed

`Optimize`

Input: Maximum number of iterations (optional)
Description: Iterate until finished or maximum iterations exceeded
Output: Whether the optimization is completed

`Add-Update`

Input: Code to be executed at the end of each design iteration
Description: Allows user provided code to be executed between each design iteration (*e.g.,* code for displaying optimization status)
Output: None

Figure 4.7: Optimization Package Interface

Well defined interfaces make software more modular. The interface described in Figure 4.7 makes it possible to replace one optimization package with another without changing the form of the problem specification. Each optimization package represents an *implementation* which conforms to the interface described in Figure 4.7. For example, we have created a C++

class which conforms to the interface in Figure 4.7 and which calls DOT [15] to perform the underlying optimization.

Another object in our optimization system is the one which describes the design variables. This object is characterized by the interface in Figure 4.8. For the example presented in Chapter 3, the `Vector` method would return h, T_∞ and L. In addition, the lower limit on h returned by the `GetLimits` methods would be zero.

Design Space Interface

`Initialize`
Input: Number of design variables and their initial values
Description: Initialize internal data
Output: None

`Vector`
Input: None
Description: Provides current design vector
Output: A vector of scalar values

`GetLimits`
Input: Design Variable Number
Description: Provides limits for each design variable
Output: Upper and Lower limits

`SetLimits`
Input: Design Variable Number, Upper limit, Lower limit
Description: Allows user to set limits for each design variable
Output: None

Figure 4.8: Design Space Interface

The most important part of the process is the abstraction for the problem specification. Figure 4.9 shows the information required for the problem specification. For the example presented in Chapter 3, the `Space` method would return a reference to the design specification mentioned previously, while the `Cost` function would compute the objective function using Equation 3.6. Figure 4.10 gives an overview of the relationship between the interfaces discussed so far.

Problem Specification Interface

`Initialize`
Input: Design Space Information
Description: Associates a design space with the problem specification
Output: None

`Space`
Input: None
Description: Provides a description of the design space, see Figure 4.8
Output: Design Space Information

`Cost`
Input: Gradient Info Flag
Description: Calculates cost function and gradient
Output: Cost function and gradient

`Constraints`
Input: Gradient Info Flag
Description: Calculates values for all constraint functions and their gradients if flag is set
Output: Constraint functions and gradients

Figure 4.9: Problem Specification Interface

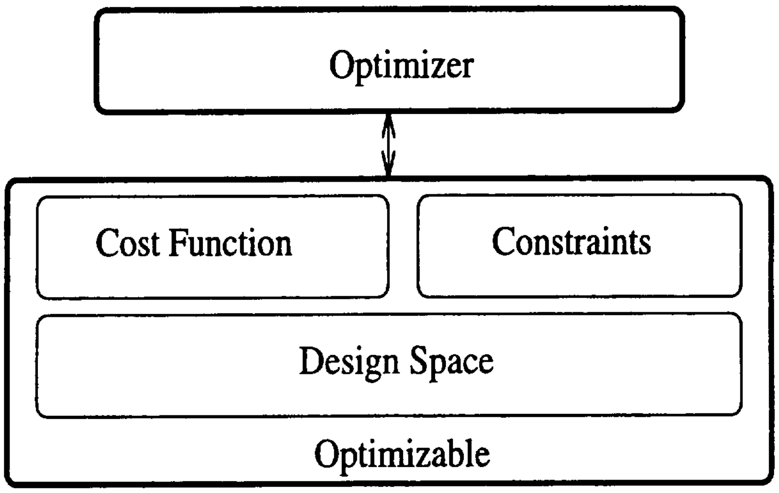


Figure 4.10: Important Concepts

4.4 Simulation Abstraction

Figure 4.11 shows an interface that should be compatible with nearly any kind of simulation software. The methods in this interface, such as `Solve-Initial` and `TimeStep`, allow other objects to control the simulation, stopping at critical times to make additional calculations (*e.g.,* calculating the objective function) based on the simulation results. Note that one of the functions of the simulation software is to return a description of the simulation results at each time step. Figure 4.12 shows the interface for simulation results. The need for a generalization of results comes about because simulators can produce a variety of *secondary variables.* For example a simulator may provide both displacements and stresses, and each is considered a "result" in our model.

Simulation Software Interface

`Initialize`

Input: Design information

Description: Information about the design is saved so the simulator can request the values of design variables during subsequent simulations

Output: None

`Reset`

Input: None

Description: Resets all internal variables. Indicates a new simulation is to be started.

Output: None

`Time`

Input: None

Description: Used to determine the current time in the computational domain

Output: Simulation Time

`Solve-Initial`

Input: Sensitivity flag

Description: This function determines the initial conditions (or steady-state solution), and the sensitivity of the initial conditions if the sensitivity flag is set

Output: Whether the simulation completed successfully

`TimeStep`

Input: Sensitivity flag, Target time

Description: Instructs the simulation software to perform a time step, but not to step past the target time. Sensitivities are calculated if the sensitivity flag is set

Output: Whether the simulation is complete

`Result`

Input: Result Index

Description: Returns a description of the simulation results for the current time step.

Output: Results description

Figure 4.11: Simulation Software Interface

Simulation Results Interface

`NumNodes`

Input: None
Description: Returns number of nodes in system for which the result applies. Each node may have multiple degrees of freedom
Output: Number of nodes (integer)

`NumEq`

Input: None
Description: Number of equations (unknowns) in the system
Output: Number of equations (integer)

`NumDOFS`

Input: Node number
Description: Indicates how many degrees of freedom there are associated with this node
Output: Number of degrees of freedom (integer)

`EqNum`

Input: Node number and Degree of freedom type
Description: Given a specific node and degree of freedom, returns the corresponding equation (unknown) number.
Output: Equation number (integer)

`GetSol`

Input: Equation number
Description: Returns the scalar value of the unknown associated with a particular equation
Output: Scalar

`GetDUDB`

Input: Equation number, Design variable
Description: Returns the sensitivity of the unknown associated with a particular equation with respect to a design variable
Output: Vector

Figure 4.12: Simulation Results Interface

4.5 Integrating Simulation and Optimization

The benefit of having designed simulation and optimization interfaces is that the integrating code (the "glue" between the two) is the same for all problems. There is no need to be aware of implementation details, nor is knowledge of the particular simulation or optimization package required. This section describes the objective interface as well as integrating code.

Figure 4.13 shows the interface for objective specification. The `TimeStep` method exists so the objective can be notified when each time step is complete. In addition using the `NextTime` method, the objective is able to provide specific times it would like simulation results for. This information is then returned to the object controlling the simulation and this object makes sure that the objective object's `TimeStep` method is called at that time. The logic for this integrating code, written in procedural style, is shown in Figure 4.14. The object which interfaces with both the simulation package and the objective is called *SimOpt*.

Figure 4.15 shows the various objects involved. In addition, Figure 4.16 shows the specifics steps that take place for solving this problem in an object oriented style. These figures demonstrate how the various interfaces shown in Figures 4.7-4.13 interact.

Objective Interface

`Initialize`

Input: A simulator description, and design vector
Description: Stores the simulator and design description for later use and initializes internal variables
Output: None

`TimeStep`

Input: Sensitivity flag
Description: Tells the objective that the simulator has just completed a time step, calculate contribution to objective if necessary. Also, calculate gradient if sensitivity flag is set
Output: None

`GetStatus`

Input: None
Description: Returns whether the objective function calculation has been completed
Output: None

`NextTime`

Input: None
Description: Indicates the next target time (in the computational domain) when the simulator will need to stop (for the purposes of calculating the objective).
Output: Time

`NumCon`

Input: None
Description: Indicates the number of constraints associated with this objective
Output: Number of constraints (integer)

`Results`

Input: None
Description: The objective function value (for example) is typically accumulated from one time step to the next. This function returns the accumulated result.
Output: Value of objective (gradient if applicable) and constraints (gradients if applicable)

Figure 4.13: Objective Interface

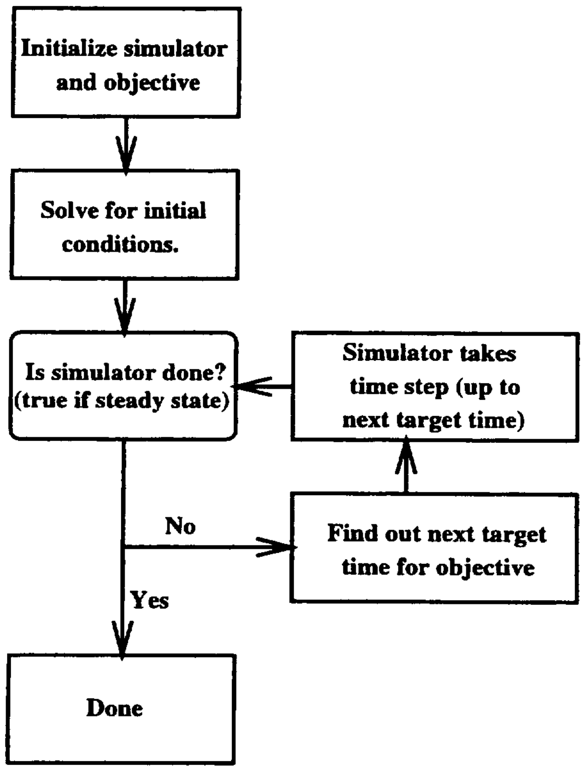

Initialize simulator and objective
Solve for initial conditions.
Is simulator done? (true if steady state)
Simulator takes time step (up to next target time)
No
Find out next target time for objective
Yes
Done

Figure 4.14: Integrating Logic

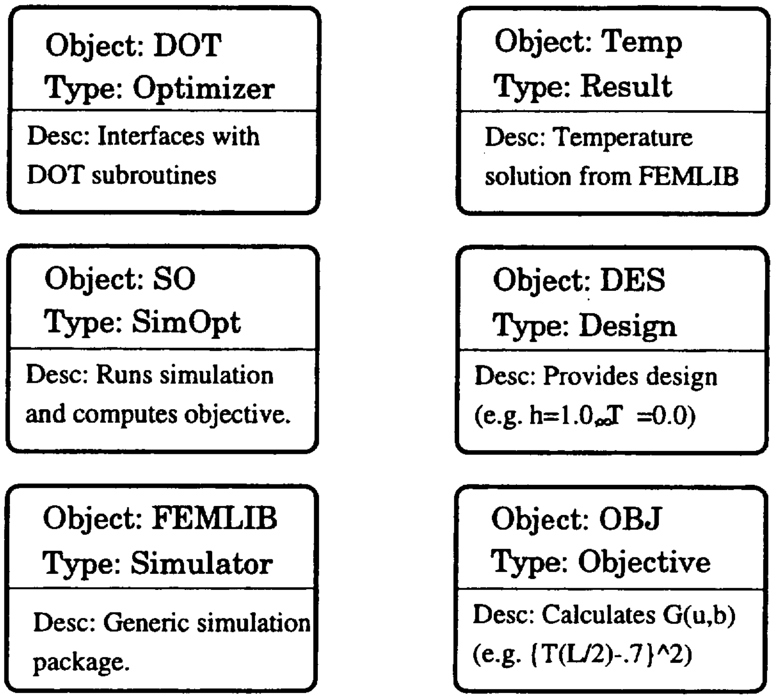

Object: DOT
Type: Optimizer
Desc: Interfaces with DOT subroutines
Object: Temp
Type: Result
Desc: Temperature solution from FEMLIB
Object: SO
Type: SimOpt
Desc: Runs simulation and computes objective.
Object: DES
Type: Design
Desc: Provides design (e.g. h=1.0,T∞=0.0)
Object: FEMLIB
Type: Simulator
Desc: Generic simulation package.
Object: OBJ
Type: Objective
Desc: Calculates G(u,b) (e.g. {T(L/2)-.7}^2)

Figure 4.15: Sample Objects

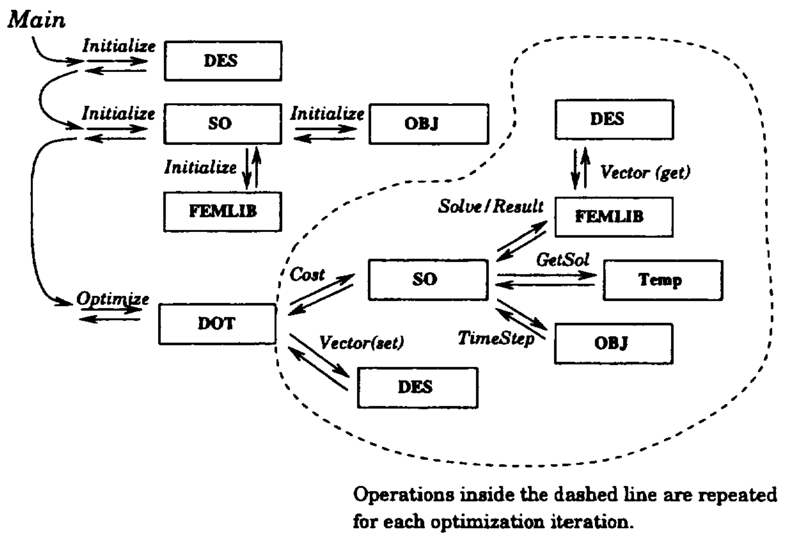


Figure 4.16: Sample Flow

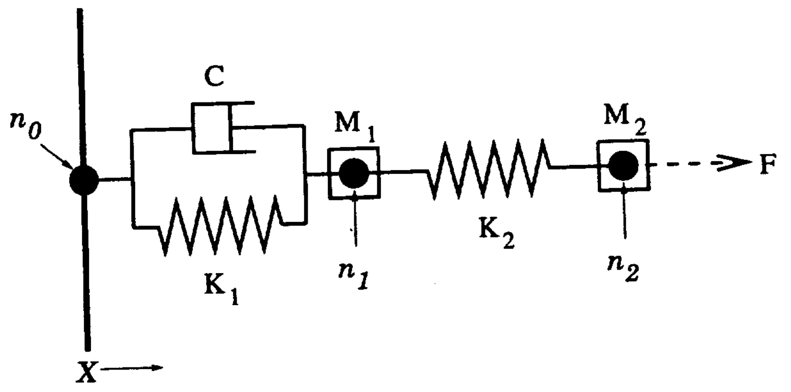


Figure 4.17: Mechanical Example

4.7.1 Nodes

A node is defined as a collection of degrees of freedom. In the example shown in Figure 4.17, the degrees of freedom would be the displacements, in the x direction, of nodes n_0, n_1 and n_2.

The interface for node objects is shown in Figure 4.18. Equation numbers for a node's degrees of freedom can be determined by using the `dof` method. For finite element problems, a node may have an associated spatial location. The spatial location of the node is provided by the `GetPos` method, and its dependence on design variables is expressed by the `Par` method. It is important to note that these spatial locations **do not** represent degrees of freedom. Some problems, like our example in Figure 4.17 have displacement degrees of freedom, but these would be associated with values returned by the `dof` method and not the `GetPos` or `Par`.

Node Interface

`Initialize`

Input: Nodal Coordinates
Description: Initializes the spatial positions of nodes
Output: None

`Par`

Input: A design variable
Description: Calculates the sensitivity of the node's spatial position, $\{x, y, z\}$, with respect to a particular design variable
Output: $\frac{\partial x}{\partial b}, \frac{\partial y}{\partial b}, \frac{\partial z}{\partial b}$ and a flag indicating whether all derivatives are zero

`dof`

Input: A degree of freedom type
Description: Returns the equation associated with the particular degree of freedom type for this node
Output: Equation number (integer)

`GetPos`

Input: None
Description: Returns the spatial coordinates of the node
Output: Nodal coordinates (three reals)

Figure 4.18: Node Interface

4.7.2 Components

A description of the interface for component objects can be found in Figure 4.19. The quantity returned by the `Mass` method has two derivative quantities, the tangent mass contribution returned by the `TanMass` method and the explicit design sensitivity returned by the `ParMass` method. This same holds true for the `Damp`, `Stiff` and `Force` methods.

Element objects contribute to the constraints of the system. FEMLIB constructs constraints for each degree of freedom according to the following general equation:

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{K}\mathbf{x} = \mathbf{f} \tag{4.2}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{C}$ is the damping matrix, $\mathbf{K}$ is the stiffness matrix and $\mathbf{f}$ is the force vector. Equation (4.2) represents a second-order system of differential equations where each individual equation is a constraint for a specific degree of freedom in the system.

At each time step, we form a system of equations typically derived from a conservation principle. Each component makes its contribution, in turn, to this system. Initially the system of ordinary differential equations (ODEs) contains no coefficients, *i.e.*,

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{4.3}$$

Our example in Figure 4.17 contains two `Spring` objects, a `Dashpot` object, two `PointMass` objects and a `PointForce` object. The system of ODEs is constructed by assuming the sum

Component Interface

`Mass`

Input: Domain object, transient context

Description: This function computes the mass matrix ($\mathbf{M}$) contributions for various degrees of freedom. The transient context allows quantities to be calculated in the context of previous time steps.

Output: Mass matrix contribution

`ParMass`

Input: Domain object, design variable, transient context

Description: Same as the Mass function except the quantity calculated is $\frac{\partial \mathbf{M}}{\partial b_k}$

Output: $\frac{\partial \mathbf{M}}{\partial b_k}$ matrix contribution

`TanMass`

Input: Domain object, design variable, derivative specification, transient context

Description: Same as the Mass function except the quantity calculated is $\frac{\partial M_{ij}}{\partial v_k} u_j$ where $\mathbf{v}$ and $\mathbf{u}$ are provided in the derivative specification.

Output: $\frac{\partial M_{ij}}{\partial v_k} u_j$ matrix contribution

the functions **Damp**, **ParDamp**, **TanDamp**, **Stiff**, **ParStiff** and **TanStiff** are similar to the ones listed above and will not be described

`Force`

Input: Domain object, transient context

Description: Computes the force vector

Output: Force vector contribution

`ParForce`

Input: Domain object, design variable, transient context

Description: Computes the $\frac{\partial \mathbf{f}}{\partial b_k}$ vector

Output: $\frac{\partial \mathbf{f}}{\partial b_k}$ vector contribution

`TanForce`

Input: Domain object, design variable, derivative specification, transient context

Description: Computes the tangent force vector contribution $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$

Output: $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$ matrix contribution

Figure 4.19: Component Interface

of the forces at each node is zero. Each spring affects the x displacement (degree of freedom) at two nodes and makes a contribution to the equations associated with those degrees of freedom. The system of ODEs is formed by collecting contributions from each component by calling the **Mass**, **Damp**, **Stiff** and **Force** method of each component.

After adding the contributions of the springs, due to the `Stiff` method of the `Spring` class, the system of ODEs becomes:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} + \begin{bmatrix} K_1 & -K_1 & 0 \\ -K_1 & K_1 + K_2 & -K_2 \\ 0 & -K_2 & K_2 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{4.4}$$

Next, we consider the dashpot whose contribution, due to the **Damp** method of the **Dashpot** class, appears in the **C** matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} C & -C & 0 \\ -C & C & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} + \begin{bmatrix} K_1 & -K_1 & 0 \\ -K_1 & K_1 + K_2 & -K_2 \\ 0 & -K_2 & K_2 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{4.5}$$

Then, we consider the point masses, produced in the **Mass** method of the `PointMass` class, and our system of ODEs is complete:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & M_1 & 0 \\ 0 & 0 & M_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} C & -C & 0 \\ -C & C & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} + \begin{bmatrix} K_1 & -K_1 & 0 \\ -K_1 & K_1 + K_2 & -K_2 \\ 0 & -K_2 & K_2 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{4.6}$$

Lastly, we add in our load F, due to the `Force` method of the `PointForce` class, which gives the final form for our system of ODEs:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & M_1 & 0 \\ 0 & 0 & M_2 \end{bmatrix} \begin{Bmatrix} \ddot{x}_0 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix} + \begin{bmatrix} C & -C & 0 \\ -C & C & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} + \begin{bmatrix} K_1 & -K_1 & 0 \\ -K_1 & K_1 + K_2 & -K_2 \\ 0 & -K_2 & K_2 \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ F \end{Bmatrix} \tag{4.7}$$

Although the notion of a component was originally set up to deal with finite elements, a variety of systems, like the one shown above, can be modeled with this generalized approach (*e.g.*, electrical circuits, mechanisms). In fact, one possible application of this generalized formulation is for simulating mixed systems containing drastically different objects. For example, it would be possible to simultaneously simulate an electrical circuit and the heat transfer due to resistive elements in the circuit. This would allow a single package to simultaneous consider several aspects of a complex system.

4.7.3 Properties

Properties represent scalar values which help to describe the behavior of components. In our example, the properties in the system would be K_1, K_2, C, M_1, M_2 and F. Requirements for computing sensitivity with respect to the design and solution are mandated by components having the same requirement. For example, since a `PointMass` object provides methods `ParMass` and `TanMass` the component must have a way to determine $\frac{\partial M_1}{\partial b_j}$ and $\frac{\partial M_1}{\partial \mathbf{u}}$. The property interface is shown in Figure 4.20. We see a method `Eval` which returns a computed quantity, *e.g.,* a springs stiffness, and two other methods `TanEval` and `ParEval` that return that same quantity differentiated by the solution and design variables, respectively. By allowing properties to have complex relationships to design and solution variables we gain flexibility in being able to define that relationship in each problem as we see fit. This gives us greater expressiveness when posing our simulation and optimization problems, and it allows us to substitute a non-linear spring into our system by defining a subclass of property without the need to modify any existing code.

In the simplest case, properties take on the value of a design variable. However properties may also be related to design variables through more complex relationships. For example, the continuous ambient temperature distribution visible to a surface of convection elements may be a spline characterized by design variables. For such cases, the derivative of the ambient temperature with respect to a particular design variable must be computed in order to compute $\frac{\partial \mathbf{R}}{\partial b_j}$. In general, property objects compute their scalar values and sensitivities of these values as functions of the solution and design.

