FTL REPORT R86-1

# ON THE NUMERICAL SOLUTION OF SIMULTANEOUS, NON-LINEAR EQUATIONS IN COMPUTER-AIDED PRELIMINARY DESIGN

Mark A. Kolb

FTL COPY, DON'T REMOVE
33-412, MIT .... 02139
Single sided for copies

JANUARY 1986

MIT

DEPARTMENT
OF
AERONAUTICS
&
ASTRONAUTICS

FLIGHT TRANSPORTATION
LABORATORY
Cambridge, Mass. 02139

FLIGHT TRANSPORTATION LABORATORY REPORT R86-1

# ON THE NUMERICAL SOLUTION OF SIMULTANEOUS, NON-LINEAR EQUATIONS IN COMPUTER-AIDED PRELIMINARY DESIGN

by

MARK A. KOLB
S. B., Massachusetts Institute of Technology
(1984)

33-412, MIT .... 02139
single sided for copying

© Massachusetts Institute of Technology, 1986

This report reproduces a thesis submitted on January 17, 1986, to the Department of Aeronautics and Astronautics of the Massachusetts Institute of technology in partial fulfillment of the requirements for the degree of Master of Science.

- blank page -

ON THE NUMERICAL SOLUTION OF
SIMULTANEOUS, NON-LINEAR EQUATIONS
IN COMPUTER-AIDED PRELIMINARY DESIGN

by

MARK A. KOLB

# Abstract

In the solution of problems in preliminary design, it often becomes necessary to solve systems of equations. An immediate example from aeronautical design is the frequent need to perform several iterations around the gross takeoff weight in order to determine a consistent value for this design variable.

The *Paper Airplane* program is the result of research by the MIT Flight Transportation Laboratory into automating the computational aspects of preliminary design, with the object of freeing the designer to concentrate on the more creative aspects of the design process. *Paper Airplane* is written in Common LISP (augmented by the object-oriented programming "flavor" construct, supported by NIL and ZetaLisp), and implemented on both a VAX–11/750 and a Texas Instruments EXPLORER Lisp Machine. In its original form, *Paper Airplane* possessed the capability of transforming declarative knowledge about design relationships into imperative form; i.e., the declarative statement, $w = f_1(x, y, z)$, could be understood to imply that $x = f_2(w, y, z)$, $y = f_3(w, x, z)$, and $z = f_4(w, x, y)$, as well. Understanding of this concept was simulated by applying the Newton-Raphson method to *numerically* invert design relationships. This capability enabled *Paper Airplane* to solve design problems which could be reduced to sets of single, independent equations. However, no techniques were available for cases in which iteration was required to solve the design relationships—i.e., when the system could only be reduced to two or more simultaneous equations.

Towards the end of providing this additional capability, three numerical techniques for solving sets of two equations in two unknowns were examined. In the course of investigating these approaches, the need for explicit representation of the steps by which design relationships were to be used to calculate values for design variables was recognized, and appropriate software representations were developed.

The first of these techniques, the so-called "fixed-point" method, involves assuming a value for one variable, based upon which the equations are solved, enabling calculation of a new value for the variable whose value was assumed. This calculated value is used as

the assumed value for the next iteration, repeating this process until, if possible, the value calculated at the end of an iteration is sufficiently close to the value assumed at the beginning of the iteration.

In the "simultaneous Newton-Raphson" method, a value is again assumed for one variable, and design relationships are chained together in such a way as to make it possible to calculate two values for a second variable. The assumed value is updated by calculating the intersection of tangent lines determined by the derivatives of the two chains of design relationships, until the values calculated by both chains are identical.

Upon establishing that the two former methods rely upon invalid assumptions concerning the linearity of design relationships, a third method, the "logarithmic distribution" method was examined. In setting up the problem, an approach parallel to that used in the "simultaneous Newton-Raphson" method is taken, but the solution to the system of equations is determined by testing logarithmically distributed values over progressively smaller search intervals. Due to difficulties associated with design relationships which are multi-valued in certain directions, this technique required modification to the procedure whereby design relationships are numerically inverted, using the Newton-Raphson method. It was found that by making the selection of the initial guess value for the Newton-Raphson procedure stochastic, multi-valued relationships could be prompted to use the appropriate branches, when necessary. To more effectively deal with multi-valued branches, application of symbolic algebra techniques and further modifications to the Newton-Raphson procedure are proposed.

# Acknowledgements

I would like to acknowledge the contributions to this work of the following individuals, and express to them my sincere gratitude and appreciation: Prof. Antonio Elias, my thesis advisor, for introducing this engineer to the joys of LISP and the challenges of Computer-Aided Preliminary Design, for prompting, suggesting, and correcting; Dr. John Pararas, a programmer's programmer, for knowing how to do things, and being willing to explain; the MIT Department of Aeronautics and Astronautics, for granting the fellowship without which this work would not have begun; Mark Sapossnek, a friendly competitor, for sharing his approach to the problem, and for his interest in *our* approach; Solly Ezekiel and Ron Lajoie, men I may call my colleagues, for advice and assistance from the student's perspective; the staff and congregation of Park Street Church for their guidance and inspiration, and particularly to members Peter and Pam Brown, Joe and Paula Grabowski, Wayne and Debbie Koch, and Jim and Kathy Verhulst, for their encouragement and prayers; David and Sanna Kolb, my parents, for their patience, understanding, confidence, and concern, and for their never-failing readiness to help; and, most especially, Jean Marie Kolb, my wife, for her care, her support, and her love—she is truly God's most "gracious gift" to me.

# Contents


**Abstract** ii

**Acknowledgements** iv

**1 Introduction** 1

1.1 Computer-Aided Preliminary Design 1

1.2 The Problem of Generalization 2

1.3 Paper Airplane 4

1.3.1 Introduction to Paper Airplane 4

1.3.2 Function Application, Inversion, and the Newton-Raphson Method 4

1.3.3 Research Problem and Solution Constraints 8

1.4 Related Work 9

1.5 A Note on Notation 11

**2 The Fixed-Point Approach and Explicit Representation of the Computational Agenda** 13

2.1 The Fixed-Point Method 13

2.2 Implementation of the Fixed-Point Method 16

2.3 Computational Agenda Construction 18

2.3.1 Forced Path Construction 18

2.3.2 Loop Construction 22

2.3.3 Overconstraining Functions 27

2.4 Representation of the Agenda Steps 28

2.4.1 Agenda Entries 30

2.4.2 Loop Headers 31

2.5 Execution of the Computational Agenda using the Fixed-Point Method 32

2.6 Recovery from Instabilities 33

2.6.1 Satisfying the Convergence Criterion 33

2.6.2 Implementation Problems 35

2.6.3 Interactions between Multiple Loops 37

**3 The Simultaneous Newton-Raphson Approach** 41

3.1 The Simultaneous Newton-Raphson Method 41

3.2 Implementation of the Simultaneous Newton-Raphson Method 44

3.2.1 Changes to the Computational Agenda Loop Header Structure . . . . 44
3.2.2 Changes to Computational Agenda Loop Construction . . . . . . . . . 46
3.2.3 Execution of the (Revised) Computational Agenda using the Simultaneous Newton-Raphson Method . . . . . . . . . . . . . . . . . . . . . . 51
3.3 Problems Associated with the Simultaneous Newton-Raphson Method . . . . 53

**4 The Logarithmic Distribution Approach** **56**
4.1 The Logarithmic Distribution Method . . . . . . . . . . . . . . . . . . . . . . 56
4.1.1 Application of the Logarithmic Distribution Method . . . . . . . . . . 56
4.1.2 Logarithmic Distribution over an Interval . . . . . . . . . . . . . . . . 59
4.1.3 Implementation of the Logarithmic Distribution Method . . . . . . . . 61
4.2 Performance of the Logarithmic Distribution Method . . . . . . . . . . . . . . 61
4.2.1 Extreme Values for the Forcing Variable . . . . . . . . . . . . . . . . . 61
4.2.2 Multi-Valued Design Functions . . . . . . . . . . . . . . . . . . . . . . 62
4.2.3 Adapting for Multi-Valued Design Functions . . . . . . . . . . . . . . 65

**5 Concluding Remarks** **69**
5.1 Research Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.2 Suggestions for Further Investigation . . . . . . . . . . . . . . . . . . . . . . 71
5.2.1 A Hybrid Analytical/Numerical Approach . . . . . . . . . . . . . . . . 71
5.2.2 Solutions to Larger Systems of Simultaneous Equations . . . . . . . . 73
5.2.3 Recognizing Multi-Valued Functions Numerically . . . . . . . . . . . . 74

**A Test Design Set** **76**
A.1 Design Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.2 Design Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.3 Base Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
A.4 Computational Agenda . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
A.5 Derived Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

**B Glossary of New Terms** **81**

# List of Figures


1.1 The Newton-Raphson technique for locating zeros. . . . . . . . . . . . . . . . . 6

2.1 The fixed-point iterative technique for locating intersections. . . . . . . . . . 15
2.2 The computational agenda structure for the example design set. . . . . . . . 29
2.3 A computational agenda structure with nested and tangled loops. . . . . . . . 39

