Development of CASCADE - a Test Simulator for Modelling Multidisciplinary Design Optimization Problems in Distributed Computing Environments

by
Kevin F. Hulme

A thesis submitted to the
Faculty of the Graduate School of
the State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of
Master of Science

January 23, 1996

Acknowledgments

First and foremost, I would like to thank my family, for their constant love and support. Without them, this work would not have been possible.

I would also like to thank Dr. Christina Bloebaum, for introducing me to the field of Multidisciplinary Design Optimization, and for her faith in accepting me as her graduate student. She has been a very supportive advisor, and is a pleasure to work for.

The 6 1/2 year road that has led to the completion of this work has been a long one. I would like to thank my friends for their support, and for bearing with me through the difficult times.

And finally, a special thank you to my editor.

Table of Contents

Acknowledgements	ii
List of Tables	v
List of Figures	vii
Abstract	ix
1. Introduction	1
2. Multidisciplinary Design Optimization	6
Optimization	6
Multidisciplinary Design Optimization Background	7
Decomposition	8
Global Sensitivity Equations	10
Design Synthesis	13
Design Structure Matrix and Scheduling	15
System Reduction	18
Closure	20
3. Parallel Processing and Distributed computing	21
Background	21
Parallel Virtual Machine (PVM)	23
Relation to MDO	25
Closure	26
4. Complex Application Simulator for the Creation of Analytical Design Equations (CASCADE)	27
System construction and convergence	27
I. Inputs	*27*
II. Calling program preliminaries	*29*
III. Determination of the nature of each term	*30*
IV. Determination of the magnitudes and sensitivities of each term	*36*
V. Term magnitude check	*38*
VI. Coefficient magnitude check	*39*
VII. Equation magnitude check	*40*
VIII. System convergence	*44*
Additional post convergence features	46
I. Average coefficient values	*46*
II. Equations written to subroutines	*47*
III. Sensitivity computation	*51*
IV. Term statistics	*56*
Closure	57

5. CASCADE implementation using PVM 58
System construction and convergence 58
Closure 61

6. Example problems created by using CASCADE 62
Input file 62
"Parameters" file 63
Output magnitudes 67
Sensitivity information 68
Converged system written to subroutines 70
Closure 73

7. Macroscopic analysis of numerous CASCADE outputs 74
Equation construction and convergence - CPU time 74
Iterations to convergence 77
Effects of Normalization on GSE solution 81
Closure 84

8. Conclusions 85
Summary 85
Future work 89

References 91

Appendix I. CASCADE program listing 93

Appendix II. CASCADE User's manual 149

Appendix III. Chapter 6 example system cases: full listing 153

List of Tables

Table 3.1: Essential PVM routine calls

Table 4.1: Parameter *sscho*

Table 4.2: Parameter *numdv*

Table 4.3: Exponent values corresponding to dvmag values

Table 4.4: Effect of the exponent on the magnitude of the term

Table 4.5: Parameter *dvexp*

Table 4.6: Parameter *numbv*

Table 4.7: Parameter *bvexp*

Table 4.8: Parameter *sign*

Table 4.9: Example term with design variable dependency

Table 4.10: Example term with output equation coupling

Table 4.11: Magnitudes of the two example terms

Table 4.12: Local sensitivities for the two example terms

Table 4.13: Character strings used for writing equations to subroutines

Table 4.14: Example write statement for one term, written in FORTRAN

Table 6.1: Input files for the two system cases

Table 6.2a: Parameters file excerpt for the first system case

Table 6.2b: Parameters file excerpt for the second system case

Table 6.3: Output magnitudes for the two system cases

Table 6.4a: Total derivative matrix for the first system case

Table 6.4b: Total derivative matrix for the second system case

Table 6.5a: Equation-subroutines for the first system case

Table 6.5b: Equation-subroutines for the second system case

Table 7.1: CPU time required per iteration, for the PVM systems

Table 7.2: Condition number comparison of normalized / un-normalized data

List of Figures

Figure 2.1: Traditional serial design approach

Figure 2.2: Hierarchic system model

Figure 2.3: Non-hierarchic system model

Figure 2.4: Hybrid-hierarchic system model

Figure 2.5: Subsystems interactions flowchart

Figure 2.6: Non-hierarchic design synthesis

Figure 2.7: Five subsystem, hybrid-hierarchic interaction

Figure 2.8a: Randomly oriented DSM

Figure 2.8b: Reordered GSM

Figure 2.9: 7 module GSM a) with 7 crossovers b) without crossovers

Figure 2.10: 3 non-hierarchic subsystems

Figure 2.11: GSM for figure 11 system

Figure 3.1: Workstation network

Figure 3.2: Hierarchic or non-hierarchic structured systems

Figure 3.3: Classification of the decomposition approaches

Figure 4.1: Convergence check flowchart

Figure 4.2: Form of the Global Sensitivity Equations

Figure 6.1: Block drawing depiction of the second system case couplings

Figure 7.1: **Single computer CPU time results**

Figure 7.2: **Virtual machine CPU time results, using PVM**

Figure 7.3: **Single computer Iteration results**

Figure 7.4: **Virtual machine Iteration results, using PVM**

Figure 7.5: **Sensitivity CPU times vs. system size**

Figure 8.1: **Direct message passing to the Network layer**

Abstract

Industry is constantly pursuing faster and cheaper means for designing and manufacturing goods and services. The emerging field of Multidisciplinary Design Optimization (MDO) attempts to simplify the design of a large, complex system, by dividing the system into a series of smaller, simpler, and coupled subsystems. Each subsystem can be thought of as a participating design group of a large scale design. An example would be the *aerodynamics* division of the design of an aircraft. One goal of MDO is to analyze these subsystems concurrently, thus speeding up the design time of the overall product.

In addition to decomposing a large system, the field of MDO strives to further simplify the system, using techniques such as task scheduling and system reduction. Task scheduling seeks to find the optimum sequence of subsystems in which to perform a system analysis, so as to gain convergence most quickly. System reduction seeks to identify relatively weak couplings between subsystems, and then either temporarily suspend them (during the system analysis), or eliminate them altogether.

These methodologies require extensive testing, either on analytical representations of real-life systems, or on the actual system itself, prior to implementation. Due to the complexity of these real-life systems, it is not practical to perform methodology feasibility studies on the analytical and numerical representations of the true system. Hence, some representative yet efficient means of determining the feasibility and robustness of MDO methods is crucial. This thesis describes the design of a test simulator, called CASCADE (Complex Application Simulator for the Creation of Analytical Design Equations), that is capable of randomly generating and then converging a complex system of analytical equations, of user-specified size. CASCADE-generated systems can be used, for example, to test the ordering and system reduction strategies that were described above. CASCADE has been designed to operate in a parallel computing environment (using Parallel Virtual Machine), as the field of MDO itself is perfectly suited for such a setting.

Chapter 1

Introduction

The design methodology in worldwide industry is rapidly changing. United States aircraft, automotive, and electronics industries, among others, are constantly searching for ways to improve the efficiency of their design cycles to meet time and cost demands. The traditional serial design approach is characterized by a sequential design cycle, with respect to the participating design groups. A design is formulated in a given design group and passed to the next group, who uses the previous groups' output parameters as input parameters. Because of both inefficiency and design system complexity, this approach has become obsolete, in favor of the Concurrent Engineering approach.

Concurrent Engineering is a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. The interaction of all participating engineering groups throughout the design cycle is a truly *multidisciplinary* effort. Many of the recently developed capabilities to address concurrent design have stemmed from the emerging area of Multidisciplinary Design Optimization, or MDO. The MDO approach is intuitive: divide one large task into a group of smaller, interrelated (coupled), and more manageable tasks [16]. The large task is often referred to as a *system*, and the smaller, interrelated tasks are called *subsystems*. Each subsystem contains *design variables*, as well as additional unknown outputs, often referred to as *behavior variables*. These subsystem variables are collectively referred to as *modules* of the system. This method

was established by applying a linear decomposition method to a hierarchical (top-down) system.

Most design cycles contain participating groups that interact laterally. Such design cycles are thus non-hierarchic in nature. The Global Sensitivity Equation (GSE) method was the first approach to extend the concepts of the linear decomposition method to non-hierarchical systems [18,1]. This method uses *local* sensitivities (derivatives that are computed within each subtask) to compute total system sensitivities.

An MDO issue that is the focus of much research today is the concept of scheduling (or sequencing). Scheduling is a methodology that reorders the design tasks (modules) in a given system, to allow for maximum efficiency in the execution of the design [14]. The efficiency of a system can be increased if certain problem dependent parameters are minimized, such as cost, CPU time, *feedbacks*, or *cross-overs*. A feedback occurs when a system module requires information from another module that is located later in the design sequence. A cross-over occurs when the feedbacks of two modules intersect, without any transfer of information.

Another area of current research within the field of MDO is the concept of system reduction, through coupling suspension and elimination. As explained, the decomposition of a large system results in a series of smaller subsystems, that are interrelated through couplings. Because of the enormity of many engineering systems, there is a need to minimize the complexities of the system, and thus the time for both design system convergence and sensitivity analysis. It would be advantageous to find an analytical means for quantifying the strengths of these couplings. Couplings that are found to be weak could be suspended for a

portion of the system analysis, or eliminated outright. This concept provides the foundation for system reduction strategies.

Industry is primarily interested in the way that the various participating design groups communicate. An efficient and natural way for design groups to pass information back and forth is via *distributed processing*. The concept of distributed processing assures that the system design tasks are computationally distributed among the participating design groups. In this way, distributed processing extends the principals of MDO to a computer network. This methodology allows for parallel communication between the design groups, and hence provides greater efficiency than a sequential computing approach. A modern day approach that is used to achieve distributed processing is the Parallel Virtual Machine (PVM) coding language. PVM uses library calls and message passing to distribute tasks amongst the individual computer hosts on the network.

It is quite clear that the field of MDO has a great deal of potential to provide methodologies that can be used in industry. For this to happen, these MDO-methodologies must be tested extensively. Researchers typically use previously-generated analytical systems to test their methodologies. It would be convenient for these researchers to possess a simulator that is capable of generating, converging, and then further analyzing an analytical representation of a complex engineering system. The simulator should be robust and capable of representing a wide range of complex systems. The simulation should likewise be arbitrary (random), as many design scenarios are often presented with issues that were initially unpredicted. Lastly, the simulation should be realistic, in that it should allow for a distributed processing communication architecture.

Given the above motivation, this thesis discusses the design and creation of an MDO-type simulator of the described form, called ***CASCADE*** (Complex Application Simulator for the Creation of Analytical Design Equations). CASCADE can be used to generate complex systems comprised of analytical equations, of user specified size. Thereafter, CASCADE employs a system analysis to iteratively converge the generated system. To add realism to the simulation, this process can be made to take place in a distributed environment, using the PVM coding language. After the system has converged, CASCADE uses the GSE method to compute the total sensitivities off all output responses, with respect to all inputs (design variables). This sensitivity information could potentially be used to analyze coupling strengths for possible suspension/elimination. CASCADE writes each converged output (behavior variable) equation to a separate subroutine. Researchers could potentially experiment with this sequence of subroutines to further investigate coupling strengths, or to investigate the sequencing issue.

Chapter 2 takes a closer look at the field of Multidisciplinary Design Optimization. The methodologies of the Global Sensitivity Equations, task scheduling, and coupling suspension and elimination are investigated in greater detail.

Chapter 3 presents a background on the principals of distributed processing and parallel computing. The coding methodology of Parallel Virtual Machine (PVM) is investigated, as PVM is the chosen means for achieving a distributed computational environment for this research.

Chapter 4 analyzes in detail the development and operation of CASCADE, the application simulator that has been created by the author. The design considerations that went into the creation of CASCADE are discussed.

Chapter 5 looks into the incorporation of PVM code within the framework of CASCADE. CASCADE was initially designed to operate sequentially, on a single computer. The principals of parallel processing were incorporated as a final phase.

Chapter 6 presents results from two analytical systems that were created by CASCADE. Chapter 7 analyzes these results, and compares and contrasts the implications of the results on a more global level.

Chapter 8 presents concluding remarks on the accomplishments of this research, and proposes suggestions for future work. The Appendix contains a program listing of CASCADE, a user's manual for the entire simulator package, and two example system cases that were created by using CASCADE.

Chapter 2

Multidisciplinary Design Optimization

Optimization

The term *optimize* makes one think of analogous phrases such as *maximize* or *make the best.* In fact, optimization is a concept that has significance in most of our everyday lives. For example, in engineering we wish to "produce the best quality of life, using the resources that are available" [24]. A problem that involves the concept of optimization has several important parameters. The *objective function* is the cost function that is being extremized (either minimized or maximized). The concept of optimizing a structure implicitly suggests that there is some design freedom to change the structure [2]. The *design variables* are changeable parameters that signify a potential for change. The *constraints* are limitations on the design space. Constraints can be of numerous forms, including equality, inequality, and side constraints. A typical optimization problem has the mathematical form of equation 2.1.

$$
\begin{array}{llll}
\text{Minimize} & F(X) & & \\
\text{Subject to} & g_j(X) \le 0 & j=1,\dots l & \\
 & h_k(X) = 0 & k=1,\dots m & [2.1] \\
\text{and} & X_i^L \le X_i \le X_i^U & i=1,\dots n &
\end{array}
$$

The objective function and/or constraints of any given optimization problem could conceivably be a function of the design variables and/or the subsystem outputs of a highly coupled, decomposed system. With this general understanding of the discipline of optimization, the specialized field of Multidisciplinary Design Optimization can be investigated in detail.

Multidisciplinary Design Optimization background

United States industry has responded to competitive pressures that have resulted in losses in market share due to high product development times, at high costs. Many corporations are making use of the concurrent engineering approach, which causes product developers, from the outset of design, to consider all elements of the product life cycle, from conception through disposal. This requires constant interaction between all participating design groups throughout the entire design cycle. This approach is multidisciplinary, and makes the traditional serial design approach, depicted in figure 2.1, obsolete [13].

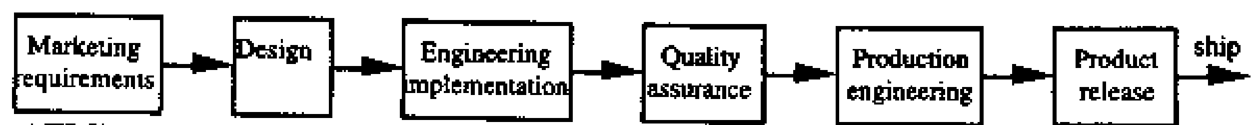

Figure 2.1: Traditional serial design approach

In the early 1980's, Sobieski laid the foundations for a field that is now known as Multidisciplinary Design Optimization, or MDO. The fundamental objective of MDO is to develop an improved design capability, while considering disciplinary interactions for synergistic effects [3]. The underlying principle of MDO is quite simple: divide a large task, into a sequence of smaller, interrelated (complex or coupled) and more individually manageable tasks [18]. In engineering optimization problems, the large task is commonly

known as a *system*, and the smaller, interrelated tasks are known as *subsystems*. Each subsystem contains design variable inputs, as well as additional unknown outputs (sometimes called *behavior variables*), that are unique to that subsystem.

Decomposition

To accomplish the division of the system, Sobieski applied a linear decomposition method [18] for large-scale multidisciplinary problems. This methodology is applicable to hierarchic (top-down) systems, such as the one depicted in figure 2.2.

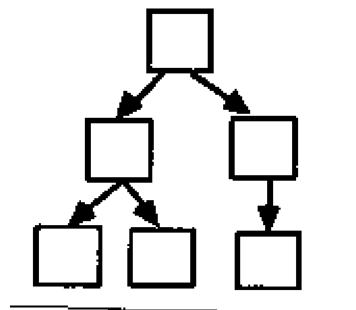

Figure 2.2: Hierarchic system model

In such a hierarchic system, there is a definite ordering to the execution of each module to produce a final and exact result. The breakdown of *most* complex engineering systems does not result in a top-down system model, but rather a non-hierarchic system model, as depicted in figure 2.3.

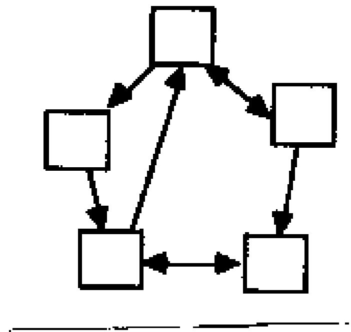

Figure 2.3: Non-hierarchic system model

A non-hierarchic system contains lateral couplings, which essentially means that the system has no discernable *starting point*. Between many of the modules in a non-hierarchic system, there exists two-way couplings. (The output of one module is the input to a second module, and vice-versa.) The system analysis for a non-hierarchic system requires an *initial guess* to the magnitude of each output module. Clearly, such a system requires iteration to gain convergence. Some decomposed systems exhibit traits from both hierarchic and non-hierarchic systems, and are called hybrid-hierarchic, as seen in figure 2.4.

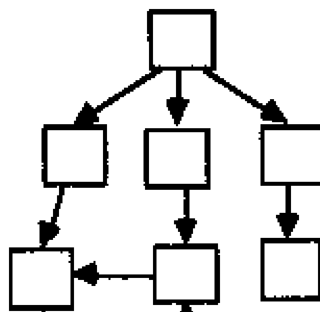

Figure 2.4: Hybrid-hierarchic system model

The Global Sensitivity Equation (GSE) method was the first to successfully extend Sobieski's concept of hierarchic modularity to non-hierarchic systems, as early as 1990 [20,22,1,2].

Global Sensitivity Equations

A sensitivity is defined as a change in an output value, with respect to a given input value. System sensitivities are required to gain system improvement by prescribing a change in the subsystem design variables. A sensitivity analysis can be a computationally lengthy (and thus costly) procedure, and must therefore be efficient. The Global Sensitivity Equation (GSE) approach defines total derivatives of the output quantities in terms of *local sensitivities*. These local sensitivities are partial derivatives of each subsystem's outputs, with respect to its inputs. For this research, local sensitivities are computed analytically. For complex output functions, a numerical procedure, such as finite difference methods, are often required to attain these sensitivities.

To illustrate the mathematics of the GSE method, consider the two subsystem schematic seen in figure 2.5. This schematic can be thought of as an abstraction of a real-life system. For example, the design of a large, multidisciplinary mechanical system, such as an automobile, contains interactions between numerous disciplines, such as structures and aerodynamics. Subsystem A could be thought of an abstraction of the structures discipline; subsystem B, the aerodynamics discipline. Subsystem A has two sets of inputs: design variables X_A, and the output (coupling) from subsystem B, Y_B . Similarly, subsystem B has design variables X_B and the output (coupling) from subsystem A, Y_A, as its inputs.