Property Interface

`IsConst`

Input: None

Description: Tells whether the property is a constant value. This allows for many optimizations to be made by the calling object. However, even if the property is constant, then tangent and partial derivative methods must still return arrays initialized to zero because the caller is not obligated to consider possible optimizations.

Output: Returns a flag indicating whether it is constant.

`IsFunc`

Input: Degree of freedom, transient context

Description: Tells whether a property is a function of a particular degree of freedom. This method is also used to allow certain optimization but is subject to the same restrictions as above.

Output: Returns a flag indicating whether it is a function of the degree of freedom

`Eval`

Input: Transient context, domain, set of nodes, Gauss points

Description: Calculates the property at each Gauss point

Output: Returns value at each Gauss point

`TanEval`

Input: Transient context, degree of freedom, domain, set of nodes, Gauss points

Description: Calculates the derivative a property with respect to a particular degree of freedom

Output: Returns derivative value at each Gauss point

`ParEval`

Input: Transient context, design variable, domain, set of nodes, Gauss points

Description: Calculates the derivative a property with respect to a particular design variable

Output: Returns derivative value at each Gauss point

Figure 4.20: Property Interface

Several common property types have already been implemented. The most obvious is a property which is constant. In addition, a property which is equal to a particular design variable, called a design property, has been implemented. One of the more advanced properties is one which is constructed piecewise in time, by data points which are themselves material properties.

4.7.4 Domains

The computational domain for simulation, or simply "domain", is also generalized for the purposes of flexibility and reuse. The domain is a collection degrees of freedom and constraints. Unlike the other objects mentioned, there are no computations performed by the domain objects. Figure 4.21 shows how generic information such as the number of equations (`NumEqs`), and number of nodes (`NumNodes`) can be accessed. In addition, the mapping from nodes and degrees of freedom to actual equations is also accessible using the `EqNum` method. Finally, the bookkeeping functions `AddNode` and `AddElem` are also present. The assembling of equations, as seen in Equations 4.3-4.7, is performed by considering, in turn, each component in the domain.

Domain Interface

`NumNodes`
Input: None
Description: Number of nodes in the domain
Output: Number of nodes (integer)

`NumEqs`
Input: None
Description: Number of equations (unknown) in the domain
Output: Number of equations (integer)

`NumNodalDOFS`
Input: Node number
Description: Number of degrees of freedom associated with a particular node.
Output: Number of degrees of freedom (integer)

`NodalDOF`
Input: Node number and index
Description: Returns the degree of freedom type for the $index^{th}$ degree of freedom associated with a particular node.
Output: Degree of freedom type (integer)

`EqNum`
Input: Node number and degree of freedom type
Description: Returns the equation number for a particular node and degree of freedom
Output: Equation number (integer)

`AddNode`
Input: Node description
Description: Instructs domain to keep track of this node
Output: None

`AddElem`
Input: Element description
Description: Instructs domain to keep track of this element
Output: None

`NodeSet`
Input: None
Description: Returns the collection of nodes in the domain
Output: Set of nodes

`ElemSet`
Input: None
Description: Returns the collection of elements in the domain
Output: Set of elements

Figure 4.21: Domain Interface

4.7.5 Integrators

The system of equations described in Section 4.7.6 may arise from a finite difference time integration scheme such as forward or backward Euler applied to a nonlinear system of second order ordinary differential equation of the general form:

$$\mathbf{M}(\mathbf{u}(\mathbf{b}), \mathbf{b})\ddot{\mathbf{u}} + \mathbf{C}(\mathbf{u}(\mathbf{b}), \mathbf{b})\dot{\mathbf{u}} + \mathbf{K}(\mathbf{u}(\mathbf{b}), \mathbf{b})\mathbf{u} = \mathbf{f}(\mathbf{u}(\mathbf{b}), \mathbf{b}) \tag{4.8}$$

The integrator converts this equation into a system of nonlinear algebraic equations for the solution at the next time step, *e.g.,*

$$\mathbf{A}\mathbf{u}^{n+1} = \mathbf{p} \tag{4.9}$$

This system of nonlinear algebraic system of equations can easily be made to conform to the interface described in Figure 4.22.

There are a number of integration schemes possible (see Sections 1.4 and 1.5). For a given integration scheme, we want to be able to compute not only the solution, but its sensitivities. The integrator objects act as "middlemen" between the underlying nonlinear second-order differential equations, *e.g.,* Equation 4.7, and the desired system of nonlinear algebraic equations. They must perform not only a transformation, which forms the residual vector given the underlying second-order system, but they must also transform sensitivity information. In other words, whatever transformation the integrator performs to calculate the residual, it must also perform a similar transformation for sensitivity information (tangent matrix and pseudo-load) as well.

4.7.6 Residual Systems

Recall our general approach to solving systems of nonlinear equations from Chapters 1 and 3. FEMLIB assumes that general nonlinear residual and sensitivity equations may be written in the following form (from Section 2.3):

$$\frac{\partial \mathbf{R}}{\partial \mathbf{u}}(\mathbf{u}(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b})\Delta \mathbf{u}_i = -\mathbf{R}(\mathbf{u}_i(\mathbf{b}), \mathbf{v}(\mathbf{b}), \mathbf{b}) \tag{4.10}$$

$$\frac{d\mathbf{u}}{db_k} = -\left[\frac{\partial \mathbf{R}}{\partial \mathbf{u}}\right]^{-1} \frac{\partial \mathbf{R}}{\partial b_k} \tag{4.11}$$

These equations can be used to solve any nonlinear system for which the residual ($\mathbf{R}$) and tangent matrix ($\frac{\partial \mathbf{R}}{\partial \mathbf{u}}$) can be computed. In addition, if the pseudo-load ($-\frac{\partial \mathbf{R}}{\partial b_k}$) is available sensitivity information may also be computed. Figure 4.22 shows the interface for such a system (boundary conditions have been omitted for brevity). The Newton-Raphson algorithm used in FEMLIB uses the interface in Figure 4.22, and is independent of the implementation details of the particular system of equations being solved.

When performing a Newton-Raphson iteration, the residual and tangent matrix are requested using the `Residual` and `Tangent` methods. Then the solution can be retrieved and replaced using the `Get` and `Set` methods. Finally, the sensitivities can be computed using the pseudo-load, $-\frac{\partial \mathbf{R}}{\partial b_k}$, provided by `ParRes`.

Residual System Interface

`Initialize`

Input: Description of solution
Description: Memory to be used for the solution vector is allocated to the residual system
Output: None

`Residual`

Input: None
Description: Compute residual given current value of the solution
Output: The residual vector

`ParRes`

Input: Design variable b_j
Description: Calculate $\frac{\partial \mathbf{R}}{\partial b_j}$
Output: A vector representing $\frac{\partial \mathbf{R}}{\partial b_j}$

`Local`

Input: None
Description: Return a description of the current solution
Output: Solution information, (*e.g.*, Number of equations, current solution, sensitivity information)

`Num`

Input: None
Description: Return number of unknowns for the system
Output: Number of unknowns (integer)

`Get`

Input: Unknown number
Description: Determine unknown's current value
Output: Unknown value (real)

`Set`

Input: Unknown number and value
Description: Set unknowns current value
Output: None

`Tangent`

Input: None
Description: Compute and return tangent matrix
Output: Tangent matrix

Figure 4.22: Residual System Interface

4.7.7 Transformations

The objects presented in the next two sections are useful in the context of finite element formulations. Transformations are designed to be used for Gaussian integration where an integral is simplified by mapping it onto a regular geometry. The transformation is generalized, and the interface that results can be seen in Figure 4.23.

Abstracting transformations allows the underlying mathematical models present in integrals to be independent of the transformations (*i.e.*, one-, two-, three-dimensional and axisymmetric formulations). Note that the interface presented in Figure 4.23 allows transformations to be a function of design, but not of the solution.

The `At` method in Figure 4.23 directs the Transformation object to compute, at a specified point on the "local element", the Jacobian, its inverse and its determinant. Likewise, the `ParAt` method directs the object to compute the design sensitivity of the Jacobian, its inverse and its determinant.

Transformation Interface

`At`

Input: A point in space
Description: Computes some internal values for representing a transformation at a particular point
Output: None

`ParAt`

Input: A point in space, design variable
Description: Compute some internal values for representing the sensitivity of the transformation at a particular point
Output: Returns a flag indicating whether the transformation is a function of the design variable

`Pos`

Input: None
Description: Exports information about the point of last 'At' or 'ParAt' call
Output: A point in space

`NumDim`

Input: None
Description: Provides knowledge of how many dimensionality of the transformation
Output: Number of dimensions (integer)

`J, Ji, ParJ, ParJi`

Input: two axes x_i and x_j
Description: These methods return the entry in the Jacobian ($\mathbf{J}$), inverse Jacobian ($\mathbf{J}^{-1}$), partial Jacobian ($\frac{\partial \mathbf{J}}{\partial b}$) and partial inverse Jacobian ($\frac{\partial \mathbf{J}^{-1}}{\partial b}$) matrices (respectively) at the point of the last 'At' or 'ParAt' call. Which entry is determined by the two axes arguments.
Output: Scalar

`detJ, PardetJ`

Input: None
Description: These two methods return the values of the Jacobian determinant ($|\mathbf{J}|$) and partial Jacobian determinant ($\frac{\partial |\mathbf{J}|}{\partial b}$) respectively
Output: Scalar

Figure 4.23: Transformation Interface

4.7.8 Basis Functions

Basis Function Interface

`NumNodes`
Input: None
Description: Number of basis functions provided. In a finite element formulation, this is equivalent to the number of nodes in an element
Output: Number of basis functions (integer)

`NumDim`
Input: None
Description: Number of arguments each basis function takes. In a finite element formulation, this is equivalent to the number of dimensions the element spans
Output: Number of arguments (integer)

`Eval`
Input: Spatial coordinate
Description: Computes values for all basis functions at a particular point
Output: Values of basis functions and their derivatives with respect to their arguments

Figure 4.24: Basis Function Interface

Another topic, related specifically to finite element formulations, is basis functions. When using a finite element formulation, basis functions are used to approximate the solution between nodes. The choice of basis functions can influence the accuracy of the results since the basis functions are used to interpolate degrees of freedom over the elements. The interface for basis functions is shown in Figure 4.24. As in transformations and properties, many basis functions are particularly common. Linear basis functions for one, two and three dimensions have been implemented.

4.7.9 Testers

As noted in Section 4.6, the most difficult problem in debugging sensitivity calculations is locating the source of the error. The various objects described in Sections 4.7.1 through 4.7.8 calculate a quantity, and the derivative of that quantity with respect to the solution and design vectors. It is possible to verify the consistency of these results by performing a finite difference calculation on the computed quantity.

For example, if we have a given property describing a non-linear spring whose value is given by:

$$k = K_1 \Delta x + K_2 \Delta x^2 \tag{4.12}$$

then we can compute the sensitivities in one of two ways. The first way would be to call the `ParEval` or `TanEval` methods. Thus, if we wanted to evaluate $\frac{\partial k}{\partial b_j}$ the non-linear spring property would compute

$$\frac{\partial k}{\partial b_j} = \frac{\partial K_1}{\partial b_j} \Delta x + \frac{\partial K_2}{\partial b_j} \Delta x^2 \tag{4.13}$$

where K_1 and K_2 may be given in turn by other property objects. The second way we could compute $\frac{\partial k}{\partial b_j}$ would be to use a finite different approximation, *e.g.*,

$$\frac{\partial k}{\partial b_j} = \frac{k(\mathbf{b}) - k(\mathbf{b} + \Delta b_j)}{\Delta b_j} \tag{4.14}$$

The former technique requires that the code for computing Equation 4.13 be correct. The latter approach, although expensive, will be give accurate results if Δb_j is small enough to avoid truncation error and largest enough to avoid round-off error. By computing $\frac{\partial k}{\partial b_j}$ both ways we can make sure that the former technique has been implemented correctly. This approach can be taken for nearly every object which returns a computed value, both tensor and scalar.

The objects which do this testing are referred to as *testers*. For many of the classes described there is another class of objects meant to test for this "derivative consistency". Since the interfaces to the classes described in Sections 4.7.1 through 4.7.9 are the same for all instances, the testers operate independently of the implementation details of any given instance. In other words, the ability to test a given a property is independent of whether it represents, for example, a piece-wise linear function or a cubic spline.

4.7.10 Summary

The "big picture" of FEMLIB, is that separating out the various concepts permits the different aspects of the problem to be independent. For example, given a working model of the example shown in Figure 4.17 we can substitute any variety of non-linear springs into the system without having to change any of the existing code. The goal is to reuse as much of the supporting code (*e.g.,* integration schemes, linear solvers) while retaining the ability to add new components as the need arises.

4.8 User Interfacing

One way to construct a problem description is to write a C++ program by creating and connecting objects from FEMLIB. This becomes tedious because every change in the problem statement requires recompilation of the C++ code describing the problem. For this reason, TCL [40] has been used to create an interpreted front end for problem descriptions. TCL scripts can be written to resemble conventional input files for FEM packages.

However, there are two important differences between the capabilities of the TCL interpreter and a typical FEM package. The first difference is that a TCL interpreter can operate in either batch mode, where a group of operations are performed to completion, or in an interactive mode where operations are performed interactively. The second difference is that while TCL controls the simulation, the simulation is only one aspect of the TCL front end. Typical interactive tasks within the TCL front end might examine sensitivities at individual time steps, restart the simulation with a different design, optimize for a particular objective, *etc.*

Figure 4.25 shows an example of an input script for the TCL front end. This script represents the problem solved in Chapter 3, where the conductivity was temperature dependent. The important thing to notice in Figure 4.25 is how the simulation parameters, the simulation, and the objective are all defined.

For example, part 1, the declares the design variables. An "=" preceding the value of a design variable indicates it should be considered constant (*i.e.*, not modified during an optimization). Part 2 contains statements about what algorithms should be used for solving the nonlinear system of equations, linear systems of equations and sensitivities.

Starting in part 3 we start to see statements specific to the finite element method. Part 3 itself declares how the degrees of freedom will be interpolated over the finite elements. For example, the statement

```
field T -dof temp -bf lbf42
```

```
# Part 1: Declare design variables and name them
parameter T_infty =0.0
parameter H .05
# Part 2: Define what algorithms will be used
senses ilu -check 1 -tol 1e-5
integration steady -tol 1e-8 -log CS.out
solver nr -lsolve slbc -iter 75

# Part 3: Define fields
field T -dof temp -bf lbf42
field Ts -dof temp -bf lbf21
fieldprop air -conv -temp1 Ts -temp2 T_infty -h H
fieldtype q
# Part 4: Additional properties
property K -table -field T -data [list {-10.0 2.0} {10 0.0}]
# Part 5: Describe governing equations
model m1 cond -dof temp -prop K -ngp {2 2}
model m2 flux -dof temp -flux q -ngp {2}
model m3 flux -dof temp -flux air -ngp {2}
# Part 6: Define transformations
trans cart -type cart -bf lbf42
trans carts -type cart -bf lbf21

# Part 7: Instantiate nodes
node n1 0.0 0.0 0.0
...
node n121 10.0 10.0 0.0
# Part 8: Instantiate elements
element -name quad1 -trans cart -nodes {n1 n2 n13 n12} \
    -field "T {n1 n2 n13 n12}" -model m1
...
element -name surf1 -trans carts -nodes {n112 n111} \
    -field "Ts {n112 n111}" -model m3

# Part 9: Boundary conditions
essent -node n1 -dof temp -val 1.0
...
essent -node n11 -dof temp -val 1.0
# Part 10: Simulation objective
solgoal -node n61 -dof temp -val .7
```

Figure 4.25: Simple TCL Script

defines a field T which interpolates temperature (**temp**) using a **L**inear **B**asis **F**unction for **4** noded elements in **2** dimensions. Of course, use of a given basis function must be consistent with the number of nodes in the element and the dimension of the problem. Part 4 shows one way a nonlinear material property can be defined. Part 5 shows how we declare what "governing equations" we will use. For example, the statement

```
model m1 cond -dof temp -prop K -ngp {2 2}
```

defines a new "model" called m1 which uses the governing equation for heat conduction with K (from part 4) as the thermal conductivity property employing 2x2 Gaussian integration. Next, in part 6, we declare the transformation used to transform the governing equations from the local to the global elements. Up until now, we have not defined a specific finite element problem, instead we have laid the building blocks on which to construct our simulation.

In part 7 we begin to define a *specific* problem by declaring the spatial locations of our nodes. Part 8 is a particularly good example of how we use a "compositional" style to piece the specific behavior we want. For example, the statement

```
element -name quad1 -trans cart -nodes {n1 n2 n13 n12} \
    -field "T {n1 n2 n13 n12}" -model m1
```

creates an element quad1 which uses the transformation cart. In addition quad1 is attached to nodes n1 n2 n13 and n12. The nodes appear twice because the -nodes directive indicates nodes used for spatial interpolation while the -field directive indicates nodes used to interpolate the field T and these two transformations are independent of each other. Finally, the -model directive attaches a particular governing equation to the element.

The remainder of our example is not specific to the finite element method. Part 9 shows how essential boundary conditions are imposed. Finally, part 10 gives an example of how optimization objectives can be declared along side the simulation description.

Chapter 5

Radiation Modeling

5.1 View Factors

We consider non-participating radiative heat transfer, where many surfaces radiate to each other. In addition, we assume that each surface absorbs all incident radiation. For each pair of surfaces A and B, the *view factors*, $F_{A\to B}$ and $F_{B\to A}$ measure the fraction of surface B visible to surface A, and the fraction of surface A visible to surface B, respectively. In this chapter we discuss how these view factors appear in the calculation of radiative heat transfer, and describe new methods for calculating them for axisymmetric cylinders and annular disks.

To accurately model many solidification processes it is necessary to model the radiative heat transfer between different components. This requires the calculation of view factors between the different surfaces participating in radiative heat transfer. These view factors can be calculated analytically for simple geometries [22] or numerically [41, 42]. Regardless of how they are computed, for large problems the number of view factors involved results in a significant amount of time being spent in calculating the view factors.

The total heat radiated from surface A to B may be expressed (*cf.* Lienhard [22]) as:

$$Q_{A\to B} = \sigma \int_A \int_B \left(T_A^4 - T_B^4\right) \frac{\cos\beta_A \cos\beta_B}{\pi s^2} dA\, dB \tag{5.1}$$

where T is the temperature of the surfaces, β represents the angle between the surface normal and a line connecting the two surfaces, and s is the distance between the two surfaces. We assume that the temperature distribution is constant over each surface, which reduces Equation (5.1) to:

$$\begin{aligned} Q_{A\to B} &= \sigma\left(T_A^4 - T_B^4\right) \int_A \int_B \frac{\cos\beta_A \cos\beta_B}{\pi s^2} dA\, dB && (5.2) \\ &= \sigma(T_A^4 - T_B^4) F_{A\to B} A_A && (5.3) \end{aligned}$$

where A_A is the area of surface A. The "geometric" relationship between surfaces A and B is represented by $F_{A\to B}$, the *view factor* from surface A to B. We recognize from Equation (5.2) that:

$$F_{A\to B} = \frac{1}{A_A} \int_A \int_B \frac{\cos\beta_A \cos\beta_B}{\pi s^2} dA\, dB \tag{5.4}$$

There are several useful properties associated with view factors [22]. The simplest, called the *enclosure property*, is:

$$\sum_\alpha F_{A\to\alpha} = 1 \quad \forall A \tag{5.5}$$

where α represents, in turn, every surface visible to A. Another useful property, called the *reciprocity property*, comes from the fact that the heat transfered from surface A to surface B must balance the heat transfered from surface B to surface A. The reciprocity property is stated mathematically as:

$$A_A F_{A\to B} = A_B F_{B\to A} \tag{5.6}$$

This property is useful because it reduces the number of view factors which must be calculated via Equation (5.4). Finally, if surfaces B and C are disjoint surfaces, then we may use the *addition property*:

$$F_{A\to(B+C)} = F_{A\to B} + F_{A\to C} \tag{5.7}$$

5.2 Calculation

For our example problem, it will be necessary to compute view factors for components in an axisymmetric, co-axial cylindrical assembly. While numerical techniques may be used in cases where the geometry is so complex that analytical formulations are intractable, they are often quite expensive to perform. For cylindrical assemblies, it is possible to compute view factors analytically by combining analytical expressions due to Brockman [43] with the above properties. To our knowledge, most of these expressions have not been derived previously.