3.1 The simultaneous Newton-Raphson method for locating intersections. . . . . 42
3.2 The computational agenda structure for the example design set, modified for application of the simultaneous Newton-Raphson method. . . . . . . . . . . . 52

4.1 The logarithmic distribution method for locating intersections. . . . . . . . . 57
4.2 Deterministic propagation of forcing values for $W_g$, yielding a pair of computational loop *branch* values for $C_{L\text{cr}}$. . . . . . . . . . . . . . . . . . . . . . . . 64

# List of Tables

A.1 Design variables for the test design set. . . . . . . . . . . . . . . . . . . . . . . . . . 77
A.2 Design functions for the test design set. . . . . . . . . . . . . . . . . . . . . . . . . 78
A.3 Base variables for the test design set. . . . . . . . . . . . . . . . . . . . . . . . . . . 79
A.4 Computational agenda resulting from the base variables. . . . . . . . . . . . . . 79
A.5 Derived variables for the test design set. . . . . . . . . . . . . . . . . . . . . . . . . 80

# Chapter 1

# Introduction

## 1.1 Computer-Aided Preliminary Design

An engineering design problem—such as the design of a physical system or device—is typified, at least in the stage of *preliminary* design, by a set of design parameters, and a set of relationships governing these parameters. Frequently, these relationships may be modeled as mathematical functions of the parameters as variables, in which case the parameters may be referred to as **design variables**, and the relationships may be referred to as **design functions**. For example, when designing a house, selecting length, width, and number of floors determines the total floorspace, by the mathematical relationship

$$total\ floorspace = length \times width \times number\ of\ floors$$

For the preliminary design of a large system, such as a new aircraft or space vehicle, one can envision the need for literally hundreds of relevant design variables for describing the dimensions, weights, performance characteristics, and design specifications of the planned device, and an equally large number of design functions for describing the relationships between these design variables. The design process may then be characterized as

1. Choosing a relevant set of design variables and design functions for describing the design problem.
2. Determining a set of design variables for which values may be specified, i.e., choosing a set of **base variables**.
3. Choosing values for these base variables.
4. Applying the design functions to determine values for the remaining design variables, i.e., the **derived variables**.

5. Examining these calculated values to evaluate the overall "goodness" of the resulting design.

Depending on the outcome of this final step, the process may need to be repeated, accompanied by either selecting new values for the base variables and repeating steps 3–5, or, indeed, selecting a new set of base variables and starting over from step 2.[1] In either of these two cases, step 4, in which the design functions are applied to produce new values for the derived variables, must be repeated. Indeed, for most design problems this calculation procedure will be repeated many times, as the designer attempts to produce as "good" a design as possible. If the number of design functions is large, which, as suggested above, it very well may be, then each iteration may require a substantial amount of computation. When many iterations are desired, the computational task can quickly become unwieldy, if done by hand. For this reason, it becomes desirable to have a computer perform this task for the designer, relieving him from the tedious job of arithmetic, and allowing him to focus his attention on the more creative aspects involved in the design process (e.g., deciding which design variables and design functions are appropriate for the problem, choosing the base variables and assigning values for them, and evaluating the computed results). Thus, the notion of **Computer-Aided Preliminary Design** (or **CPD**, if you like acronyms) is introduced: Computer-Aided Preliminary Design is the use of computers to perform the mundane computational requirements of a preliminary design problem.

## 1.2 The Problem of Generalization

Of course, the notion of using computers to repeatedly perform a set of computations is not new; that is one of the things computers happen to do very well. Making use of this talent in the domain of preliminary design, even (and, one might say, *particularly*) in the field of aeronautics and astronautics, is not a new idea [4,11,13]. However, such applications have characteristically taken the form of a program written for the design of a very specific type of vehicle, for which the **design set** (i.e., the design variables and design functions), *and* a specific set of base variables, have been chosen in advance, these choices being inextricably embedded in the program's code.[2] Such programs typically allow the user/designer to vary

[1] In extreme cases, a designer may even wish to return to step 1, and choose completely new design variables and design functions.

[2] For example, in the GATEP (General Aviation, Twin-Engined, Propeller-driven aircraft) program described in Reference [11], parametric studies may be conducted around a base-line design of a twin-engined, propeller-driven, general aviation aircraft, based on user-inputs specifying the aircraft geometry, payload, required range or fuel available, and power plant operating characteristics. Given values for the required base variables, the GATEP program will calculate aircraft component weights, lift and drag characteristics, and takeoff,

the values of at least some of the base variables; in terms of the outline of the design process presented on page 1, this means that these kinds of programs explicitly perform steps 1, 2, and 4, leaving steps 3 and 5 to the designer. If the designer wishes to alter the design set, or to change which variables he wishes to use as base variables, either the program must be re-coded, or an entirely new program must be developed. Use of a computer program is introduced to help manage the computational complexities resulting from the use of a large set of design parameters, but in doing so, programs such as these limit the methods by which the designer may attack the problem.

With the advent of symbolic processing, however, it has become realistic to develop a more general-purpose computer-aided preliminary design program, which allows the designer complete control of the design set with which he is working: freedom to add, remove, or alter design variables and design functions, and change which design variables should be used as base variables (and, therefore, which design variables should have their values calculated). Such freedom permits the designer to try many different approaches to solving the design problem, since the program no longer forces him to use one particular approach (i.e., a "frozen" set of design functions, design variables, and base variables). With such a program, the computer is restored to its "proper role": it is responsible for performing calculations, not for making design decisions (or, at least, decisions about how the designing should be accomplished).

It is easy to see the desirability of such a system from the designer/user's point of view. And it is perhaps just as easy to imagine some of the difficulties a programmer might encounter in trying to produce such a system. Of obvious interest are two questions:

- How may the design functions and design variables be represented in such a program in order to permit the designer freedom in choosing which design variables are to be used as base variables?

- Given a design set for which a set of design variables have been designated as base variables (referred to as a **design path**), how can the program use the available design functions to calculate values for the derived variables (i.e., generate a **computational path**)?

It has been the goal of this research to provide an answer to this latter question, based on the results of previous research concerning the former. This previous research was performed by Prof. Antonio L. Elias, thesis supervisor for the current work, and resulted in the original

landing, climb, and range performance.

version of the *Paper Airplane* computer program. The results of this earlier work will be discussed in the next section.

## 1.3 Paper Airplane

### 1.3.1 Introduction to Paper Airplane

*Paper Airplane* is a LISP program for the manipulation of arbitrary, user-defined design functions and design variables, the original version of which was written in 1981 by Prof. Antonio L. Elias, of the Massachusetts Institute of Technology. This version included representations for both arbitrary design functions and arbitrary design variables, and was also able to compute values for those derived variables which were related to user-specified base variables by so-called "perfectly constrained" design functions. In this context, a **perfectly constrained** design function refers to a design function for which values are known for all associated design variables; thus a value may readily be calculated for the remaining (i.e., unknown) design variable.[3]

This was achieved by representing design functions as LISP functions and by designating a **computational state** for each design variable. Base variables were assigned the state[4] "I" ("initialized"), all other design variables initially receiving the state "F" ("free"). Any design functions involving only one such "free" design variable (that is, any perfectly constrained design functions) were then applied to calculate a value for the free design variable, updating the state of this design variable from "F" to "C" ("computed"). Computed values were used to further constrain any unused design functions, until the supply of perfectly constrained design functions was exhausted.

### 1.3.2 Function Application, Inversion, and the Newton-Raphson Method

In this section, a description of the means by which design functions are used to calculate values for free design variables in the *Paper Airplane* program is presented. First of all, design functions are defined in the form, $w = f(x, y, z)$, where $w$, $x$, $y$, and $z$ are design variables related by the design function, $f$. Note that, as defined, $f$ is a function which computes a value for $w$, given the values of $x$, $y$, and $z$. Design variable $w$ may be considered to be the **computed variable** of design function $f$, while $x$, $y$, and $z$ are the **input variables** of design function $f$. If values are known for all the input variables of a design function, then

[3]This follows because, by definition, a perfectly constrained design function represents a system of one equation in one unknown.

[4]Except where indicated otherwise, reference to a design variable's **state** is meant to indicate the design variable's computational state, and not, for example, its numerical value.

the value of the design function's computed variable is readily calculated by applying the design function as it has been defined—i.e., by computing $f(x,y,z)$ and assigning the result as the value of $w$. In this case, the value of $w$ may be calculated numerically, by simply having the computer evaluate the function $f$ at the current values of $x$, $y$, and $z$. As long as $f$ is a single-valued function of $x$, $y$, and $z$, $f(x,y,z)$ is readily calculated: no analytical understanding of the operations involved in the function $f$ (i.e., multiplications, divisions, additions, exponentiations, integrations, etc.), or of the relationships between these operations (e.g., that division is the inverse of multiplication, that multiplication is associative, etc.), is necessary.