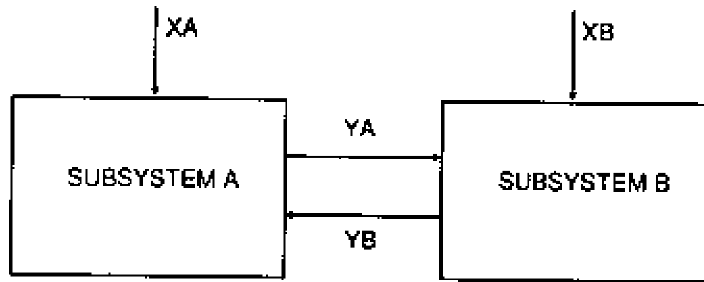

Figure 2.5: Subsystems interactions flowchart

Symbolically, this relationship is depicted as follows:

$$A((X_A,Y_B),Y_A) = 0$$
$$B((X_B,Y_A),Y_B) = 0 \qquad [2.2]$$

This expressions can be rewritten explicitly as follows:

$$Y_A=(X_A,Y_B)$$
$$Y_B=(X_B,Y_A) \qquad [2.3]$$

Expanding equation [2.3] with a first order Taylor series gives:

$$\frac{dY_A}{dX_A} = \frac{\partial Y_A}{\partial X_A} + \frac{\partial Y_A}{\partial Y_B}\frac{dY_B}{dX_A}$$
$$\frac{dY_B}{dX_B} = \frac{\partial Y_B}{\partial X_B} + \frac{\partial Y_B}{\partial Y_A}\frac{dY_A}{dX_B} \qquad [2.4]$$

Applying the chain rule to equation [2.3] gives:

$$\frac{dY_A}{dX_B} = \frac{\partial Y_A}{\partial Y_B}\frac{dY_B}{dX_B}$$

$$\frac{dY_B}{dX_A} = \frac{\partial Y_B}{\partial Y_A}\frac{dY_A}{dX_A} \qquad [2.5]$$

Finally, equations [2.4] and [2.5] can be represented as follows:

$$\begin{bmatrix} 1 & -\frac{\partial YA}{\partial YB} \\ -\frac{\partial YB}{\partial YA} & 1 \end{bmatrix} \begin{bmatrix} \frac{dYA}{dXA} & \frac{dYA}{dXB} \\ \frac{dYB}{dXA} & \frac{dYB}{dXB} \end{bmatrix} = \begin{bmatrix} \frac{\partial YA}{\partial XA} & 0 \\ 0 & \frac{\partial YB}{\partial XB} \end{bmatrix} \qquad [2.6]$$

Equations [2.6] are the representation for the GSE [2].

The leftmost square matrix on the left side of equation [2.6] is known as the Global Sensitivity Matrix (GSM), and is comprised of the couplings between interacting subsystems. In other words, the GSM is a matrix of the partial derivatives of all output equations with respect to all other output equations. The dimension of this matrix is (n×n), where n is the total number of output equations in the system. The matrix on the right hand side of the equation is a matrix of partial sensitivities of all system outputs, with respect to all system design variables. The dimension of this matrix is (n×m), where m is the total number of system design variables. The *center* matrix (the rightmost matrix on the left side of the equation) is the desired matrix of total derivatives. These derivatives provide an indication of how a change in one or more design variables will affect all of the outputs of the system.

Having computed the GSM and the right hand side matrix, the total derivatives are computed by using LU-decomposition with numerical conditioning. Before this can take place, the components of both matrices must be scaled, or normalized [10, 3]. This is because the magnitudes of the design variables (**X**) and the subsystem outputs (**Y**) are often of widely-varying magnitude. For equation [2.6] above, the normalization *form* for the partial derivative terms of the GSM is as follows:

$$\frac{\partial Y_A'}{\partial Y_B} = \frac{\partial Y_A}{\partial Y_B} \cdot \frac{Y_B}{Y_A} \qquad [2.7]$$

Note that the normalized partial derivative is *primed*, on the left side of equation [2.7]. Following solution of the GSE, the unscaled total derivatives are recovered as follows:

$$\frac{dY_A}{dX_B} = \frac{dY_A'}{dX_B} \cdot \frac{Y_A}{X_B} \qquad [2.8]$$

Design Synthesis

With the given background on engineering optimization, system decomposition, and sensitivity analysis, the design synthesis for generic non-hierarchic multidisciplinary problems is presented. The methodology is illustrated in figure 2.6.

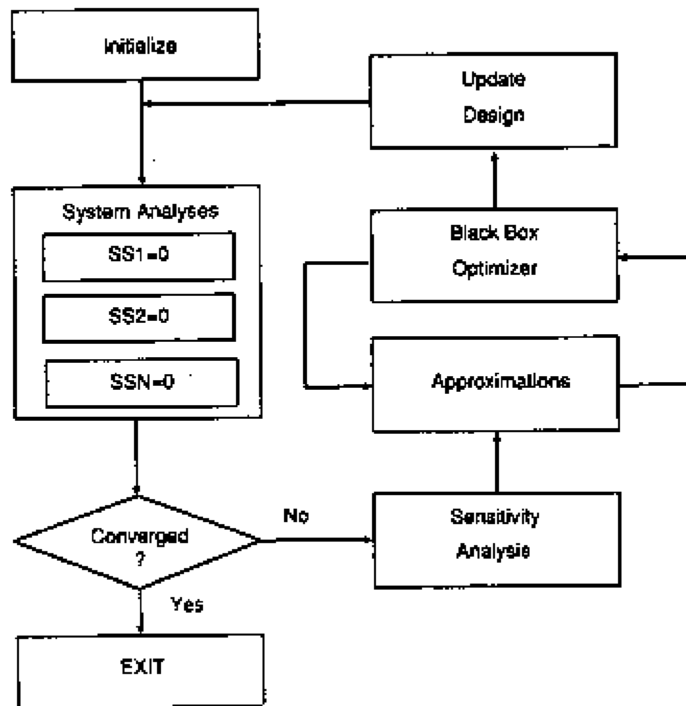

Figure 2.6: Non-hierarchic design synthesis

With a given set of design variables, the system outputs are initialized, which allows for an initial system analysis of all the decomposed subsystems. System convergence is checked thereafter; if the system has not converged, sensitivities are computed, and fed to an optimizer. The optimizer uses the sensitivities to better the design by perturbing the system design variables. This updated design is once again fed to the system analyzer. The process repeats itself until the optimum design has converged.

The simulator program that has been created in the present research focuses on two segments of this design cycle: the system analysis and the sensitivity computation. A system of user specified size is constructed, and thereafter converged. Sensitivities of the converged system of equations are then computed, by using the GSE method. This information can then

be used (by researchers in the MDO community) to compare and contrast ways to converge the system more efficiently. This time savings may be achieved by system reordering (via task sequencing), or by system reduction strategies (via coupling suspension and elimination).

Design Structure Matrix and Scheduling

Consider the five subsystem hybrid-hierarchic system of figure 2.7.

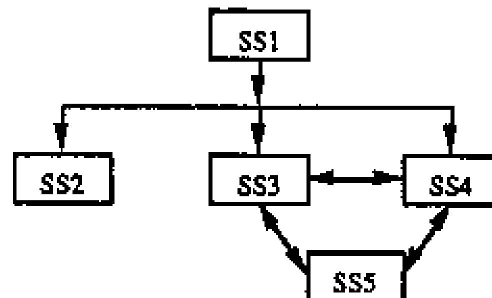

Figure 2.7: Five subsystem, hybrid-hierarchical interaction

This complex system can be represented as a square Design Structure Matrix (DSM), wherein each of the subsystems is denoted as a box, along the diagonal [23]. (A more complicated DSM could be composed, where each box along the diagonal represents each *module* [output equation or design variable] of each and every subsystem). Figure 2.8a depicts a randomly ordered DSM for the system of figure 2.7.

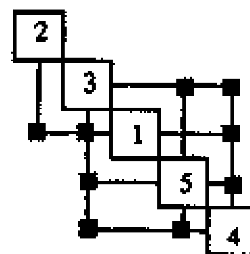

Figure 2.8a: Randomly oriented DSM

To understand this figure fully, one must compare it back to figure 2.7. From figure 2.7, it is clear that subsystem 3 requires inputs from subsystems 1,4 and 5, and transmits outputs (as inputs) to subsystems 4 and 5. Note the ordering of figure 2.8a. This information is translated to that figure as follows: subsystem 3 provides information via *feedforwards* to subsystems 4 and 5, and receives *feedbacks* from modules 1,4 and 5. From this example, it is clear that feedbacks are pieces of information that are required from subsystems that are located *downstream* in the solution process. An initial guess and an iterative framework are hence required to converge a system with feedbacks.

One area of interest within the field of MDO is known as task sequencing, or scheduling. Optimal scheduling of a system is the result of reordering the sequence of boxes in the DSM, so as to maximize the efficiency of the system analysis. This is accomplished primarily by reducing the GSM feedbacks. Figure 2.8b is an illustration of a re-ordered DSM of the figure 2.7 system. Using this ordering strategy, the number of feedbacks has been reduced from 5 to 3. This reduction will reduce the number of iterative loops required for system convergence.

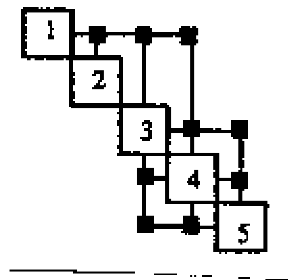

Figure 2.8b: Reordered GSM

Another (although less important) objective of a task sequencer is to reduce the number of *crossovers* in the DSM. A crossover occurs when one modules feedback crosses over that of another, without exchanging information at the intersection [17, 15]. Crossovers tend to obscure the convergence process in any system that contains them. This causes additional iteration and computational expense. The removal of crossovers increases the clarity of the convergence procedure, allowing for greater control of the process from the design managers standpoint. Figure 2.9a illustrates a 7 module GSM, that contains 7 crossovers. The reordered GSM of figure 2.9b sees the elimination of all crossovers.

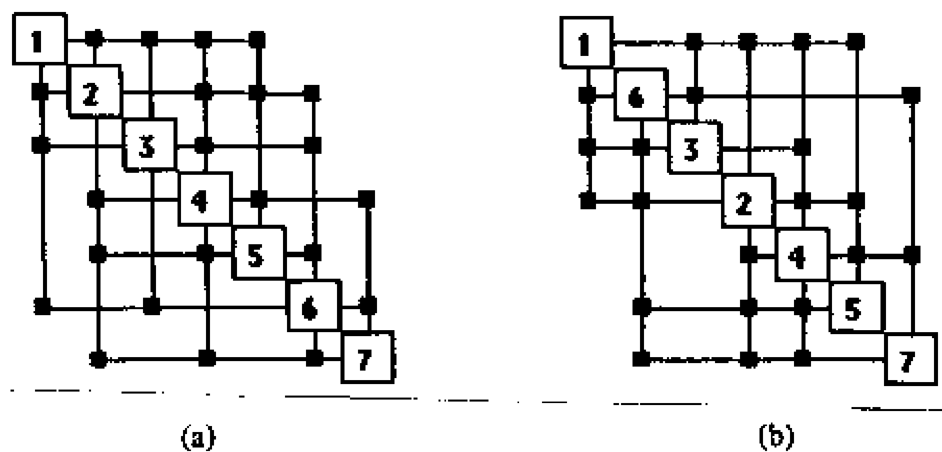

Figure 2.9: 7 module GSM a) with 7 crossovers b) without crossovers

A final benefit of scheduling is to aide in identifying weak links. Once a system has been optimally scheduled, the sensitivities of the couplings can be analyzed. Those that are found to be comparatively weak could be suspended for a select number of iterations of the convergence procedure, or eliminated altogether.

System Reduction

Consider next the coupled system of figure 2.10. The system is seen to have 3 subsystems, each having anywhere from 2 - 4 outputs each, and either 2 or 3 design variables.

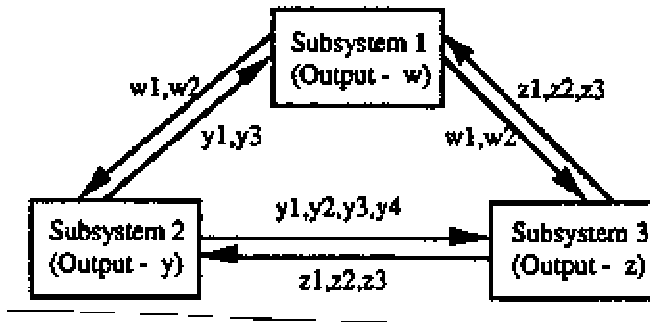

Figure 2.10: 3 non-hierarchic subsystems

The *full* GSM, whereby each box is a module (design variable or output equation) is seen in figure 2.11. It has been optimally sequenced, so as to minimize feedbacks, by using DeMAID.

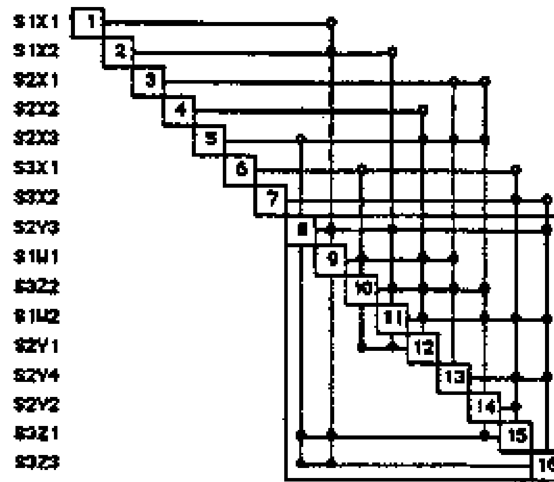

Figure 2.11: GSM for figure 11 system

The circles depicted in figure 2.11 represent the couplings between the various modules. At this stage, the system designer needs to know the relative strength of all of these couplings. This knowledge may allow for the temporary suspension, or the outright elimination of select couplings.

Bloebaum's method for assessing coupling strength is based on local sensitivities [3]. This method finds the numerical value of every coupling sensitivity, using the previously explained GSE method. These sensitivities are normalized, and then compared. Modules that are found to have the smallest set of normalized sensitivities are considered to be *weak couplings*. These couplings are then considered for temporary suspension or outright elimination, providing that they do not have a substantial impact further downstream. This determination is made based on downstream coupling strengths.

Recall that many optimization problems have objective functions and constraint functions that *have outputs of complex systems as their variables*. Hence, the drawback of the local sensitivity method for identifying weak couplings is that it does not relate the local coupling strengths to their effect on the global problem, namely the objective function and constraints. A coupling may have a relatively small normalized sensitivity, but may still have a large effect on the objective function or constraints. This coupling may therefore have a major impact on the accuracy of the design solution.

More recently, researchers have been looking for ways to assess coupling strengths based on total sensitivities. Miller and Bloebaum have developed such a method; their analysis creates a separate coupling strength for the objective function and each constraint [16]. This work suggests that temporary suspension of certain couplings only occur *every*

other cycle, due to the difficulty in predicting which constraints will be active, during any given system analysis iteration. In addition, this work suggests that the criterion for the total elimination of a coupling be based on a user defined percentage of the objective function magnitude.

Closure

Given the above background, it is clear that MDO is a field with a great deal of potential for industrial implementation. It is equally clear that the methods and theories that have been presented must be tested extensively, using a wide range of problems. Before discussing the simulator that has been developed in this research, a background will be presented on *distributed computing* and *parallel processing*. Both are techniques that are imperative for the implementation and growth of the field of MDO. This background will help to place the discussion of the simulator in the appropriate context.

Chapter 3

Parallel Processing and Distributed Computing

Background

The term *parallelism* denotes the possibility of executing several operations simultaneously [5]. *Parallel processing* is the method of having many small tasks solve one large problem. The origins of parallel processing can be traced back to the early 1960's, when researchers were fascinated by the challenge of solving problems while assuming the existence of a parallel environment. Of course, no such environment existed at the time, and researchers had no idea how or if these principals could be applied to scientific problems. This method has emerged as a key enabling technology in modern computing. The applicability of parallel processing to MDO is readily apparent, based on the background that has been presented.

It seems obvious that the system solution process could be rapidly accelerated by simultaneously executing certain stages of a lengthy sequential process. However, the conversion of a sequential algorithm to a parallel algorithm is not a simple one. In addition, this conversion does not usually provide the maximum attainable parallelism for a given problem. The simple fact that there is no trivial correspondence between sequential and parallel computing opens new horizons for scientific investigation [5].

The past several years have witnessed an ever-increasing acceptance of parallel processing. The acceptance has been facilitated by two major developments: Massively

Parallel Processors (MPP's) and distributed computing. MPP's are the world's most powerful type of computer. They combine thousands of CPU's (in a single cabinet) and gigabytes worth of RAM. *Distributed computing* is a process whereby a set of computers connected by a network are used collectively to solve a single large problem [9]. The method of distributed computing has an advantage over the implementation of an MPP; that is cost. MPP's could cost millions of dollars; in contrast, the cost in interconnecting a local set of networked computers is minimal. Figure 3.1 illustrates the general nature of a distributed computing environment, with two *upper-level* computers allocating tasks to the six *lower-level* machines.

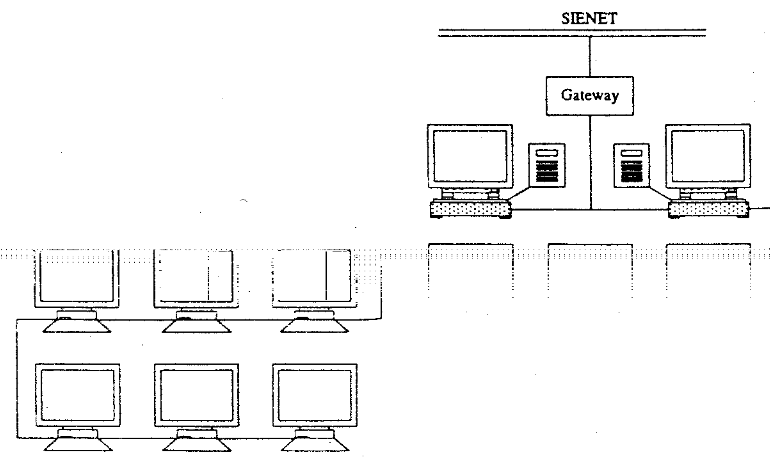

Figure 3.1: Workstation network

To be effective, distributed computing requires high communication speeds. Over the past 20 years, network speeds have increased dramatically. Among the primary advances in computer network technology is the standard Ethernet based network, developed in the early 1980's. Ethernet uses a digital signal with a bandwidth (speed) of 10 Mega-bits per second.