Brockman considered radiation between two concentric cylinders of equal length and two annular disks, as illustrated in Figure 5.1. These two cylinders define the indicated surfaces b, i, o and t, corresponding to the bottom, inner, outer and top surfaces, respectively. Brockman parameterizes the view factors in terms of outer radius, R_o, inner radius, R_i, top

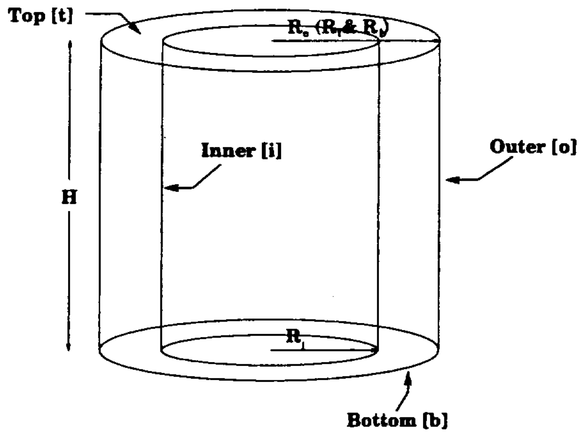


Figure 5.1: Two Concentric Cylinders

radius, R_t, bottom radius R_b and cylinder height, H. It is convenient to define the following terms before presenting the view factors:

$$X_o = \frac{R_o}{H} \tag{5.8}$$
$$X_i = \frac{R_i}{H} \tag{5.9}$$
$$X_t = \frac{R_t}{H} \tag{5.10}$$
$$X_b = \frac{R_b}{H} \tag{5.11}$$
$$Y_{to} = \sqrt{X_o^2 - X_i^2} + \sqrt{X_t^2 - X_i^2} \tag{5.12}$$
$$Y_{tb} = \sqrt{X_t^2 - X_i^2} + \sqrt{X_b^2 - X_i^2} \tag{5.13}$$

Brockman provides the following expressions for the view factors:

$$F_{o\to o} = \begin{cases} 0, & \text{when } X_o = X_i \\ \frac{1}{2X_o}\left(1 + 2X_o - \sqrt{1+4X_o^2}\right), & \text{when } X_i = 0 \\ \frac{1}{\pi X_o}\left(\cos^{-1}\frac{X_i}{X_o} + 2X_i \tan^{-1}\left(2\sqrt{X_o^2 - X_i^2}\right) + \right. & \\ \left. \pi(X_o - X_i) - \sqrt{(1+4X_o^2)}\tan^{-1}\left(\frac{\sqrt{(1+4X_o^2)(X_o^2-X_i^2)}}{X_i}\right)\right), & \text{otherwise} \end{cases} \tag{5.14}$$

$$F_{o\to i} = \begin{cases} 0, & \text{when } X_i = 0 \\ 1, & \text{when } X_o = X_i \\ \frac{1}{\pi X_o}\left\{\frac{1}{2}(X_o^2 - X_i^2 - 1)\cos^{-1}\frac{X_i}{X_o} + \pi X_i - \right. & \\ \frac{\pi}{2}(X_o^2 - X_i^2) - 2X_i \tan^{-1}\sqrt{X_o^2 - X_i^2} + & \\ \sqrt{(1+(X_o+X_i)^2)(1+(X_o-X_i)^2)} * & \\ \left. \tan^{-1}\sqrt{\frac{(1+(X_o+X_i)^2)(X_o-X_i)}{(1+(X_o-X_i)^2)(X_o+X_i)}}\right\}, & \text{otherwise} \end{cases} \tag{5.15}$$

$$F_{t\to o} = \begin{cases} 0, & \text{when } X_t = X_i \\ \frac{1}{\pi(X_t^2-X_i^2)}\Big(\frac{1}{2}(X_o^2-X_i^2)\Big(\pi-\cos^{-1}\frac{X_i}{X_o}\Big) & \\ 2X_i\Big(\tan^{-1}\Big(\sqrt{X_o^2-X_i^2}+\sqrt{X_t^2-X_i^2}\Big)- & \\ \tan^{-1}\Big(\sqrt{X_o^2-X_i^2}\Big)\Big)-\frac{1}{2}\cos^{-1}\frac{X_i}{X_t}+ & \\ \sqrt{(1+(X_o+X_t)^2)(1+(X_o-X_t)^2)}* & \\ \tan^{-1}\Big(\sqrt{\frac{(1+(X_o+X_t)^2)(Y_{to}^2-(X_o-X_t)^2)}{(1+(X_o-X_t)^2)((X_o+X_t)^2-Y_{to}^2)}}\Big)- & \\ \sqrt{(1+(X_o+X_i)^2)(1+(X_o-X_i)^2)}* & \\ \tan^{-1}\Big(\sqrt{\frac{(1+(X_o+X_i)^2)(X_o-X_i)}{(1+(X_o-X_i)^2)(X_o+X_i)}}\Big)\Big), & \text{when } X_t = X_o \\ \frac{1}{2X_t^2}\Big(\sqrt{(1+(X_t+X_o)^2)(1+(X_t-X_o)^2)}\Big)- & \\ 1-X_o^2+X_t^2), & \text{when } X_i = 0 \\ \frac{1}{\pi(X_t^2-X_i^2)}\Big(\frac{1}{2}(X_o^2-X_i^2)(\pi-\cos^{-1}\frac{X_i}{X_o})- & \\ 2X_i\Big(\tan^{-1}\Big(\sqrt{X_o^2-X_i^2}+\sqrt{X_t^2-X_i^2}\Big)- & \\ \tan^{-1}\sqrt{X_o^2-X_i^2}\Big)-\frac{1}{2}\cos^{-1}\frac{X_i}{X_t}+ & \\ \sqrt{(1+(X_o+X_t)^2)(1+(X_o-X_t)^2)}* & \\ \tan^{-1}\Big(\sqrt{\frac{(1+(X_o+X_t)^2)(Y_{to}^2-(X_o-X_t)^2)}{(1+(X_o-X_t)^2)((X_o+X_t)^2-Y_{to}^2)}}\Big)- & \\ \sqrt{(1+(X_o+X_i)^2)(1+(X_o-X_i)^2)}* & \\ \tan^{-1}\Big(\sqrt{\frac{(1+(X_o+X_i)^2)(X_o-X_i)}{(1+(X_o-X_i)^2)(X_o+X_i)}}\Big)- & \\ (X_o^2-X_t^2)\tan^{-1}\Big(\frac{X_o+X_t}{X_o-X_t}\Big)\sqrt{\frac{Y_{to}^2-(X_o-X_t)^2}{(X_o+X_t)^2-Y_{to}^2}}\Big), & \text{otherwise} \end{cases} \tag{5.16}$$

$$F_{t \to b} = \begin{cases} 0, & \text{when } X_b = X_i \\ \frac{1}{2X_t^2}\left(1 + X_t^2 + X_b^2 - \sqrt{(1+(X_t+X_b)^2)(1+(X_t-X_b)^2)}\right), & \text{when } X_i = 0 \\ \frac{1}{\pi(X_t^2-X_i^2)}\left(\frac{1}{2}(X_t^2 - X_i^2)\cos^{-1}\frac{X_i}{X_b} + \frac{1}{2}(X_b^2 - X_i^2)\cos^{-1}\frac{X_i}{X_t} + \right. & \\ 2X_i\left(\tan^{-1}\left(\sqrt{X_t^2 - X_i^2} + \sqrt{X_b^2 - X_i^2}\right) - \right. & \\ \left.\tan^{-1}\sqrt{X_t^2 - X_i^2} - \tan^{-1}\sqrt{X_b^2 - X_i^2}\right) - & \\ \sqrt{(1+(X_t+X_b)^2)(1+(X_t-X_b)^2)} * & \\ \tan^{-1}\left(\sqrt{\frac{(1+(X_t+X_b)^2)(Y_{tb}^2-(X_t-X_b)^2)}{(1+(X_t-X_b)^2)((X_t+X_b)^2-Y_{tb}^2)}}\right) + & \\ \sqrt{(1+(X_t+X_i)^2)(1+(X_t-X_i)^2)} * & \\ \tan^{-1}\left(\sqrt{\frac{(1+(X_t+X_i)^2)(X_t-X_i)}{(1+(X_t-X_i)^2)(X_t+X_i)}}\right) + & \\ \sqrt{(1+(X_b+X_i)^2)(1+(X_b-X_i)^2)} * & \\ \left.\tan^{-1}\left(\sqrt{\frac{(1+(X_b+X_i)^2)(X_b-X_i)}{(1+(X_b-X_i)^2)(X_b+X_i)}}\right)\right), & \text{otherwise} \end{cases} \tag{5.17}$$

For our applications, we will assume that $R_b = R_t = R_o$.

View Factor	Origin
$F_{o \to i}$	Equation (5.15)
$F_{o \to o}$	Equation (5.14)
$F_{t \to o}$	Equations (5.16)
$F_{t \to b}$	Equations (5.17)
$F_{i \to o}$	Reciprocal of $F_{o \to i}$
$F_{o \to t}$	Reciprocal of $F_{t \to o}$
$F_{b \to t}$	Reciprocal of $F_{t \to b}$
$F_{t \to i}$	Enclosure (1-$F_{t \to o}$-$F_{t \to b}$)
$F_{i \to t}$	Reciprocal of $F_{t \to i}$
$F_{b \to i}$	Symmetry using $F_{t \to i}$
$F_{i \to b}$	Reciprocal of $F_{b \to i}$

Table 5.1: View Factors for Two Concentric Cylinders

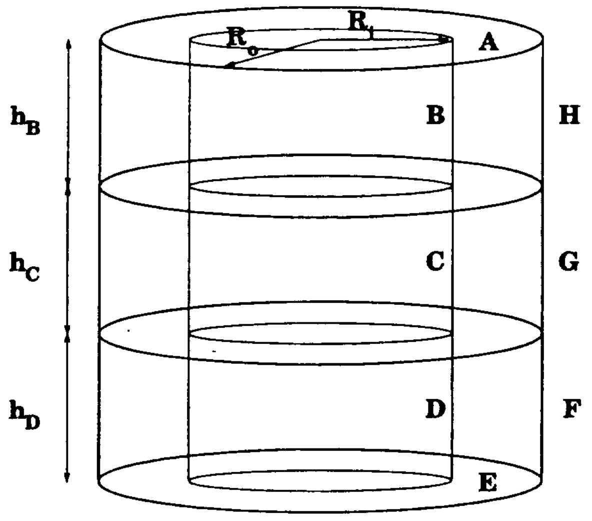


Figure 5.2: Complex Concentric Cylinders

Unfortunately, the geometry shown in Figure 5.1 is not sufficiently general for our application. We therefore generalize the case considered by Brockman to the one presented in Figure 5.2. All possible configurations of two concentric co-axial cylinders radiating to each other are shown in Figures 5.3. Likewise, all possible configurations of an annular disk and cylinder radiating to each other are shown in Figure 5.4. The view factors for the cases in Figures 5.3 and 5.4 can be expressed in terms of the view factors for the surfaces in Figure 5.2. So we begin by showing how we can compute all combinations of view factors for Figure 5.2.

Several of these view factors correspond directly to view factors in Table 5.1. For example, the view factors, $F_{G \to C}$, $F_{H \to B}$ and $F_{F \to D}$ are equivalent to $F_{o \to i}$. The full list of these view factors is given in Table 5.2. Several more view factors can be derived using the addition

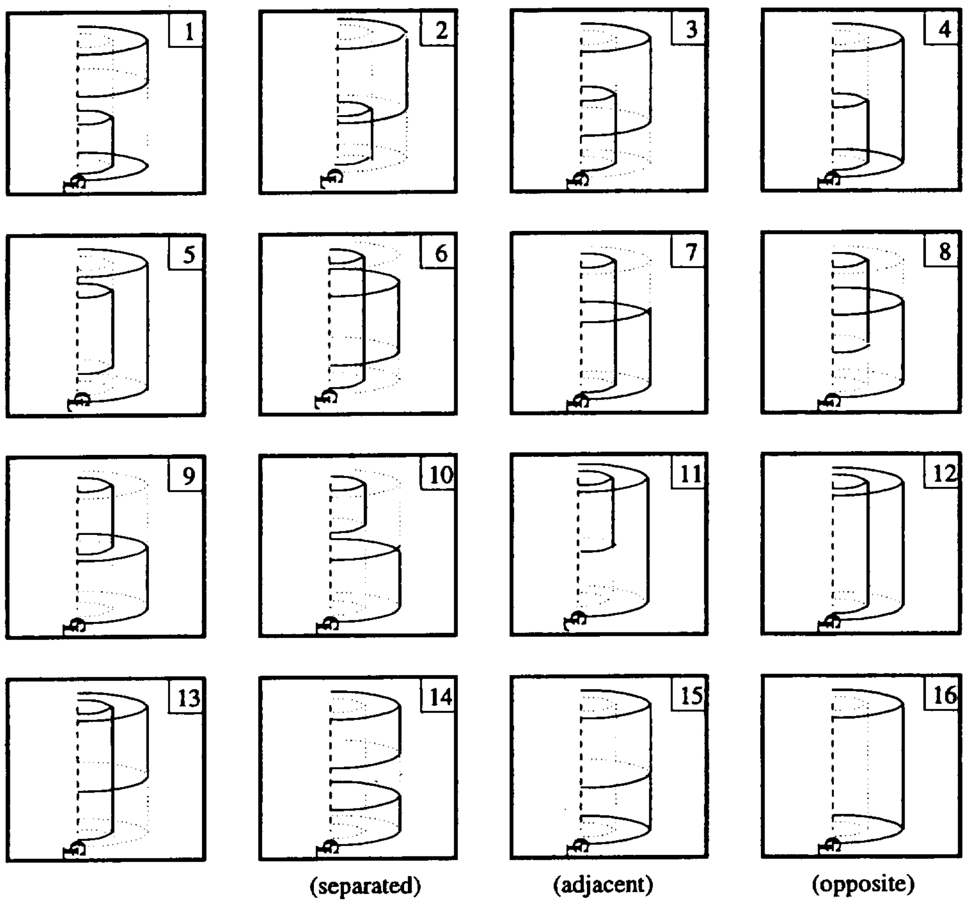


Figure 5.3: Configurations for Two Concentric Cylinders

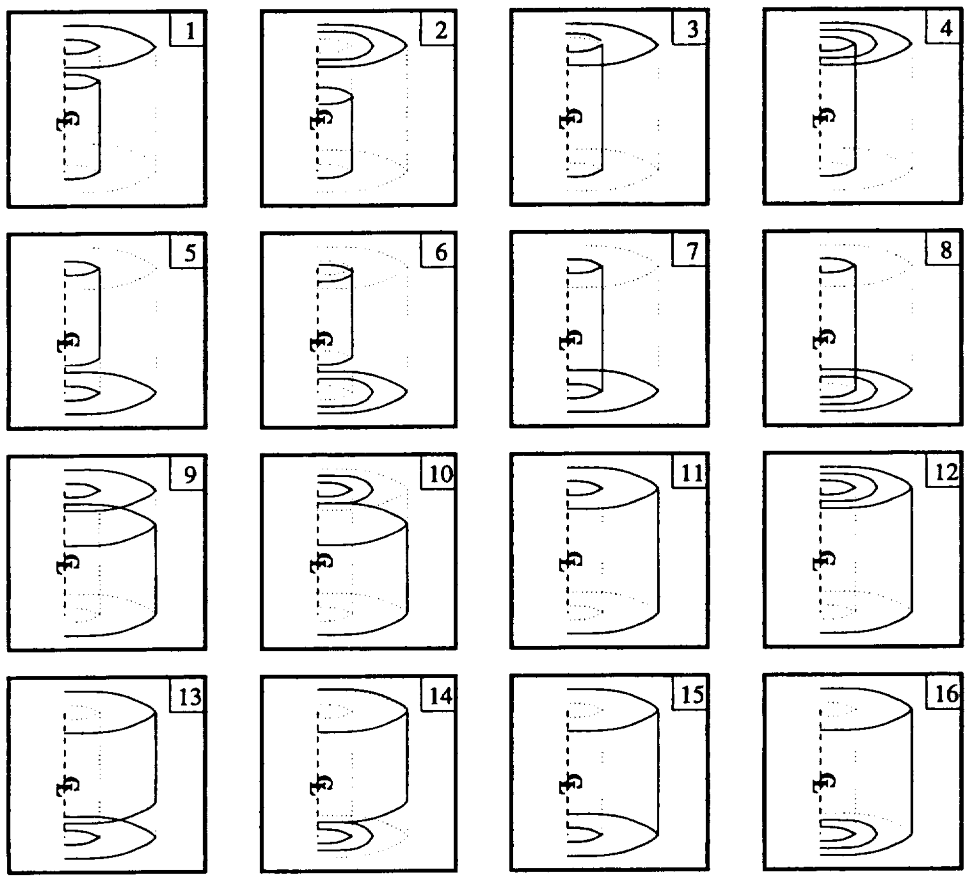


Figure 5.4: Configurations of a Concentric Cylinder and Annular Disk

property. For example, the view factor $F_{E\rightarrow H}$ can be expressed as:

$$F_{E\rightarrow H} = F_{E\rightarrow H+G+F} - F_{E\rightarrow G+F} \tag{5.18}$$

where $F_{E\rightarrow H+G+F}$ and $F_{E\rightarrow G+F}$ can both be computed directly from $F_{b\rightarrow o}$. The remaining composite view factors are listed in Table 5.3. Additionally, the view factor $F_{G\rightarrow A}$ is the reciprocal of the view factor $F_{A\rightarrow G}$ which can be computed from $F_{t\rightarrow o}$ as shown in Table 5.2.

Finally, Table 5.5 contains all of the remaining view factors, which require more complex manipulations. As an example, let us consider the derivation of view factor $F_{F\rightarrow B}$. Looking at Figure 5.2, we see that surface B sees surfaces A, H, G, E and F. We already know, from Tables 5.2-5.5, the view factors for $F_{B\rightarrow A}$, $F_{B\rightarrow H}$, $F_{B\rightarrow G}$ and $F_{B\rightarrow E}$. We may now use the enclosure property to express $F_{B\rightarrow F}$ as:

$$F_{B\rightarrow F} = 1 - F_{B\rightarrow A} - F_{B\rightarrow H} - F_{B\rightarrow G} - F_{B\rightarrow E} \tag{5.19}$$

Note that we can then compute $F_{F\rightarrow B}$ by using $F_{B\rightarrow F}$ and the reciprocal property.

It is important to note that every view factor in Tables 5.2-5.5 depends only on view factors previously listed in the table, or in a previous table. This avoids any "circular" dependencies between the view factors.

Table 5.6 lists all the configurations shown in Figure 5.3 and describes how each can be expressed in terms of the view factors shown in Tables 5.1-5.5. Likewise, Table 5.7 lists all the configurations shown in Figure 5.4 and describes how each can be expressed in terms of the view factors shown in Tables 5.1-5.5.

View Factor	Derived from
$F_{B\to A}$	$F_{i\to t}$
$F_{B\to H}$	$F_{i\to o}$
$F_{B\to Y}$	$F_{i\to b}$
$F_{C\to Y}$	$F_{i\to t}$
$F_{C\to G}$	$F_{i\to o}$
$F_{C\to X}$	$F_{i\to b}$
$F_{A\to B}$	$F_{t\to i}$
$F_{G\to C}$	$F_{o\to i}$
$F_{D\to X}$	$F_{i\to t}$
$F_{D\to F}$	$F_{i\to o}$
$F_{F\to D}$	$F_{o\to i}$
$F_{D\to E}$	$F_{i\to b}$
$F_{X\to C}$	$F_{b\to i}$
$F_{C\to X}$	$F_{i\to b}$
$F_{X\to G}$	$F_{b\to o}$
$F_{G\to X}$	$F_{o\to b}$
$F_{X\to A}$	$F_{b\to t}$
$F_{A\to H}$	$F_{t\to o}$
$F_{H\to A}$	$F_{o\to t}$
$F_{E\to F}$	$F_{b\to o}$
$F_{F\to E}$	$F_{o\to b}$
$F_{A\to G}$	$F_{t\to o}$
$F_{E\to G}$	$F_{b\to o}$
$F_{H\to Y}$	$F_{o\to b}$
$F_{Y\to H}$	$F_{b\to o}$
$F_{H\to H}$	$F_{o\to o}$
$F_{G\to G}$	$F_{o\to o}$
$F_{F\to F}$	$F_{o\to o}$
$F_{E\to D}$	$F_{b\to i}$
$F_{D\to E}$	$F_{i\to b}$

Table 5.2: Simple View Factors

View Factor	Derived from
$F_{X\to H}$	$F_{b\to o}$
$F_{E\to H}$	$F_{b\to o}$
$F_{E\to B}$	$F_{b\to i}$

Table 5.3: View Factors using the Addition Property

View Factor	Derived from
$F_{H \to B}$	$F_{B \to H}$
$F_{G \to A}$	$F_{A \to G}$
$F_{G \to E}$	$F_{E \to G}$

Table 5.4: View Factors using the Reciprocal Property

View Factor	Derived from
$F_{X \to B}$	Enclosure property $(F_{X \to C}, F_{X \to G}, F_{X \to H} and F_{X \to A})$
$F_{B \to X}$	Reciprocal property
$F_{B \to G}$	Enclosure property $(F_{X \to C}, F_{B \to H}, F_{B \to A} and F_{B \to X})$
$F_{G \to B}$	Reciprocal property
$F_{C \to H}$	Symmetry $(F_{B \to G})$
$F_{H \to C}$	Reciprocal property
$F_{D \to G}$	Symmetry $(F_{C \to H})$
$F_{G \to D}$	Reciprocal property
$F_{H \to D}$	Symmetry $(F_{F \to B})$
$F_{D \to H}$	Symmetry $(F_{B \to F})$
$F_{B \to E}$	Reciprocal property
$F_{B \to F}$	Enclosure property $(F_{B \to A}, F_{B \to H}, F_{B \to G} and F_{B \to E})$
$F_{F \to B}$	Reciprocal property
$F_{H \to X}$	Reciprocal property
$F_{H \to G}$	Enclosure property $(F_{H \to X}, F_{H \to C}, F_{H \to B}, F_{H \to H} and F_{H \to A})$
$F_{G \to H}$	Reciprocal property
$F_{H \to E}$	Reciprocal property
$F_{H \to F}$	Enclosure property $(F_{H \to A}, F_{H \to B}, F_{H \to C}, F_{H \to D}, F_{H \to E}, F_{H \to H} F_{H \to G})$
$F_{A \to C}$	Addition property $(F_{A \to B})$
$F_{C \to A}$	Reciprocal property
$F_{A \to D}$	Symmetry $(F_{A \to C})$
$F_{D \to A}$	Symmetry $(F_{C \to A})$
$F_{E \to C}$	Symmetry $(F_{E \to D})$
$F_{C \to E}$	Reciprocal property
$F_{C \to F}$	Symmetry $(F_{B \to G})$
$F_{F \to C}$	Symmetry $(F_{G \to B})$

Table 5.5: View Factors Using Previously Derived View Factors

Configuration (inner to outer)	Utilizes
1	$F_{D\to H}$
2	$F_{C\to H}$
3	Enclosure, Reciprocal and Addition Property
4	Enclosure Property ($F_{C\to X}$ and $F_{C\to A}$)
5	Addition Property ($F_{C\to H}, F_{C\to G}$ and $F_{C\to F}$)
6	Reciprocal Property ($F_{G\to B}, F_{G\to C}$ and $F_{G\to D}$)
7	Reciprocal Property ($F_{G\to B}$ and $F_{G\to C}$)
8	Enclosure, Reciprocal and Addition Property
9	$F_{B\to G}$
10	$F_{B\to F}$
11	Addition Property ($F_{B\to H}$ and $F_{B\to G}$)
12	$F_{B\to H}$
13	Reciprocal Property ($F_{H\to B}$ and $F_{H\to C}$)
14 (outer to outer)	$F_{H\to F}$
15 (outer to outer)	$F_{H\to G}$
16 (outer to outer)	$F_{H\to H}$

Table 5.6: Table of Configurations from Figure 5.3

Configuration (inner to disk)	Utilizes
1	$F_{C\to A}$
2	Addition Property ($F_{C\to A}$)
3	$F_{B\to A}$
4	Addition Property ($F_{B\to A}$)
5	$F_{C\to E}$
6	Addition Property ($F_{C\to E}$)
7	$F_{D\to E}$
8	Addition Property ($F_{D\to E}$)
9	$F_{G\to A}$
10	Addition and Reciprocal Property ($F_{t\to o}$)
11	$F_{H\to A}$
12	$F_{o\to t}$
13	$F_{G\to E}$
14	Addition and Reciprocal Property ($F_{b\to o}$)
15	$F_{F\to E}$
16	$F_{o\to b}$

Table 5.7: Table of Configurations from Figure 5.4

Appendix B contains *Mathematica* scripts for computing all the view factors discussed in this chapter. This allows the reader to see in detail the manipulations required to obtain the view factors described.