What if the value of a design function's computed variable is already known? It may have been initialized,[5] or it may have been calculated using another design function. How may the information represented by such a design function be used? The **explicit direction** of the design function $w = f(x,y,z)$ is to compute a value for $w$, given values for $x$, $y$, and $z$. However, it may be desirable to use the design function in a different **computational direction**—i.e., to apply the numerical relationship represented by the design function to calculate a value for one of the design function's input variables. For the example cited, using a computational direction different from the design function's explicit direction would entail using the relationship $w = f(x,y,z)$ to compute a value for either $x$, $y$, or $z$; the computational direction would be the same as the explicit direction if the relationship were used to compute a value for $w$.

If values are known for the computed variable and all but one of the input variables, then the design function must somehow be "inverted," in order to use the design function to calculate a value for the remaining, unknown input variable. Such a necessity may suggest the need for the capability to rearrange the sequence of operations within the design function in order to "analytically invert" the design function, thus requiring an understanding of the relationships between the various operations which may be part of the design function; for example,

$$a = b \times c \Longrightarrow \begin{cases} b &= \frac{a}{c} \\ c &= \frac{a}{b} \end{cases}$$

and

$$a = b + c \Longrightarrow \begin{cases} b &= a - c \\ c &= a - b \end{cases}$$

Implementing this kind of understanding of what may be called the **atomic operations** of a design function is the focus of current and past research by a number of investigators

[5] In which case the computed variable of the design function is also a base variable of the design set.

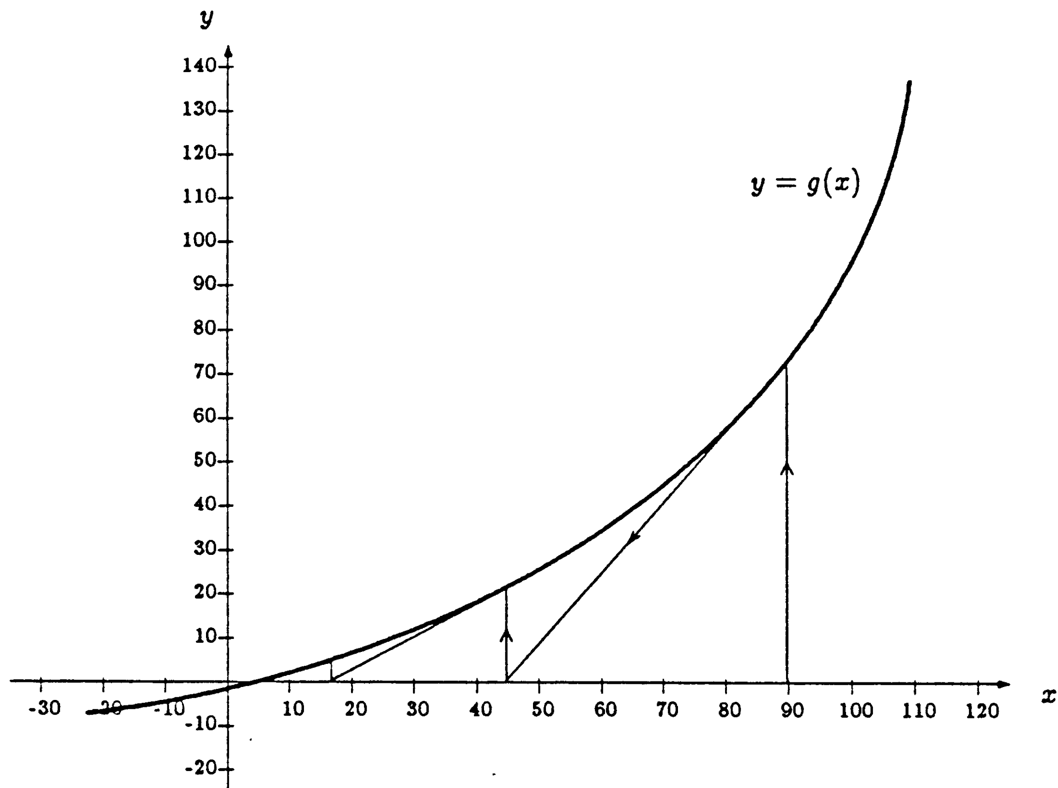



Figure 1.1: The Newton-Raphson technique for locating zeros.

[15,18].[6] However, use of these analysis techniques restricts the form design functions may take: namely, design functions must be represented by combinations of analytic operations. In real-world applications, however, design functions may often[7] take the form of numerical integrations or table-lookups, which cannot be represented by analytic operations. To allow greater flexibility in the choice of design functions, *Paper Airplane* takes a numerical approach toward design function inversion, by applying the Newton-Raphson method.

The Newton-Raphson method is a technique for numerically locating the zeros of an arbitrary function, and is represented graphically in Figure 1.1.[8] For an arbitrary function

[6] And has even resulted in the release of a commercial software product, called *TK!Solver* [24].

[7] At least in the field of aeronautics and astronautics ...

[8] Good introductory treatments of the Newton-Raphson method are presented in Chapter 2, Sections 10.3–

$g(x)$, the Newton-Raphson method consists of

1. Choosing a value for $x$, referred to as $x_i$, which is presumed to be near one of the zeros of $g(x)$.

2. Computing $g(x_i)$ and $g'(x_i)$.

3. If $g(x_i) \approx 0$, then $x_i$ is the desired zero.

4. If $g(x_i) \neq 0$, then a new value for $x$ is chosen, $x_{i+1}$, according to where the tangent to $g(x)$ at $x_i$ intersects the $x$ axis, i.e.,

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

   and the process is repeated.

This procedure may be applied to a design function in order to calculate the value of one of the design function's input variables by using the Newton-Raphson method to locate zeros of the *difference* between the value of the design function—given some **search value** for the input variable—and the known value of the computed variable. In terms of the example used above, $w = f(x, y, z)$, if design function $f$ is to be used to calculate a value for $x$ when the values of $w$, $y$, and $z$ are known, the Newton-Raphson method may be applied to locate the value for $x$ for which $F(x; w, y, z) = f(x, y, z) - w = 0$.[9]

Of course, there are certain problems with using this method. First of all, the success of this technique for numerically locating zeros is highly dependent upon the initial value used to start off the search. For this reason, *Paper Airplane* requires typical **upper** and **lower values**, and an **order of magnitude** value, to be associated with each design variable;[10] this approach mimics the way a human would solve the problem by hand, applying his knowledge of the problem domain to guess a reasonable initial value. Problems will also arise when

10.5, of Reference [7], and Chapter 10, Section 8, of Reference [9]. The indicated sections of Reference [7] also introduce such notions as the convergence and stability of numerical analysis techniques, and even describe a technique very similar to the "fixed-point" method, to be discussed in Chapter 2 of this thesis. The material presented in Reference [9] includes the description of an extension of the Newton-Raphson method for the solution of simultaneous equations which is related to the "simultaneous Newton-Raphson" method discussed in Chapter 3 of this thesis.

[9]Note that since, in this case, the value of $w$ is known and constant (as are the values of $y$ and $z$), it will be the case that $\frac{dF}{dx} = \frac{df}{dx}$. This derivative is needed for updating the value of $x$ while searching for zeros of $F$.

[10]Using this information, the initial value is determined by searching a list comprised of (1) the order of magnitude of the design variable, and (2) a twelve-interval logarithmic distribution (see Section 4.1.2) twice-extended beyond the design variable's upper and lower values, for the value for which the difference between the value of the design function and the (known) value of the design function's computed variable is closest to zero.

the search encounters a stationary point in the function for which the zero is sought, since the formula indicated above for updating the search value is invalid (because $g'(x)$ would be zero). And, of course, the method will fail when the function being examined has no zeros; this corresponds to the case where it impossible for a design function to return a value equal to the value which the computed variable already has. (In terms of the example, this is the case for which there is no value of $x$ for which $f(x, y, z) = w$ at the current values of $w$, $y$, and $z$.) Finally, in the case where the function does have zeros, the Newton-Raphson method is capable of finding only one of them: given a particular initial value for the **search variable** (i.e., the variable which is varied in trying to locate a zero), the method, if it finds a zero, will always find the same zero. If the design function which is to be "numerically inverted" by applying the Newton-Raphson method happens to be multi-valued in this inverted direction, for a single initial value for the search variable, only one of the possible "correct" values for the search variable will be computed.

### 1.3.3 Research Problem and Solution Constraints

With the use of the Newton-Raphson method in cases where design functions must be inverted, it is possible to solve a design function for any one of its associated design variables, given values for all the other **associated variables** of the design function. Thus, values for some subset of the derived variables of a design path may be calculated using those design functions which are perfectly constrained, as described in Section 1.3.1, by applying the numerical techniques discussed above. After the supply of perfectly constrained design functions has been exhausted, however, some unused design functions may remain, and some of the derived variables may not have calculated values. It may still be possible to use these design functions to calculate values for those derived variables by solving a set of simultaneous equations. It is at this point that the capabilities of the original version of *Paper Airplane* ended. Thus, this research has focused on the adaptation of the techniques used in *Paper Airplane* to the solution of simultaneous equations.