More recent networking advances include the development of Asynchronous Transfer Mode, or ATM. ATM utilizes an optical signal, and has communication rates as fast as 2500 Mega-bits per second [12]. Though very costly, ATM would be the ideal network platform for distributed computing, with its high-speed inter-communication rates.

Two notions that are common to most parallel processing schemes are the notions of concurrence and message passing. *Concurrence* is the simultaneous participation of entities (design groups) in the quest towards a common goal. One of the main obstacles to concurrence, through the use of parallel computers, is to efficiently implement a means for intercommunicating among the different computing elements [5]. This is most often accomplished by using a technique known as message passing. Other paradigms have been used, including shared memory and parallelized compilers. Message passing has become the paradigm of choice; most multiprocessors support it, and most languages and software systems use it.

Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) system uses message passing to allow programmers to exploit distributed computing, using any of a wide variety of computer types. A key concept in PVM is that it makes a collection of individual computers (referred to as *hosts*) appear as one large *virtual machine*. PVM transparently handles all message routing, data conversion, and task scheduling across a network of (often times) incompatible computer architectures [9]. The PVM computing model is simple, and general. The user writes his or her application as a collection of cooperating tasks. (This is an idealization. Often times, the user must break his sequential program into an analogous parallel model. As previously

discussed, this is not a simple undertaking. More information will be presented on this topic in Chapter 5). Typically, an initiating task, commonly referred to as the *master* task, will spawn a series of *slave* tasks, that are used to accomplish all of the computational "grunt work". These slave tasks receive information from the master, perform their computations, and then send the newly computed information back to the master task. The master task will then gather, organize, and interpret the information received from all of the slaves, and then make a decision accordingly (terminate the program, or iterate).

Tasks access PVM resources through a library of standard routines. These routines allow for the initialization and termination of tasks across the network, as well as communication amongst them. Most to all PVM applications contain calls to a select few *essential* routines, which are listed and defined in table 3.1:

pvmfmytid **- enrolls the process into PVM, and obtains the task ID of the current process**
pvmfparent **- obtains the task ID of the parent that spawned the current process**
pvmfspawn **- spawns a new task**
pvmfinitsend **- clears the send buffer, and prepares it for sending a new message**
pvmfpack **- packs data into the send buffer**
pvmfsend **- sends the packed data**
pvmfrecv **- receives the packed data**
pvmfunpack **- unpacks the sent data from the active message buffer**
pvmfexit **- exits the current process from PVM**

Table 3.1: Essential PVM routine calls

All PVM routines are written in the C programming language. However, these routines are available for implementation in both FORTRAN and C programming applications. The

routine names in table 3.1 are calls to these routines from a FORTRAN application, such as those that were used for the present research.

Relation to MDO

The primary goal of MDO is to efficiently divide a large engineering system into a series of smaller, more tractable, yet coupled subsystems. This is called decomposition. Due to their immanent, natural parallelity, decomposition methods are generally suitable for parallel processing. In fact, parallel processing itself can be considered as a very general decomposition approach [7].

As discussed in Chapter 2, the decomposition of a system will result in either a hierarchic structure, or more often a non-hierarchic structure (Figure 3.2).

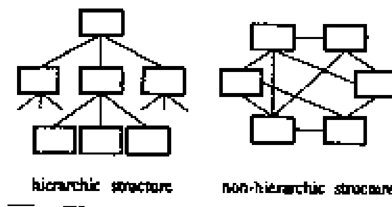

Figure 3.2: Hierarchically or non-hierarchically structured systems

This form of decomposition, where the variables and equations are decomposed, can be more specifically referred to as a *model decomposition*. The goals of a model decomposition are to reduce problem sizes, provide a more convenient problem definition, and use specialized solution methods within the subsystems. Similarly, the implementation of parallel processing can be thought of a decomposition known as *computational decomposition*. The benefit of a such a decomposition is primarily a reduction in computational time. Computational decomposition tends to simulate an asynchronous communication environment, whose benefits were mentioned earlier in Chapter 3. Very often, computational decomposition is

used in combination with model decomposition, where the independent subproblems are solved in parallel [7], as shown in figure 3.3.

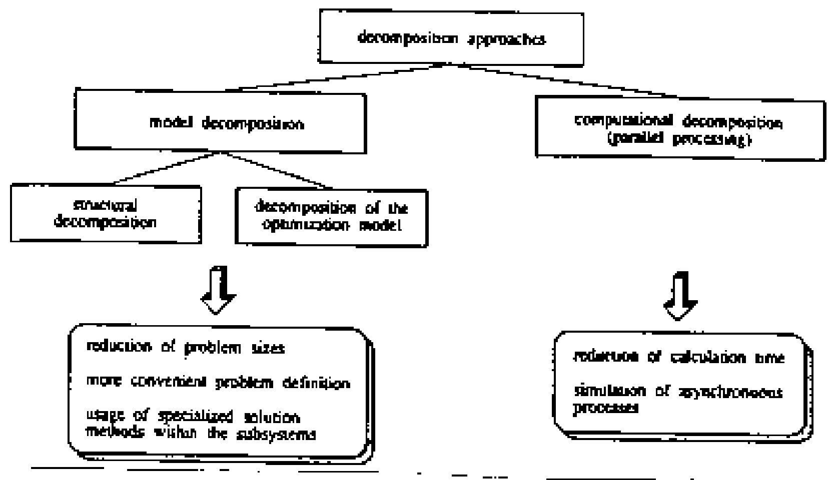

Figure 3.3: Classification of the decomposition approaches

Closure

Thus far, a background on MDO has been presented, and the potential for its industrial implementation has been made clear. Moreover, the suitability of MDO techniques for parallel processing in a distributed computing environment has been verified. Given the theoretical background, and its applicability to a state-of-the-art computational platform, this paper is now ready to present details on the simulator program that has been created. CASCADE (Complex Application Simulator for the Creation of Analytical Design Equations) will implement the principals of MDO to build and construct an analytical design setting, consisting of coupled equations. By using PVM, the parallelity of the created system will be exploited through the computational distribution of its subtasks.

Chapter 4

Complex Application Simulator for the Creation of Analytical Design Equations (CASCADE)

System construction and convergence

I. Inputs

The first task of the *CASCADE package* ("package" of FORTRAN files and subroutines) is to accept a series of user-inputs, that will define the current program execution. These options are answered by the user by executing a program called *cascin.f.* Upon executing this program, the user is first prompted to answer a few preliminary options; first, a seed value for the random number generator. The random number generator is used for the many arbitrary decisions that CASCADE will make. An IMSL, Gaussian-distributed generator is used, that returns a real value between 0 and 1. The second preliminary question that the user is asked is whether or not distributed computing (with the aid of PVM) will be employed to construct and converge the system. If PVM is to be used, a slightly different code methodology is employed, and will be discussed in Chapter 5. This chapter presents the single-machine, sequential code methodology for system analysis.

With these preliminaries, the user is asked to define the size of the system that is to be constructed. A range of 1 to 99 subsystems is allowed, each potentially having anywhere from 1 to 99 equations within. The reason for the maximum limit of 99 is that array dimensioning prevented the use of numbers sufficiently larger than 99, for these two options.

The number of subsystems is represented by a parameter called *numss*. The number of equation per subsystem is represented in an array called *bvperss*. Based on the total number of equations in the system, the user is told whether or not a sensitivity analysis will be permitted for the present run. (Presently, dimensioning constraints prevent the computation of a Global Sensitivity Matrix of size greater than 999 × 999). If the size of the system is within range, the user is then asked whether or not a sensitivity analysis is desired. If a sensitivity analysis is desired, the user is asked whether or not to accept the matrix-conditioning defaults for attaining the total derivatives via LU-decomposition.

In brief, the next two groups of user-options are comprised of writing to screen options, and writing to file options. The final group of user options pertain to convergence. The user is first asked to specify a convergence criterion, ranging from 0.0001 to 1.0. The upper limit of 1.0 seems to be the maximum allowable convergence deviation, while 0.0001 seems to be an acceptable minimum criterion, especially when considering the affects of numerical imprecision. In other words, the system will converge when **all equation magnitudes**, on the current iteration, have converged to within the aforementioned numerical criterion, from their respective magnitudes on the previous iteration. The user also enters *convergence-ease* parameters. In other words, if the system is having problems converging, the initial convergence criterion can be eased 10-fold, 100-fold, and/or can be reset to 1.0, after a user-specified number of iterations. All of the user-input parameters are written to a file, called *cascin.dat*. This file is required to initiate the main (calling) program.

This section presents a brief overview of the user-inputs that are required to initiate CASCADE. For a detailed explanation of each user input option, refer to the User's Manual, that is available in Appendix II.

II. Calling program preliminaries

The calling program portion of the CASCADE package is called *casccall.f*. The primary purpose of this program is to call all of the subroutines that perform the various tasks. Initially, the program reads the parameters from the newly created input file, that was described in section I. Following this, the random number generator is again used to generate 4 more sets of quantities. These are: the number of design variables per subsystem, [an integer value between 1 and 5] (called array *totdv*); the magnitudes of the design variables, [a real value between 0 and 10] (called array *dvmag*); the number of terms per output equation, [an integer value between 1 and 20] (called array *termperss*); and the initial magnitudes of each output-term coefficient, [a real value between 0 and 1] (called array *coeffmag*). The ranges for these 4 quantities were chosen somewhat arbitrarily. Having numerically solved numerous smaller analytical, complex systems prior to writing this program, the author obtained a feel for the nature of these quantities. Recall that the system to be created will be non-hierarchical in nature, and thus has no starting point from which to work *downward*. Before the system construction process can begin, the magnitude of every output equation is thus initialized to zero. The array that these output magnitudes are stored in is called *bvmag*.

At this point in the program, the first of 3 "clocks" is initiated, so as to measure CPU time. The 3 clocks that are used measure: 1.) the time required to build and converge the

system of equations, 2.) the time required to solve for the sensitivities of the system, and 3.) the time required to execute the entire program, once the variables are initialized, and the main loop is entered. All CPU times are measured in seconds, using an IMSL library routine.

III. Determination of the nature of each term

At this point, the first main subroutine of the program is entered: subroutine *determ*. In short, this subroutine determines the nature of every term, in every equation, in every subsystem, in the constructed system. For each term, the first parameter that is decided upon is the coupling, which is stored in an array called *sscho*. An integer random number is returned, varying between 0 and the number of subsystems in the system. If *sscho* returns a zero, then the coupling is not a coupling to another subsystem, but instead a coupling to a design variable. If *sscho* returns any other number, that number represents the subsystem of the given terms coupling. It must be mentioned that this coupling is forced to be or not to be certain values in certain situations. The first term of every output equation is forced to be a design variable. Hence, *sscho* for the first term of every equation will be zero. The coupling of every term is enforced never to be equal to the subsystem in which that term resides. In other words, the value of *sscho* for a term in subsystem number 26, will never be equal to 26. For a summary of parameter *sscho*, see table 4.1.

sscho: the coupling nature of each term in the system

Range: from 0 to the number of subsystems in the system (*numss*)

sscho = 0 : design variable dependency

sscho = (any non-zero number) : dependency to that output equation number

Enforcements: first term in every equation - sscho = 0 (design variable dependent)

output equation dependency of a term never involves the subsystem of that term

Table 4.1: Parameter *sscho*

If, for a given term, a value of zero for *sscho* is returned, the dependency is to a design variable, and not to an equation magnitude from another subsystem, as discussed above. Two additional parameters must be determined, for this chosen dependency. The first is the design variable *number*, stored in an array called *numdv*. Recall that every subsystem has from between 1 and 5 design variables associated with it, as decided by the random number generator, and stored in array *totdv*. Hence the value of *numdv* for the present term is again decided by the random number generator, and ranges from between 1 and the value of *totdv* for the subsystem in which the present term resides. For example, if subsystem number 6 contains 4 design variables {totdv(6) = 4}, then every term in subsystem number 6 that has a design variable dependency will have a *numdv* value of between 1 and 4. In short, a chosen design variable dependency has 4 values to choose from, **in the present subsystem**. For a summary of parameter numdv, refer to table 4.2.

numdv:	**The chosen design variable number, when the term dependency is to a design variable i.e. when sscho = 0**
Range:	**From 1 to 5, the previously chosen number of design variables in the present subsystem**

Table 4.2: Parameter *numdv*

The second parameter that must be determined for a design variable dependency is the exponent of the term. The exponent of such terms are stored in an array called *dvexp*. Once again, the random number generator is called to return a value of dvexp, for every term, from an integer value between 0 and 6. The integer that is returned corresponds to an exponent as summarized by table 4.3.

dvexp or bvexp	*exponent*
0	*1*
1	*2*
2	*-1*
3	*1/2*
4	*1/3*
5	*1/4*

Table 4.3: Exponent values corresponding to dvmag values

The 6 exponent choices above were chosen so as to have various ways to **alter the magnitude of the term, by way of its exponent**. In other words: an exponent of "1" will not alter the term magnitude, an exponent of "2" will increase the magnitude of the term, an exponent of "-1" will essentially *invert* the magnitude of the term, and the three fractional exponents will slightly lower the magnitude of the term. To show how these exponents could potentially change the magnitude of a term, consider the following scenario. Consider that

a given term is coupled to design variable number 3 of the subsystem that it belongs to. The value of that design variable is 7.83. The coefficient of this term in 0.879. Table 4.4 summarizes how the value of the term would change, after applying the exponent.

$0.879 * 7.83^{1} = 6.88$
$0.879 * 7.83^{2} = 53.89$
$0.879 * 7.83^{-1} = 0.112$
$0.879 * 7.83^{1/2} = 2.46$
$0.879 * 7.83^{1/3} = 1.75$
$0.879 * 7.83^{1/4} = 1.47$

Table 4.4: Effect of the exponent on the magnitude of the term

As discussed earlier, the first term of every output equation is enforced to be a design variable (*sscho* = 0). A second enforcement on the first term of every equation is that the exponent be *substantial*; either 1 or 2. Hence, the value of *dvexp* for the first term in every output equation is enforced to be, correspondingly, 0 or 1. For a summary of parameter *dvexp*, refer to table 4.5.

dvexp: The chosen exponent code for a design variable coupled term (when sscho = 0)
Range: From 0 to 6; see table 4.3 for correspondence
Enforcements: First term of every equation must have a dvexp value of 0 or 1 (an exponent of 1 or 2, respectively)

Table 4.5: Parameter *dvexp*

If, for a given term, any non-zero value for *sscho* is returned, the dependency of the term is to an equation magnitude from another subsystem, as discussed above. Again, two additional parameters must be determined, for this present coupling. The first is the equation

number of the chosen subsystem coupling, which is stored in an array called *numbv*. In other words, with which equation of the chosen *sscho* subsystem value is the coupling associated? Recall that the number of equations for a given subsystem are stored in an array called *bvperss*. Hence, for a given non-zero value of *sscho* for the present term, the random generator is called to return a value of *numbv* for this term, from between 1 and the value of *bvperss* for subsystem *sscho* {i.e. the value of *bvperss* (*sscho*)}. For example, let there be 12 equations in subsystem number 32; i.e. bvperss (32) = 12. Then, if for a given term (**not in subsystem number 32**), a coupling value for *sscho* of 32 is chosen, the value of *numbv* that will be returned by the random number generator will be between 1 and 12. Again, the quantity *12* represents the number of equations in subsystem number 32. For a summary of parameter *numbv*, refer to table 4.6.

numbv: The chosen output equation number dependency, when the coupling of the present term is to another subsystem (when sscho $\neq 0$)
Range: From 0 to the number of subsystems (numss) in the system
Enforcements: none

Table 4.6: Parameter *numbv*

Similar to the design variable dependencies, an exponent is also associated with the output equation couplings. These values are stored in an array called *bvexp*. The correspondence between the random number generator-chosen values of *bvexp* and the exponent itself are identical to the correspondence between the *dvexp* values and their exponents; refer to table 4.3. For a summary of parameter *bvexp*, refer to table 4.7.

bvexp: The chosen exponent code for an output equation-coupled term (when sscho ≠ 0)

Range: From 0 to 6; see table 4.3 for correspondence

Enforcements: none

Table 4.7: Parameter *bvexp*

Regardless of the outcome of *sscho*, every equation term has a *sign* associated with it. The sign of each term is stored in an array called *sign*. The random number generator is called to choose an integer value between 0 and 2. If *sign* equals 0 or 1, then the term is positive. If *sign* equals 2, then the term is negative. Again, certain enforcements are made. The first term of every output equation is always positive. Hence, the value of *sign*, for the first term in every equation is always 0 or 1. Consecutive negative terms are to be avoided, for reasons which will soon become apparent. Hence, if *sign* equals 2 for term number 12 in equation number 7 of subsystem number 3, then term number 13 (of the same equation and subsystem) will have a *sign* value of either 0 or 1. For a summary of parameter sign, refer to table 4.8.

sign: The sign code for each term in the system.

Range: sign = 0 or 1 - the term is positive

sign = 2 - the term is negative

Enforcements: the first term of every equation (which, is enforced to be a design variable) has a positive sign (sign = 0 or 1)

consecutive negative terms in the same output equation are not allowed

Table 4.8: Parameter *sign*

Before exiting the *determ* subroutine, the chosen sign of each term is "connected" to the coefficient of each term. For example, if the coefficient of a given term was determined

to be 0.578 (*coeffmag* = 0.578), and the sign of that term was deemed negative (*sign* = 2), the new value for *coeffmag* for that term is -0.578. If, instead, the value of *sign* for that term was a 0 or a 1, then the value of *coeffmag* would have remained the same.

IV. Determination of the magnitudes and sensitivities of each term

At this point, the characteristics of every term in the system have been determined. The second subroutine, *mags* (short for magnitudes) is now entered. It is in this subroutine that the magnitudes of each output equation term, and the sensitivity of each term are computed. This subroutine first decides whether each given term is a design variable dependency (designated by an *sscho* value of zero) or an output equation coupling (designated by any non-zero *sscho* value). The magnitude of each term is computed into an array called *termmag*. If the term has a design variable dependency, the magnitude of the chosen design variable is raised to its chosen exponential power, and that quantity is multiplied by the chosen exponent of the term. If the term has an output equation coupling, then the chosen subsystem / subsystem equation (of the coupling) is raised to its chosen power, and multiplied by its chosen coefficient.