5.3 Numerical Implications

Modeling radiative heat transfer has an important impact on the structure of the resulting matrix (*cf.* Gartling [44]). We consider steady-state problems for the sake of brevity, although the same analysis applies to transient problems. Each equation in a finite element formulation represents a constraint on the unknowns of the system. Given a node n_i in a heat conduction problem (with no radiative heat transfer), the constraint on the temperature at node n_i will involve only those nodes, n_j, in the mesh for which n_i and n_j appear in the same element.

However, when we consider radiative heat transfer, it is possible for a nodal temperature to be affected by a much larger set of other nodes. The constraint on node n_i's temperature will then contain terms involving n_j if both nodes are part of the same element *or* connected to radiative surface elements which can "see" each other (*i.e.*, have non-zero view factors).

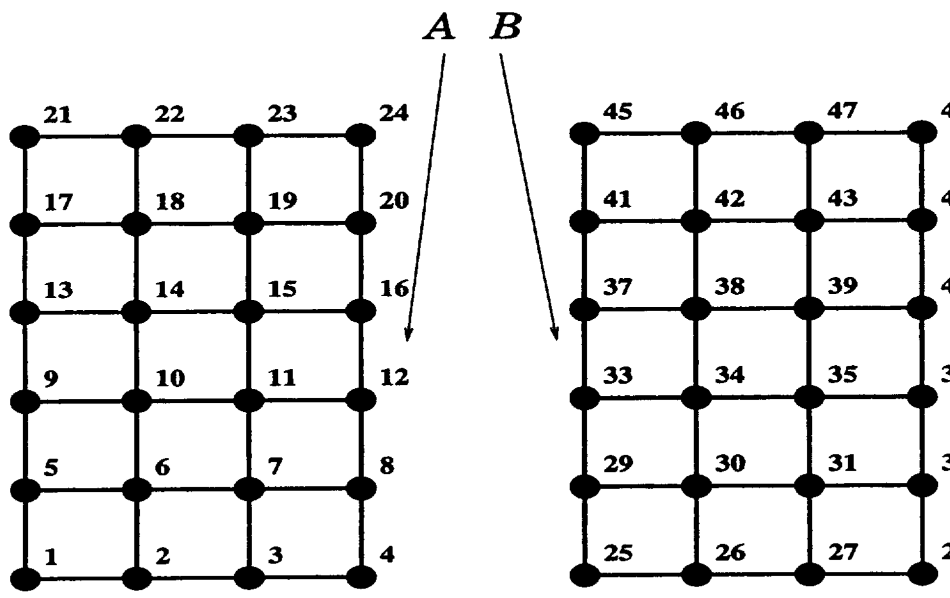


Figure 5.5: Example Mesh

To illustrate this point, Figure 5.5 shows a sample mesh consisting of two separate bodies. Figure 5.6a shows the tangent matrix structure when only heat conduction is considered. Notice that there are never more than 9 non-zero terms for any given equation. On the other hand, Figure 5.6b shows the structure of the tangent matrix when radiative heat transfer occurs between edges A and B. Notice the "coupling" terms in both the upper right and lower left quadrants resulting from the radiative heat transfer. Although we do not consider the case of participating media, it is worth noting that the matrix is potentially *fully dense* because of the interaction with additional nodes in the participating media.

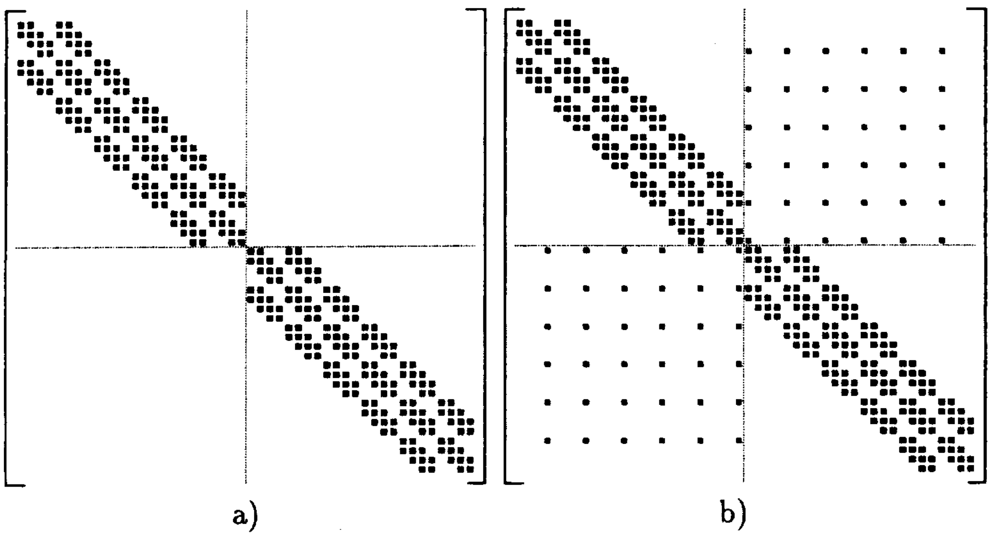
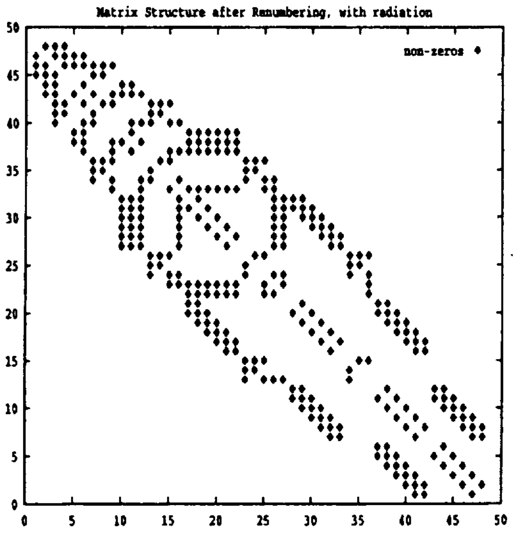
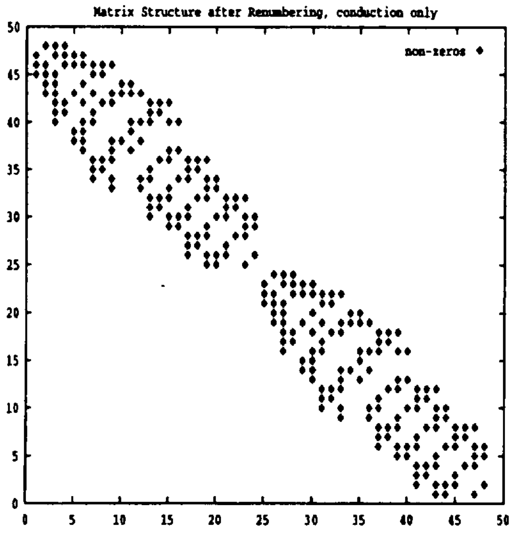


Figure 5.6: Matrix Structure (A ■ represents a non-zero entry)

Such coupling terms cause trouble for factorization methods because they inflate the envelope of the tangent matrix, even with node renumbering, as can be seen in Figure 5.7. For our example, the envelope for the matrix contains 792 elements with radiation while it contains only 435 without radiation.



with radiation



without radiation

Figure 5.7: Structure after renumbering

In the case of Krylov subspace methods, however, the addition of the coupling terms increases only slightly the amount of work necessary to compute the needed matrix-vector products. However, even though the cost of the matrix-vector product is not increased significantly, in a problem with many coupling terms, if the condition number of the tangent matrix increases, the convergence of the conjugate gradient algorithm is usually slower. For the system shown in Figure 5.5, given a uniform conductivity of 1.0 and the following boundary conditions:

$$T_1 = 1400K$$

$$T_{25} = 1500K$$

we find that the condition number of the tangent matrix is 162.54 for conduction only, but it jumps to 1219.3 when radiation is included. Nevertheless, we show later that the Krylov subspace methods are much more efficient than factorization methods for solving our example problem.

Chapter 6

Applications

6.1 Introduction

One field which has long been of commercial interest and where considerable modeling work has been done, is the study of solidification processes. Initially, the focus was to develop accurate models of the solidification process, see Kurz and Fisher [45]. This work was then followed by modeling of crystal growth in Bridgman furnaces [46] by Crochet *et al.* [47,48] and Alexander *et al.* [49]. In addition, Atherton *et al.* [50] examined the effect of radiative heat transfer on crystal growth. Finally, modeling of general phase changes using FEM has been done by Bathe [51], Voller *et al.* [52–54] and Dantzig [55].

Given accurate simulation models of solidification and the ability to provide sensitivity information for these models, attention has recently turned to the optimization of solidification processing. Work presented by Tortorelli *et al.* [37,38] included optimization applications of crystal growth and casting, respectively. For this work the commercial finite element simulation code FIDAP [39] and the optimization package ADS [14] were used. In both cases, most of the effort in setting up the optimization problem involved either modifying FIDAP to calculate sensitivities, or passing objective information between FIDAP and ADS. In this work, the authors had access to the source code for FIDAP. If a commercial code does not compute the tangent matrix accurately, it would be impossible to even consider adapting the commercial code to calculate analytical sensitivities in non-linear problems without access to the source code.

To demonstrate the usefulness of optimization in solidification processing, we present a realistic application. We consider the Bridgman growth of a semiconductor crystal using NASA's Crystal Growth Furnace (CGF). The crystal is encased in a cartridge which is heated inside the furnace, and solidification is accomplished by translating the cartridge with respect to the furnace. The process can be optimized by adjusting the power input to the heater coils to provide a desired temperature profile. This application involves radiative heat transfer in a complex geometry, and the calculation of the solution alone is numerically challenging.

6.2 Example Problem

6.2.1 Problem Description

Before demonstrating the crystal growth application, we first present a simpler problem to introduce many of the concepts involved. Figure 6.1 shows a schematic of the physical problem to be solved. The material to be solidified is represented by a 15cmx5cm slab. The

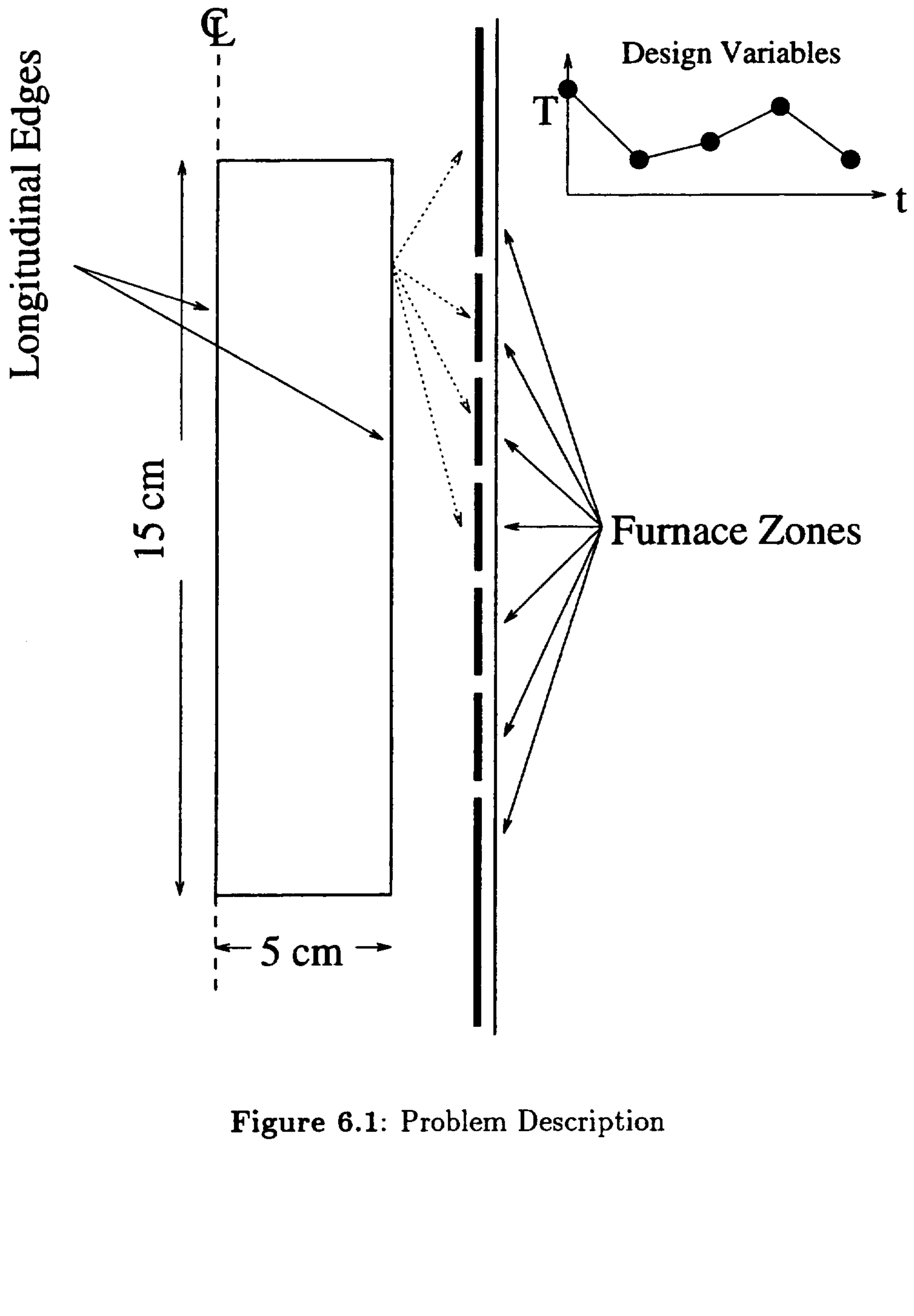


Figure 6.1: Problem Description

unknowns in the problem are the temperatures in the slab. The furnace wall (shown on the right in Figure 6.1) is represented as a set of boundary conditions. The furnace is divided into seven zones and the temperature is considered to be piecewise uniform over each zone. The temperature of each zone is also interpolated in time by four piecewise linear segments (as shown in the upper right of Figure 6.1). The points defining these segments are the design variables. The five control points in each zone and seven distinct zones, produce a total of 35 design variables.

6.2.2 Objectives

The following are our goals for solidification process:

- Uniform axial interface velocity across the radial direction.
- Maintain a specified constant temperature gradient normal to the solidification interface.
- Minimize radial temperature gradient.

We translate these goals into the desired temperature distributions along both the edge and center line, as shown in Figure 6.2. For example, if the temperature distribution is the same along the edge and centerline, then the radial temperature gradient is negligible. Likewise, if the temperature distribution in Figure 6.2 can be achieved at the edge and centerline, then we will satisfy the constraints on temperature gradient and velocity at the solidification interface.

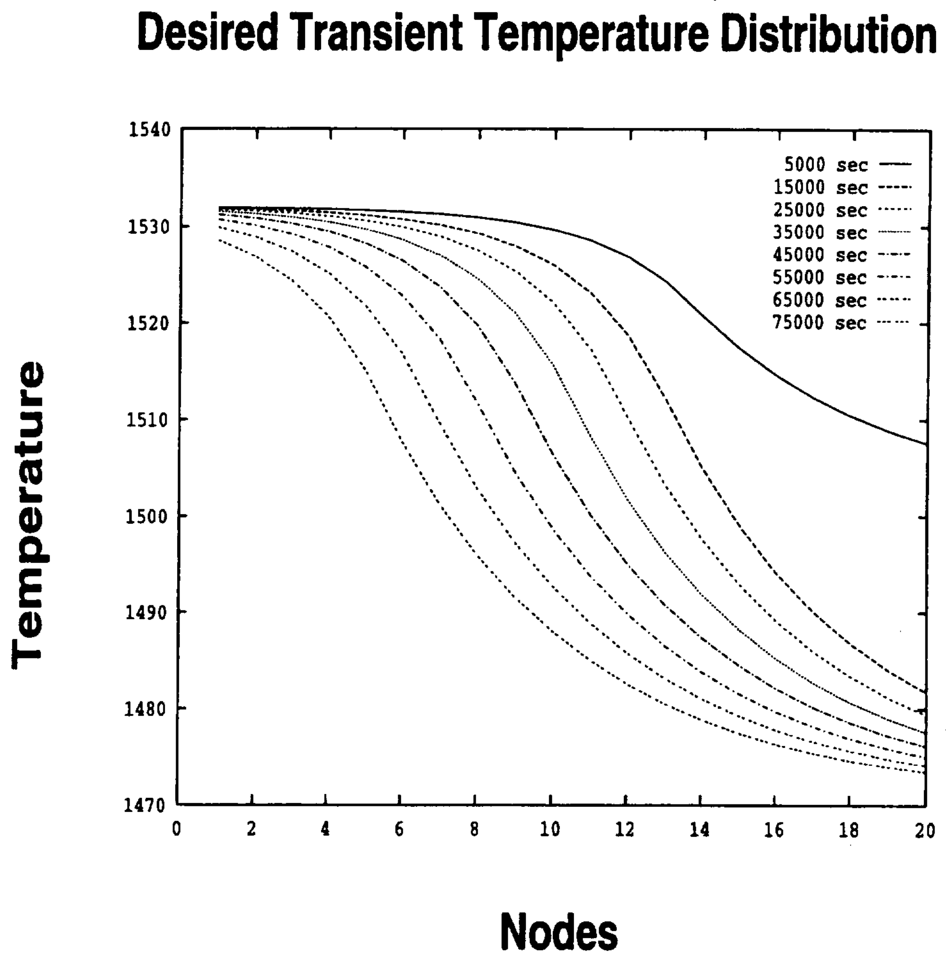


Figure 6.2: Ideal Longitudinal Temperature Distributions

Because the distributions shown in Figure 6.2 represent the best possible result, we define our objective function as the root mean square error between the simulation results and the desired distributions shown in Figure 6.2. The objective function is posed mathematically as:

$$G(T(\mathbf{n},\mathbf{t},\mathbf{b}),\mathbf{b}) = \sqrt{\frac{\sum_{i=1}^{N_t}\sum_{j=1}^{N_n}(T(n_j,t_i,\mathbf{b}) - \bar{T}(n_j,t_i))^2}{N_t * N_n}} \tag{6.1}$$

where N_t is the number of time steps in the simulation, N_n is the number of nodes along the longitudinal edges, n_j is the j^{th} node on the longitudinal edges, t_i is the time at the i^{th} edge and $\bar{T}(n_j,t_i)$ is the the value of the function shown in Figure 6.2 at node n_j and time t_i.

6.2.3 Results

This optimization problem was solved using FEMLIB, and the DOT library [15]. The sensitivities were computed by direct differentiation. Figure 6.3 shows the computed optimal temperature distribution along the outer longitudinal edge (the edge closest to the furnace wall) compared with the functions shown in Figure 6.2. It can be seen from this figure that it is possible, by using optimization, to specify a desired result for a simulation and then systematically and efficiently determine the optimal values for simulation parameters so as to achieve this result. In cases with the potential for multiple optimal designs, additional objectives can be placed on the system to distinguish between designs.