As discussed in Section 1.3.2, *Paper Airplane* uses only numerical techniques for analyzing design functions. *Paper Airplane* can only examine the inputs and outputs, and thus treats the design function as a sort of "black box," into which values for design variables may be sent, and a value for another design variable is returned. The task of this research has been to develop a method for solving simultaneous equations—specifically, sets of design functions reducible to two equations in two unknowns—under the ground rules established in *Paper Airplane* for specifying and solving design problems:

- Design functions may only be analyzed numerically, by applying input values and ob-

serving the output value. Thus, no information about the atomic operations performed by the design function is available, and function inversion must be accomplished numerically, using, for example, the Newton-Raphson method. Additionally, various "numerical experiments" may be performed on design functions, allowing, for example, estimation of the derivative of a design function at a given point.

- Design functions are specified as single-valued expressions for computing a value for a particular design variable (the design function's computed variable), based on the values of certain other design variables (the design function's input variables). While such specification of a design function requires that the design function be single-valued in the direction in which it is defined (i.e., computing a value for the computed variable based on the values of the input variables), no such restriction ***necessarily*** applies to the use of a design function in an inverted direction. Additionally, no restrictions are placed on whether design functions are linear or even analytical.
- The definition of a design variable may include such information about the design variable as typical upper and lower values, and an order of magnitude value.

## 1.4 Related Work

As indicated, past work in automating the computational aspects of preliminary design problems has primarily focused on developing individual computer programs for the solution of specific design problems (i.e., fixed design sets), based upon a specific sequence of computational steps (i.e., a fixed computational path, with fixed input variables). Examples of such programs may be found in References [4,11,13].

Recently, two different trends in the application of computers to problems in aerospace preliminary design have developed. The older of these two trends represents the attempt to integrate the operation of numerous individually-designed, large-scale program modules. In such a system, each program module is the equivalent of a single, "classical" CPD program (for example, a computational fluid dynamics program for subsonic wing design, a finite-element method program for fuselage structural design, or a blade-design program for turboprop engines); use of these modules is presided over by some form of "executive" program, which provides the interface between the program modules and the designer/user. In these systems, the executive program typically has access to a large database of information, which usually includes

1. the design variables required by each module;

2. the units for the design variables expected by each module;

3. the format of the input and output files required for running each module; and

4. results of previous runs of the various program modules.

Equipped with such knowledge, when asked to run a particular module, the executive program can prompt the designer/user for the needed inputs, pre-process this input for use by the program module, and then display the results as requested by the user, perhaps calling an auxiliary graphics or geometric modelling module.

An early, very ambitious attempt at such a system was NASA's Integrated Programs for Aerospace Design (IPAD), development of which began in the early 1970's [6]. IPAD was intended to incorporate as features

- simple, standardized means of adding additional modules as they become available, and
- a uniform "programming" language for specifying how the modules are to be called, allowing for iteration and conditional branching.

The Interactive Design and Evaluation of Advanced Spacecraft (IDEAS) program, also developed at NASA [22], combines several structural, thermal, drag, deployment, cost, and performance analysis modules, and graphics and animation capabilities, to compare eight different Space Station configurations. Plans for Boeing Aerospace Company's PDTool (Preliminary Design Tool) system include the capability to determine, based on the specified inputs, which modules may be run, and in what sequence the modules should be run in order to take advantage of values calculated in other modules [14].

In systems such as IPAD, IDEAS, and PDTool, the preliminary design process is modelled as the sequential application of several "operational modules"[6] or "technology codes"[14], each of which embodies a large set of discipline-specific design functions. The second current trend in Computer-Aided Preliminary Design is associated with those systems which take the alternative approach of modelling the preliminary design process at the level of individual design functions. For example, the Aircraft Design & Analysis System (ADAS), developed at the Delft University of Technology [2,21], allows the user to create a design set through access to an extensive database of design functions associated with the preliminary design of subsonic aircraft (derived from the text of Reference [20]). However, in using these design functions, ADAS requires the user to specify the computational path.

Two other examples of systems which follow this second trend are *Paper Airplane* (see Section 1.3.1) and MARKSYMA, a rule-based CPD system developed at the Rensselaer Polytechnic Institute [15]. Both *Paper Airplane* and MARKSYMA require the user to define his

own design functions, but once the base variables are chosen, these two systems are capable of automatically generating an appropriate computational path. MARKSYMA applies symbolic algebra techniques in establishing a computational path, while *Paper Airplane* relies solely on numerical techniques. The conceptual model behind both of these systems is the notion of functional "constraint propagation," first developed in the doctoral research of Steele [18].

A system which borrows traits from both trends is currently under development at Lockheed-Georgia. This system, called GRADES [16], relies on a fixed set of design functions and design variables, which are applied one at a time in order to build up individual software "models" of aircraft components, combining these elements to represent the improving aircraft design. Each aircraft component is modelled as a single software object, with which is associated knowledge about the mathematical equations governing the component's physical parameters (e.g., dimensions, weight, lift and drag characteristics, etc.) and any interactions between these parameters and any hierarchically superior components of the aircraft under design. (For instance, relocating the main wing will automatically change the location of the main wing fuel tanks, wing flaps, and any engines which may be mounted on the wing, as well as upgrade the location of the aircraft's center of gravity.) GRADES is similar to systems such as IDEAS and PDTool in that the design set is fixed (the design set is determined by which components are selected), but it is also similar to systems such as MARKSYMA and *Paper Airplane* insofar as the design functions associated with each component are applied individually, as the components are manipulated in the developing design.

## 1.5 A Note on Notation

In this thesis, phrases which appear in **boldface** refer to concepts associated with Computer-Aided Preliminary Design and the *Paper Airplane* program, which are independent of the method(s) used in processing design sets. In addition, such phrases will appear in boldface only when they are *first* introduced to the reader. Examples of such general-application phrases which have already been encountered are "design set," "computational state," and "atomic operations."

Phrases which appear in *italics*, however, are understood to have application to only one of the processing methods discussed in this thesis, or to have a different application for different processing methods. Such phrases will be italicized for *all* occurrences in the text. Examples of such method-dependent phrases (none of which have yet to be encountered) are "*loop variable*," "*first branch*," and "*forcing entry*."

It is unfortunate that most computer science work requires the introduction of an inordi-

nate number of ad hoc terms. To aid the reader, all such new phrases presented in this work are summarized in a glossary to be found in Appendix B.

# Chapter 2

# The Fixed-Point Approach and Explicit Representation of the Computational Agenda

## 2.1 The Fixed-Point Method

As indicated, a design problem may typically be defined by

- a set of parameters: the design variables;
- a set of functional relationships between those parameters: the design functions;
- and a set of constraints on some of the values of those parameters.

The design process can be viewed, then, as the solution of a set of simultaneous equations, which are based on the design functions, for a set of values—the values of the design variables.

The set of simultaneous equations which must be solved will depend upon which of the design variables are given initial values (as specified, presumably, by constraints on the design which restrict the possible values of certain of the design variables, as suggested above). Values must be calculated for those variables which are not assigned initial values, by means of the design functions. Thus, the design functions represent the equations of the set of simultaneous equations which must be solved, and those variables which are *not* assigned initial values (i.e., the derived variables) represent the unknowns for which values are sought.

An iterative scheme has been developed, which has been dubbed the "fixed-point" approach [5], whereby values are guessed for those variables which have not been initialized, and these guesses are subsequently replaced by values computed using the design functions. Since these computed values may have been based on values guessed for other design variables, the process is repeated using the computed values as the guessed values for the next

iteration. When the values of all the design variables are observed to converge from one iteration to the next (i.e., the values computed during the iteration are approximately equal to the values guessed for the variables at the beginning of the iteration), a solution to the set of simultaneous equations has been found.[1]

Observation has shown, however, that such convergence is by no means a guaranteed result of applying this procedure [12]. Convergence depends upon the direction in which the design functions are processed:[2] specifically, convergence depends upon the selection of the design variable for which a new value is to be calculated using a particular design function. And, of course, choice of the design variable to be calculated using a particular design function will affect the order in which the design functions may be used.

To illustrate this point, consider two design functions, $f$ and $g$, relating the two design variables, $x$ and $y$, neither of which have been initialized, such that

$$\begin{aligned} y &= f(x) \\ y &= g(x) \end{aligned}$$

Figure 2.1 illustrates the numerical process by which the fixed-point method might be used to solve this set of two equations in two unknowns. In Figure 2.1, the two heavy lines represent the curves of points which individually satisfy the above two functional relations between $x$ and $y$. As the figure indicates, the point of intersection of these two curves (and thus the solution to the pair of simultaneous equations) is determined by first guessing a value for $y$, using $f$ to calculate a corresponding value for $x$, using this computed value for $x$, along with the function $g$, to calculate a new value for $y$, and repeating this process until the values for $x$ and $y$ converge to the values at the intersection point. In Figure 2.1, the thin line with the arrowheads indicates this process of revising the values of the two variables until the intersection point is found.

Note however, that if the computational directions of these two functions were reversed—i.e., if $f$ were used to update the value of $y$ and $g$ were used to update the value of $x$ (instead of the other way around, as described above)—then instead of moving towards the intersection point, each revision of the values of these two variables would be successively *farther* away from the intersection point, effectively reversing the direction of the arrowheads on the thin line in Figure 2.1. Thus, the success or failure of the fixed-point method depends upon appropriate selection of the computational direction for each design function.