To attempt to clarify the procedure thus far; the determination of the nature of each term, and the subsequent determinations of the magnitude and sensitivity of each term, 2 example terms are presented. Assume, for the purpose of this illustration, that an arbitrary system has 3 subsystems (*numss* = 3), each having 3 equations (*bvperss* () = 3). Both terms belong to the first equation of the first subsystem, which has 2 design variables (*dvperss* (*1*) = 2). The first term of the equation has a design variable dependency (*sscho* = 0) to the second design variable (*numdv* = 2). The exponent of the first term is 2 (*dvexp* = 1), and the

sign of the term is positive (*sign* = 1). The coefficient of the first term was chosen to be 0.286. The second term of the equation has an output equation coupling to the first equation (*numbv* = 1) of the third subsystem (*sscho* = 3). The exponent of this second term is 1/3 (*bvexp* = 4), and the sign of the term is negative (*sign* = 2). The exponent of the second term was chosen to be 0.993. The form of the first term is seen in table 4.9 (part a) and, given the above parameters, its representation lies below it (in part b). Realize that *ss*, *eqn*, and *term* are array variables for the current subsystem, subsystem equation, and equation term, respectively.

$$\textbf{a) termmag (ss,eqn,term) = dvmag (ss,numdv (ss,eqn,term))}^{\text{dvexp (ss,eqn,term)}}$$

$$\times \textbf{ coeffmag(ss,eqn,term)}$$

$$\textbf{b) termmag (1,1,1) = dvmag } (1,2)^2 \times 0.286$$

Table 4.9: Example term with design variable dependency

The form and representation of the second term are seen in table 4.10.

$$\textbf{a) termmag (ss,eqn,term) = bvmag (sscho (ss,eqn,term),}$$

$$\textbf{numbv (ss,eqn,term))}^{\text{bvexp (ss,eqn,term)}} \times \textbf{ coeffmag (ss,eqn,term)}$$

$$\textbf{b) termmag (1,1,2) = bvmag } (3,1)^{0.333} \times \; - 0.993$$

Table 4.10: Example term with output equation coupling

On the first iteration, the value of *bvmag (3,1)* will be 1.0; the value of *dvmag (1,2)* might be 6.34 (recall that it was a previously determined real random number, from between 1 and 10).

Using these values, the magnitudes of termmag (1,1,1) and termmag (1,1,2) are 11.50 and -0.993, respectively. Refer to table 4.11.

$\mathbf{termmag\ (1,1,1) = 0.286 * 6.34^2 = 11.50}$
$\mathbf{termmag\ (1,1,2) = -0.993 * 1.0^{0.333} = -0.993}$

Table 4.11: Magnitudes of the two example terms

Sensitivities for every output term are stored in an array called *sens*. These local derivatives are computed analytically. The derivatives are taken (for each term) with respect to the coupling, be it a coupling to a design variable or to another output equation. For illustrative purposes, the sensitivities for the above 2 terms are listed in table 4.12.

$\mathbf{sens\ (1,1,1) = 0.286 \times ((2) \times dvmag\ (1,2)^1) = 3.62}$
$\mathbf{sens\ (1,1,2) = -0.993 \times ((0.333) \times bvmag\ (3,1)^{-0.666}) = -0.331}$

Table 4.12: Local sensitivities for the two example terms

V. Term magnitude check

After the magnitude of each equation term is computed, the program determines whether or not the magnitude is within the allowable range. For reasons which will be explained later in this chapter, the *allowable range* is $+1.0 \leq termmag \leq +500$ for positive terms, and $-0.25 \geq termmag \geq -100.0$ for negative terms. If the magnitude of the term is within the allowable range, the program proceeds. If not, the coefficient of the term is either raised or lowered 10-fold (as appropriate), and the term magnitude and sensitivity are re-computed. This procedure repeats itself until the magnitude of the term is within the

allowable range. Adjustment of the coefficients typically occurs often on the early iterations, and subsides for the later iterations.

VI. Coefficient magnitude check

After the magnitudes of every equation term in the system have been computed, the program enters a subroutine called ***termrange***. As the name implies, this subroutine places constraints on the magnitudes of each term through the magnitude of the coefficients. It is clear from the previous paragraph that the potential exists for the initial output term coefficients to be heavily adjusted. Coefficients could be substantially raised or lowered, well out of the initial range of 0 to 1. For this reason, there should be reasonable bounds for the coefficient magnitudes, such that they do not expand to infinity or diminish to zero. For reasons that will shortly be explained, the set limits on both the positive and negative coefficients are as follows:

$$+0.000001 \leq \textit{coeffmag} \leq +10000.0$$

$$-0.000001 \geq \textit{coeffmag} \geq -10000.0$$

If, while in this subroutine, a term is found to be in violation of one of these coefficient constraints, then the coefficient is set to be identically equal to *that* constraint value, and the magnitude of the term and sensitivity of the term (***termmag*** and ***sens***, respectively) are raised or lowered, accordingly. (Both ***termmag*** and ***sens*** are linear functions of the coefficient magnitude).

The ***termmag*** subroutine is always entered on the early iterations, when the system is still "developing". However, if, after a certain user defined number of iterations the program must ease the convergence criterion because the system won't converge to the initial

criterion, the program ceases to enter the *termmag* subroutine. While all of the aforementioned restrictions on the coefficient magnitudes are ideal, they are not absolutely necessary. System convergence is imperative.

VII. Equation magnitude check

Upon exiting the *termmag* subroutine, all equation terms have acceptable magnitudes, which include coefficients that themselves have acceptable magnitudes. Now, the magnitude of each and every **output equation** can be calculated. After the characteristics of the **final term** of a given output equation have been determined, the magnitude of the entire output equation (*bvmag*) is determined by simply adding together all of the term-magnitudes that comprise that equation. To prevent divergence and arithmetic difficulties, the upper and lower bounds for the magnitude of each output equation have been set to be 9999 and 0, respectively. The lower value of 0 simply assures that all output equations are positive in magnitude. The upper bound of 9999 was chosen simply to prevent equation magnitudes of greater than 4 figures, and was a heuristical decision. These limitations are applied in a subroutine called *eqnrange*. Before this subroutine can be discussed, the reasoning for these limitations will be presented.

An upper bound is needed, to prevent output equations that have terms with exponents of 2 (*bvexp* = 1) from increasing without bound. For instance, let subsystem number 2, equation number 3, term number 4 have a coupling with subsystem number 5, equation number 4. In addition, let this coupling be ***reciprocal***. In other words, a term in subsystem number 5, equation number 4 (say, for example, term number 6) has a coupling to subsystem 2, equation number 3. Using the jargon terms that were presented earlier:

$$termmag\,(2,3,4) \sim bvmag\,(5,4)^2$$

$$termmag\,(5,4,6) \sim bvmag\,(2,3)^2$$

If the magnitudes of the output equations *bvmag* (5,4) and *bvmag* (2,3) are initially *large*, these couplings would most definitely drive both equations to infinity. For instance, let the magnitude of both equations, after 1 iteration, be 100. With no other terms considered (Recall that the above terms are terms number 4 and 6, respectively, in equations that could potentially have 20 terms, and hence 20 such reciprocal couplings.), the magnitudes of these terms on the second iteration would be $100^2 = 10000$; $10000^2 = 100000000$ on the third iteration, and so on. It is clear that an ***upper bound must be set***, to prevent this likely scenario. As previously noted, 9999.0 has been chosen as the upper bound.

A lower bound is needed to assure that instances of mathematical impossibility will never be reached. Recall that any given equation term will have 1 of a random choice of 6 different exponent values. Three of these 6 choices require that the term being exponentiated be non-negative. For instance, let the exponent of a given equation term be 1/3; hence *bvexp* = 4. Let this term be coupled to subsystem number 4, equation number 3. If the present term is term number 2, of equation number 1, of subsystem number 5, the relevant expression for the term is as follows:

$$termmag\,(5,1,2) \sim bvmag\,(4,3)^{1/3}$$

The magnitude of *bvmag* must be positive. A negative term raised to a fractional power is undefined. The same logic can be applied to the 1/2 and 1/4 exponents, which are 2 of the

other 5 exponent choices. To avoid the possibility of such a scenario, the magnitude of every output equation has been imposed to be greater than 0.0.

The magnitude of each output equation is controlled by controlling each term of the equation. This is accomplished by controlling the magnitude of each term-coefficient. Bounds on the magnitude of each term and the coefficient of each term were imposed earlier in this chapter, without explaining the reasoning behind the bounds. Recall that the magnitudes of each equation term are set to be $+1.0 \leq termmag \leq +500$ for positive terms, and $-0.25 \geq termmag \geq -100.0$ for negative terms. The upper bound for positive terms was attained by dividing the maximum allowable magnitude of a given equation, by the maximum number of terms allowed in that equation; $+9999/20 \approx 500$. The lower bound of +1.0 for the magnitude of each positive equation term is used to enforce that every equation term in the system is *substantial* and *contributing*, to some degree. (A term with a magnitude of 0.0000057 is so small that it has zero contribution, for all intents.) The corresponding bounds for negative equation terms are set to be lower, by a factor of between 4 and 5. Recall that the lower bound on the entire equation is 0.0. Negative equations are to be avoided, hence negative terms are enforced to have narrower bounds than positive terms.

Recall that the minimal and maximal bounds on the equation-term coefficients were set to be 0.000001 and 10000.0, respectively, for both positive and negative terms. The reasoning can now be explained. For example, let a given term have a coupling to another output equation that has attained its maximum magnitude (+9999). Let the exponent of this term be 2. The magnitude of this term, without the coefficient figured in, would be *large*; more specifically, $+9999^2 \approx 100{,}000{,}000$. A coefficient of +0.000001 multiplied by this

magnitude approximately yields +100, which **enforces the magnitude of the term back into its allowable range** [+1.0 ≤ *termmag* ≤ +500]. If the sign of the coefficient were negative, the same logic would hold. The magnitude of the term would then be approximately -100, which is also within the allowable range for a negative term [-0.25 ≥ *termmag* ≥ -100.0].

Conversely, let the exponent of the same term be -1. Then, the magnitude of the term would be *small* instead of large. With the coefficient initially neglected, $(+9999)^{-1} \approx +0.0001$. **To make this term *substantial*, and within the allowable range** [+1.0 ≤ *termmag* ≤ +500] for a positive term, a maximum coefficient of +10000.0 can be multiplied by it, approximately yielding +1.0. If the sign of the coefficient were negative, the same logic would hold. The term magnitude would then be approximately -1.0, which is also within the allowable range for a negative term [-0.25 ≥ *termmag* ≥ -100.0].

Recall that the CASCADE program enters a subroutine called *eqnrange* to impose limitations on the magnitude of each output equation. If the magnitude of a given output equation is found to have a value that is greater than +9999, then changes are made to each term that comprises that equation, depending on the sign of each term. If the given term is positive, then the magnitude of that term is enforced to be identically +500.0, and the coefficient and sensitivity of that term are scaled proportionally. Recall that the quantity +500.0 comes from the division of +9999 (maximum equation magnitude) by 20 (maximum number of terms per equation). Negative terms are left alone. Heuristics have found that lowering the magnitude of an output equation is best achieved by *either* decreasing the positive terms *or* by increasing the negative terms, but not by altering both simultaneously.

If the magnitude of a given output equation is found to be less than 0.0, then certain other changes are made to the terms that comprise the equation, depending on the sign of each term. If the term is negative, then the magnitude of that term is enforced to be identically -1.0, and the coefficient and sensitivity of that term are scaled proportionally. If the term is positive, then the magnitude of that term is enforced to be identically +10.0, and the coefficient and sensitivity of that term are scaled proportionally. These actions will ensure a positive equation, with *manageable* term-magnitudes, for the present iteration.

After every term in the system has gone through the *determ, mags, termrange*, and *eqnrange* subroutines, the system is **sufficiently constructed** for the present iteration. System convergence must now be checked. It is likely that the latter 3 of these 4 subroutines will be encountered at least 40 times, to successfully converge a large system.

VIII. System convergence

The subroutine that checks for system convergence is called *converge*. The first thing that this subroutine does is that it sets the convergence criterion for the current iteration. Recall that the user enters both an initial convergence criterion, as well as convergence ease parameters. These parameters tell the program when (after how many passes) it may ease the initial convergence criterion 10-fold, 100-fold, and when it may check for convergence of all equations to within "1.0". The newly-computed magnitudes of each output equation (*bvmag*) for the present iteration are checked against the magnitudes of the corresponding output equation from the previous iteration. These *old* magnitudes are stored in an array called *bvmagold*. **On the first iteration, there are no old equation magnitudes to compare to,** hence each equation magnitude of the *bvmagold* array is set to equal to each corresponding

equation magnitude of the *bvmag* array, **plus** 10 times the convergence criterion, such that the system will not falsely converge. The absolute value of the differences between the magnitudes of *bvmag* and *bvmagold*, for each output equation, are then checked. If this difference is less than or equal to the present convergence criterion, for every output equation, then the system has converged, and the program will proceed. Upon convergence, the magnitude of each output equation can be written to an output file (called *outeqns.dat*), if so desired.

If, for one ore more equations, the difference between *bvmag* and *bvmagold* is greater than the convergence criterion, then one of two things will happen. The program first checks to see if the initial convergence criterion has been eased. 1) If the convergence criterion has not been eased, or if the convergence criterion has been eased but the maximum number of iterations **has not** been exceeded, then the values of *bvmagold* are set to equal the corresponding values of *bvmag* (for use in the next iteration), and the program is sent back to the *mags* subroutine, for re-computation of each equation term of the system. The program will then enter the *termrange* and *eqnrange* subroutines before entering the *converge* subroutine again, on the next iteration. 2) If the convergence criterion has been eased, and all system equations have not yet converged to within "1.0" of their previous value (after the user specified number of iterations), the program will abruptly exit. This feature was applied to ensure that a non-converging system will eventually exit the program loop. Before exiting, the program will compute the size of the largest (absolute value) difference between *bvmag* and *bvmagold*, for every equation. This quantity will clearly be greater than 1.0, and will give

some indication as to how close the system was to converging after that many iterations. The convergence check procedure is illustrated in the figure 4.1 flowchart.

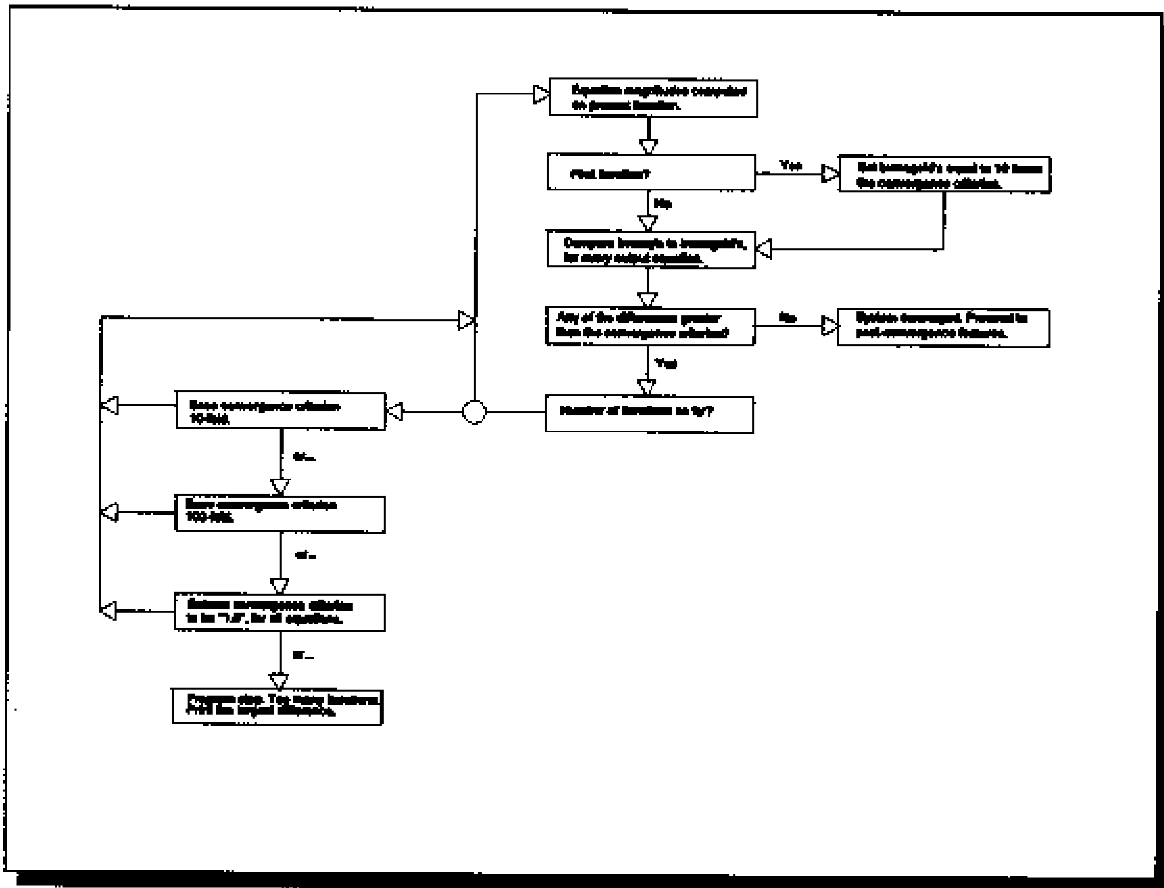

Figure 4.1: Convergence check flowchart

Additional post-convergence features

I. Average coefficient values

After the system has been constructed and has converged, the program will proceed and perform various other features. The first such feature attained in a subroutine called *average*, which computes the average positive and negative term coefficients of the system

Recall further that many term-coefficients were likely modified, to ensure that the coefficient itself, the term that the coefficient is part of, and the entire equation associated with the term, have acceptable magnitudes. It may therefore be insightful (to understand the statistics of the converged system) to look at the numerical average of the positive and negative coefficients of the converged system. Because every equation is enforced to be positive, it is expected that the average positive coefficient will be greater than the average negative coefficient.

The averages are attained by first computing the number of positive and negative terms, stored in quantities *numpos* and *numneg*. This is accomplished by simply counting the number of terms that have a *sign* value of 0 or 1 for positive, and the number having a *sign* value of 2 for negative. Next, quantities *sumpos* and *sumneg* are computed by adding all of the positive and negative coefficient values, respectively. The numerical averaged positive and negative coefficients, called *avgpos* and *avgneg* respectively, simply equal the divisions of *sumpos/numpos* and *sumneg/numneg*, respectively.

II. Equations written to subroutines

The second feature that can be employed to the converged system is a very important one. The user is initially given the option to write each output equation of the converged system to a separate subroutine. (Such subroutines could be used in a system reduction analysis, to view the effects of coupling suspension and elimination. Alternatively, each converged equation (*module*) of the subroutine could be perturbed in some fashion, optimally sequenced, and then re-converged, likely in a more rapid manner). This feature is performed in a subroutine called *eqnsubs*.