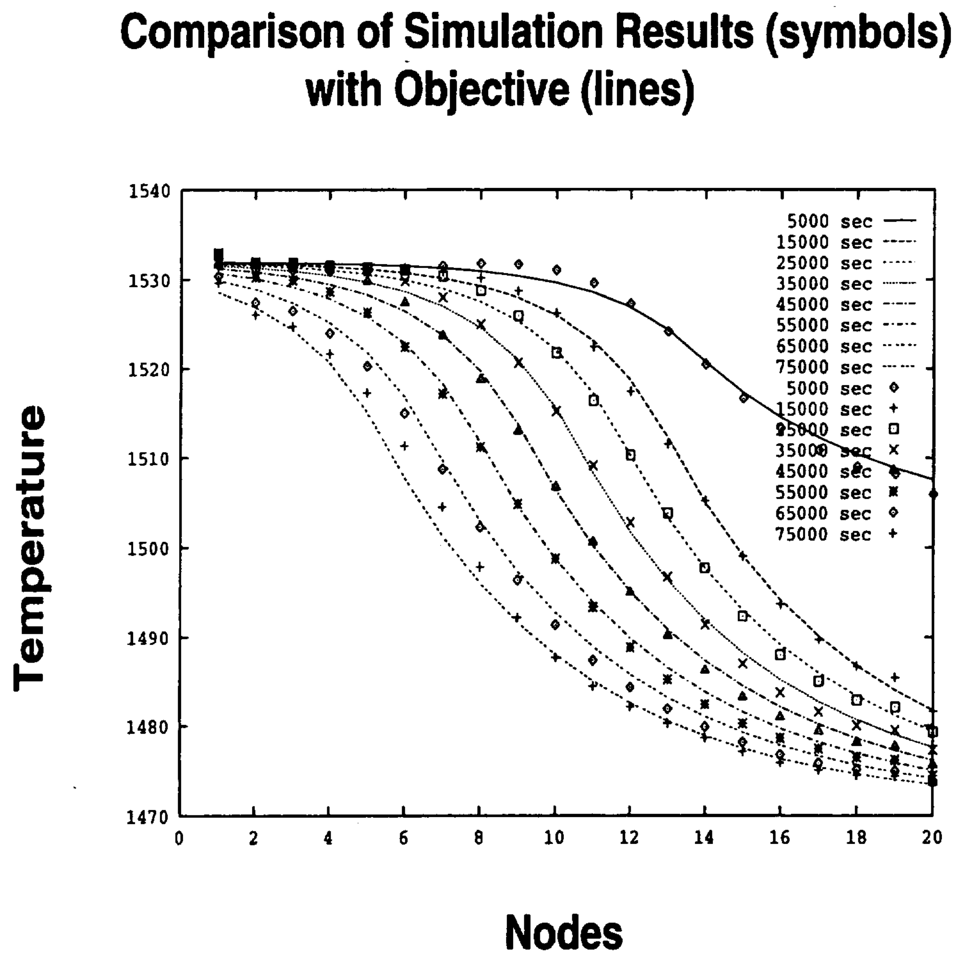


Figure 6.3: Optimization Results

6.3 Crystal Growth

We now consider a steady-state model of a crystal growth process and how optimization and sensitivity analysis have been applied to improve the crystal growth process. The solidification analysis is based on work by Watring [56] and Chait [57], using NASA's Crystal Growth Furnace (CGF).

6.3.1 Apparatus

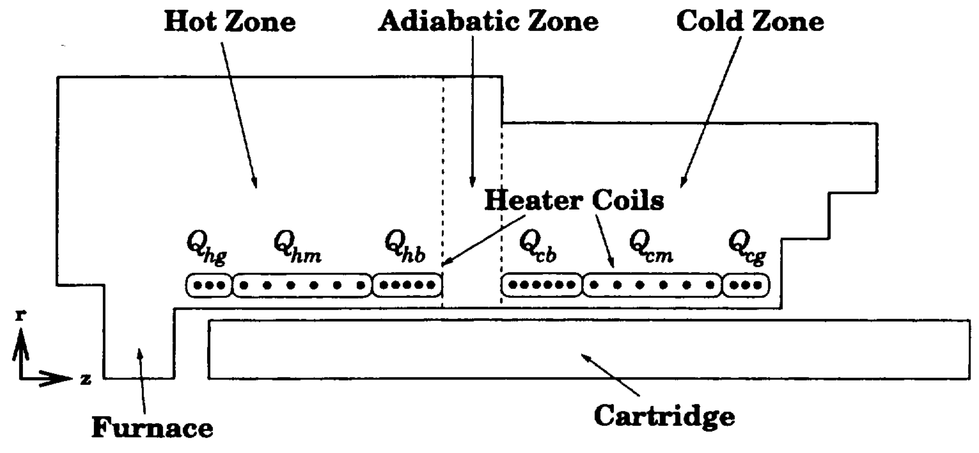


Figure 6.4: Furnace/Cartridge Apparatus

Figure 6.4 shows a schematic containing both the furnace and the cartridge. The CGF is approximately 55cm long and has an outer radius of 12cm. The 2.56cm diameter cartridge is made of molybdenum and fits inside the 3cm diameter furnace bore. The aspect ratio of the furnace has been altered in Figure 6.4 to distinguish the different zones.

6.3.1.1 Cartridge

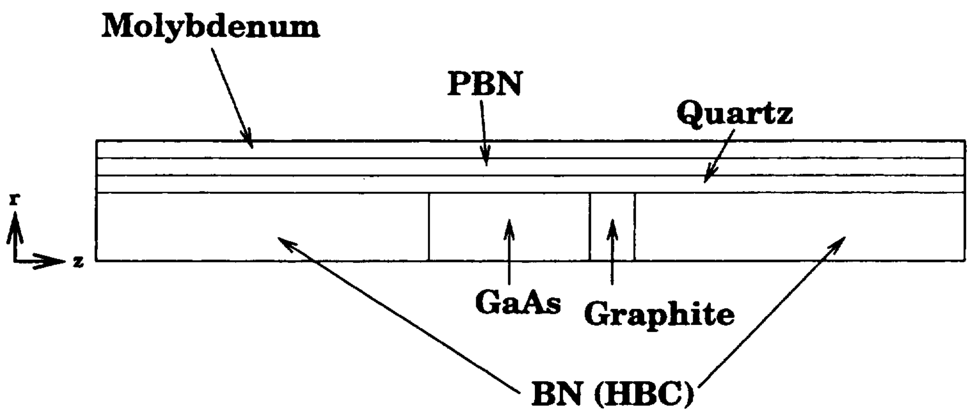


Figure 6.5: Cartridge Schematic

The cartridge encloses an ampoule containing Gallium Arsenide (GaAs), from which the crystal is to be grown, and various container materials. The exterior of the cartridge is made from layers of molybdenum, quartz and pyrolytic boron nitride (PBN). Inside these protective layers is the GaAs ampoule, bracketed by graphite and Boron Nitride (BN) to plug the ends of the cartridge (see Figure 6.5). The properties for the materials used in our model can be found in Appendix C. We model both the furnace and the cartridge, and the models can be used either together or independently. The simulations use axisymmetric transformations to take advantage of the cylindrical symmetry.

A schematic of the FEM model for the cartridge can be found in Figure 6.6. In addition, because of the expected steep temperature gradients near the solidification interface, the mesh has been refined in the area of solidification to assure accuracy.

6.3.1.2 Furnace

The furnace is constructed from various ceramic insulating materials and contains three separate heating zones. The "hot zone" is the section of the furnace which operates above

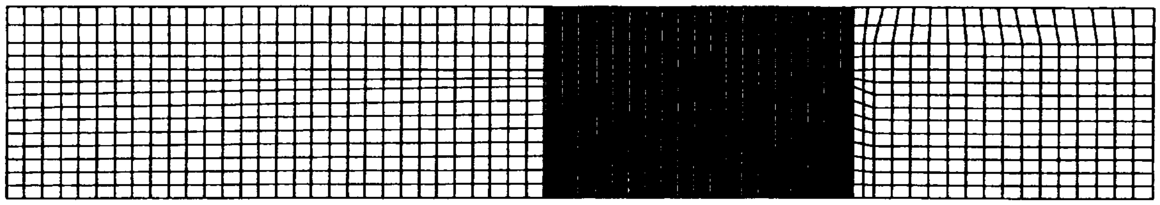


Figure 6.6: Cartridge Mesh

the melting temperature of GaAs, while the "cold zone" operates below this temperature. Between these two zones is an "adiabatic zone" where radial heat transfer between the cartridge and the furnace is kept to a minimum. The axial temperature gradient in the adiabatic zone is very steep because within this narrow zone the temperature must go from hot to cold. During this transition within the adiabatic zone, the gallium-arsenide solidifies.

Figure 6.7 shows a schematic diagram of the furnace. The inside of the furnace bore is composed of zirconia in the adiabatic zone and beryllia in hot and cold zones. The furnace bore is then surrounded by a layer of zirconia and a final layer of Min-K TE-1800. Figure 6.8 shows the finite element mesh used for the furnace.

Radiative heat transfer occurs between the inside of the furnace bore and the outside of the cartridge. To compute the heat transfer, it is necessary to calculate view factors as described in Section 5. The exterior of the furnace is surrounded by water filled cooling pipes, hence the outer furnace surface is covered with surface elements to model the forced convection. The heater coils are modeled as volumetric heat sources along the inside of the furnace bore. Finally, radiative heat transfer is included for the alumina at the end of the

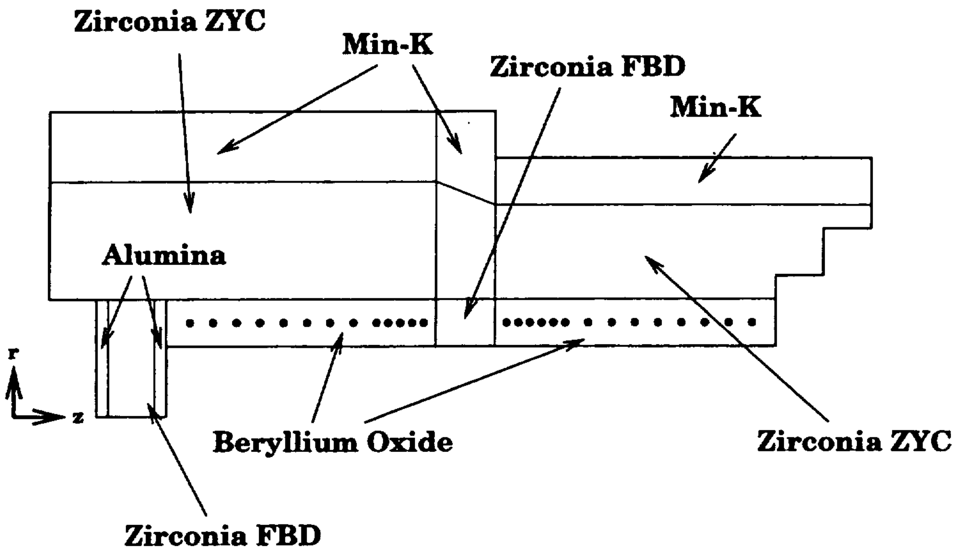


Figure 6.7: Furnace Schematic

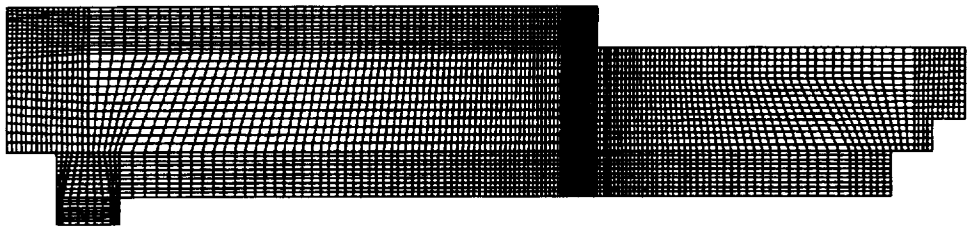


Figure 6.8: Furnace Mesh

hot zone because it is significantly hotter than the ambient. There are no essential boundary conditions.

6.3.2 Objective

When growing crystals, the two main process parameters we wish to control are, the growth rate and the temperature gradient in the liquid at the interface. The GaAs crystals are grown at rate of around 1 μm per second and with a temperature gradient at the solidification interface between 1 and 10 K/mm [45]. The crystal satisfies the Stefan condition [45] at the interface:

$$k_s \nabla T_s \cdot \hat{\mathbf{n}} - k_l \nabla T_l \cdot \hat{\mathbf{n}} = \rho L_f \mathbf{V} \cdot \hat{\mathbf{n}} \tag{6.2}$$

where k_s is the conductivity of the isotropic solid, ∇T_s is the temperature gradient in the solid, k_l is the conductivity of the isotropic liquid, ∇T_l is the temperature gradient in the liquid, L_f is the latent heat of fusion, ρ is the density, $\mathbf{V}$ is the velocity of the solidification front and $\hat{\mathbf{n}}$ is the normal to the solidification interface. Dividing by $k_l \nabla T_l \cdot \hat{\mathbf{n}}$ yields:

$$\frac{k_s \nabla T_s \cdot \hat{\mathbf{n}}}{k_l \nabla T_l \cdot \hat{\mathbf{n}}} - 1 = \frac{\rho L_f V \cdot \hat{\mathbf{n}}}{k_l \nabla T_l \cdot \hat{\mathbf{n}}} \tag{6.3}$$

Using a growth velocity of 1 μm/sec, temperature gradients from 1-10 K/mm and the material properties given in the appendix, the right-hand side of Equation 6.3 evaluates to between $55.4 * 10^{-3}$ and $5.54 * 10^{-3}$. This indicates that the contribution of latent heat evolution (*i.e.*, interface motion) is negligible in comparison to heat conduction. We therefore treat the problem as if it were steady-state.

We compute the Rayleigh Number for this problem as follows:

$$\frac{\rho C_p g l^3 \beta \Delta T}{\mu k} = \frac{\left(5.71 * 10^3 \frac{kg}{m^2}\right)^2 \left(2840 \frac{J}{kgK}\right) \left(10^{-3} \frac{m}{s^2}\right) (.3\ m)^3 \left(10^{-5} \frac{1}{K}\right) (400\ K)}{\left(1.7 * 10^3 \frac{kg}{ms}\right) \left(18 \frac{W}{mK}\right)} = .326 \tag{6.4}$$

Because the Rayleigh Number is so low, we can neglect modeling of bouyancy driven convection in the liquid. Finally, we compute the Peclet Number:

$$\frac{\rho C_p V l}{k} = \frac{\left(5.71 * 10^3 \frac{kg}{m^2}\right) \left(2840 \frac{J}{kgK}\right) \left(10^{-6} \frac{m}{s}\right) (.3\ m)}{\left(18 \frac{W}{mK}\right)} = .270 \tag{6.5}$$

Since the Peclet Number is so low, we can also ignore any forced convection that occurs in the liquid.

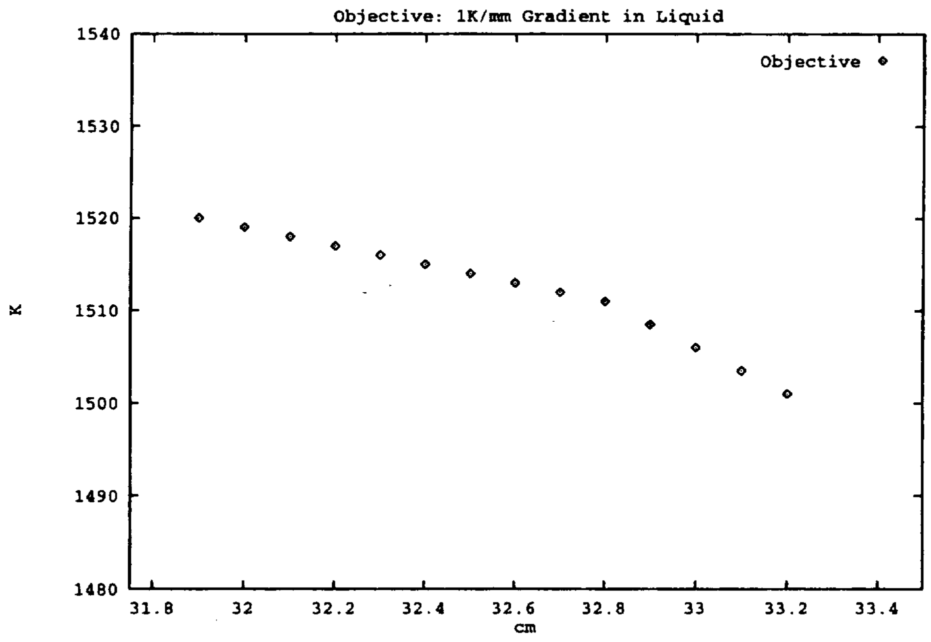


Figure 6.9: Crystal Growth Objective Function for $\nabla T = \frac{1K}{mm}$

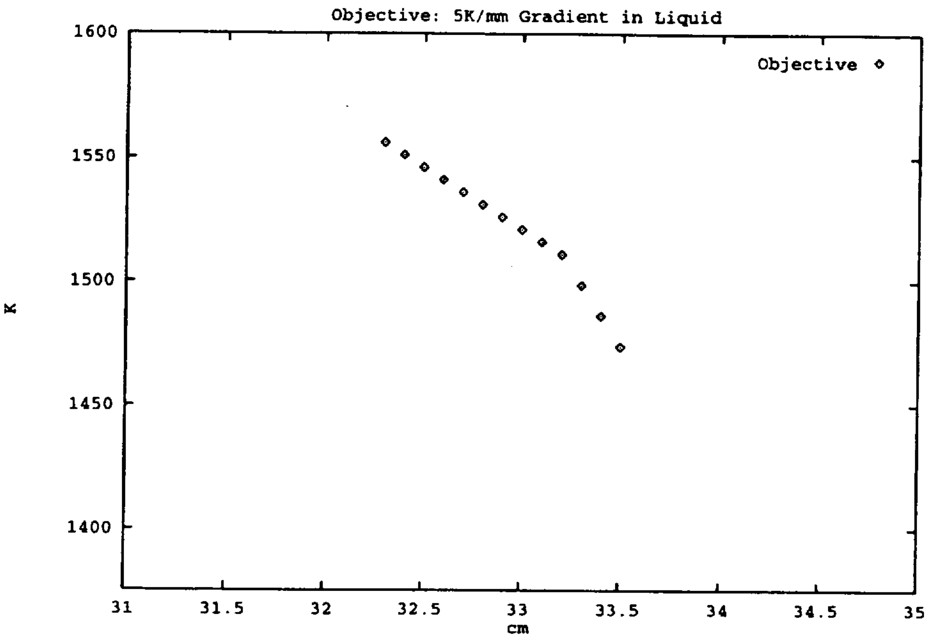


Figure 6.10: Crystal Growth Objective Function for $\nabla T = \frac{5K}{mm}$

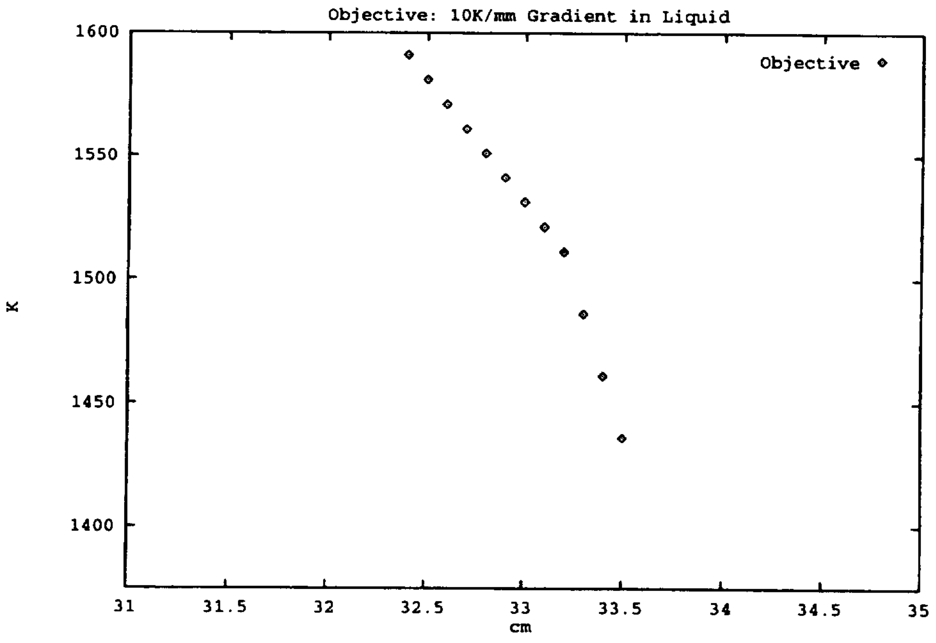


Figure 6.11: Crystal Growth Objective Function for $\nabla T = \frac{10K}{mm}$

As in the previous example, we pose a "desirable" temperature distribution for the ampoule, shown in Figures 6.9-6.11. The independent variable in these figures is the longitudinal position of the temperature profile. There are several interesting characteristics to this objective function. First, we attempt to specify the temperature distribution in the ampoule only, avoiding the problem of over-constraining the system. Using equation Equation (6.3), and neglecting the heat evolution term, we have the following expression for the temperature gradient in the solid:

$$\nabla T_s \cdot \hat{\mathrm{n}} = \frac{k_s}{k_l} \nabla T_l \cdot \hat{\mathrm{n}} \tag{6.6}$$

Using this expression we can construct the desired temperature distribution near the interface (see Figures 6.9-6.11). Lastly, we compute our objective by taking the differences between our desired temperature distribution and the actual ampoule temperatures both at the surface of the ampoule and at its center. Forcing the surface and center of the ampoule to the same temperature distribution leads to a flat interface.