[1] The "fixed point" of a function $f$ is that value of $x$ for which $f(x) = x$; the approach taken here for solving the simultaneous equations is to input sets of values for the unknowns until those same values are returned.

[2] Where "processing" refers to the use of a design function to compute a new value for a design variable based upon the current values of the other design variables associated with the design function.

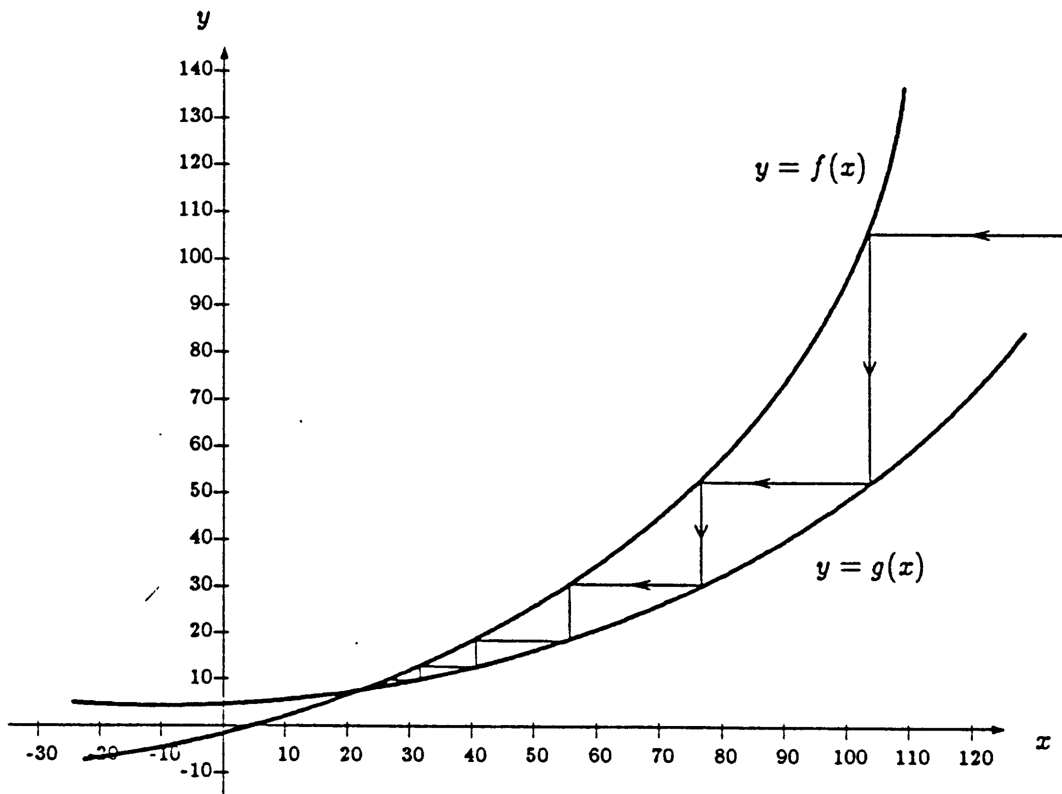



Figure 2.1: The fixed-point iterative technique for locating intersections.

Simple graphical analysis of the possible types of intersections of two continuous curves indicates that the choice of computational direction depends on the absolute values of the slopes of the two curves. In fact, the criterion turns out to be

$$\left|\left(\tfrac{dy}{dx};f\right)\right| > \left|\left(\tfrac{dy}{dx};g\right)\right| \implies \begin{cases} f \longrightarrow x \\ g \longrightarrow y \end{cases}$$

$$\left|\left(\tfrac{dy}{dx};f\right)\right| < \left|\left(\tfrac{dy}{dx};g\right)\right| \implies \begin{cases} f \longrightarrow y \\ g \longrightarrow x \end{cases}$$

where "$\left(\frac{dy}{dx};f\right)$" indicates the first derivative of design variable $y$ with respect to design variable $x$, as calculated using design function $f$, and "$f \longrightarrow x$" indicates that design function $f$ is to be used to calculate a value for design variable $x$, based on the current values of all the other design variables associated with $f$. This result suggests that, while it may not be possible to anticipate the proper way each design function should be used in trying to solve a pair of simultaneous equations, it should be possible to correct any observed divergence by suitable re-direction and re-ordering of the appropriate design functions.

This suggests that, for a given design path (which is determined by the design set, and which of the design variables are base variables in the design set), it is possible to generate multiple computational paths (i.e., sequences of design functions calculating values for design variables). In the example above, either $f$ computes $x$ and $g$ computes $y$, or $g$ computes $x$ and $f$ computes $y$—two computational paths are available for solving the problem. The convergence of a computational path depends on the values assigned to the base variables, and the initial "guess" values for the unknowns, insofar as these values affect the derivatives of any design functions which are to be solved by the fixed-point approach. Once these derivatives have been calculated,[3] convergence of the computational path can be predicted, and "correction" of the computational path may be performed where needed.

## 2.2 Implementation of the Fixed-Point Method

In order to incorporate the fixed-point method for solving simultaneous equations into the *Paper Airplane* program, two major changes in the program's architecture were necessary. First, a means for assigning "guess" values to design variables was needed. This was accom-

[3]Presumably, all derivatives are calculated by numerical means, since, again, the fixed-point approach assumes that no knowledge of the atomic operations within a design function is available. A second-order approximation to the first derivative of a function $h(z)$ with respect to variable $z$ is

$$\frac{dh}{dz} = \frac{h(z+\Delta z) - h(z-\Delta z)}{2\Delta z} + O[(\Delta z)^2]$$

plished by simply creating a new computational state[4] for design variables: a "G" ("guess") state was added, signifying that the current value of a design variable is a provisional value, subject to replacement by a calculated value (in contrast to an I-variable—a design variable whose state is "I"—which has been assigned a value which may not be changed). With the introduction of this "G" state, it becomes possible to insist that design variables *always* have a value, making the value of all un-initialized variables (i.e., derived variables) provisional by assigning the "G" state to these variables. This makes the "F" state unnecessary, but also requires definition of some kind of default value for all design variables. The design variable's order of magnitude, specification of which is already required by *Paper Airplane* (see page 7), was chosen to also serve as the default value for derived variables.[5]

In addition, the original version of *Paper Airplane* made design function and design variable processing choices concurrent with actual value calculation. Upon selecting a design function for processing, the decision was forgotten, making an eventual re-ordering of the computational path very expensive. For this reason, it was necessary to separate the process of computational path *determination* from the process of computational path *execution*. To facilitate this separation, the choices made during computational path determination were represented explicitly within a computational path "structure," referred to as a **computational agenda**, as a first step in developing procedures for recovering from the computational path instabilities described above. There are two immediate advantages to having a computational agenda structure:

- When reordering of the computational path is deemed appropriate, a record of past decisions is immediately available and this plan may be modified to reflect whatever new ordering is recommended.

- The computational path need not be recomputed with each iteration (as was necessary in the original version of *Paper Airplane*), thus speeding up design path processing.

In the remainder of this chapter, the detailed implementation of this computational agenda structure, and the problems which were ultimately encountered in applying the fixed-point approach to design problems, will be discussed.

---

[4]See page 4.

[5]Note that these guess values need not be consistent—i.e., they need not satisfy the design functions. This is why such values are merely "guessed." Consistent values are determined through calculations, which update both the value and the state of design variables.

## 2.3 Computational Agenda Construction

As indicated in the preceding sections of this chapter, the computational path is dependent upon which variables are assigned input values (i.e., which variables are base variables to the design set), and which variables' values must be computed (i.e., which variables are derived variables), since this determines the unknowns for the set of simultaneous equations. Once the unknowns are determined, a plan must be developed for computing values for these unknowns using the design functions.

This research has involved representing this plan as an explicit computational agenda structure. This structure is implemented using the LISP `flavor` facility,[6] which allows for the definition of the following attributes of the computational agenda structure:

> *forced path:* an ordered set of functions which are perfectly constrained by the base variables.
>
> *loops:* a set of ordered sets of functions which comprise a computational loop; iteration over these functions is necessary as none of these functions are perfectly constrained by the base variables.
>
> *overconstraining functions:* a set of all the overconstraining functions (see Section 2.3.3) which have been discovered during the course of the construction of the agenda.

### 2.3.1 Forced Path Construction

*Paper Airplane* design functions may be represented by equations of the form $x_3 = f_1(x_1, x_2)$, where $f_1$ represents a design function of the parameters $x_1$ and $x_2$ (these functional parameters are themselves design variables), which, in its explicit direction, computes a value for the design variable $x_3$. Although the form of the equation used here suggests that the values of $x_1$ and $x_2$ must be known in order to determine the value of $x_3$, *Paper Airplane* is able to numerically invert equations of this form when necessary (see Section 1.3.2), and thus a value may be computed for any one of the design variables associated with the design function $f_1$—namely $x_1$, $x_2$, and $x_3$—as long as the values of the other two design variables are known. A design function for which the values of all but one of the associated variables is known is considered to be "perfectly constrained"[7]: a value may be readily calculated for

[6] `Flavors` are LISP constructs for object-oriented programming, and are available in both NIL (New Implementation of LISP)[3] and ZetaLISP (Lisp Machine LISP)[17]. Alas, `flavors` are not part of the Common LISP standard[19] ...