If the user decides to write each equation to a subroutine, the user is given the choice of what computer language the subroutines will be written in either FORTRAN 77 or ANSI C. The equation will then be written to a file called *eqns.for* or *eqns.c*, as the case may be. The first item that is written to each subroutine is the title. The titles of each subroutine are *eqnxxxx*, where "xxxx" ranges from 0001 to a maximum of 9999. These 4 numbers represent the total number of equations in the system, listed sequentially. Following the title, the output equation (*bvmag*) and design variable (*dvmag*) array dimensions are written to the subroutine, and the *bvmag common* array declaration is written to the subroutine. This statement is needed in the subroutine so that each output equation need not be explicitly declared in each subroutine. (There could be 9999 of them!) Both of these operations are used for writing FORTRAN subroutines, and are not required for writing ANSI C subroutines. Instead, in ANSI C programming, a subroutine is initiated by typing "{". Hence, this symbol is written to an ANSI C subroutine (and is obviously not required for writing a FORTRAN subroutine). Next, expressions for the design variable magnitudes, of the subsystem that contains the given output equation, are written to the subroutine. Recall that the number of design variables per subsystem ranges from 1 to 5. Hence, from 1 to 5 design variable quantities are written to each subroutine.

The output equation itself is written to the subroutine by using character strings. Each term of the output equation is written to the subroutine as follows. Character string *char2* stores the sign of the term. Hence, a *sign* value of either 0 or 1 for a given term would yield a *char2* value of "+"; a *sign* value of 2 would yield a *char2* value of "-". Character string *char3* depends on whether the coupling of the present term involves a design variable, or

another output equation. If the coupling involves a design variable, then *char3* = "*dvmag*(". Otherwise, an output equation coupling yields a *char3* value = "*bvmag*(". The first argument of the coupling is stored in a real array called *one*. If the coupling of the term involves a design variable, then the value of *one* would be the value of the subsystem that the present term belongs to. If the coupling involves another output equation, then the value of *one* would be the chosen subsystem coupling (*sscho*) of the given term. String *char4* is a comma, ",", and is used to separate the two arguments of the coupling. The second argument of the coupling is stored in a real array called *two*. If the present coupling involves a design variable, then the value of *two* equals the chosen design variable number (*numdv*) of the subsystem of the present term. If the coupling involves another output equation, then the value of *two* equals the output equation number of the subsystem that was chosen for the coupling (*numbv*). String *char5* is an end parenthesis, ")", and is used to complete the coupling arguments. Finally string *char6* stores the exponent of the term-coupling, and thus depends on either parameter *dvexp* or parameter *bvexp*. For example, if the exponent of the present *design variable coupled* term is 1/2 (*dvexp* = 3), then *char6* would equal: "**.50". Similarly, if the exponent of the present *output equation-coupled* term is -1 (*bvexp* = 2), then *char6* would equal: "**-1.". (The above example has assumed that the subroutine is being written in FORTRAN 77. An analogous procedure would be employed for the writing of an ANSI C subroutine). To clarify this paragraph, table 4.13 summarizes the character string values, for both design variable-coupled and output equation-coupled terms. The table again assumes the writing of FORTRAN 77 subroutines.

character:	design variable coupling	output equation coupling
char2	+ or -	+ or -
char3	dvmag(	bvmag(
one	(subsystem number)	(sscho)
char4	,	,
two	(numdv)	(numbv)
char5	)	)
char6	(dvexp)	(bvexp)

Table 4.13: Character strings used for writing equations to subroutines

To ensure a clear understanding of this methodology, an example term is presented. Assume that term number 3, of equation number 2, of subsystem number 1, is written to a FORTRAN 77 subroutine. (This subroutine equation may contain from between 1 and 20 terms. This example shows the procedure only for term number 3. Hence, in this example, this equation is assumed to have *at least* 3 terms.) Let this term have a coefficient of 0.582, and be positive (*sign* = 0). Assume that this term has a coupling to another output equation - subsystem number 4 (*sscho* = 4), equation number 2 (*numbv* = 2). Let the exponent of the coupling be 1 (*bvexp* = 0). The write statement that would be used to write this term to the subroutine would appear as in table 4.14.

write · · · , *char2(3), coeffmag(1,2,3), char3(3), one(3), char4(3), two(3),*

char5(3), char6(3), · · ·

where:

***char2*(3) = "+", *coeffmag*(1,2,3) = 0.582, *char3*(3) = "*bvmag*(", *one*(3) = 4,**

***char4*(3) = ",", *two*(3) = 2, *char5*(3) = ")", and *char6*(3) = "**1.0".**

Table 4.14: Example write statement for one term, written in FORTRAN

For each output equation (and hence, for each subroutine) the above procedure is repeated for all 20 terms of the equation. If the equation contains less than 20 terms, then zeros are added to the end of the equation, in place of the previously explained character string quantities. Finally, a *completion* is written to the subroutine. In FORTRAN, this includes the *return* and *end* statements. In ANSI C, only a "}" symbol is required, to complete the subroutine.

III. Sensitivity computation

The third of the post-convergence features of the CASCADE package is also very important. Total sensitivities can be computed, using a subroutine called *derivs*. Recall the form of the Global Sensitivity Equations first presented in equation [2.6], and repeated in figure 4.2, for clarity:

Observe that the total derivatives can be attained by first attaining the local output-output sensitivities of the Left Hand Side (LHS) matrix, and the local output-input sensitivities of the Right Hand Side (RHS) matrix. With the magnitudes of these two matrices known, the total derivative matrix can be attained by using an LU-decomposition with numerical conditioning.

First, the LHS matrix is computed. First, all diagonal elements of the matrix are assigned a value of unity. The LHS matrix is a square matrix, and the diagonal terms represent the sensitivity of each output equation, with respect to itself, which is simply 1.0. The off-diagonal elements are computed one at a time, by cycling through the rows of the LHS matrix. Recall that each output equation "Y" contains from 1 to 20 terms within. This implies that each output "Y" (and hence each row of the LHS matrix) could have anywhere from 1 to 20 non-zero elements, which represent the coupling sensitivities of the current "Y" value, to the other output equations. An example is presented to clarify this methodology. Let there exist a 3 subsystem system, each having 2 equations. The LHS matrix will thus be (6×6), and will have the following form:

$$LHS = \begin{bmatrix} sens_{11,11} & sens_{11,12} & sens_{11,21} & sens_{11,22} & sens_{11,31} & sens_{11,32} \\ sens_{12,11} & sens_{12,12} & sens_{12,21} & sens_{12,22} & sens_{12,31} & sens_{12,32} \\ sens_{21,11} & sens_{21,12} & sens_{21,21} & sens_{21,22} & sens_{21,31} & sens_{21,32} \\ sens_{22,11} & sens_{22,12} & sens_{22,21} & sens_{22,22} & sens_{22,31} & sens_{22,32} \\ sens_{31,11} & sens_{31,12} & sens_{31,21} & sens_{31,22} & sens_{31,31} & sens_{31,32} \\ sens_{32,11} & sens_{32,12} & sens_{32,21} & sens_{32,22} & sens_{32,31} & sens_{32,32} \end{bmatrix} \qquad [4.1]$$

where, for instance, $sens_{11,21}$ is the sensitivity of subsystem number 1, equation number 1, to subsystem number 2, equation number 1, and so forth. For example, let equation number 1 of subsystem number 1 contain 4 terms. The first term is, as always, a design variable ("X")

coupling, and has no contribution to this matrix. The second term has a coupling to subsystem number 3, equation number 2. The computed value of this sensitivity was found to be 0.069. The third term has a coupling to subsystem number 2, equation number 1. The computed value of this sensitivity was found to be -0.480. Lastly, the fourth term is another design variable coupling, and has no contribution to this matrix. Hence, the first row of the LHS matrix, which results from the couplings of subsystem number 1, equation number 1, to other output equations, appears as follows:

$$LHS(output1,1) = \begin{bmatrix} 1 & 0 & -0.480 & 0 & 0 & 0.069 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad [4.2]$$

Recall that the diagonal elements of the matrix always have a value of unity, as the sensitivity of any output, with respect to itself, equals 1.0. CASCADE uses similar methodology to compute the other rows of the LHS matrix. All off-diagonal elements of the LHS matrix are multiplied by -1, to put the matrix in the form of figure 4.2. Using techniques that were described earlier (see equation 2.6), the LHS matrix is then normalized, term by term, to improve its numerical condition for the forthcoming LU-decomposition.

The computation of the RHS matrix uses similar methodology, but involves all of the design variable couplings instead of the output couplings. Recall that the first term of every

output equation is enforced to be a design variable. Therefore, every output equation "Y" will contain at least one coupling to a design variable "X". It is possible that a select few of the other terms of each output equation could as well have a design variable coupling. Again, the off-diagonal elements are computed one at a time, by cycling through the rows of the RHS matrix. The same example that was used for the LHS matrix will be used for the RHS matrix, to clarify this methodology. Recall that the system has 3 subsystems, each having 2 equations. Let the first subsystem contain 4 design variables, the second, 1 design variable, and the third, 2 design variables. Hence, the RHS matrix will be (6×7); 6 rows symbolizing the 6 equations in the system, and 7 columns representing the 7 design variables in the system. The form of the matrix is as follows:

$$RHS = \begin{bmatrix} sens_{11,11} & sens_{11,12} & sens_{11,13} & sens_{11,14} & sens_{11,21} & sens_{11,31} & sens_{11,32} \\ sens_{12,11} & sens_{12,12} & sens_{12,13} & sens_{12,14} & sens_{12,21} & sens_{12,31} & sens_{12,32} \\ sens_{21,11} & sens_{21,12} & sens_{21,13} & sens_{21,14} & sens_{21,21} & sens_{21,31} & sens_{21,32} \\ sens_{22,11} & sens_{22,12} & sens_{22,13} & sens_{22,14} & sens_{22,21} & sens_{22,31} & sens_{22,32} \\ sens_{31,11} & sens_{31,12} & sens_{31,13} & sens_{31,14} & sens_{31,21} & sens_{31,31} & sens_{31,32} \\ sens_{32,11} & sens_{32,12} & sens_{32,13} & sens_{32,14} & sens_{32,21} & sens_{32,31} & sens_{32,32} \end{bmatrix} \quad [4.3]$$

where, for instance, $sens_{11,14}$ is the sensitivity of subsystem number 1, equation number 1, to subsystem number 1, design variable number 4, and so forth. Recall that in the previous example, equation number 1 of subsystem number 1 contains 4 terms. Recall again that both the first and fourth terms of this output equation have design variable couplings. Let term number 1 be coupled to design variable number 3, (of the subsystem to which it belongs; subsystem number 1). Similarly, let term number 4 be coupled to design variable number 3,

of subsystem number 1. The computed values of these sensitivities were found to be 7.93 and -3.66, respectively. Hence, the first row of the RHS matrix, which results from the couplings of subsystem number 1, equation number 1, to its design variables, appears as follows:

$$RHS(output1,1) = \begin{bmatrix} 0 & 0 & 4.27 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad [4.4]$$

Notice that there exists only non-zero element, due to the design variable couplings of output equation number 1 of subsystem number 1. In this example, the equation contained only 4 terms, 2 of which were design variables. Both of the design variable couplings were to the *third* design variable of subsystem number 1. Hence, the numerical value of RHS (1,3) is simply the sum of these two sensitivities. All other elements of this row equal zero. CASCADE uses similar methodology to compute the other rows of the RHS matrix. Using techniques that were described earlier (again, see equation 2.6), the RHS matrix is normalized, term by term, to improve its numerical condition for the forthcoming LU-decomposition.

With the LHS and RHS matrices computed, there is enough information to compute the matrix of total derivatives. The dimension of this matrix will be (m×n), where m is the total number of equations in the entire system, and n is the total number of design variables

in the system. Given the LHS and RHS matrices, the *GLMCOIT* package of subroutines is used to compute the total derivative matrix (called *dydx*), by using LU-decomposition. The two matrices can be either the normalized or *not normalized* versions; it is up to the user. The effects of matrix normalization will be discussed in a later chapter. If the normalized matrices are used in GLMCOIT to attain the total derivatives, the *total derivative* matrix that is obtained will also be normalized, and called *normdydx*. This matrix must also be un-*normalized*, term by term, to obtain the actual total derivative matrix (*dydx*). See equation 2-7 for these term-recovery techniques.

Before exiting the *sens* subroutine, the user is given the option to write the computed total derivatives to an output file, called *total.dat*. Also written to this file is the *condition number* of the computed total derivative matrix. If *not-normalized*, poorly conditioned matrices were used to compute the total derivatives, then the condition number will be relatively large, and the reliability of the numerical estimate will be relatively small. If normalized and conditioned matrices were used to attain the total derivatives, the condition number will likely be small, and the reliability of the numerical estimate will likely be large.

IV. Term statistics

The fourth and final post-convergence feature of the CASCADE package is the option to write the chosen "term-parameters" [illegible] *Parameters* to an output file, called *params.dat*. This feature is executed in a subroutine called *param*. Written to this file first are a series of 3 CPU times: the time required to build and converge the system of equations; the time required to compute the total sensitivities of the system; and the time required to execute the entire program. Other statistical parameters that are written to this file include the average coefficients (for

both positive and negative terms), and the total number of iterations required for convergence. The primary block of information written to this file are the term parameters themselves. For each and every term of the system, the values for *sscho, numbv, bvexp, numdv, dvexp, sign*, and *coeffmag* are tabulated in this data file. In addition, the magnitudes of all of the design variables, for each subsystem in the system, are appended to this file. This file provides the user with concise insight to the nature of the system that was randomly generated.

Closure

This chapter has provided an in-depth description of the development and operation of the CASCADE package, and the various design considerations therein. CASCADE, as has been explained thus far, was initially designed to construct and converge the system of equations sequentially, on one computer. With this framework, the operation of CASCADE has been expanded to a parallel network of computers, and has made use of distributed computing techniques with the aid of Parallel Virtual Machine (PVM). Chapter 5 will focus on the discussion of this transition.

Chapter 5

CASCADE implementation using Parallel Virtual Machine (PVM)

System construction and convergence

The principals of distributed computing have been used to extend the operation of a portion of the CASCADE package, from a sequential to a parallel computing environment. Specifically, a parallel machine is used to construct, and then sequentially converge a system of equations. The post-convergence features that were described in the last chapter will still be performed on a single computer.

The files associated with the execution of CASCADE in parallel are called *cascmast.f* and *cascslav.f*. The former is the master program, and the latter is the slave program, using the jargon that was presented in Chapter 3. The execution of CASCADE in parallel, with these two files, begins by executing the master program, *cascmast.f* , on a single machine. As with the sequential version, the program first reads the input file (*cascin.dat*) parameters, and proceeds to initialize a number of parameters of its own, with the aide of the random number generator. These parameters include the number of design variables per subsystem, the magnitudes of the design variables, the number of terms per output equation, and the initial magnitudes of the term-coefficients and the output equations.

Once all of the preliminaries have been taken care of on the *master node*, distributed computing can take place. PVM is initiated within the master program by issuing the

pvmfmytid command. Before doing so, a *no more work flag*, called *nmwf*, is set to equal a positive integer. As long as the system is in either a construction or "not-converged" state, there is more work to be done by the slave tasks, and this parameter will remain positive.

Next, the *pvmfconfig* command is issued, mainly to detect the number of host computers that are available on the virtual machine. The number of separate slave tasks that can be issued is equal to the total number of hosts mounted on the virtual machine (called parameter *hosts*) minus 1. Quantity (*hosts* - 1) is referred to as *nhost*. Remember that one of the hosts on the virtual machine is executing the master program itself; hence the value of quantity *nhost*. Next, an *nhost* number of tasks are spawned; in other words, the slave program *cascslav.f* is spawned *nhost* times. Each instance of the slave program is executing on a separate machine, and is sent a variety of parameters with which to perform a variety of computations. These parameters include: a seed for random number generation; **a subsystem number**; the no more work flag value; the number of output equations per the present subsystem; an array containing the number of terms per output equation; the iteration number; the number of design variables per the present subsystem; the array containing the term coefficient magnitudes; the arrays containing the design variable and output equation magnitudes; and the array containing the output equation magnitudes from the previous iteration. These parameters are sent to the slave tasks by issuing a sequence of three commands; *pvmfinitsend* to initialize the send buffer, *pvmfpack* to pack the data into the buffer (either as *integer4*, *real4*, or *real8* for double precision), and *pvmfsend* to send the data to the slaves.

Each slave program that was spawned by the master must issue two preliminary commands: *pvmfmytid* to enroll in PVM as a slave, and then *pvmfparent* to find the task

identification number of the program that spawned the task (in this case, the task id of *cascmast.f*). Thereafter, each slave program must first receive, and then unpack the data that was sent to it by the master. The commands are *pvmfrecv* and *pvmfunpack*, respectively. After receiving the data, each slave decides whether or not to proceed. This is decided by the value of the no more work flag, *nmwf*. As long as this value is positive, there is more work to be done. Each slave then proceeds, **handling only one subsystem**, to call the *determ*, *mags*, *termrange*, and *eqnrange* subroutines, to determine the equation-term parameters, and then compute and alter the corresponding magnitudes of the quantities involved, as was previously described. The newly computed and the re-calculated data within each slave program is then sent back to the master program for analysis. For CASCADE, this data includes: the coupling arrays *sscho, numbv, bvexp, numdv, dvexp*, and *sign*; the coefficient magnitude array; the sensitivity array; the output equation magnitude array; and the task id of the slave itself, called *mytid*, locally. The slave then proceeds to the top of the program, and awaits either more work or a negative no more work flag, telling the slave to exit.

The master then receives the newly computed and re-calculated data back from each slave. If there are more subsystems to be solved for the present iteration, then the master sends each *completed* slave a new subsystem number, and associated parameters, for more calculations. If all subsystems have been sent out to the various slaves for analysis, the master node will simply wait until it has received newly-computed data back from all of the slaves, for all of the subsystems. Once all subsystems have been processed by the slaves, and all information sent back to and received by the master, a convergence check can take place. This check takes place sequentially, on one computer. If the system has not converged, the

distributed-computing process is repeated, using the updated magnitudes of every output equation in the system until convergence. If the system is found to have converged, the program proceeds, on one computer, through the *average, eqnsubs, derivs*, and *param post-convergence* subroutines. The master program assigns the no more work flag a negative value, and sends this value to all of the slaves, allowing them to exit PVM, using the *pvmfexit* command. The master program then issues this command, itself, terminating the current run of CASCADE in parallel.

Closure

This chapter has outlined the procedure for operating the CASCADE package in a distributed-computing framework, using PVM. This operation stemmed from an extension of CASCADE's sequential operation, on one computer, as was fully explained in Chapter 4. The next chapter will present various example systems that were created by using CASCADE, in both sequential and distributed environments. The presentation of these example systems will likely clarify the details of the last two chapters, and will provide a guideline for the analysis and comparison of CASCADE's infinitely possible output systems.