6.3.3 Results

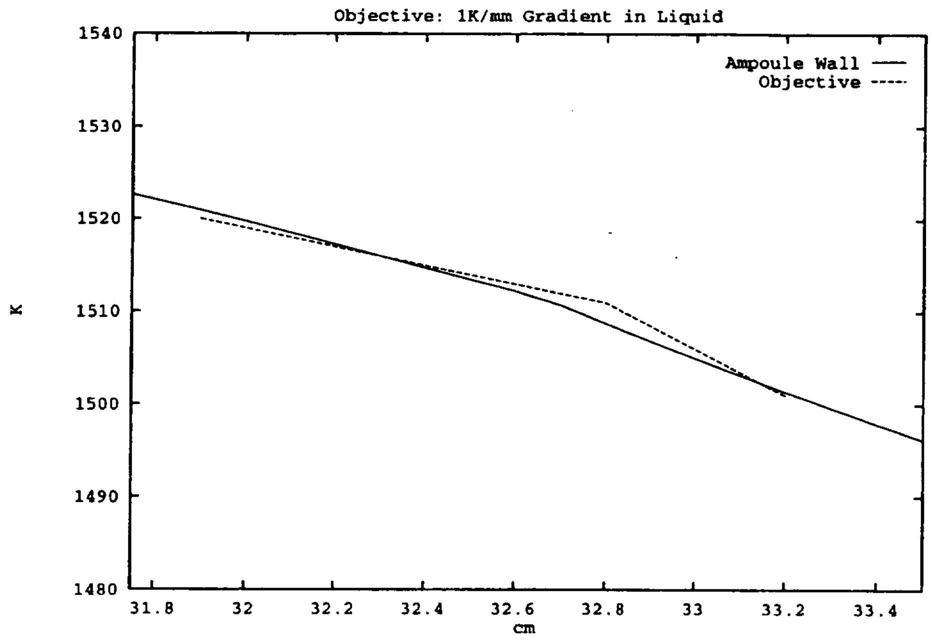
For all the results presented in the section, the DOT optimization package [15] was used. In addition, the objective function gradients were computed using the adjoint method as discussed in Section 2.4. Table 6.12 shows various statistics for the optimization of the different objective functions. In addition, Figures 6.13-6.15 compare the desired results with those obtained through optimization. In each of the figures below there are two parts. Part *a* will show a comparison between the ampoule wall temperature distribution and our ideal temperature distribution. Likewise, part *b* will show a comparison between the centerline temperature distribution and our ideal temperature distribution.

Interface Temperature Gradient	Function Evaluations	Gradient Evaluations	Initial Objective	Final Objective
1,000K/m	88	16	714,089	101
5,000K/m	51	9	$6.593 * 10^6$	1088
10,000K/m	44	7	$6.937 * 10^6$	11940

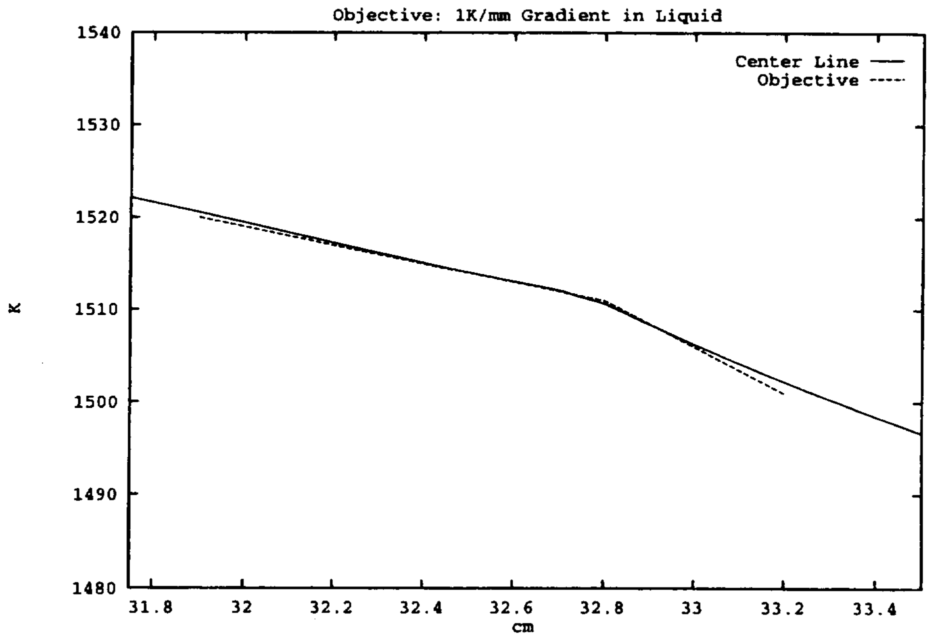
Figure 6.12: Optimization Statistics

As we can see, the steeper the temperature gradient becomes, the more difficultly we have matching our ideal temperature distribution. In addition, note that the temperature distributions on the ampoule wall and at the centerline appear to be offset in the longitudinal direction. This offset is due to radial heat losses in the adiabatic zone. Any heat loss in the radial direction translates into a non-zero temperature gradient component, $\frac{\partial T}{\partial r}$, in the radial direction.

We take this opportunity to compare the various linear solution solving algorithms discussed in Sections 1.3.2 and 2.5. Many benchmarks exist which compare the various tech-

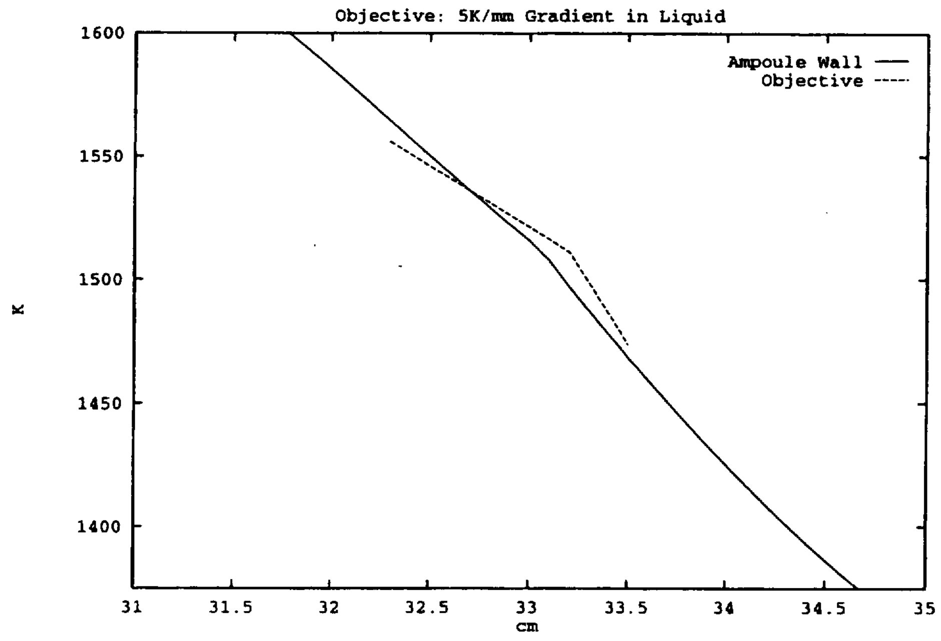


a)

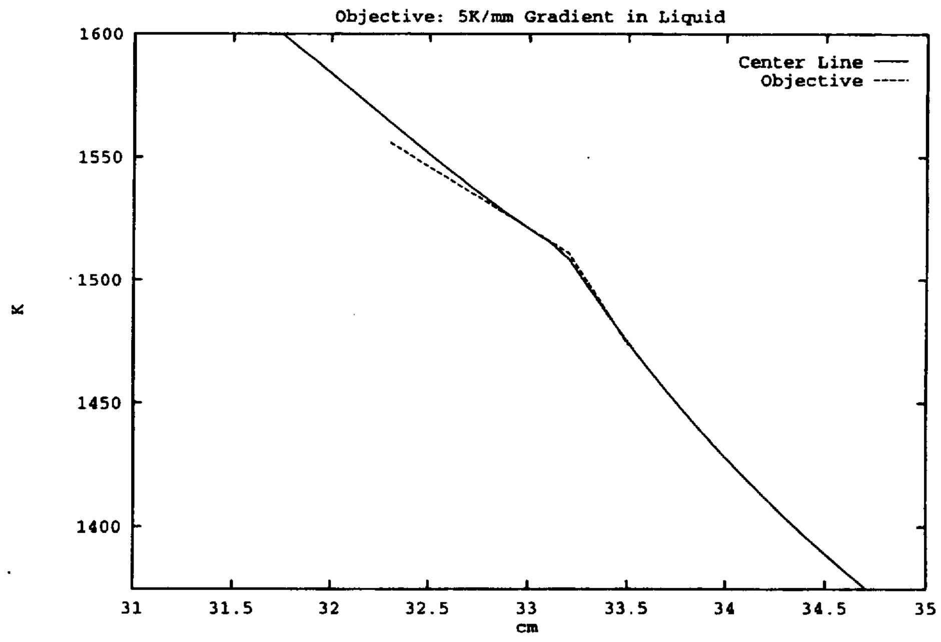


b)

Figure 6.13: Optimization Results for 1,000K/m

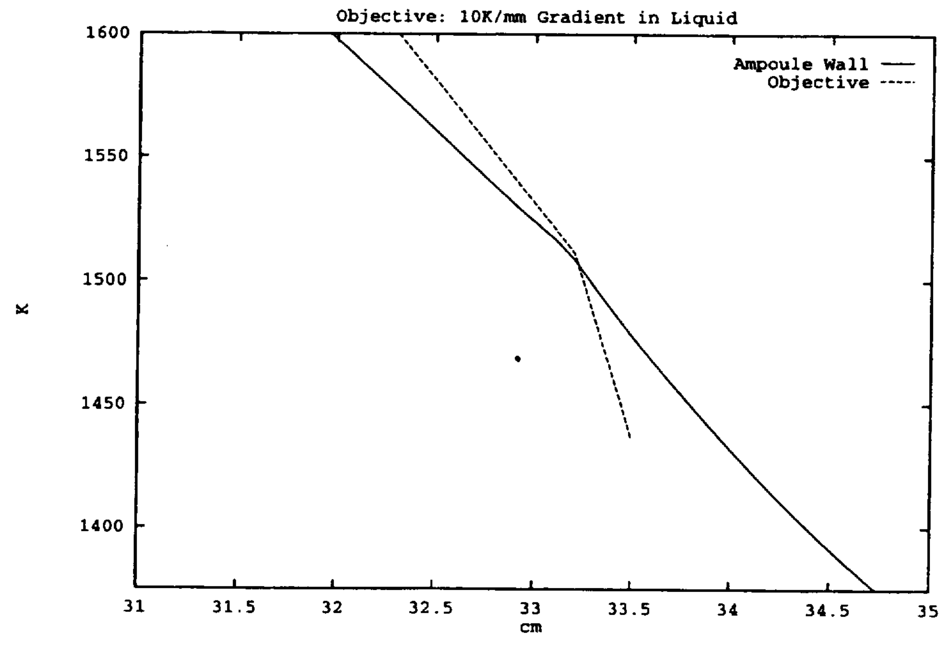


a)

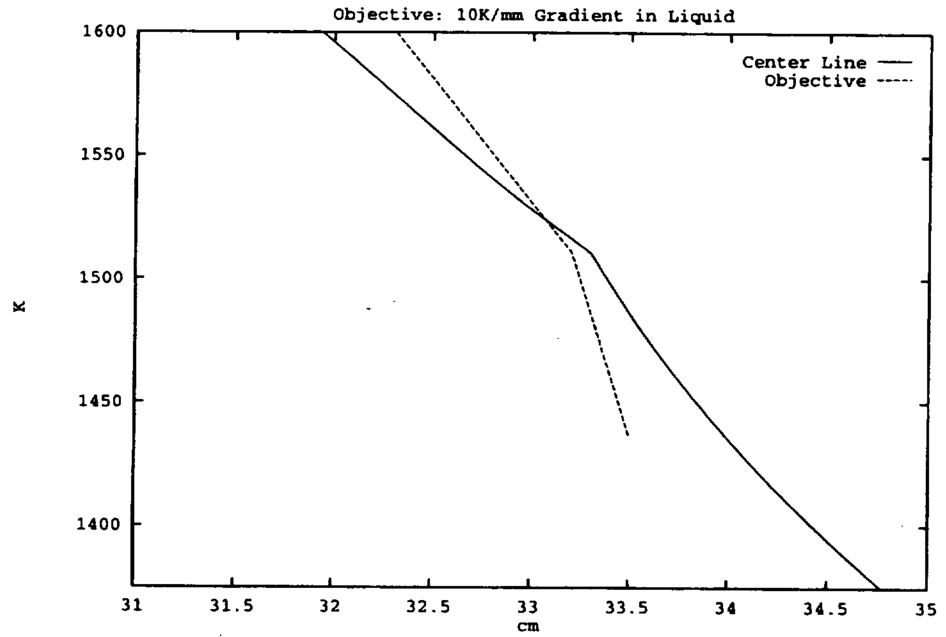


b)

Figure 6.14: Optimization Results for 5,000K/m



a)



b)

Figure 6.15: Optimization Results for 10,000K/m

niques discussed, but the structural properties of the tangent matrix for the present radiative heat transfer problem provide an interesting complication. Using the LU decomposition algorithm from the Yale Sparse Matrix Package (YSMP) [58] and the biconjugate gradient algorithm (BiCG-STAB) with various preconditioners from SparseLib++ [59] our best results came from using an ILU preconditioned biconjugate gradient algorithm.

Solution Scheme	Single System (avg)	Factor/Precondition	Additional RHS (avg)
YSMP	105.7 seconds	104.486 seconds	1.11 seconds
BiCG-STAB w/DP	121.6 seconds	.77 seconds	120.83 seconds
BiCG-STAB w/ILUP	28.7 seconds	.621 seconds	28.1 seconds

Figure 6.16: Linear Solver Comparison

Table 6.16 shows how the various techniques compare for our CGF problem. The first column indicates the algorithm used, where DP stands for diagonal preconditioning and ILUP stands for incomplete LU preconditioning. The next three columns contain CPU times for the solving of a single linear system. The second and third columns are broken down into the time required to factor or precondition the matrix, which is necessary once for each matrix, and the cost of solving each right hand side, respectively.

The results shown in Figure 6.16 have important ramifications for senstivitivity analysis. For example using the adjoint method, where a single right hand side must be solved, the BiCG-STAB algorithm with ILU preconditioning is over three times faster than YSMP. However, using direct differentiation with three (or more) design variables, YSMP would be preferred because it would take 109 seconds (105.7+1.11*3) versus 113 seconds (28.7+28.1*3) for BiCG-STAB with ILU preconditioning.

Chapter 7

Conclusion and Future Work

There were three principle components to the work presented. The first was a formalized approach to integrate optimization, sensitivity information and analysis from a software point of view. Next, we discussed the implications of radiation view factors and introduced some closed form solutions for computing view factors of co-axial axisymmetric cylinders. Finally we combined these two components to solve two example applications.

The work done in integrating software has two components. First, we define a protocol for communication between optimization and simulation packages. Next, we formalize many of the calculations which take place during analysis to accommodate computing sensitivity information. Potential future work for the former would be to develop a set of tools, using our protocol, to allow the user to specify a variety of objectives. These tools would then be able to provide the implicit derivatives of the user objectives (and constraints) with respect to solution and design. The development of graphical user interfaces (GUIs) would make it easier for users to create optimization problem specifications and visualize their results.

Future work related to formalizing the analysis and sensitivity calculations might focus on refining the approach for more diverse systems. In addition, there are many useful algorithms which could be added to the framework. For example, the ability to perform eigenmode analysis would be very useful. The more capabilities added to the system, the larger the potential pool of applications becomes. Another useful component to our simulation system would be pre and post-processing capabilities.

The radiation modeling could also be refined to include more complex formulations. For example, we assume no reflection or transmittance of radiation in our model. Research into computing sensitivities in the presence of reflection and transmittance would be the next logical step.

Finally, the applications themselves could also be refined. In generating our CGF model, we made some assumptions in an effort to simplify our model. One result of this is that the power consumption in our furnace is substantially higher than in the actual furnace (1500-1700W vs. 800W). Much of this is probably the result of simplifications made to the model and a more complete model would be of benefit. In addition, the actual CGF can be assembled with a heat extraction plate which makes it easier to obtain $10K/mm$ gradients in the furnace. The addition of a heat extraction plate to our model would probably allow us to get closer to a $10K/mm$ gradient at the interface.

Overall, this work represents a successful mixing of the components we have discussed. Additional research can build upon this work to gain a more complete understanding of the concepts involved.

Appendix A

Class Definitions

A.1 Optimizer and OptUpdate Class Definitions

Figure A.1 is the abstract class definition for all optimization software packages and corresponds to the function description presented in Figure 4.7. Figure A.2 essentially implements a form of closure which can be called after every design iteration.

```
class   Optimizer
{
    protected:
        Optimizable &opt;
        SimpleSet<OptUpdate *>  updates;

        void DoUpdates();
    public:
        Optimizer(Optimizable &o);
        void Add(OptUpdate *ou) { updates+=ou; }
        enum Status { Done, NotDone };
        virtual Status Iterate(int num, int &suc) = 0;
        virtual Status Optimize();
        virtual Status Optimize(int max);
        virtual ~Optimizer() { }
};
```

Figure A.1: Optimizer Class Definition

```
class   OptUpdate
{
    public:
        virtual void update(Optimizer *) = 0;
};
```

Figure A.2: OptUpdate Class Definition

A.2 Optimizable Class Definitions

Figure A.3 is the class definition for the problem specification. The functional behavior is like that described in 4.9.

```
class   Optimizable
{
    private:
        Optimizable(const Optimizable&);
        Optimizable &operator=(const Optimizable&);
    protected:
        Optimizable();
    public:
        virtual ~Optimizable() { }

        // Optimizable Interface
        virtual DesignSpace     &Space() = 0;
        virtual const Function  &Cost(int grad=1) = 0;
        virtual const FCSet     &Constraints(int grad=1) = 0;
};
```

Figure A.3: Optimizable

A.3 Simulator and Result Class Definitions

```
class   Simulator
{
    protected:
        Design  *design;
    public:
        enum Status { Done, NotDone };
        Simulator(Design *d) : design(d) { }
        Design  *GetDesign() { return design; }

        // Simulator interface
        virtual void        Init() = 0;
        virtual double      Time() const = 0;
        virtual Status      Solve_Init(int sense) = 0;
        virtual Status      TimeStep(int sense, double tmesh) = 0;
        virtual Result*     Result(int sol) = 0;
        virtual ~Simulator() { }
};
```

Figure A.4: Simulator

```
class   Result
{
    protected:
        Simulator   *sim;
    public:
        Result(Simulator *);
        virtual ~Result() { }

        virtual int     NumNodes() = 0;
        virtual int     NumEq() = 0;
        virtual int     NumDOFS(int node) = 0;
        virtual int     EqNum(int node, int dof) = 0;
        virtual double  GetSol(int eq) = 0;
        virtual double  GetDUDB(int eq, int bn) = 0;
        virtual void    GetSol(double *);
        virtual void    GetDUDB(double *, int bn);
//      virtual void    SetSol(int eq, double val) = 0;
};
```

Figure A.5: Result

A.4 Objective Class Definition

```
class   Objective
{
    public:
        enum    Status { Done, NotDone };
        static  inline int Equal(double d1, double d2, double eps=1e-9);

    protected:
        Simulator   &sim;
        Design      &des;

    public:
        Objective(Simulator &, Design &);
        virtual ~Objective() { }

        // Objective interface
        virtual void    Init() = 0;
        virtual void    TimeStep(int) = 0;
        virtual Status  GetStatus() = 0;
        virtual double  NextTime() = 0;
        virtual int     NumCon() = 0;
        virtual void    Results(CacheFunction& cost, const SCFSet& cset) = 0;
};
```