[7] Thus, with replacement of the "F" state with the "G" state, a perfectly constrained design function is a design function for which only one of the variables associated with it is a G-variable (a design variable whose state is "G").

the remaining design variable.

Some subset of the design functions of a given design set will be perfectly constrained by the base variables and those design variables which may be directly computed using the base variables. These are the design functions which are said to comprise the forced path of the computational agenda. Since these functions are perfectly constrained, values for the uninitialized variables associated with these functions may be computed immediately, and no iteration will be needed; these computed values are guaranteed to satisfy the forced path design functions, and will be used to further constrain those functions which cannot be included in the forced path.

To illustrate these concepts, the following example design set is introduced:

$$
\begin{aligned}
x_3 &= f_1(x_1, x_2) \\
x_4 &= f_2(x_1, x_2, x_3) \\
x_5 &= f_3(x_3, x_6) \\
x_6 &= f_4(x_1, x_5) \\
x_6 &= f_5(x_2, x_5, x_7) \\
x_7 &= f_6(x_2, x_5) \\
x_8 &= f_7(x_3, x_4)
\end{aligned}
$$

This design set has seven design variables—$x_1, x_2, \ldots, x_6, x_7$—and six design functions—$f_1, f_2, \ldots, f_5, f_6$. Design variables $x_2$ and $x_3$ are to be initialized (i.e., assigned values by the designer), and this is indicated by appending a superscript "I" to these variables. This superscript "I" represents the state of the design variable where, as indicated above, "I" stands for "initialized." For consistency, a state is also needed for the uninitialized design variables. This is the "G" state, introduced in the preceding section: for a design variable which is uninitialized, information about the appropriate value for the design variable (at least before any processing is done) can only be guessed. Thus the design set may now be

written as

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1(x_1^{\mathrm{G}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{G}} &= f_2(x_1^{\mathrm{G}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_1^{\mathrm{G}}, x_5^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{G}} &= f_7(x_3^{\mathrm{I}}, x_4^{\mathrm{G}})
\end{aligned}
$$

Now, since $x_2$ and $x_3$ have been initialized, design function $f_1$ is perfectly constrained, and a value for $x_1$ may be computed. Thus $f_1$ becomes the first design function in the forced path of this design set's computational agenda. Since the agenda is only being *constructed*, it is not necessary to actually make the computation needed to determine the appropriate value for $x_1$—indeed, the values of the design variables do not affect the way in which the computational path is initially constructed; it is only the states of the design variables which affect agenda construction.[8] (And once the computational agenda is constructed, the values of all the design variables may be determined by processing the completed agenda.) However, as far as completing the agenda is concerned, it must be indicated that a computed value is now available for $x_1$, and this is done by changing this variable's state from "G" to "K" which implies that a computed value is now available for the design variable.[9] Design variable $x_1^{\mathrm{K}}$ also appears in the relations for $f_2$ and $f_4$, so its state will be changed in those two equations as well. Additionally, some indication that the design function $f_1$ has been added to the computational agenda should be given, and for this reason $f_1$ is tagged as having been

[8]However, execution of the computational agenda may uncover numerical conditions which require further revision to the computational agenda. Such numerical conditions *are* dependent upon design variable values.

[9]The state "K" here signifies a *hard* computation (i.e., a *k*omputation) which follows directly from the values of the base variables; the state "C" is reserved for *soft* computations which do not follow *directly* from the values of the base variables, but require iteration, as well.

"used," by appending a superscript "U." Thus the design set may now be written as

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{G}} &= f_2(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{G}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{G}} &= f_7(x_3^{\mathrm{I}}, x_4^{\mathrm{G}})
\end{aligned}
$$

Using $f_1$, a reliable value for $x_1$ has been obtained, and now the equation involving $f_2$ is observed to be perfectly constrained. Design function $f_2$ may be added to the computational agenda to calculate $x_4$, and the design set may now be written as

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{G}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{G}} &= f_7(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}
$$

Note that it was not until after construction of the forced path began that $f_2$ was observed to be perfectly constrained. The design function could not be added to the forced path until those design functions which were found to be *immediately* perfectly constrained by the base variables had been included in the forced path. Similarly, inclusion of $f_2$ in the forced path allows $f_7$ to be added to the forced path, to calculate a value for $x_8$. Design function $f_7$ is the last design function which may be added to the forced path, however, since no more design functions are observed to be perfectly constrained. At the end of forced path construction,

then, the state of the design set may be written as

$$\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{G}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}$$

### 2.3.2 Loop Construction

As indicated in Section 2.1, to find values for those variables whose values cannot be calculated in the forced path, iteration loops must be constructed. The iteration technique adopted here is, as mentioned earlier, the fixed-point approach, in which

1. Values are assumed for certain of the variables.

2. Values are calculated for the remaining variables.

3. Finally, based on these calculations, new values are calculated for the variables originally assigned assumed values.

The variables which are initially given assumed values (step 1) are referred to as the *loop variables*. The fixed-point approach continues with

4. The new values calculated for the *loop variables* are used as the assumed values for these variables for the next iteration.

5. Iteration over these steps continues until the values calculated for the *loop variables* at the end of a given iteration are sufficiently close to the values used as assumed values for the *loop variables* at the start of the iteration.

This approach is based upon consideration of how a human might attempt to solve a set of non-linear design equations, none of which were perfectly constrained. In aircraft design, for instance, when calculating the weights of the various aircraft components, one often starts by assuming a value for the total aircraft weight at takeoff—referred to as the "gross takeoff weight"—then calculating the components weights, and finally summing these component weights to find a new value for the gross takeoff weight. This process is continued until the

sum of the component weights is sufficiently close to the value for gross takeoff weight that was assumed in order to compute these component weights in the first place.

In constructing a computational agenda, a means must be found for representing this iterative process. This is the role of the *loops* attribute of the agenda structure. The *Paper Airplane* implementation of the fixed-point approach follows these steps:

1. Choose a *loop variable* by determining which G-variable occurs most often in the functions which have yet to be assigned to the agenda.

2. Identify those design functions which are perfectly constrained (based upon the assumption that a value is available for the *loop variable*), and assign them to the agenda.

3. Continue this process until one of these four conditions are met:

   (a) a design function is found for which values have been calculated for all the associated variables except for one, which happens to the *loop variable*, in which case this function is used to "close" the loop, and a new loop is begun;[10] or

   (b) no more perfectly constrained functions are found, in which case a new loop must be begun without having "closed" the current loop; or

   (c) only overconstraining functions remain to be processed, in which case loop construction has been completed; or

   (d) there are no functions left to add to the agenda, in which case the computational agenda has been completed.

A loop is said to be "closed" when a design function has been found which may be used to calculate a value for the *loop variable*.

Note that by choosing the G-variable which occurs most often in the remaining design functions as the loop-variable, the analogy with a human designer is continued: in aircraft design, gross takeoff weight occurs very frequently in design functions. There is nothing mysterious about using frequency of occurrence as the deciding factor, though: by assuming a value for the most frequently-occurring variable, the maximum number of functions will end up being perfectly constrained and available for processing. In this way the total number of loops is kept small; with fewer loops, instabilities should be easier to correct.

[10] Actually, any perfectly constrained functions available when the loop is "closed" may be added to the end of the current loop, without requiring commencement of a new loop.

To return to the example, after constructing the forced path, the design set looked like this (see page 22)

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{G}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}
$$

There are four design functions yet to be included in the computational agenda, and three G-variables remain: $x_5^{\mathrm{G}}$, $x_6^{\mathrm{G}}$, and $x_7^{\mathrm{G}}$. Design variable $x_5$ occurs in all four of the remaining design functions, $x_6$ occurs in three, and $x_7$ occurs in just two. Thus, $x_5$ is chosen as the first *loop variable*, and its state is changed from "G" to "L." The design set may now be written as

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{L}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{L}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{L}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{L}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}
$$

Once the *loop variable* is chosen, loop construction begins. By changing the state of $x_5$ from "G" to "L," it is now the case that there is just one G-variable associated with each of the design functions $f_3$, $f_4$, and $f_6$; these three design functions have been tentatively made to be perfectly constrained since, for the time being, a value has been assumed for the *loop variable*, $x_5$.

Any one of these three functions—$f_3$, $f_4$, and $f_6$—may now be added to the agenda as part of the loops attribute. These three functions may be used to compute values for the three remaining variables—$x_6$, $x_7$, and, eventually, $x_5$, itself—but great care must be taken in how these three functions are added to the agenda. In particular, either $f_3$ or $f_4$ must be used to "close" the loop—computing the new value for $x_5$—or else they would be redundant

(since both would compute $x_6$, the only G-variable associated with either of the two design functions). By adding only one function to the agenda at a time, this redundancy is avoided, because one of these two functions may be added to the agenda to compute a value for $x_6$, and the other may be used to close the loop by computing a value for $x_5$.