Chapter 6

Example problems created by using CASCADE

<u>Input file</u>

Recall that the first thing that the user must do to run CASCADE is create an input file, called *cascin.dat*. This is accomplished by answering a series of system related questions, by executing a program called *cascin.f*. This chapter will present two *system cases*. Very generally, case 1 was created sequentially on one computer, and case 2 was created by using distributed computing (PVM). The input files for the two cases are seen in table 6.1.

Case 1	Case 2
-1804	-5089
3	3
2	2
2	2
3	3
2	2
7	7
1	1
1	1
1	1
1.0E+11	1.0E+11
0	0
2	2
2	2
2	2
2	2
1	1
1	1
1	1
1	1
1	2
5	5
100	100
200	200
400	400

Table 6.1: Input files for the two system cases

These numbers provide the numerical values and the option flags that are required to execute CASCADE. *The italicized values towards the top of the table* reflect that both of these system cases have 3 subsystems, each with 2 equations in the first and third subsystems, and each with 3 equations in the second subsystems. Hence, there are a total of 7 output equations per system.

"Parameters" file

Table 6.2a is an excerpt from the parameters file (called *params.dat*) that was created for the first system case.

CPU time required to build and converge the system (seconds)= 0.75260
CPU time required to acquire the sensitivities (seconds)= 0.12721
The time required to execute the entire program (seconds)= 1.06768

SS# 1 BV# 1 term# 1
sscho= 0; numbv=**; bvexp=*; numdv=1; dvexp=5; sign=0; coefficient= 0.826497
SS# 1 BV# 1 term# 2
sscho= 2; numbv= 1; bvexp=1; numdv=*; dvexp=*; sign=1; coefficient= 0.000094
SS# 1 BV# 1 term# 3
sscho= 2; numbv= 1; bvexp=0; numdv=*; dvexp=*; sign=1; coefficient= 0.064643
SS# 1 BV# 1 term# 4
sscho= 2; numbv= 1; bvexp=4; numdv=*; dvexp=*; sign=2; coefficient= -0.537248
SS# 1 BV# 1 term# 5
sscho= 0; numbv=**; bvexp=*; numdv=2; dvexp=3; sign=0; coefficient= 5.189545

SS# 3 BV# 2 term# 1
sscho= 0; numbv=**; bvexp=*; numdv=4; dvexp=1; sign=0; coefficient= 0.765490
SS# 3 BV# 2 term# 2
sscho= 0; numbv=**; bvexp=*; numdv=4; dvexp=3; sign=1; coefficient= 4.665359
SS# 3 BV# 2 term# 3
sscho= 2; numbv= 1; bvexp=5; numdv=*; dvexp=*; sign=0; coefficient= 2.905074
SS# 3 BV# 2 term# 4
sscho= [illegible]; numbv= [illegible]; bvexp=[illegible]; numdv=[illegible]; dvexp=[illegible]; sign=1; coefficient= [illegible]
SS# 3 BV# 2 term# [illegible]
sscho= 1; numbv= 1; bvexp=1; numdv=*; dvexp=*; sign=2; coefficient= -0.000183
SS# 3 BV# 2 term# 6
sscho= 1; numbv= 2; bvexp=3; numdv=*; dvexp=*; sign=0; coefficient= 1.455039
SS# 3 BV# 2 term# 7
sscho= 1; numbv= 1; bvexp=0; numdv=*; dvexp=*; sign=2; coefficient= -0.042755
SS# 3 BV# 2 term# [illegible]
sscho= 0; numbv=**; bvexp=*; numdv=2; dvexp=0; sign=0; coefficient= 172.523101
SS# 3 BV# 2 term# 9
sscho= 1; numbv= 1; bvexp=3; numdv=*; dvexp=*; sign=1; coefficient= 2.067727
SS# 3 BV# 2 term# 10
sscho= 0; numbv=**; bvexp=*; numdv=4; dvexp=3; sign=2; coefficient= -0.434359
SS# 3 BV# 2 term# 11
sscho= 0; numbv=**; bvexp=*; numdv=4; dvexp=1; sign=0; coefficient= 0.473740
SS# 3 BV# 2 term# 12
sscho= 0; numbv=**; bvexp=*; numdv=2; dvexp=4; sign=0; coefficient= 25.814882
SS# 3 BV# 2 term# 13
sscho= 0; numbv=**; bvexp=*; numdv=4; dvexp=1; sign=2; coefficient= -0.047374
SS# 3 BV# 2 term# 14
sscho= 2; numbv= 3; bvexp=5; numdv=*; dvexp=*; sign=1; coefficient= 2.543852
SS# 3 BV# 2 term# [illegible]
sscho= 0; numbv=**; bvexp=*; numdv=1; dvexp=0; sign=0; coefficient= 1.059242

Design variables for the current system of equations:

Subsystem #: 1; dvmag(1) = 5.156
Subsystem #: 1; dvmag(2) = 1.734
Subsystem #: 1; dvmag(3) = 9.702
Subsystem #: 2; dvmag(1) = 9.098
Subsystem #: 3; dvmag(1) = 9.441
Subsystem #: 3; dvmag(2) = 0.058
Subsystem #: 3; dvmag(3) = 4.188
Subsystem #: 3; dvmag(4) = 4.594

The average positive coefficient= 335.9032899
The average negative coefficient= -0.7483856
The system converged after 22 passes.

Table 6.2a: Parameters file excerpt for the first system case

The heading of the file shows the user various CPU times for the present system. This excerpt shows the parameters for the first and last equations in this system; subsystem number 1, output equation number 1, and subsystem number 3, output equation number 2. For each and every term of the output equation, the six coupling values are listed (*sscho, numbv, bvexp, numdv, dvexp,* and *sign*), as well as the value of the term-coefficient, for the converged system. Note that a term that has a non-zero value for *numbv* and *bvexp* will **not have values** for *numdv* and *dvexp*, and vice-versa. (A term is coupled **either** to a subsystem design variable **or** to another subsystem output equation; not to both). This nullity is denoted by asterisks in the data file. Appended to the end of the parameters file are the numerical values for every design variable in the system (there are 8 total), the values of the average positive and negative coefficients, and the number of iterations that were required for convergence. Table 6.2b lists similar data, for the second system case. In this system, note that there are 6 total design variables.

CPU time required to build and converge the system (seconds)= 0.37043
CPU time required to acquire the sensitivities (seconds)= 0.03642
The time required to execute the entire program (seconds)= 0.46114

SS# 1 BV# 1 term# 1
sscho= 0; numbv=**; bvexp=*; numdv=3; dvexp=4; sign=1; coefficient= 4.581233
SS# 1 BV# 1 term# 2
sscho= 3; numbv= 2; bvexp=2; numdv=*; dvexp=*; sign=1; coefficient= 678.811371
SS# 1 BV# 1 term# 3
sscho= 3; numbv= 1; bvexp=1; numdv=*; dvexp=*; sign=0; coefficient= 0.000078
SS# 1 BV# 1 term# 4
sscho= 3; numbv= 1; bvexp=3; numdv=*; dvexp=*; sign=0; coefficient= 2.572560

SS# 3 BV# 2 term# 1
sscho= 0; numbv=**; bvexp=*; numdv=1; dvexp=1; sign=0; coefficient= 1.373290
SS# 3 BV# 2 term# 2
sscho= 2; numbv= 2; bvexp=2; numdv=*; dvexp=*; sign=2; coefficient= -880.118236
SS# 3 BV# 2 term# 3
sscho= 2; numbv= 1; bvexp=2; numdv=*; dvexp=*; sign=1; coefficient= 2147.589773
SS# 3 BV# 2 term# 4
sscho= 2; numbv= 3; bvexp=3; numdv=*; dvexp=*; sign=0; coefficient= 4.540861
SS# 3 BV# 2 term# 5
sscho= 1; numbv= 2; bvexp=2; numdv=*; dvexp=*; sign=1; coefficient= 8251.212239
SS# 3 BV# 2 term# 6
sscho= 2; numbv= 1; bvexp=3; numdv=*; dvexp=*; sign=1; coefficient= 8.121039
SS# 3 BV# 2 term# 7
sscho= 0; numbv=**; bvexp=*; numdv=1; dvexp=5; sign=1; coefficient= 3.085842
SS# 3 BV# 2 term# 8
sscho= 2; numbv= 1; bvexp=2; numdv=*; dvexp=*; sign=0; coefficient= 637.509406
SS# 3 BV# 2 term# 9
sscho= 2; numbv= 3; bvexp=4; numdv=*; dvexp=*; sign=1; coefficient= 6.200842
SS# 3 BV# 2 term# 10
sscho= 2; numbv= 1; bvexp=1; numdv=*; dvexp=*; sign=2; coefficient= -0.000756

Design variables for the current system of equations:
Subsystem #: 1; dvmag(1) = 5.300
Subsystem #: 1; dvmag(2) = 3.377
Subsystem #: 1; dvmag(3) = 3.647
Subsystem #: 2; dvmag(1) = 3.497
Subsystem #: 2; dvmag(2) = 1.911
Subsystem #: 3; dvmag(1) = 1.425

The average positive coefficient= 632.7095337
The average negative coefficient= -157.0396881
The system converged after 23 passes.

Table 6.2b: Parameters file excerpt for the second system case

To provide some indication as to the complexities of the couplings, even in these small and simple systems that have been created, observe figure 6.1. The figure is a block diagram that illustrates the module inter-relations *for the table 6.2b system*. Realize that **the couplings of only 2 of the 7 modules are drawn**: those of the first and last equations, of the 7-equation system.

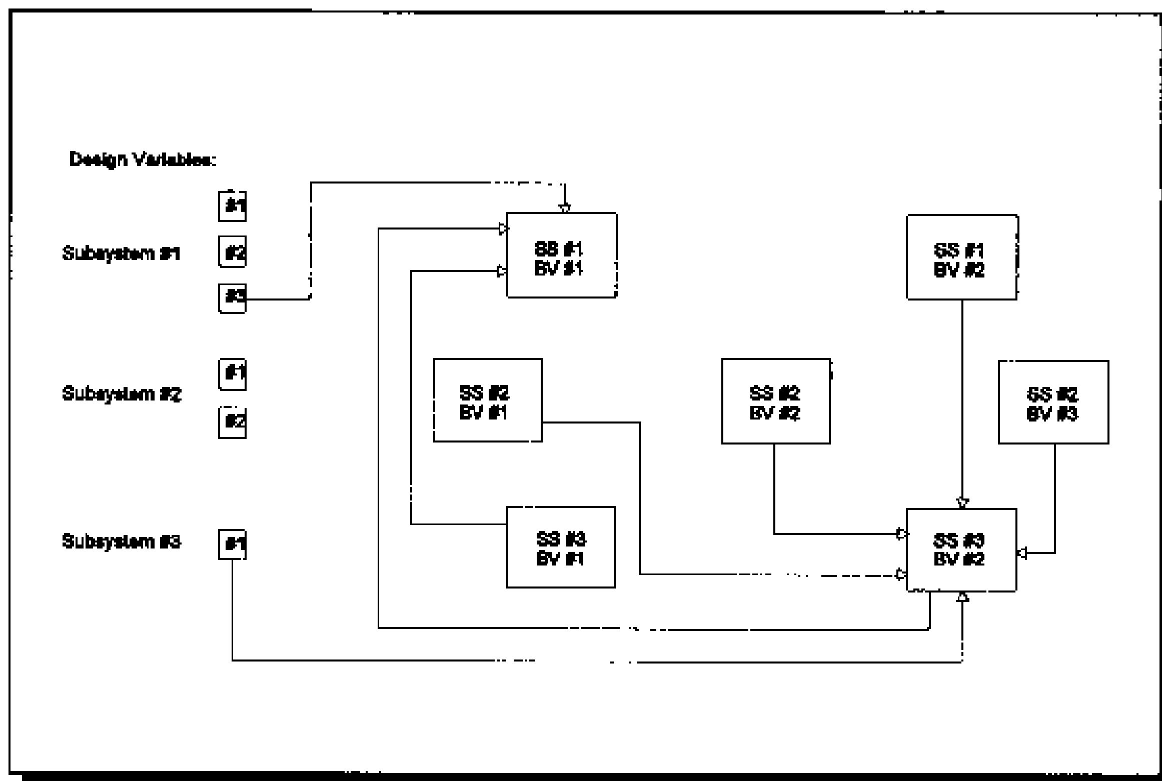

Figure 6.1: Block drawing depiction of the second system case couplings

<u>Output magnitudes</u>

While the parameters file provides the user with the detailed nature of each term, the *outeqns.dat* file shows the user the final, converged magnitudes of every equation in the system. Again, these magnitudes represent the summation of the magnitudes of every term

in each equation. Table 6.3 shows the final magnitudes of the 7 output equations for the two system cases.

Case 1		Case 2	
bvmag(1, 1)=	129.1590	bvmag(1, 1)=	90.9539
bvmag(1, 2)=	597.9459	bvmag(1, 2)=	688.1819
bvmag(2, 1)=	865.3136	bvmag(2, 1)=	58.7005
bvmag(2, 2)=	17.1665	bvmag(2, 2)=	734.9493
bvmag(2, 3)=	901.0440	bvmag(2, 3)=	196.0485
bvmag(3, 1)=	310.3985	bvmag(3, 1)=	527.6104
bvmag(3, 2)=	160.1847	bvmag(3, 2)=	223.5547

Table 6.3: Output magnitudes for the two system cases

Note primarily that all output equations are positive. This (as discussed in Chapter 4) prevents undefined exponentiations, that would be encountered when attempting to raise a *negative* output equation to a fractional power.

Sensitivity information

Recall that the user has the option to compute the total derivatives of every output equation in the system, with respect to every input (design variable). Table 6.4a lists these total derivatives for the first system case. The option to compute the derivatives was chosen when executing the input file. When the user decides to compute sensitivities, he or she must also make two other decisions. First, whether or not to use normalization techniques in attaining the total derivatives. Second, whether or not to use the default LU-decomposition matrix conditioning parameters for attaining the total derivative matrix, from the left and right hand side matrices. For both of the system cases presented here, normalization techniques were employed, and the default decomposition parameters were used.

The condition number of this estimate is: 19.63779

Element	Value
dY/dX(1, 1) =	-0.1144795939
dY/dX(1, 2) =	3.0181250672
dY/dX(1, 3) =	0.0681485832
dY/dX(1, 4) =	5.9125843048
dY/dX(1, 5) =	0.3731823564
dY/dX(1, 6) =	81.2975006104
dY/dX(1, 7) =	-0.0095281163
dY/dX(1, 8) =	4.2234354019
dY/dX(2, 1) =	-1.0068091154
dY/dX(2, 2) =	5.9812073708
dY/dX(2, 3) =	0.3918518126
dY/dX(2, 4) =	16.0213813782
dY/dX(2, 5) =	0.4674587250
dY/dX(2, 6) =	102.3620376587
dY/dX(2, 7) =	-0.0270002671
dY/dX(2, 8) =	5.3177456856
dY/dX(3, 1) =	-0.7770560980
dY/dX(3, 2) =	4.6563863754
dY/dX(3, 3) =	0.3028360009
dY/dX(3, 4) =	26.2741107941
dY/dX(3, 5) =	1.6583331823
dY/dX(3, 6) =	361.2666625977
dY/dX(3, 7) =	-0.0423406698
dY/dX(3, 8) =	18.7679386139
dY/dX(4, 1) =	0.0000000000
dY/dX(4, 2) =	0.0000000000
dY/dX(4, 3) =	0.0000000000
dY/dX(4, 4) =	0.5568355322
dY/dX(4, 5) =	0.0000000000
dY/dX(4, 6) =	0.0000000000
dY/dX(4, 7) =	0.0000000000
dY/dX(4, 8) =	0.0000000000
dY/dX(5, 1) =	-1.6777621508
dY/dX(5, 2) =	10.0151004791
dY/dX(5, 3) =	0.6534714699
dY/dX(5, 4) =	33.4415092468
dY/dX(5, 5) =	0.9081594348
dY/dX(5, 6) =	198.0003662109
dY/dX(5, 7) =	-0.0277222525
dY/dX(5, 8) =	10.2861919403
dY/dX(6, 1) =	-1.0207289457
dY/dX(6, 2) =	6.8604078293
dY/dX(6, 3) =	0.4053069353
dY/dX(6, 4) =	18.7997589111
dY/dX(6, 5) =	0.6369510889
dY/dX(6, 6) =	132.0486755371
dY/dX(6, 7) =	0.1757465601
dY/dX(6, 8) =	6.8599777222
dY/dX(7, 1) =	-0.0436308160
dY/dX(7, 2) =	0.2620775700
dY/dX(7, 3) =	0.0170102250
dY/dX(7, 4) =	0.8510763645
dY/dX(7, 5) =	1.0921310186
dY/dX(7, 6) =	236.9910430908
dY/dX(7, 7) =	-0.0013054742
dY/dX(7, 8) =	12.3117723465

Table 6.4a: Total derivative matrix for the first system case

Take note of two aspects of the information in this file. First, the relative *small* size of the condition number. With a small system such as this one, matrix ill-conditioning is not expected, and hence a small condition number is intuitive. Moreover, normalization techniques were employed in attaining this matrix. The purpose of normalizing each matrix element is to enforce all matrix elements to be of the same order of magnitude, thus reducing the likelihood of an ill-conditioned matrix. Second, note the dimension of the matrix: (7×8). The 7 rows represent the 7 subsystems in the system; the 8 columns represent the 8 design variables in the system. Similar data is seen in table 6.4b for the second system case. Note that the condition number is of similar magnitude, and that the matrix is now only of order (7×6), as the second system contains only 6 design variables.

```
The condition number of this estimate is:    17.54262

dY/dX( 1, 1) =    0.0000000000
dY/dX( 1, 2) =   -0.0568428747
dY/dX( 1, 3) =    0.7164396048
dY/dX( 1, 4) =    0.1165984273
dY/dX( 1, 5) =   -0.0095997136
dY/dX( 1, 6) =    1.5605872869
dY/dX( 2, 1) =    0.0000000000
dY/dX( 2, 2) =    6.6628861427
dY/dX( 2, 3) =   -0.8881819248
dY/dX( 2, 4) =    1.4089199305
dY/dX( 2, 5) =    0.0009145011
dY/dX( 2, 6) =   12.0316715240
dY/dX( 3, 1) =    0.0000000000
dY/dX( 3, 2) =    0.0557988882
dY/dX( 3, 3) =   -0.0062947720
dY/dX( 3, 4) =   -1.2396259308
dY/dX( 3, 5) =   -0.0001429401
dY/dX( 3, 6) =    0.4106028080
dY/dX( 4, 1) =    0.0000000000
dY/dX( 4, 2) =   -0.1698314250
dY/dX( 4, 3) =    0.7532086372
dY/dX( 4, 4) =    2.6747479439
dY/dX( 4, 5) =   -0.7584438920
dY/dX( 4, 6) =   19.8240299225
dY/dX( 5, 1) =    0.0000000000
dY/dX( 5, 2) =   -0.2484287769
dY/dX( 5, 3) =    0.0948477909
dY/dX( 5, 4) =    1.1564078331
dY/dX( 5, 5) =   -0.0026999870
dY/dX( 5, 6) =    6.6165437698
dY/dX( 6, 1) =    0.0000000000
dY/dX( 6, 2) =   -0.4291304052
dY/dX( 6, 3) =    0.5240863562
dY/dX( 6, 4) =    0.9095312953
dY/dX( 6, 5) =   -0.0694639236
dY/dX( 6, 6) =   11.8183383942
dY/dX( 7, 1) =    0.0000000000
dY/dX( 7, 2) =   -0.1923779994
dY/dX( 7, 3) =    0.0402156822
dY/dX( 7, 4) =    0.6933003068
dY/dX( 7, 5) =   -0.0018028717
dY/dX( 7, 6) =    5.6571822166
```