Figure A.6: Result

Appendix B

Mathematica View Factor Code

```
(* Given a two concentric cylinders with longitudinal length H, inner
   radius Ri, outer radius Ro, and annular ends of radius Rt and Rb

Taken from : "Analytic angle factors from the radiant interchange
among the surface elements of two concentric cylinders"

Directed to my attention by Robert McDavid.
*)

(* View factor from outer cylinder to itself *)

Fooaux1[Xo_,Xi_] := If[Xo==Xi, 0,
If[Xi==0, 1/(2*Xo)*(1+2*Xo-Sqrt[(1+4*Xo^2)]),
1/(Pi*Xo)*(Pi*(Xo-Xi)+ArcCos[Xi/Xo]
-Sqrt[(1+4*Xo^2)]*ArcTan[Sqrt[(1+4*Xo^2)(Xo^2-Xi^2)]/Xi]
+2*Xi*ArcTan[2*Sqrt[Xo^2-Xi^2]])]]

Fooaux2[Xo_,Xi_] := 1/(Pi*Xo)*(Pi*(Xo-Xi)+ArcCos[Xi/Xo]
-Sqrt[(1+4*Xo^2)]*ArcTan[Sqrt[(1+4*Xo^2)(Xo^2-Xi^2)]/Xi]
+2*Xi*ArcTan[2*Sqrt[Xo^2-Xi^2]])

Foo[Ro_,Ri_,Rt_,Rb_,H_] = Block[{Xo=Ro/H, Xi=Ri/H}, Fooaux1[Xo, Xi]]

(* View factor from outer cylinder to inner cylinder *)
```

```
Foiaux1[Xo_, Xi_] :=
If[Xo==Xi, 1,
If[Xi==0, 0,
1/(Pi*Xo)*(.5*(Xo^2-Xi^2-1)*ArcCos[Xi/Xo]+Pi*Xi
-(Pi/2)*(Xo^2-Xi^2)-2*Xi*ArcTan[Sqrt[Xo^2-Xi^2]]
+Sqrt[((1+(Xo+Xi)^2)*(1+(Xo-Xi)^2))]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/((1+(Xo-Xi)^2)*(Xo+Xi))]])]]

Foiaux2[Xo_, Xi_] := 1/(Pi*Xo)*(.5*(Xo^2-Xi^2-1)*ArcCos[Xi/Xo]+Pi*Xi
-(Pi/2)*(Xo^2-Xi^2)-2*Xi*ArcTan[Sqrt[Xo^2-Xi^2]]
+Sqrt[((1+(Xo+Xi)^2)*(1+(Xo-Xi)^2))]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/((1+(Xo-Xi)^2)*(Xo+Xi))]])


Foi[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Xo=Ro/H, Xi=Ri/H}, Foiaux1[Xo,Xi]]

(* View factor from inner cylinder to outer cylinder *)
Fio[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Ai=2*Pi*Ri*H, Ao=2*Pi*Ro*H},
(Foi[Ro,Ri,Rt,Rb,H]*Ao)/Ai]

(* View factor from top annulus to outer cylinder *)
Ftoaux1[Xt_,Xo_,Xi_,Yto_] :=
If[Xi==0, (1/(2*Xt^2))*(Sqrt[(1+(Xt+Xo)^2)*(1+(Xt-Xo)^2)]-1-Xo^2+Xt^2),
If[Xt==Xi, 0,
If[Xt==Xo,
1/(Pi*(Xt^2-Xi^2))*(.5*(Xo^2-Xi^2)*(Pi-ArcCos[Xi/Xo])
-2*Xi*(ArcTan[Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]]
-ArcTan[Sqrt[Xo^2-Xi^2]])-.5*ArcCos[Xi/Xt]
+Sqrt[(1+(Xo+Xt)^2)*(1+(Xo-Xt)^2)]
*ArcTan[Sqrt[((1+(Xo+Xt)^2)*(Yto^2-(Xo-Xt)^2))/
((1+(Xo-Xt)^2)*((Xo+Xt)^2-Yto^2))]]
-Sqrt[(1+(Xo+Xi)^2)*(1+(Xo-Xi)^2)]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/
((1+(Xo-Xi)^2)*(Xo+Xi))]]),
1/(Pi*(Xt^2-Xi^2))*(.5*(Xo^2-Xi^2)*(Pi-ArcCos[Xi/Xo])
-2*Xi*(ArcTan[Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]]
-ArcTan[Sqrt[Xo^2-Xi^2]])-.5*ArcCos[Xi/Xt]
+Sqrt[(1+(Xo+Xt)^2)*(1+(Xo-Xt)^2)]
*ArcTan[Sqrt[((1+(Xo+Xt)^2)*(Yto^2-(Xo-Xt)^2))/
((1+(Xo-Xt)^2)*((Xo+Xt)^2-Yto^2))]]
-Sqrt[(1+(Xo+Xi)^2)*(1+(Xo-Xi)^2)]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/((1+(Xo-Xi)^2)*(Xo+Xi))]]
-(Xo^2-Xt^2)*ArcTan[((Xo+Xt)/(Xo-Xt))
*Sqrt[(Yto^2-(Xo-Xt)^2)/((Xo+Xt)^2-Yto^2)]])]]]
```

```
Ftocase1[Xt_,Xo_,Xi_,Yto_] :=
(1/(2*Xt^2))*(Sqrt[(1+(Xt+Xo)^2)*(1+(Xt-Xo)^2)]-1-Xo^2+Xt^2)

Ftocase2[Xt_,Xo_,Xi_,Yto_] := 1/(Pi*(Xt^2-Xi^2))*
(.5*(Xo^2-Xi^2)*(Pi-ArcCos[Xi/Xo])
-2*Xi*(ArcTan[Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]]
-ArcTan[Sqrt[Xo^2-Xi^2]])-.5*ArcCos[Xi/Xt]
+Sqrt[(1+(Xo+Xt)^2)*(1+(Xo-Xt)^2)]
*ArcTan[Sqrt[((1+(Xo+Xt)^2)*(Yto^2-(Xo-Xt)^2))/
((1+(Xo-Xt)^2)*((Xo+Xt)^2-Yto^2))]]
-Sqrt[(1+(Xo+Xi)^2)*(1+(Xo-Xi)^2)]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/
((1+(Xo-Xi)^2)*(Xo+Xi))]])

Ftocase3[Xt_,Xo_,Xi_,Yto_] := 1/(Pi*(Xt^2-Xi^2))*
(.5*(Xo^2-Xi^2)*(Pi-ArcCos[Xi/Xo])
-2*Xi*(ArcTan[Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]]
-ArcTan[Sqrt[Xo^2-Xi^2]])-.5*ArcCos[Xi/Xt]
+Sqrt[(1+(Xo+Xt)^2)*(1+(Xo-Xt)^2)]
*ArcTan[Sqrt[((1+(Xo+Xt)^2)*(Yto^2-(Xo-Xt)^2))/
((1+(Xo-Xt)^2)*((Xo+Xt)^2-Yto^2))]]
-Sqrt[(1+(Xo+Xi)^2)*(1+(Xo-Xi)^2)]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/((1+(Xo-Xi)^2)*(Xo+Xi))]]
-(Xo^2-Xt^2)*ArcTan[((Xo+Xt)/(Xo-Xt))
*Sqrt[(Yto^2-(Xo-Xt)^2)/((Xo+Xt)^2-Yto^2)]])

Ftoaux2[Xt_,Xo_,Xi_,Yto_] :=
1/(Pi*(Xt^2-Xi^2))*(.5*(Xo^2-Xi^2)*(Pi-ArcCos[Xi/Xo])
-2*Xi*(ArcTan[Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]]
-ArcTan[Sqrt[Xo^2-Xi^2]])-.5*ArcCos[Xi/Xt]
+Sqrt[(1+(Xo+Xt)^2)*(1+(Xo-Xt)^2)]
*ArcTan[Sqrt[((1+(Xo+Xt)^2)*(Yto^2-(Xo-Xt)^2))/
((1+(Xo-Xt)^2)*((Xo+Xt)^2-Yto^2))]]
-Sqrt[(1+(Xo+Xi)^2)*(1+(Xo-Xi)^2)]
*ArcTan[Sqrt[((1+(Xo+Xi)^2)*(Xo-Xi))/((1+(Xo-Xi)^2)*(Xo+Xi))]]
-(Xo^2-Xt^2)*ArcTan[((Xo+Xt)/(Xo-Xt))
*Sqrt[(Yto^2-(Xo-Xt)^2)/((Xo+Xt)^2-Yto^2)]])


Fto[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Xt=Rt/H, Xo=Ro/H, Xi=Ri/H,
Yto=Sqrt[Xo^2-Xi^2]+Sqrt[Xt^2-Xi^2]}, Ftoaux1[Xt,Xo,Xi,Yto]]

(* View factor from outer cylinder to top annulus *)
```

```
Fot[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Ao=2*Pi*Ro*H, At=Pi*Rt^2-Pi*Ri^2},
(Fto[Ro,Ri,Rt,Rb,H]*At)/Ao]

(* Symmetry between top and bottom *)
Fbo[Ro_,Ri_,Rt_,Rb_,H_] := Fto[Ro,Ri,Rb,Rt,H]

(* View factor from outer cylinder to bottom annulus *)
Fob[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Ab,Ao},
Ab = Pi*Rb^2-Pi*Ri^2; Ao = 2*Pi*Ro*H;
(Fbo[Ro,Ri,Rt,Rb,H]*Ab)/Ao]

(* View factor from top annulus to bottom annulus *)
Ftbaux1[Xt_,Xb_,Xi_,Ytb_] :=
If[Xb==Xi, 0,
If[Xi==0, (1/(2*Xt^2))*(1+Xt^2+Xb^2-Sqrt[(1+(Xt+Xb)^2)*(1+(Xt-Xb)^2)]),
1/(Pi*(Xt^2-Xi^2))*(.5*(Xt^2-Xi^2)*ArcCos[Xi/Xb]
+.5*(Xb^2-Xi^2)*ArcCos[Xi/Xt]
+2*Xi*(ArcTan[Sqrt[Xt^2-Xi^2]+Sqrt[Xb^2-Xi^2]]
-ArcTan[Sqrt[Xt^2-Xi^2]]-ArcTan[Sqrt[Xb^2-Xi^2]])
-Sqrt[(1+(Xt+Xb)^2)*(1+(Xt-Xb)^2)]
*ArcTan[Sqrt[((1+(Xt+Xb)^2)*(Ytb^2-(Xt-Xb)^2))/
((1+(Xt-Xb)^2)*((Xt+Xb)^2-Ytb^2))]]
+Sqrt[(1+(Xt+Xi)^2)*(1+(Xt-Xi)^2)]
*ArcTan[Sqrt[((1+(Xt+Xi)^2)*(Xt-Xi))/
((1+(Xt-Xi)^2)*(Xt+Xi))]]
+Sqrt[(1+(Xb+Xi)^2)*(1+(Xb-Xi)^2)]
*ArcTan[Sqrt[((1+(Xb+Xi)^2)*(Xb-Xi))/((1+(Xb-Xi)^2)*(Xb+Xi))]])]]

Ftbaux2[Xt_,Xb_,Xi_,Ytb_] := 1/(Pi*(Xt^2-Xi^2))*(.5*(Xt^2-Xi^2)*ArcCos[Xi/Xb]
+.5*(Xb^2-Xi^2)*ArcCos[Xi/Xt]
+2*Xi*(ArcTan[Sqrt[Xt^2-Xi^2]+Sqrt[Xb^2-Xi^2]]
-ArcTan[Sqrt[Xt^2-Xi^2]]-ArcTan[Sqrt[Xb^2-Xi^2]])
-Sqrt[(1+(Xt+Xb)^2)*(1+(Xt-Xb)^2)]
*ArcTan[Sqrt[((1+(Xt+Xb)^2)*(Ytb^2-(Xt-Xb)^2))/
((1+(Xt-Xb)^2)*((Xt+Xb)^2-Ytb^2))]]
+Sqrt[(1+(Xt+Xi)^2)*(1+(Xt-Xi)^2)]
*ArcTan[Sqrt[((1+(Xt+Xi)^2)*(Xt-Xi))/
((1+(Xt-Xi)^2)*(Xt+Xi))]]
+Sqrt[(1+(Xb+Xi)^2)*(1+(Xb-Xi)^2)]
*ArcTan[Sqrt[((1+(Xb+Xi)^2)*(Xb-Xi))/((1+(Xb-Xi)^2)*(Xb+Xi))]])


Ftb[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Xt=Rt/H, Xb=Rb/H, Xi=Ri/H,
Ytb=Sqrt[Xt^2-Xi^2]+Sqrt[Xb^2-Xi^2]}, Ftbaux1[Xt,Xb,Xi,Ytb]]
```

```
(* View factor from bottom annulus to top annulus *)
Fbt[Ro_,Ri_,Rt_,Rb_,H_] := Block[{At=Pi*Rt^2-Pi*Ri^2, Ab=Pi*Rb^2-Pi*Ri^2},
(Ftb[Ro,Ri,Rt,Rb,H]*At)/Ab]

(* View factor from top annulus to inner cylinder *)
Fti[Ro_,Ri_,Rt_,Rb_,H_] := 1-Fto[Ro,Ri,Rt,Rb,H]-Ftb[Ro,Ri,Rt,Rb,H]

(* View factor from inner cylinder to top annulus *)
Fint[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Ai=2*Pi*Ri*H, At=Pi*Rt^2-Pi*Ri^2},
(Fti[Ro,Ri,Rt,Rb,H]*At)/Ai]

(* Symmetry between top and bottom *)
Fbi[Ro_,Ri_,Rt_,Rb_,H_] := Fti[Ro,Ri,Rb,Rt,H]

(* View factor from outer cylinder to bottom annulus *)
Fib[Ro_,Ri_,Rt_,Rb_,H_] := Block[{Ab=Pi*Rb^2-Pi*Ri^2, Ai=2*Pi*Ri*H},
(Fbi[Ro,Ri,Rt,Rb,H]*Ab)/Ai]


(* Additional derivations by me *)

(* These functions require the ones from the paper by Blockman *)
<<paper.m

(* Given a cylinder labeled as follows ...

a => top annular disk
b, c, d => inner cylinder segments (top to bottom)
e => bottom annular disk
f, g, h => outer cylinder segments (bottom to top)

We have the following relations...

*)

(*
===== Section 1 =====

The following section contains the list of explicit view factors we
already know from paper.m
*)

Fba[Ri_,Ro_,Hb_,Hc_,Hd_] := Fint[Ro,Ri,Ro,Ro,Hb]
Fbh[Ri_,Ro_,Hb_,Hc_,Hd_] := Fio[Ro,Ri,Ro,Ro,Hb]
```

```
Fby[Ri_,Ro_,Hb_,Hc_,Hd_] := Fib[Ro,Ri,Ro,Ro,Hb]

Fcy[Ri_,Ro_,Hb_,Hc_,Hd_] := Fint[Ro,Ri,Ro,Ro,Hc]
Fcg[Ri_,Ro_,Hb_,Hc_,Hd_] := Fio[Ro,Ri,Ro,Ro,Hc]
Fcx[Ri_,Ro_,Hb_,Hc_,Hd_] := Fib[Ro,Ri,Ro,Ro,Hc]

Fab[Ri_,Ro_,Hb_,Hc_,Hd_] := Fti[Ro,Ri,Ro,Ro,Hb]
Fgc[Ri_,Ro_,Hb_,Hc_,Hd_] := Foi[Ro,Ri,Ro,Ro,Hc]

Fdx[Ri_,Ro_,Hb_,Hc_,Hd_] := Fint[Ro,Ri,Ro,Ro,Hd]
Fdf[Ri_,Ro_,Hb_,Hc_,Hd_] := Fio[Ro,Ri,Ro,Ro,Hd]
Ffd[Ri_,Ro_,Hb_,Hc_,Hd_] := Foi[Ro,Ri,Ro,Ro,Hd]
Fde[Ri_,Ro_,Hb_,Hc_,Hd_] := Fib[Ro,Ri,Ro,Ro,Hd]

Fxc[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbi[Ro,Ri,Ro,Ro,Hc]
Fcx[Ri_,Ro_,Hb_,Hc_,Hd_] := Fib[Ro,Ri,Ro,Ro,Hc]

Fxg[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbo[Ro,Ri,Ro,Ro,Hc]
Fgx[Ri_,Ro_,Hb_,Hc_,Hd_] := Fob[Ro,Ri,Ro,Ro,Hc]

Fxa[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbt[Ro,Ri,Ro,Ro,Hb+Hc]

Fah[Ri_,Ro_,Hb_,Hc_,Hd_] := Fto[Ro,Ri,Ro,Ro,Hb]
Fha[Ri_,Ro_,Hb_,Hc_,Hd_] := Fot[Ro,Ri,Ro,Ro,Hb]

Fef[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbo[Ro,Ri,Ro,Ro,Hd]
Ffe[Ri_,Ro_,Hb_,Hc_,Hd_] := Fob[Ro,Ri,Ro,Ro,Hd]

Fag[Ri_,Ro_,Hb_,Hc_,Hd_] := Fto[Ro,Ri,Ro,Ro,Hb+Hc]-Fto[Ro,Ri,Ro,Ro,Hb]
Feg[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbo[Ro,Ri,Ro,Ro,Hc+Hd]-Fbo[Ro,Ri,Ro,Ro,Hd]

Fhy[Ri_,Ro_,Hb_,Hc_,Hd_] := Fob[Ro,Ri,Ro,Ro,Hb]
Fyh[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbo[Ro,Ri,Ro,Ro,Hb]

Fhh[Ri_,Ro_,Hb_,Hc_,Hd_] := Foo[Ro,Ri,Ro,Ro,Hb]
Fgg[Ri_,Ro_,Hb_,Hc_,Hd_] := Foo[Ro,Ri,Ro,Ro,Hc]
Fff[Ri_,Ro_,Hb_,Hc_,Hd_] := Foo[Ro,Ri,Ro,Ro,Hd]

Fed[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbi[Ro,Ri,Ro,Ro,Hd]
Fde[Ri_,Ro_,Hb_,Hc_,Hd_] := Fib[Ro,Ri,Ro,Ro,Hd]

(*
===== Section 2 =====
This section contains functions related to composite surfaces.
*)

Fxh[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbo[Ro,Ri,Ro,Ro,Hb+Hc]-Fbo[Ro,Ri,Ro,Ro,Hc]

Feh[Ri_,Ro_,Hb_,Hc_,Hd_] :=
Fbo[Ro,Ri,Ro,Ro,Hb+Hc+Hd]-Fbo[Ro,Ri,Ro,Ro,Hc+Hd]

Feb[Ri_,Ro_,Hb_,Hc_,Hd_] :=
Fbi[Ro,Ri,Ro,Ro,Hb+Hc+Hd]-Fbi[Ro,Ri,Ro,Ro,Hc+Hd]

(*
===== Section 3 =====

These functions come from reversing existing relations.
*)

Fhb[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ah=2*Pi*Ro*Hb,Ab=2*Pi*Ri*Hb},
Ab*Fbh[Ri,Ro,Hb,Hc,Hd]/Ah]

Fga[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ag=2*Pi*Ro*Hc,Aa=Pi*Ro^2-Pi*Ri^2},
Aa*Fag[Ri,Ro,Hb,Hc,Hd]/Ag]

Fge[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ag=2*Pi*Ro*Hc,Ae=Pi*Ro^2-Pi*Ri^2},
Ae*Feg[Ri,Ro,Hb,Hc,Hd]/Ag]

(*
===== Section 4 =====

We shouldn't see calls to any of the functions in paper.m from now on
*)

Fxb[Ri_,Ro_,Hb_,Hc_,Hd_] := 1-(Fxc[Ri, Ro, Hb, Hc, Hd]+
Fxg[Ri, Ro, Hb, Hc, Hd]+Fxh[Ri, Ro, Hb, Hc, Hd]+Fxa[Ri, Ro, Hb, Hc, Hd])
Fbx[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ax=Pi*Ro^2-Pi*Ri^2,Ab=2*Pi*Ri*Hb},
Ax*Fxb[Ri,Ro,Hb,Hc,Hd]/Ab]

Fbg[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ax=Pi*Ro^2-Pi*Ri^2,Ab=2*Pi*Ri*Hb},
1-(Fbh[Ri,Ro,Hb,Hc,Hd]+Fba[Ri,Ro,Hb,Hc,Hd]+Fbx[Ri,Ro,Hb,Hc,Hd])]

Fgb[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ag=2*Pi*Ro*Hc, Ab=2*Pi*Ri*Hb},
Ab*Fbg[Ri,Ro,Hb,Hc,Hd]/Ag]

Fch[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbg[Ri,Ro,Hc,Hb,Hd]
```

```
Fhc[Ri_,Ro_,Hb_,Hc_,Hd_] := Fgb[Ri,Ro,Hc,Hb,Hd]

Fdg[Ri_,Ro_,Hb_,Hc_,Hd_] := Fch[Ri,Ro,Hc,Hd,Hb]
Fgd[Ri_,Ro_,Hb_,Hc_,Hd_] := Fhc[Ri,Ro,Hc,Hd,Hb]

Fhd[Ri_,Ro_,Hb_,Hc_,Hd_] := Ffb[Ri,Ro,Hd,Hc,Hb]

Fdh[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbf[Ri,Ro,Hd,Hc,Hb]

Fbe[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ae=Pi*Ro^2-Pi*Ri^2, Ab=2*Pi*Ri*Hb},
Ae*Feb[Ri,Ro,Hb,Hc,Hd]/Ab]

Fbf[Ri_,Ro_,Hb_,Hc_,Hd_] := 1-(Fba[Ri,Ro,Hb,Hc,Hd]
+Fbh[Ri,Ro,Hb,Hc,Hd]+Fbg[Ri,Ro,Hb,Hc,Hd]+Fbe[Ri,Ro,Hb,Hc,Hd])

Ffb[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Af=2*Pi*Ro*Hd, Ab=2*Pi*Ri*Hb},
Ab*Fbf[Ri,Ro,Hb,Hc,Hd]/Af]

Fhx[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ah=2*Pi*Ro*Hb, Ax=Pi*Ro^2-Pi*Ri^2},
Ax*Fxh[Ri,Ro,Hb,Hc,Hd]/Ah]

Fhg[Ri_,Ro_,Hb_,Hc_,Hd_] := 1-(Fhx[Ri,Ro,Hb,Hc,Hd]+Fhc[Ri,Ro,Hb,Hc,Hd]
+Fhb[Ri,Ro,Hb,Hc,Hd]+Fha[Ri,Ro,Hb,Hc,Hd]+Fhh[Ri,Ro,Hb,Hc,Hd])

Fgh[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ah=2*Pi*Ro*Hb, Ag=2*Pi*Ro*Hc},
Ah*Fhg[Ri,Ro,Hb,Hc,Hd]/Ag]

Fhe[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ah=2*Pi*Ro*Hb, Ae=Pi*Ro^2-Pi*Ri^2},
Ae*Feh[Ri,Ro,Hb,Hc,Hd]/Ah]

Fhf[Ri_,Ro_,Hb_,Hc_,Hd_] := 1-(Fha[Ri,Ro,Hb,Hc,Hd]+Fhb[Ri,Ro,Hb,Hc,Hd]
+Fhc[Ri,Ro,Hb,Hc,Hd]+Fhd[Ri,Ro,Hb,Hc,Hd]+Fhe[Ri,Ro,Hb,Hc,Hd]
+Fhh[Ri,Ro,Hb,Hc,Hd]+Fhg[Ri,Ro,Hb,Hc,Hd])

Fac[Ri_,Ro_,Hb_,Hc_,Hd_] := Fab[Ri,Ro,Hb+Hc,0,0]-Fab[Ri,Ro,Hb,Hc,0]
Fca[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Aa=Pi*Ro^2-Pi*Ri^2, Ac=2*Pi*Ri*Hc},
Aa*Fac[Ri,Ro,Hb,Hc,Hd]/Ac]

Fad[Ri_,Ro_,Hb_,Hc_,Hd_] := Fac[Ri,Ro,Hb+Hc,Hd,0]
Fda[Ri_,Ro_,Hb_,Hc_,Hd_] := Fca[Ri,Ro,Hb+Hc,Hd,0]


Fec[Ri_,Ro_,Hb_,Hc_,Hd_] := Fed[Ri,Ro,0,Hb,Hc+Hd]-Fed[Ri,Ro,Hb,Hc,Hd]
Fce[Ri_,Ro_,Hb_,Hc_,Hd_] := Block[{Ac=2*Pi*Ri*Hc, Ae=Pi*Ro^2-Pi*Ri^2},
Ae*Fec[Ri,Ro,Hb,Hc,Hd]/Ac]

Fcf[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbg[Ri,Ro,Hc,Hd,Hb]
Ffc[Ri_,Ro_,Hb_,Hc_,Hd_] := Fgb[Ri,Ro,Hc,Hd,Hb]

(*
Fdf[Ri_,Ro_,Hb_,Hc_,Hd_] := Fbh[Ri,Ro,Hd,Hc,Hb]
Ffd[Ri_,Ro_,Hb_,Hc_,Hd_] := Fhb[Ri,Ro,Hd,Hc,Hb]
*)

(* Finally, the different configurations for two concentric cylinders *)
<<complex.m

CCConfig1[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fdh[Ri,Ro,L1,Z2-(Z1+L1),L2]
CCConfig2[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fch[Ri,Ro,L1,L2,0]
CCConfig3[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z2-Z1, Hc=Z1+L1-Z2,
Hd=Z2+L2-(Z1+L1), Aa=Pi*Ro^2-Pi*Ri^2, Af=2*Pi*Ro*(Z2+L2-(Z1+L1)),
Ae=Pi*Ro^2-Pi*Ri^2, Acd=2*Pi*Ri*L2}, 1-(
(Aa/Acd)*(Fac[Ri,Ro,Hb,Hc,Hd]+Fad[Ri,Ro,Hb,Hc,Hd])+
(Af/Acd)*(Ffc[Ri,Ro,Hb,Hc,Hd]+Ffd[Ri,Ro,Hb,Hc,Hd])+
(Ae/Acd)*(Fec[Ri,Ro,Hb,Hc,Hd]+Fed[Ri,Ro,Hb,Hc,Hd]))]

(*
CCConfig3[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z2-Z1, Hc=Z1+L1-Z2,
Hd=Z2+L2-(Z1+L1), Aa=Pi*Ro^2-Pi*Ri^2, Af=2*Pi*Ro*(Z2+L2-(Z1+L1)),
Ae=Pi*Ro^2-Pi*Ri^2, Acd=2*Pi*Ri*L2}, { {Hb, Hc, Hd, Aa, Af, Ae, Acd},
(Aa/Acd), {Fac[Ri,Ro,Hb,Hc,Hd], Fad[Ri,Ro,Hb,Hc,Hd]},
(Af/Acd), {Ffc[Ri,Ro,Hb,Hc,Hd], Ffd[Ri,Ro,Hb,Hc,Hd]}, {Ri,Ro,Hb,Hc,Hd},
(Ae/Acd), {Fec[Ri,Ro,Hb,Hc,Hd], Fed[Ri,Ro,Hb,Hc,Hd]}}]
*)

CCConfig4[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z2-Z1, Hc=L2, Hd=0},
1-(Fcx[Ri,Ro,Hb,Hc,Hd]+Fca[Ri,Ro,Hb,Hc,Hd])]
CCConfig5[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z2-Z1, Hc=L2, Hd=Z1+L1-(Z2+L2)},
Fch[Ri,Ro,Hb,Hc,Hd]+Fcg[Ri,Ro,Hb,Hc,Hd]+Fcf[Ri,Ro,Hb,Hc,Hd]]
CCConfig6[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z1-Z2, Hc=L1, Hd=Z2+L2-(Z1+L1),
Ag=2*Pi*Ro*Hc, Abcd=2*Pi*Ri*L2},
Ag*(Fgb[Ri,Ro,Hb,Hc,Hd]+Fgc[Ri,Ro,Hb,Hc,Hd]+Fgd[Ri,Ro,Hb,Hc,Hd])/Abcd]
CCConfig7[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z1-Z2, Hc=L1, Hd=0,
Ag=2*Pi*Ro*Hc, Abc=2*Pi*Ri*L2},
Ag*(Fgb[Ri,Ro,Hb,Hc,Hd]+Fgc[Ri,Ro,Hb,Hc,Hd])/Abc]
CCConfig8[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=Z1-Z2, Hc=Z2+L2-Z1,
Hd=Z1+L1-(Z2+L2), Aa=Pi*Ro^2-Pi*Ri^2, Ah=2*Pi*Ro*Hb,
Ae=Pi*Ro^2-Pi*Ri^2, Abc=2*Pi*Ri*L2}, 1-(
(Aa/Abc)*(Fab[Ri,Ro,Hb,Hc,Hd]+Fac[Ri,Ro,Hb,Hc,Hd])+
(Ah/Abc)*(Fhb[Ri,Ro,Hb,Hc,Hd]+Fhc[Ri,Ro,Hb,Hc,Hd])+
(Ae/Abc)*(Feb[Ri,Ro,Hb,Hc,Hd]+Fec[Ri,Ro,Hb,Hc,Hd]))]
CCConfig9[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fbg[Ri,Ro,L2,L1,0]
CCConfig10[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fbf[Ri,Ro,L2,Z1-(Z2+L2),L1]
CCConfig11[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=L2, Hc=L1-L2, Hd=0},
Fbh[Ri,Ro,Hb,Hc,Hd]+Fbg[Ri,Ro,Hb,Hc,Hd]]
CCConfig12[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fbh[Ri,Ro,L1,0,0]
CCConfig13[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Block[{Hb=L1, Hc=L2-L1, Hd=0,
Ah=2*Pi*Ro*Hb, Abc=2*Pi*Ri*L2},
Ah*(Fhb[Ri,Ro,Hb,Hc,Hd]+Fhc[Ri,Ro,Hb,Hc,Hd])/Abc]
CCConfig14[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fhf[Ri,Ro,L1,Z2-(Z1+L1),L2]
CCConfig15[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fhg[Ri,Ro,L1,L2,0]
CCConfig16[Ri_,Ro_,Z1_,L1_,Z2_,L2_] := Fhh[Ri,Ro,L1,0,0]

CDConfig1[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fca[Rc,Ro,Zc-Zd,L,0]
CDConfig2[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fca[Rc,Ro,Zc-Zd,L,0]-Fca[Rc,Ri,Zc-Zd,L,0]
CDConfig3[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fba[Rc,Ro,L,0,0]
CDConfig4[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fba[Rc,Ro,L,0,0]-Fba[Rc,Ri,L,0,0]
CDConfig5[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fce[Rc,Ro,0,L,Zd-(Zc+L)]
CDConfig6[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fce[Rc,Ro,0,L,Zd-(Zc+L)]-Fce[Rc,Ri,0,L,Zd-(Zc+L)]
CDConfig7[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fde[Rc,Ro,0,0,L]
CDConfig8[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fde[Rc,Ro,0,0,L]-Fde[Rc,Ri,0,0,L]

CDConfig9[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fga[Ri,Rc,Zc-Zd,L,0]
(* CDConfig10[Rc_,Ri_,Ro_,Zd_,Zc_,L_] :=
Fga[Ri,Rc,Zc-Zd,L,0]-Fga[Ro,Rc,Zc-Zd,L,0] *)
CDConfig10[Rc_,Ri_,Ro_,Zd_,Zc_,L_] :=
Block[{Ao=2.0*Rc*L*Pi,At=Pi*Ro^2-Pi*Ri^2},
At*(Fto[Rc,Ri,Ro,Rc,(Zc-Zd)+L]-Fto[Rc,Ri,Ro,Rc,Zc-Zd])/Ao]
CDConfig11[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fha[Ri,Rc,L,0,0]
(* CDConfig12[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fha[Ri,Rc,L,0,0]-Fha[Ro,Rc,L,0,0] *)
CDConfig12[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fot[Rc,Ri,Ro,Rc,L]
CDConfig13[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fge[Ri,Rc,0,L,Zd-(Zc+L)]
(* CDConfig14[Rc_,Ri_,Ro_,Zd_,Zc_,L_] :=
Fge[Ri,Rc,0,L,Zd-(Zc+L)]-Fge[Ro,Rc,0,L,Zd-(Zc+L)] *)
CDConfig14[Rc_,Ri_,Ro_,Zd_,Zc_,L_] :=
Block[{Ao=2.0*Rc*L*Pi,Ab=Pi*Ro^2-Pi*Ri^2},
Ab*(Fbo[Rc,Ri,Rc,Ro,Zd-Zc]-Fbo[Rc,Ri,Rc,Ro,Zd-(Zc+L)])/Ao]
CDConfig15[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Ffe[Ri,Rc,0,0,L]
(* CDConfig16[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Ffe[Ri,Rc,0,0,L]-Ffe[Ro,Rc,0,0,L] *)
CDConfig16[Rc_,Ri_,Ro_,Zd_,Zc_,L_] := Fob[Rc,Ri,Rc,Ro,L]

(* Assuming Roa < Rob *)
(* DDConfig[Ria_,Roa_,Rib_,Rob_,H_] :=
Ftb[0,Ria,Roa,Rob,H]-Ftb[0,Ria,Roa,Rib,H] *)
DDConfig[Ria_,Roa_,Rib_,Rob_,H_] :=
Block[{Af=2.0*Pi*Rob*H,Ad=Pi*Rib^2,Aa=Pi*Roa^2-Pi*Ria^2,
Ffac=Fto[Rob,0,Roa,Rob,H], Ffc=Fto[Rob,0,Ria,Rob,H],
Fdac=Fbo[Rob,0,Rob,Rob,H], Fdc=Fbo[Rob,0,Rob,Rib,H]},
1.0-(Af*(Ffac-Ffc)-Ad*(Fdac-Fdc))/Aa]

DD[r1_,r2_,h_] :=
Block[{R1 = r1/h, R2 = r2/h, X = 1+(1+R2^2)/R1^2},
(1/2)*(X-Sqrt[X^2-4*(R2/R1)^2])]

DDConfig1[l1_,l2_,l3_,l4_,h_] :=
Block[{r1 = l1, r2 = l2, r3 = l3, r4 = l4,
R43L1234 = DD[r3+r4,l1+l2+l3+l4,h], R43L234 = DD[r3+r4,l2+l3+l4,h],
R4L1234 = DD[r4,l1+l2+l3+l4,h], R4L234 = DD[r4,l2+l3+l4,h],
R43L1 = R43L1234-R43L234, R4L1 = R4L1234-R4L234,
AR34 = Pi*(r3+r4)^2, AR4 = Pi*r4^2,
AL1 = Pi*(l1+l2+l3+l4)^2-Pi*(l2+l3+l4)^2},
(AR34*R43L1)/AL1-(AR4*R4L1)/AL1]
DDConfig2[l1_,l2_,l3_,l4_,h_] :=
Block[{r1 = l1, r2 = l2, r3 = l3, r4 = l4,
R34L234 = DD[r3+r4,l2+l3+l4,h], R34L34 = DD[r3+r4,l3+l4,h],
R4L234 = DD[r4,l2+l3+l4,h], R4L34 = DD[r4,l3+l4,h],
R34L2 = R34L234-R34L34, R4L2 = R4L234-R4L34,
AR34 = Pi*(r3+r4)^2, AR4 = Pi*r4^2,
AL2 = Pi*(l2+l3+l4)^2-Pi*(l3+l4)^2},
(AR34*R34L2)/AL2-(AR4*R4L2)/AL2]
DDConfig3[l1_,l2_,l3_,l4_,h_] :=
Block[{r1 = l1, r2 = l2, r3 = l3, r4 = l4,
R432L1234 = DD[r2+r3+r4,l1+l2+l3+l4,h],
R432L34 = DD[r2+r3+r4,l3+l4,h],
R4L1234 = DD[r4,l1+l2+l3+l4,h], R4L34 = DD[r4,l3+l4,h],
R432L12 = R432L1234-R432L34, R4L12 = R4L1234-R4L34,
AR432 = Pi*(r4+r3+r2)^2, AR4 = Pi*r4^2,
AL12 = Pi*(l1+l2+l3+l4)^2-Pi*(l3+l4)^2},
(AR432*R432L12)/AL12-(AR4*R4L12)/AL12]
DDConfig4[l1_,l2_,l3_,l4_,h_] :=
Block[{r1 = l1, r2 = l2, r3 = l3, r4 = l4,
R234L1234 = DD[r2+r3+r4,l1+l2+l3+l4,h],
R234L4 = DD[r2+r3+r4,l4,h],
R34L1234 = DD[r3+r4,l1+l2+l3+l4,h], R34L4 = DD[r3+r4,l4,h],
R234L123 = R234L1234-R234L4, R34L123 = R34L1234-R34L4,
AR234 = Pi*(r2+r3+r4)^2, AR34 = Pi*(r3+r4)^2,
AL123 = Pi*(l1+l2+l3+l4)^2-Pi*l4^2},
(AR234*R234L123)/AL123-(AR34*R34L123)/AL123]
```