Initially, any one of these three design functions involving the *loop variable* may be added to the computational agenda. Arbitrarily choosing to add $f_3$ to the agenda, the design set may be written as

$$\begin{aligned}
x_3^{\mathtt{I}} &= f_1^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}) \\
x_4^{\mathtt{K}} &= f_2^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}) \\
x_5^{\mathtt{L}} &= f_3^{\mathtt{U}}(x_3^{\mathtt{I}}, x_6^{\mathtt{C}}) \\
x_6^{\mathtt{C}} &= f_4(x_5^{\mathtt{L}}, x_1^{\mathtt{K}}) \\
x_6^{\mathtt{C}} &= f_5(x_2^{\mathtt{I}}, x_5^{\mathtt{L}}, x_7^{\mathtt{G}}) \\
x_7^{\mathtt{G}} &= f_6(x_2^{\mathtt{I}}, x_5^{\mathtt{L}}) \\
x_8^{\mathtt{K}} &= f_7^{\mathtt{U}}(x_3^{\mathtt{I}}, x_4^{\mathtt{K}})
\end{aligned}$$

Note that the state of $x_6$ has changed from "G" to "C," since the agenda now includes an entry for computing $x_6$ as part of a computation loop.[11] Now, however, design function $f_4$ has become "overconstrained": it has no G-variables left. One of its design variables is an L-variable (a *loop variable*), however, which means that $f_4$ may be used to "close" the loop, as indicated above (refer also to item 3(a) in the description of the loop-building heuristics on page 23). Thus, $f_4$ is added to the agenda, and the state of $x_5$ may be updated from "L" to "C." The design set may now be written as

$$\begin{aligned}
x_3^{\mathtt{I}} &= f_1^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}) \\
x_4^{\mathtt{K}} &= f_2^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}, x_3^{\mathtt{I}}) \\
x_5^{\mathtt{C}} &= f_3^{\mathtt{U}}(x_3^{\mathtt{I}}, x_6^{\mathtt{C}}) \\
x_6^{\mathtt{C}} &= f_4^{\mathtt{U}}(x_5^{\mathtt{C}}, x_1^{\mathtt{K}}) \\
x_6^{\mathtt{C}} &= f_5(x_2^{\mathtt{I}}, x_5^{\mathtt{C}}, x_7^{\mathtt{G}}) \\
x_7^{\mathtt{G}} &= f_6(x_2^{\mathtt{I}}, x_5^{\mathtt{C}}) \\
x_8^{\mathtt{K}} &= f_7^{\mathtt{U}}(x_3^{\mathtt{I}}, x_4^{\mathtt{K}})
\end{aligned}$$

It is no longer possible to tell that $x_5$ was a *loop variable* from looking at the design set, but since this information may be needed later, it is explicitly recorded in an appropriate slot in

[11]Recall that the state "K" denotes computation in the forced path, while the state "C" indicates computation in the iteration loops.

the computational agenda data structure (for specific details, see Section 2.4.2). Now that the loop has been closed, it might seem appropriate to begin a new loop. However, the loop calculations already included in the agenda have made it possible to solve for some of the remaining G-variables *without* starting a new loop, so instead of starting a new loop, more functions will be tacked onto the end of the current loop.[12] Only when the most highly-constrained function has two or more G-variables is it necessary to start a loop. As this is not the case here, a second loop is not needed.

Two design functions remain, $f_5$ and $f_6$, and both have exactly one G-variable. In fact, they both have the same G-variable, namely $x_7$. Only one may be added to the loop, though, so, assuming that $f_5$ is chosen for inclusion in the agenda at this point, the design set may now be written as

$$\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{C}} &= f_3^{\mathrm{U}}(x_3^{\mathrm{I}}, x_6^{\mathrm{C}}) \\
x_6^{\mathrm{C}} &= f_4^{\mathrm{U}}(x_5^{\mathrm{C}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{C}} &= f_5^{\mathrm{U}}(x_2^{\mathrm{I}}, x_5^{\mathrm{C}}, x_7^{\mathrm{C}}) \\
x_7^{\mathrm{C}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{C}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}$$

where $f_5$ has been added to the agenda at the end of the loop currently under construction, and the state of $x_7$ has been updated from "G" to "C." Only one function has yet to be included in the agenda, and this function has no G- or L-variables, so loop construction may be terminated as per item 3(c) in the description of the loop-building heuristics on page 23.

Once loop construction is completed, a sequence of calculations which may be performed to compute values for the derived variables is available. For the example presented here, the resulting series of steps is

$$\{x_2^{\mathrm{I}}, x_3^{\mathrm{I}}\} \xrightarrow{f_1} x_1^{\mathrm{K}} \xrightarrow{f_2} x_4^{\mathrm{K}} \xrightarrow{f_7} x_8^{\mathrm{K}} \xrightarrow{\text{guess}} \underbrace{x_5^{\mathrm{L}} \xrightarrow{f_3} x_6^{\mathrm{C}} \xrightarrow{f_4} x_5^{\mathrm{C}}}_{\text{loop}} \xrightarrow{f_5} x_7^{\mathrm{C}}$$

[12]Note that in processing the loop, these additional design functions need only be processed *once*—i.e., after iterative processing of the preceding steps in the loop has produced a solution value for the *loop variable*. For the sake of ease of implementation, this fact was disregarded in this implementation of the fixed-point approach, and these design functions were not considered to be any different from the other design functions included in the loop. During the course of investigating the "simultaneous Newton-Raphson" method, it was determined that a more rigorous representation of computational loop structure was needed, and the distinction between design functions over which iteration was required and design functions which merely depended upon the results of *prior* iteration was made explicit. See Chapter 3 for details.

where the single operation of using design function $f_j$ to compute a value for design variable $x_i$ is represented symbolically as "$\xrightarrow{f_j} x_i$."

### 2.3.3 Overconstraining Functions

As indicated in the above discussion, whenever a function is discovered to have no **free variables**—i.e., no G-variables or L-variables[13]—an **overconstraint** has been detected. Design functions having no free variables need not be processed, since any computations made using these functions would attempt to replace the values of variables whose values, according to the approach used here, may not be overwritten (namely, variables whose states are either "I", "K", or "C"). A list is kept of such functions under the *overconstraining functions* attribute of the computational agenda.

A design set for which the number of design functions available exceeds the number of free (i.e., uninitialized) design variables is said to be **overconstrained**, just as a set of simultaneous equations with more equations than unknowns is "overconstrained." Alternatively, if not enough design variables are initialized, then the number of unknowns will exceed the number of the equations, and the design set is said to be **underconstrained**.[14] Following this nomenclature, then, design functions which have no free variables are said to be **overconstraining functions**, because such design functions generally result from the design set being overconstrained. Such an overconstraining function is said to give rise to **incompatibilities** when the design variable values which might be computed using the overconstraining function are different from the values already assigned to those design variables. In other words, an incompatibility results when, due to the existence of an overconstraining function, there are two functions which may be used to compute the value for a particular variable, and these two computed values do not agree.

Additionally, it should be noted that an inequality between the number of free variables and the number of design functions is not the only possible source of overconstraining functions (i.e., functions with no free variables). Certain choices of base variables—for instance, initializing all the design variables associated with a particular design function—may overconstrain a particular design function, without necessarily overconstraining the design set (according to the definition of an overconstrained design set presented above). Such a choice of base variables is said to result in a **local incompatibility**. Thus, it is possible—however

---

[13]These are the only two states of design variables whose values may be replaced by computed values.

[14]It will be the case in this implementation of the fixed-point approach that when the design set is underconstrained, not all of the loops will be closed, since possible loop-closing functions are unlikely to be reduced to the condition that all the associated variables of the design function except for the *loop variable* have been either initialized or assigned to the agenda for computation by some other function.

semantically inappropriate—to have overconstraining functions although the design set is not overconstrained.

The example design set introduced earlier in this chapter is an overconstrained design set. Two design variables were initialized, leaving six free variables. There are seven equations, though. As noted at the end of the previous section, this means that after construction of both the forced path and the computational loops, one design function has yet to be included in the agenda; here, this is design function $f_6$. This design function is added to the agenda's list of overconstraining functions, which is indicated by adding a superscripted asterisk to the design function's label. Once this design function is added to the agenda as an overconstraining function, there are no design functions left to examine; all design functions are now included in the agenda, and agenda construction has been completed. Thus, at the end of agenda construction, the design set may be written as

$$\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{C}} &= f_3^{\mathrm{U}}(x_3^{\mathrm{I}}, x_6^{\mathrm{C}}) \\
x_6^{\mathrm{C}} &= f_4^{\mathrm{U}}(x_5^{\mathrm{C}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{C}} &= f_5^{\mathrm{U}}(x_2^{\mathrm{I}}, x_5^{\mathrm{C}}, x_7^{\mathrm{C}}) \\
x_7^{\mathrm{C}} &= f_6^{*}(x_2^{\mathrm{I}}, x_5^{\mathrm{C}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}$$

A pictorial representation of the completed agenda is presented in Figure 2.2; the various components of the agenda are examined in further detail in the following section, Section 2.4.