Table 6.4b: Total derivative matrix for the second system case

Converged system written to subroutines

Recall that one of the most important user options is whether or not to write each equation of the converged system to a separate subroutine. This could be beneficial to researchers who are concerned with system reduction strategies. Recall that once the user decides to create such a subroutine, the user must decide which language to write the subroutines in: FORTRAN 77 or ANSI C. For system case 1, the subroutines are written in FORTRAN (in file *eqns.for*), and for system case 2, the subroutines are written in C (in file *eqns.c*). Referring back to table 6.1, one can see that two values near the end of table are in italicized print. The "1" in the case 1 column denotes a choice of *FORTRAN*, while the "2" in the case 2 column denotes a choice of *ANSI C*. Table 6.5a is an excerpt of the subroutine file for the first system case, showing the first and last subroutine entries. Similarly, table 6.5b pertains to the second system case.

```
Subroutine eqn0001
Dimension bvmag(99,99),dvmag(99,6)
Common bvmag

dvmag( 1,1)= 5.156
dvmag( 1,2)= 1.734
dvmag( 1,3)= 9.702

bvmag( 1, 1)=                              +(       0.826497*dvmag(  1,  1)**.25)
    /+(              0.000094*bvmag( 2,  1)**2.0)+(   0.064643*bvmag( 2,  1)**1.0)
    /-(              0.537248*bvmag( 2,  1)**.33)+(   5.189545*dvmag( 1,  2)**.50)
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
    /+(   0.000000*        0+ 0        )
Return
End

Subroutine eqn0007
Dimension bvmag(99,99),dvmag(99,6)
Common bvmag

dvmag( 3,1)= 9.441
dvmag( 3,2)= 0.058
dvmag( 3,3)= 4.188
dvmag( 3,4)= 4.594

     bvmag(  3,  2)=                           +(     0.765490*dvmag(  3,  4)**2.0)
     /+(       4.665359*dvmag(  3,  4)**.50)+(     2.905074*bvmag(  2,  1)**.25)
     /+(       2.905074*bvmag(  2,  1)**.25)-(     0.000183*bvmag(  1,  1)**2.0)
     /+(       1.455039*bvmag(  1,  2)**.50)-(     0.042755*bvmag(  1,  1)**1.0)
     /+(     172.523101*dvmag(  3,  2)**1.0)+(     2.067727*bvmag(  1,  1)**.50)
     /-(       0.434359*dvmag(  3,  4)**.50)+(     0.473740*dvmag(  3,  4)**2.0)
     /+(      25.814882*dvmag(  3,  2)**.33)-(     0.047374*dvmag(  3,  4)**2.0)
     /+(       2.543852*bvmag(  2,  3)**.25)+(     1.059242*dvmag(  3,  1)**1.0)
     /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
     /+(   0.000000*        0+ 0        )+(   0.000000*         0+ 0           )
     /+(   0.000000*        0+ 0        )
Return
End
```

Table 6.5a: Equation-subroutines for the first system case

```
void eqn0001(void)
{

dvmag[ 1][1]= 5.300;
dvmag[ 1][2]= 3.377;
dvmag[ 1][3]= 3.647;

bvmag[ 1][ 1]=                          +(        4.581233*pow(dvmag[ 1][ 3],.33))
+(     678.811371*pow(bvmag[  3][  2],-1.))+(      0.000078*pow(bvmag[  3][  1],2.0))
+(          2.572560*pow(bvmag[ 3][ 1],.50))+(   0.000000*              0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(      0.000000*        0+  0            );
}


void eqn0007(void)
{

dvmag[ 3][1]= 1.425;

bvmag[ 3][ 2]=                          +(        1.373290*pow(dvmag[ 3][ 1],2.0))
-(     880.118236*pow(bvmag[  2][  2],-1.))+(  2147.589773*pow(bvmag[  2][  1],-1.))
+(      4.540861*pow(bvmag[  2][  3],.50))+(  8251.212239*pow(bvmag[  1][  2],-1.))
+(        8.121039*pow(bvmag[  2][  1],.50))+(      3.085842*pow(dvmag[  3][  1],.25))
+(   637.509406*pow(bvmag[  2][  1],-1.)) +(      6.200842*pow(bvmag[  2][  3],.33))
-(     0.000756*pow(bvmag[ 2][ 1],2.0))       +(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*           0+  0       )+(     0.000000*           0+  0        )
+(     0.000000*        0+  0            );
}
```

Table 6.5b: Equation-subroutines for the second system case

In comparing the two figures, first note that they are written in two different computer languages. Notice that the subroutine titles are numbered sequentially. The first equation of subsystem number 1 is written to subroutine *0001*, while the second equation of the third subsystem is written to subroutine *0007*. The FORTRAN subroutine (system case 1) requires the dimensioning of the *dvmag* and *bvmag* arrays, and a common declaration of the *bvmag* array. The C subroutine (system case 2) initiates by using the "{" symbol. The primary code logic of each subroutine, in both languages, is twofold: declaration of the current subsystem design variables, and the output equation assignment. The latter is based on the values of the coupling array parameters, for each and every term. These parameters have been outlined in the *params.dat* file, as previously discussed. Completion of the FORTRAN subroutine is achieved by using the *return* and *end* statements. Completion of the C subroutine is achieved by using the "}" symbol.

Closure

This chapter has presented two example system cases that have been created by CASCADE. The first system of equations was created and solved on one machine. The second system was created and solved in parallel, by using PVM. The intent of this chapter was to give the user a feel for the aesthetic of the input and output of a CASCADE generated system. Given this foundation, the next chapter will analyze the results of these example systems, and other CASCADE generated systems. In doing so, the dynamics of a randomly generated complex system can be more fully understood. For a full listing of the input/output files that have been outlined in this chapter, refer to Appendix III.

Chapter 7

Macroscopic analysis of numerous CASCADE outputs

Equation construction and convergence - CPU time

Of primary concern to a system analyst is the CPU time required to converge a complex system. Smaller CPU times result in smaller computational costs. It is therefore of great interest to reduce the time required to build and converge an analytical complex system, as much as possible. Many test systems have been generated by CASCADE to obtain a wide range of time-related results over a wide range of system sizes. Systems that were solved sequentially on one computer were solved both on a local SUN workstation (named *Hyperion*), and on a 12-processor mini-supercomputer (named *Moriarty*). For the present time, systems that were solved using distributed computing (PVM) were solved only on local SUN stations, similar in specification to Hyperion. Virtual machines having 5, 9, and 13 computers were used, so as to have 4, 8, and 12 slave machines, respectively. Figures 7.1 and 7.2 are plots of CPU time vs. system size. The former plots results for the single computer scenario, while the latter plots results for the PVM scenario.

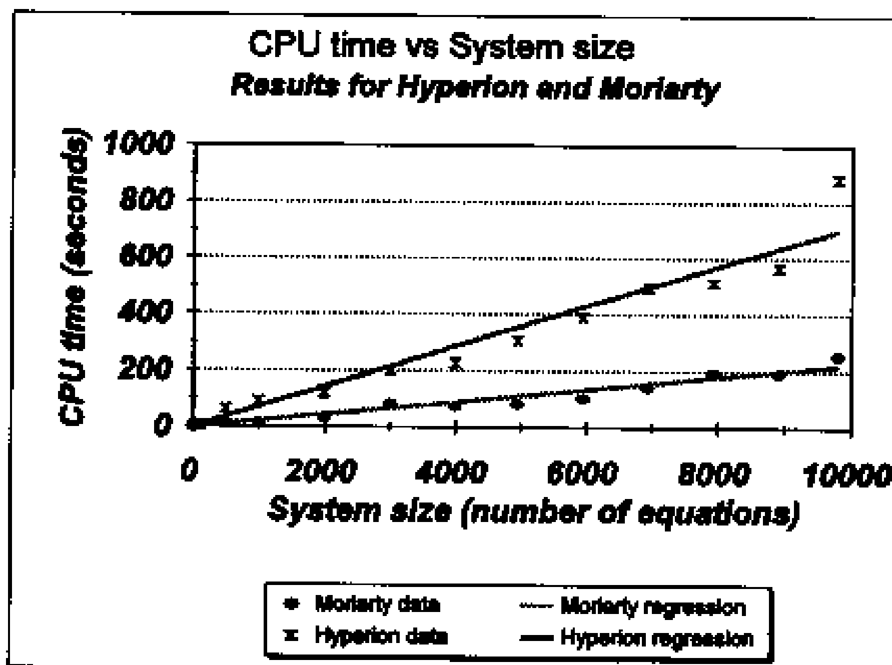

Figure 7.1: Single computer CPU time results

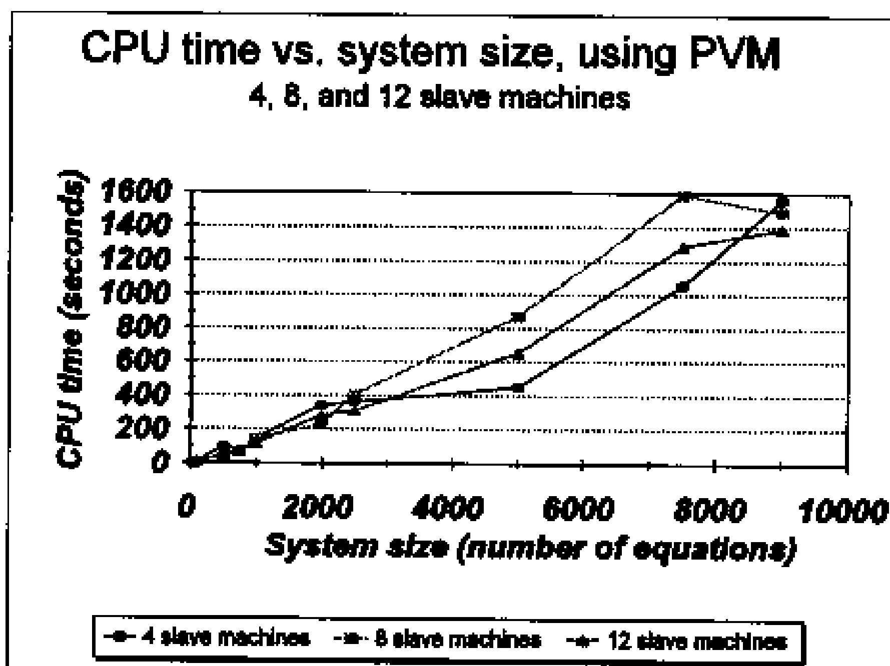

Figure 7.2: Virtual machine CPU time results, using PVM

Upon first observing figure 7.1, it is quite obvious that for all system sizes, the Moriarty supercomputer is far more efficient than a local workstation. CPU times are roughly 4 times shorter for systems solved on Moriarty than those solved on Hyperion. Moreover, there is an almost linear correlation between CPU time and system size, as the two regression curves very closely match the two sets of data points.

From several points of view, the distributed computing data does not live up to its expectations. As expected, there is a general upward trend in CPU time, with an increase in system size, for all 3 scenarios (4, 8, and 12 slave machines). However, the CPU times are *larger* than those corresponding to the single machine data of figure 7.1. At first glance, this appears to be counter-intuitive. One would think that a system that is solved my multiple machines would be constructed and converged more quickly than a system that is analyzed on one machine. This was not found to be the case.

Upon comparing the 3 curves themselves in figure 7.2, one sees that for the most part, the systems that were solved on only 4 slave machines converged most quickly, followed by systems that were solved on 12 machines, followed by systems that were solved on 8 machines, which required the *longest* time to converge. This again appears to be counter-intuitive. It would seem likely that a system solved using parallel computing techniques would solve faster, when using a greater amount of slave computers on the virtual machine. This was not found to be the case.

The above data only considers the amount of time to convergence, and does not take into consideration the number of iterations that these systems required, to converge. Before

any strong observations can be made, the iterative-related data should first be taken into account.

Iterations to convergence

The CPU-time results that were just presented are, in part, a function of the number of iterations that are required by the system, to gain convergence. It appears likely that the number of iterations to convergence is a function of both the size of the system, and more importantly, the nature of the complexities (couplings) of the system. Figures 7.3 and 7.4 plot the number of iterations to convergence vs. system size, for both single computer and PVM scenarios.

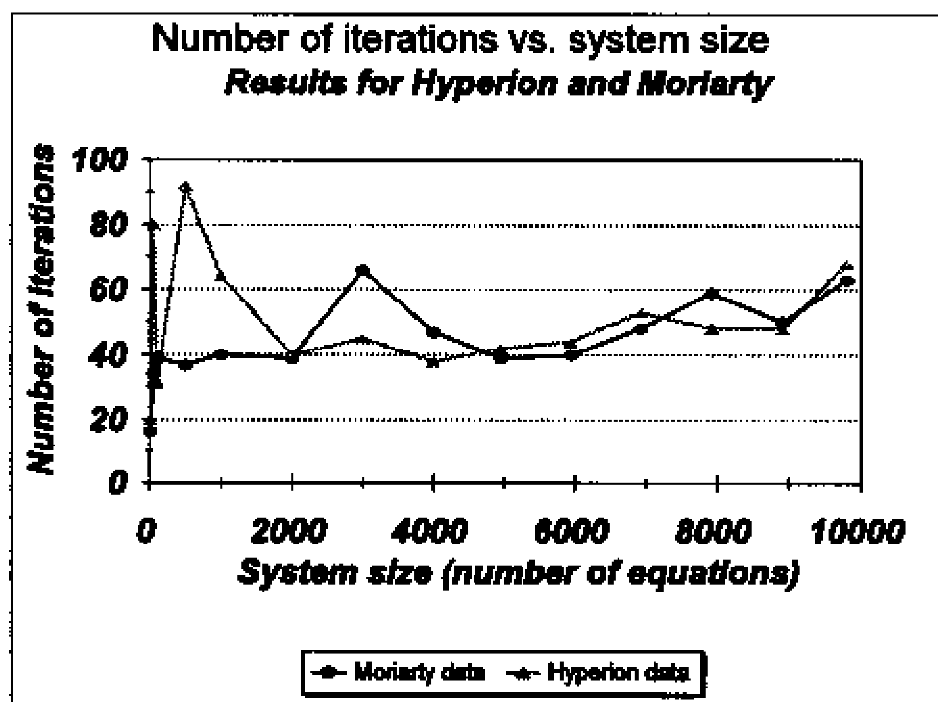

Figure 7.3: Single computer Iteration results

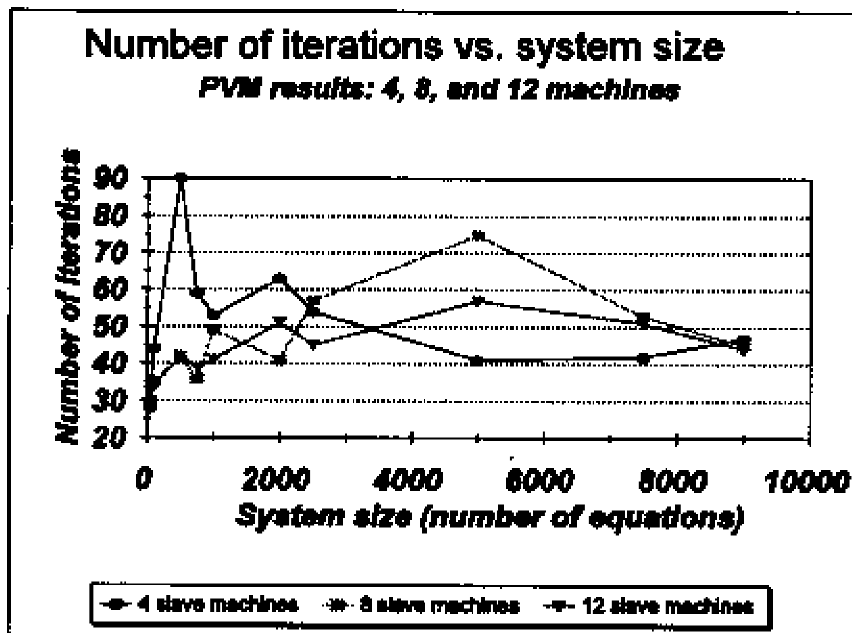

Figure 7.4: Virtual machine Iteration results, using PVM

Upon first observing figure 7.3, several observations can be made. As expected, there does appear to be an upward trend in the number of iterations to convergence, as the system size is increased. However, there are several *spikes* on the two graphs, where smaller systems required a comparatively large number of iterations to converge. For instance, the 500 subsystem system that was solved on Hyperion required over 90 iterations to converge, while the larger 2000 subsystem system required only 40 iterations to converge. A similar spike is seen on the Monarty plot, where nearly 70 iterations were required to converge a 3000 subsystem system, while only 40 iterations were required to converge a 6000 subsystem system. These discontinuities can be attributed to *excessive* coupling complexities. Often times, strong **reciprocal couplings** will be generated in a system. Such couplings are

characterized by a group of terms in several output equations that are all "strongly coupled" (an exponent of "2") amongst each other. For example, a term in output equation A is strongly coupled to terms in equations B and C, and a term in equation B is strongly coupled *back* to a term in equation A and to a term in equation C. Additionally, a term in equation C is strongly coupled *back* to terms in both equations B and A. This type of scenario might comprise a coupling loop that does not converge easily. Realize that a large system may have many such reciprocal coupling loops, and/or coupling loops of much greater size.