Appendix C

Material Properties

The material properties in this chapter were taken from Kingery *et al.* [60], Callister [61] and Pehlke *et al.* [62].

Temperature K	Conductivity $\frac{W}{mK}$
400	29.7
500	23.1
600	19.3
700	16.2
800	13.9
900	12.4
1000	11.0
1100	9.9
1200	9.1
1300	8.3
1400	7.7
1500	7.1
1511	18.0
1512	18.0

Table C.1: Gallium Arsenide Thermal Conductivity

Property	Value
Latent Heat of Fusion	175 $\frac{J}{g}$
Density	5.71 $\frac{g}{mm^3}$

Table C.2: Miscellaneous Properties for Gallium Arsenide

Temperature K	Conductivity $\frac{W}{mK}$
400	129.5
500	113.0
600	98.8
700	88.1
800	79.7
900	72.5
1100	66.8
1200	57.9
1400	52.0
1600	47.6

Table C.3: Graphite Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
400	138
500	130
600	128
700	123
800	120
900	112
1000	108
1100	106
1200	104
1300	100
1400	96
1400	94
1600	90

Table C.4: Molybdenum Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
523	.24
873	.27
1273	.34
1673	.38

Table C.5: Zirconia FBD Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
673	.09
1073	.11
1373	.14
1673	.19
1923	.25

Table C.6: Zirconia ZYC Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
100	.69
200	1.14
200	1.38
400	1.51
500	1.62
600	1.75
700	1.92
800	2.17
900	2.48
1000	2.87
1100	3.36
1200	4.00
1300	4.82
1400	6.2

Table C.7: Quartz Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
294	218.8
400	168.3
600	105.2
800	63.12
1074	29.45
1200	21.03

Table C.8: Beryllia Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
589	.3
700	.33
811	.38
923	.43
1200	.62

Table C.9: MIN-K TE-1800 Thermal Conductivity

Temperature K	Conductivity $\frac{W}{mK}$
317	35.9
417	26.4
617	15.8
817	10.4
1017	7.8
1216	6.5
1516	5.6

Table C.10: Alumina Thermal Conductivity

Temperature K	$\parallel$ Conductivity $\frac{W}{mK}$	$\perp$ Conductivity $\frac{W}{mK}$
298	104.6	1.7
773	77.1	2.1
1273	62.8	2.5

Table C.11: Pyrolytic Boron Nitride Thermal Conductivity

Property	Value
$\parallel$ Thermal Conductivity	1.7 $\frac{W}{mK}$
$\perp$ Thermal Conductivity	1.0 $\frac{W}{mK}$

Table C.12: Boron Nitride, High Boron Content (HBC) Thermal Conductivity

Property	Value
Ambient Temperature (T_∞)	300 K
Convection Coefficient h_{air}	.37 $\frac{W}{m^2K}$
Convection Coefficient h_{water}	5.0 $\frac{W}{m^2K}$
Stefan-Boltzman Constant	$5.67 * 10^{-8}$

Table C.13: Miscellaneous Properties and Coefficients

Bibliography

[1] D. S. Watkins, *Fundamentals of Matrix Computation.* New York, New York: Wiley, 1991.

[2] Gene H. Golub and Charles F. Van Loan, *Matrix Computations.* Baltimore: Johns-Hopkins, 1989.

[3] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Donagarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the solution of Linear Systems: Building blocks for Iterative Methods.* 1993.

[4] S. K. Sharma, J. W. B. Jr., and H. S. Chadha, "A client-server approach for distributed finite element analysis," *Advances in Engineering Software*, no. 17, pp. 69–78, 1993.

[5] K. K. Choi and K.-H. Chang, "Design sensitivity analysis and optimization tool for concurrent engineering," in *Concurrent Engineering: Tools and Technologies for Mechanical System Design*, 1993.

[6] D. A. Tortorelli, "Design sensitivity analysis for coupled systems and their application to concurrent engineering," in *Concurrent Engineering: Tools and Technologies for Mechanical System Design*, 1993.

[7] Panagiotis Michaleris, Daniel A. Tortorelli, and Creto A. Vidal, "Tangent operators and design sensitivity formulations for transient nonlinear coupled problems with applications to elasto-plasticity," *Department of Theoretical and Applied Mechanics, Technical Report number xxx*, 19??

[8] V. B. Venkayya, "Structural optimization: a review and some recommendations," *International Journal for Numerical Methods in Engineering*, 1978.

[9] E. Haug, K. Choi, and V. Komkov, "Design sensitivity analysis of structural systems," *Academic Press*, 1986.

[10] M. A. Austin and B. K. Voon, "Structural optimization in a distributed computing environment," *Computer Methods in Applied Mechanics and Engineering*, no. 107, pp. 173–192, 1993.

[11] Raphael T. Haftka and Zafer Gürdal, *Elements of Structural Optimization.* The Netherlands: Kluwer Academic Publishers, 1992.

[12] D. Tortorelli, *Design Sensitivity Analysis for Nonlinear Dynamic Thermoelastic Systems.* PhD thesis, University of Illinois, Urbana-Champaign, 1988.

[13] G. N. Vanderplaats, *Numerical Optimization Techniques for Engineering Design - With Applications.* New York, New York: McGraw-Hill, 1984.

[14] G. Vanderplaats, *ADS - A FORTRAN program for automated design synthesis*, 1984.

[15] V. M. A. Engineering, *DOT Users Manual.* V. M. A. Engineering, version 3.0 ed., 1992.

[16] J. Zhou and A. L. Tits, *User's Guide for FSQP.* University of Maryland, College Park, MD, version 2.0 ed., 1991.

[17] D. A. Tortorelli and R. B. Haber, "First-order sensitivity analysis for transient conduction systems by an adjoint method," *Intl. J. Numerical Methods in Engineering*, vol. 28(4), pp. 733–752, 1989.

[18] Q. Zhang and S. Mukherjee, "Design senstivity coefficients for elastic-viscoplastic problems by boundary element methods," *International Journal for Numerical Methods in Engineering*, 1991.

[19] B. G. C.-Y. Joh and R. T. Haftka, "Design optimization of transonic airfoils," in *Third International Conference on Inverse Design Concepts and Optimization in Engineering Sciences*, 1991.

[20] T. Sorensen, "Airfoil optimization with efficient gradient calculations," in *Third International Conference on Inverse Design Concepts and Optimization in Engineering Sciences*, 1991.

[21] G. W. Burgreen and O. Baysal, "Aerodynamic shape optimization using preconditioned conjugate gradient methods," *AIAA Journal*, vol. 32, no. 11, pp. 2145–2152, 1994.

[22] John H. Lienhard, *A Heat Transfer Textbook.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1987.

[23] O.-J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Common Base Language.* Oslo, Norway: Norwegian Computing Center S-22, 1970.

[24] B. Stroustrup, *The C++ Programming Language.* Addison-Wesley, second ed., 19??

[25] S. M. Omohundro, *The Sather 1.0 Specification*, 1994.

[26] C. J. Yoon, "Object-oriented development of large engineering software using clips," *Microcomputers in Civil Engineering*, no. 8, pp. 385–394, 1993.

[27] J. Buck, S. Ha, A. Lee, and D. G. Messershmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, 1994.

[28] B. Bagheri, *FEC Version 1.0*, 1990.

[29] A. Kecskméthy, "MOBILE: An object-oriented tool-set for the efficient modeling of mechatronic systems," in *Seoncd Conference on Mechatronics and Robotics*, (Duisburg/Moers, Germany), 1993.

[30] H. W. Schmidt and B. Gomes, *ICSIM: An Object-Oriented Connectionist Simulator*, 1990.

[31] D. Calhoun and A. Lewandowski, "Design of C++ Classes for Structured Modeling and Sensitivity Analysis of Dynamical Systems," in *Proceedings of the* 2^{nd} *Annual Conference on Object-Oriented Numerics*, Rogue Wave Software, 1994.

[32] A. Lewandowski, "Object-Oriented Modeling of the Natural Gas Pipeline Network," in *Proceedings of the* 2^{nd} *Annual Conference on Object Oriented Numerics*, Rogue Wave Software, 1994.

[33] B. Raphael and C. S. Krishnamoorthy, "Automating finite element development using object oriented techniques," *Engineering Computations*, vol. 10, pp. 267–278, 1993.

[34] Y. Dubois-Pélerin and T. Zimmerman, "Object-oriented finite element programming: II. a prototype program in smalltalk," *Computer Methods in Applied Mechanics and Engineering*, no. 98, pp. 361–397, 1992.

[35] Y. Dubois-Pélerin and T. Zimmerman, "Object-oriented finite element programming: III. an efficient implementation in C++," *Computer Methods in Applied Mechanics and Engineering*, no. 108, pp. 165–183, 1993.

[36] M. M. Tiller, "Sensitivity analysis for furnace temperatures in a programmable multi-zone furnace," Master's thesis, University of Illinois, Urbana-Champaign, 1993.

[37] D. A. Tortorelli, M. M. Tiller, and J. A. Dantzig, "Optimal design of nonlinear parabolic systems - part I: Fixed spatial domain with applications to process optimization," *Computer Methods in Applied Mechanics and Engineering*, vol. 113, pp. 141–155, 1994.

[38] Daniel A. Tortorelli, John A. Tomasko, Timothy E. Morthland, and Jonathan A. Dantzig, "Optimal design of nonlinear parabolic systems - part II: Variable spatial domain with applications to casting optimization," *Computer Methods in Applied Mechanics and Engineering*, 1993, accepted.

[39] Fluid Dynamic International, Incorporated, *Fidap Theoretical Manual.* Evanston, Illinois: FDI, 1991.

[40] John K. Ousterhout, *Tcl and the Tk Toolkit.* Addison–Wesley, 1994.