The distributed computing iteration results are seen in figure 7.4. These results appear to be less correlated than the single machine results of figure 7.3. First, there appears to be less of an upward trend in the iteration data as the system size increases, and more randomness to the number of iterations required to convergence. Realize that the construction and convergence of a complex system uses a slightly different methodology when distributed computing (PVM) is employed, than that of a single computer complex system. A single computer system has immediate access to the most recent changes made to subsystems that were analyzed *earlier*, on the same iteration. For example, a single computer is analyzing subsystem number 24, equation number 2. The second term of this equation is a subsystem coupling, to subsystem number 18. Subsystem number 18 was analyzed earlier than subsystem number 24 (on the present iteration), and thus the newly updated data for subsystem number 18 is already available to subsystem number 24. In the case of a PVM-analyzed system, it is often the case that 10 or more subsystems are being solved at the same time. In the example that is being presented, it may be the case that subsystems 15 through 25 are being analyzed at once. Hence, subsystem 24, for example, will be sent data for

subsystems 15-23 and for subsystem 25 from the **previous iteration**. Subsystem 24 will not have access to the most recent changes made to other subsystems that are currently being analyzed. This may, in the end, require a greater number of iterations to convergence, for the entire system. Alternatively, it may not matter much at all. This *data access differential* between the single computer and PVM results likely explains the scatter that is seen in the figure 7.4 data.

Recall from the previous subsection that CPU times for the PVM data were both a) greater than those for the single machine data, and b) un-correlated, with respect to the number of machines that were used to analyze the systems. Given the iteration data of this subsection, it may be beneficial to normalize the CPU times for the PVM data, with respect to the number of iterations to convergence. Table 7.1 presents the CPU time per iteration, for the 4, 8, and 12 slave machine scenarios. For the purposes of this analysis, data was also obtained using a 20 slave machine virtual machine, for system sizes of 1000 and 2500 equations.

System size	4 slave machines	8 slave machines	12 slave machines	20 slave machines
100	0.246	0.264	0.275	-
500	1.124	1.139	1.188	-
1000	2.730	2.810	2.814	2.944
2500	6.717	6.830	6.874	7.047
5000	11.030	11.224	11.535	-

Table 7.1: CPU time (seconds) required *per iteration*, for the PVM systems

This table indeed clarifies certain issues. Now, there does seem to be a correlation between the CPU time per iteration, and the number of slaves on the virtual machine. The correlation, however, is the opposite as that expected. With more machines, systems are taking **longer** to solve. This table indicates that at present, PVM is not providing a savings in solution time, and the reason is that **message passing is dominating the time required for convergence**. To have each subsystem solved on a separate machine, recall that a great deal of necessary information must first be sent to that machine, in order for the analysis to take place. This information includes a good deal of large, multi-dimensional arrays. In order to successfully pass these arrays (using the FORTRAN programming language), it was necessary to both a) statically dimension the arrays at a very large number, and b) send the elements of the array one-by-one. Had a better means been found to transfer the array information from the master to the slaves, it is likely that the full benefit of having a virtual machine with 10 or more slave machines would have been encountered.

Effects of Normalization on GSE solution

The sensitivity analysis of the converged system can be very useful to system-reduction researchers. The matrix of total derivatives provides an indication as to how sensitive each system output is to each system input. As discussed in the last chapter, normalization techniques are often used, so as to improve the numerical condition of the left and right hand side matrices that are used in the LU-decomposition, to attain the total derivative matrix. The benefits of this procedure will now be realized.

Figure 7.5 is a plot of the CPU time required to compute the total sensitivity matrix vs. the size of the system.

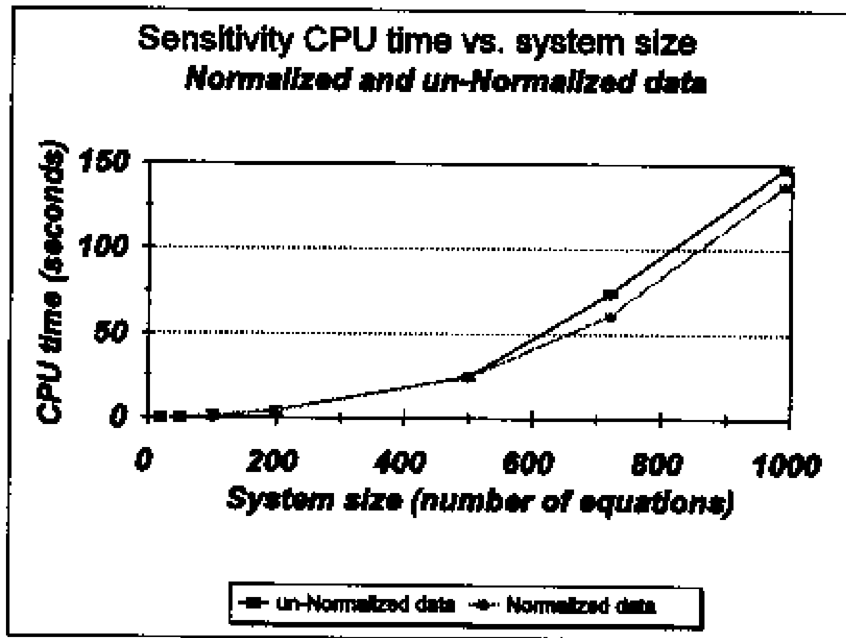

Figure 7.5: Sensitivity CPU times vs. system size

For smaller systems, the CPU times of the normalized and un-normalized systems are comparable. As the system size increases, the CPU time required to compute the sensitivity matrix is slightly shorter for the normalized matrices than for the un-normalized matrices. This reduction in CPU time is likely attributable to the numerical conditioning provided by the normalization procedure. Table 7.2 is a lists condition numbers vs. system size, for both normalized and un-normalized scenarios.

number of system equations	***condition number - normalized sensitivities***	***condition number - Un-normalized sensitivities***
10	6.418	3618.588
20	10.211	1942.187
50	25.586	3112120.500
100	855.608	885146.875
200	94.514	no solution
500	282.544	7622248.000

Table 7.2: Condition number comparison of normalized / un-normalized data

The differences in magnitude are astounding. The normalized data has matrix condition numbers that are far lower than those for the un-normalized data. Using un-normalized data, a total derivative solution could not be found for the 200 equation system example, depicted in the above table. Note that the general trend is an increase in condition number occurs with an increase in system size. As was the case with number of iterations to convergence, excessive coupling complexities could result in high condition numbers associated with small systems. This is the case with the 100 equation, normalized system in the above table.

Closure

Given the presentation of two example system cases in chapter 6, this chapter has attempted to quantify the system outputs of numerous executions of CASCADE on a more *global* scale. Foremost, this chapter has analyzed the effect of system size on both the CPU time required to build and converge a system of equations, and the number of iterations to convergence, for both sequentially created systems, and parallel-created systems using PVM. In so doing, the constructions and solutions of a sequentially created system on both a single workstation, and on a supercomputer were contrasted. Also, the comparison of system results that were attained by using distributed computing, with varying numbers of slave machines (4, 8, and 12), has taken place. Moreover, the effects of matrix normalization on the computation of the total derivatives of a system of equations were analyzed, from both a CPU-time and a numerical-conditioning standpoint. The final chapter will attempt to summarize the motivation, background, and results for this thesis, and will attempt to address recommendations for future work.

Chapter 8

Conclusions

Summary

The emerging field known as Multidisciplinary Design Optimization inherently has great potential for becoming an industrial standard for the design of large, complex systems. The basic methodology is simple: divide a large task or system into numerous smaller, but still inter-related subtasks. These subtasks can be divided among the participating design groups and solved simultaneously, in a non-hierarchic manner. The analysis of a complex system must take place on a powerful computer. Hence, it is clear that the MDO methodology lends itself well to distributed computing environments, in which one large computational task is divided among numerous computers, connected on a network.

The motivation for MDO is to reduce the time and cost required for the design process. Most complex systems are non-hierarchic in nature, and require an iterative scheme (and initial approximations) to converge. Task sequencing researchers attempt to find the optimal sequence (order) in which to analyze the system modules, so as to gain the converged solution in the least amount of time. System reduction researchers seek to effectively reduce the size of a complex system, with a minimal loss in accuracy. Researchers have attempted to temporarily suspend and/or permanently eliminate output couplings that were comparatively found to be less substantial.

Before these MDO-strategies can be implemented in the design process of such large systems as automobiles and aircraft, they must be tested. A method for analytically simulating real-life, large system couplings is necessary. The simulation should be able to predict the output sensitivities of the system; the change in the system outputs with respect to a prescribed change in the design variables of the system. The simulation should also lend itself well to a distributed computing environment. The numerous design tasks of a large system design should be computationally distributed among the participating design groups. The simulation should be *robust*; it should accommodate a wide range of system sizes and complexities. Finally, the simulation should be *random*; it should create design scenarios that may have been initially unforeseen by the system analysts. To this end, the author has designed a computer program, coded in FORTRAN, and called CASCADE (Complex Application Simulator for the Creation of Analytical Design Equations).

CASCADE accepts user inputs to randomly construct and then converge a large system of complex equations. This system of equations should be viewed as an analytical representation to a real-life design scenario. After the user tells CASCADE the desired size of the system, the *nature* (the initial coefficient, the sign, the coupling, and the exponent) of each term, of each equation, of each subsystem of the system is determined randomly, using a random number generator. The non-hierarchic system is constructed on the first iteration, and converged on iterations thereafter, after having initialized each and every output equation to a value of unity. This procedure was first implemented on a single computer format, cycling though all of the subsystems in the system, one-by-one, in a sequential manner. The procedure was later modified for solution in a distributed computing environment, using

Parallel Virtual Machine (PVM). A virtual machine is constructed, consisting of numerous local workstations. Each subsystem in the constructed system is then sent to a separate computer for analysis, such that the number of subsystems being analyzed at one time is equal to the number of computers on the virtual machine. Inherently, this appears to be more efficient, and more realistic.

Once the system is constructed and converged, either in a single computer or distributed computing manner, CASCADE offers numerous post-convergence features. The Global Sensitivity Equation method can be used to compute the total sensitivities of the system outputs, with respect to the inputs. This is done by first computing sensitivities on a local level. The matrices from which the total derivatives are computed can either be normalized or not. Normalized matrices offer a higher likelihood of an accurate solution. A second important feature is the option to write each equation of the converged system to a separate subroutine. This could be beneficial to sequencing researchers. Again, these researchers might perturb the design variables of the converged system, and then analyze the various ordering possibilities of the subroutines to see which sequence would attain a new converged solution most quickly. A final important feature is the option to write the converged system to a *parameters* file. This file provides a comprehensive listing of the nature of each term in the system that has been constructed, as well as other system statistics.

The two chapters preceding this chapter have attempted to present and discuss the output system results of executing CASCADE. The former chapter presented two example cases of small systems that were created by CASCADE. The first of the two systems was created on one computer, by sequentially cycling through the subsystems of the system. The

second of the two systems was created in a distributed computing environment, by using PVM. Excerpts of the input file, and the output files (*parameters*, output magnitudes, total sensitivities, and equations written to subroutines) were presented. The latter chapter has analyzed the results of numerous executions of CASCADE on a more global level. The single computer results saw an increase in CPU time, and an upward trend in number of iterations to convergence with an increase in system size. As expected, systems that were solved using the Moriarty supercomputer solved much faster than those solved on the local Hyperion workstation. Unfortunately, the PVM results did not live up to expectations. The CPU times for the parallel machine-generated systems were larger than those for the single computer systems. Moreover, CPU times per iteration were larger, when a larger number of slave computers were used on the virtual machine. It is likely that the time required to pass information from the master machine to the slave machines is what dominated the convergence time for the PVM scenarios. This information passing included the static dimensioning of large, multi-dimensional arrays, that could only be sent from machine to machine *element by element*.

As for the computation of the total derivative matrix by using the GSE method, matrix normalization was found to be beneficial. The condition number of the solution matrix was *lower* when using normalization techniques. This is an indication of a more reliable numerical estimate. The CPU times required to attain the normalized derivative matrix were also lower, than those required to attain un-normalized derivative matrices.

Future work

The first step that should be taken for the advancement of this research would be to improve the results associated with distributed computing and PVM. A method must be found to more effectively pass the large array data from the master to the slave. It is obvious that the use of numerous computers to solve a small portion of a large problem is more efficient than the use of only one computer to solve the entire large problem. This proves that message passing was the dominating factor in the time taken to build and converge the systems, while using parallel processing techniques. An interface is required, that will enable the PVM message passing to avoid the performance degradation due to communication through the operating system and a transport layer protocol. Refer to figure 8.1.

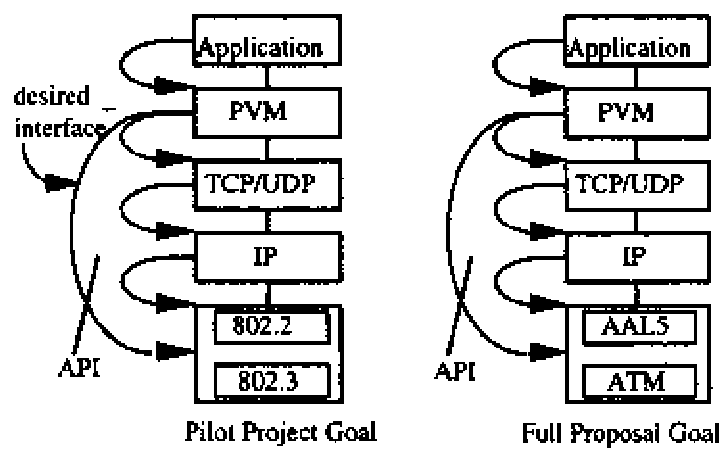

Figure 8.1: Direct message passing to the Network layer

This *desired interface* allows PVM to execute the application (CASCADE), and bypass protocol overhead of UDP and IP, and communicate directly to the low-level network interface. For this preliminary work, the low-level network has been Ethernet-based; eventually, a conversion to an ATM-based network is foreseen.

Another time-improvement could be made by setting up a virtual machine whose hosts consist of the numerous processors on a multi-processor supercomputer. This was the original intent of this research. A method was not found to achieve this scenario, and hence the multi-workstation virtual machine was implemented instead.

The size of systems created by CASCADE has been limited by the static dimensioning allowances of the FORTRAN programming language. If this were overcome, systems of infinite size could potentially be constructed and converged. This would prove that the principals of MDO could be extended to systems of any imaginable size, with couplings having any imaginable complexity.

A final idea is to physically combine the CASCADE package with the other MDO-related program methodologies that have been discussed, such as task sequencing software and coupling suspension software. An MDO-framework is envisioned, in which a complex system could be randomly built and converged, then optimally altered and re-converged.

References

[1] Bloebaum, Christina L., "Global Sensitivity Analysis in Control-Augmented Structural Synthesis," AIAA Student Journal, Summer 1989.

[2] Bloebaum, Christina L., "Formal and Heuristic System Decomposition Methods in Multidisciplinary Synthesis," Ph.D. Thesis, University of Florida, Gainsville, FL, 1991.

[3] Bloebaum, C. L., "An Intelligent Decomposition Approach for Coupled Engineering Systems". Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, Cleveland, OH, September, 1992.

[4] Cheney, Ward, and Kincaid, David, *Numerical mathematics and computing*, Brooks/Cole Publishing company, Pacific Grove, California, 1985.

[5] Codenotti, Bruno and Leoncini, Mauro, *Introduction to Parallel Processing*, Addison-Wesley, Wokingham, England, 1992.

[6] Eason, Ernest D. and Wright, Joyce E., "Implementation of Non-Hierarchic Decomposition for Multidisciplinary System Optimization." Fourth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, Cleveland, OH, September, 1992.

[7] Eschenauer, Hans A. and Weinert, Matthias, "Approximation Concepts for the Decomposition-Based Optimization of Complex Mechanical Structures on Parallel Computers," Advances in Design Automation Vol. II, ASME Pub. DE-Vol. 65-2, 337-345, 1993.

[8] Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T., *Numerical Recipes - the art of scientific computing*, Cambridge University Press, Cambridge, 1986.

[9] Geist, A., Beguelin, A., Dongerra, J., Weicheng, J., Mancheck, R., and Sunderam, V., *PVM: Parallel Virtual Machine - A user's guide and tutorial for networked parallel computing*, The MIT Press, Cambridge, Massachussets, 1994.

[10] Hajela, P., Bloebaum, Christina L., and Sobieszczanski-Sobieski, Jaroslaw, "Application of Global Sensitivity Equations in Multidisciplinary Aircraft Synthesis," Journal of Aircraft, Volume 27, No. 12, 1990, pp. 1002 - 1010.

[11] Huddleston, John V., *Introduction to Computers, FORTRAN version*, Exchange Publishing Division, Buffalo, NY, 1988.

[12] Lakshminarayan, Krishnan, "ATM Networking and Multimedia - A White Paper," Sun Microsystems Computer Corporation, Revision X, August, 1993.

[13] Maliniak, L., "Teamwork is the Key to Concurrent Design," Electronic Design, January 1991, pp. 41-54.

[14] McCulley, C. and C. L. Bloebaum, "Optimal Sequencing for Complex Engineering Systems Using Genetic Algorithms." Fifth AIAA/USAF/NASA/OAI Symposium on Multidisciplinary Analysis and Optimization, Panama City, FL, September, 1994.

[15] McCulley, Collin M., "A Genetic Tool for Optimally Sequencing the Design of Complex Engineering Systems," Masters Thesis, University Of Buffalo, Buffalo, NY, 1995.

[16] Miller, Erik, "Coupling Suspension and Elimination in Multidisciplinary Design Optimization," Masters Thesis, University of Buffalo, Buffalo, NY, 1995.

[17] Rogers, J.L., "DeMAID -- A Design Manager's Aide for Intelligent Decomposition: User's Guide," NASA Technical Memorandum 101575, March, 1989.

[18] Sobieszczanski-Sobieski, Jaroslaw, "A Linear Decomposition Method for Optimization Problems - Blueprint for Development," NASA Technical Memorandum 83248, 1982.

[19] Sobieszczanski-Sobieski, Jaroslaw, "Multidisciplinary Systems Optimization by Linear Decomposition," Symposium on Recent Experiences in Multidisciplinary Analysis and Optimization, Hampton, VA, 1984.

[20] Sobieszczanski-Sobieski, Jaroslaw, "The Sensitivity of Complex, Internally Coupled Systems," AIAA Journal, Volume 28, No. 2, pp. 153-160.

[21] Sobieszczanski-Sobieski, Jaroslaw, "Optimization by Decomposition: A Step from Hierarchic to Non-Hierarchic Systems," Second NASA/Air Force Symposium on Recent Advances in Multidisciplinary Analysis and Optimization, Hampton, VA, September, 1988.

[22] Sobieszczanski-Sobieski, Jaroslaw, Bloebaum, Christina L., and Hajela, P., "Sensitivity of Control-Augmented Structure obtained by a System Decomposition Method," AIAA Journal, Vol. 29, No. 2, February, 1991.

[23] Steward, D.V., *System Analysis and Management.* Petrocelli Books, New York, 1981.

[24] Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design: with Applications*, McGraw Hill, New York, NY, 1984.