## 2.4 Representation of the Agenda Steps

Before discussing the means by which the computational agenda is executed, mention should be made of just *how* design functions are added to *Paper Airplane* computational agenda structures to comprise the forced path and computational loops of the agenda. Within both the forced path and the loops, individual steps in the computational agenda are represented as agenda entries, as described in Section 2.4.1. Each entry points to both the entry which precedes it and the entry which follows it, and thus a chain of entries is built up to represent either the forced path or the computational loops. For the purposes of storing information which may be relevant to the processing of the computational loops, a loop header appears at the top of each loop. The structure of loop headers is described in Section 2.4.2. Recalling the description of the computational agenda given in Section 2.3, then, the *forced*

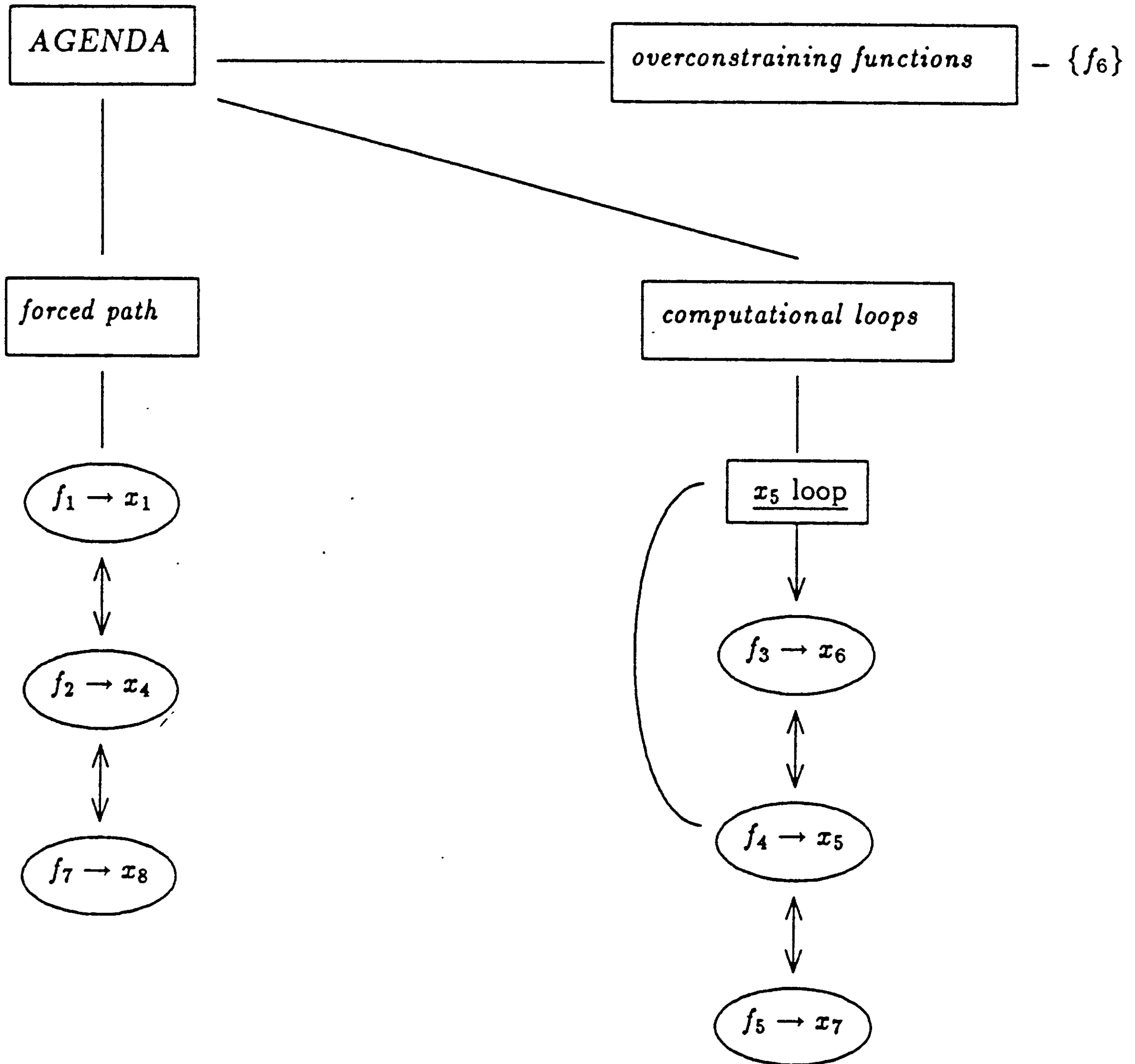



Figure 2.2: The computational agenda structure for the example design set.

*path* attribute of the computational-agenda points to the first entry of the forced path (and, through this first entry, to the rest of the forced path), while the *loops* attribute of the agenda points to a list of loop headers.

### 2.4.1 Agenda Entries

Individual steps in the computational agenda are represented by corresponding **agenda entries**. These structures are also implemented using LISP's `flavor` facility, defining the following attributes for agenda entries:

*entry function:* the function to be processed in a particular entry in the computational agenda.

*entry variable:* the variable for which a value is to be computed using the *entry function* of a particular entry in the computational agenda.

*computational direction:* the direction in which the *entry function* must be processed in order to compute a value for the *entry variable* (i.e., "`forward`" if the *entry variable* is the design variable computed by the *entry function* according to the definition given for the function, "`reverse`" otherwise).

*other functions:* a list of those functions which were just as well-constrained as the *entry function* when the *entry function* was chosen for inclusion in the agenda (i.e., a list of the alternatives available at the current step in the agenda, which may be useful for instability recovery).

*entry-used flag:* a flag used in processing the computational agenda to indicate whether or not a particular entry has been processed or not.

*entry divergence flag:* a flag used in processing the computational agenda to indicate whether or not the values computed for the *entry variable* of a particular entry were observed to diverge or converge over the last two iterations.

*previous entry:* the entry which immediately precedes a particular entry in the computational agenda.

*next entry:* the entry which immediately follows a particular entry in the computational agenda.

Information is stored concerning the means by which the agenda entry is to be processed, as well as the behavior of the agenda entry as one step in the computational path. In addition,

those alternative steps which were available at the time of agenda construction are also stored, for possible use in recovering from instabilities. Finally, it should be pointed out that because incompatible functions need not be processed, but only detected, the incompatible functions attribute of the computational agenda need store only the incompatible functions themselves, and does not require this agenda entry representation. In Figure 2.2, on page 29, each agenda entry is represented as an ellipse, labelled with the agenda function and the agenda variable.

### 2.4.2 Loop Headers

To make information about the structure of each loop readily available to *Paper Airplane* (for use in instability recovery, for example), a **loop header** appears at the top of each computational loop (which, as discussed above, is represented as a chain of agenda entries). In *Paper Airplane*, the attributes assigned to the loop header structure are as follows:

*loop variable:* the design variable chosen as the *loop variable* for the loop which follows the header; this design variable will also be the entry variable of the *closing entry* (see below).

*other variables:* a list of those design variables which also satisfied the criterion for being chosen as *loop variable* at the time the *loop variable* was actually chosen. (I.e., this attribute records other equally good choices available at the time of loop variable selection—information which could be useful in instability recovery.)

*first entry:* this attribute points to the first agenda entry of the computational loop which follows the header.

*closing entry:* this attribute points to the agenda entry which closes the loop; that is, it points to the entry which is used to calculate a new value for the *loop variable*.

*looping entry:* this attribute points to the agenda entry which enables the loop to be closed; specifically, it points to the last entry in the loop which calculates one of the C-variables of the *closing entry* and has as one of the associated variables of its entry function the *loop variable*.

*looping variable:* this attribute points to the design variable which is the entry variable of the *looping entry*.

The intent of this representation of computational loops as possessing loop headers enables the storage of various pieces of information which were used in originally constructing the

loop, including possible alternatives which were not originally used. The availability of this information is intended to aid recovery from instabilities, perhaps through intelligently exploring alternative computational paths. In Figure 2.2, on page 29, the loop header is represented by an underlined phrase, indicating the *loop variable*; note also the arc connecting the header to the *closing entry*, suggesting the iterative nature of computational loops.

## 2.5 Execution of the Computational Agenda using the Fixed-Point Method

Once a suitable computational agenda has been construction, application of the fixed-point method in processing the computational agenda is rather straightforward. First, the forced path entries are sequentially evaluated in order to calculate values for those design variables which are associated with design functions which are perfectly constrained by the base variables. The computational loops may then be processed sequentially.

To process a computational loop, a value must be assumed for the *loop variable* associated with the loop (see below). The sequence of agenda entries which comprise the loop (beginning with the *first entry*) is then evaluated, during which a new value for the *loop variable* is calculated. The loops entries are processed again, using the new value of the *loop variable*, repeating this procedure until the value calculated for the *loop variable* during one iteration of loop processing is sufficiently close to the value assumed for the *loop variable* at the beginning of the iteration.[15]

As indicated in the preceding paragraph, a value for the *loop variable* is needed to begin the first iterative pass through the loop's agenda entries. It was decided that whatever value had been previously assigned to the *loop variable* would be a suitable seed value for the fixed-point procedure. Thus, in applying the fixed-point method, the first iteration simply uses whatever the current value of the *loop variable* is as the assumed value—presumably,this will be some value calculated by an earlier application of the fixed-point method—and subsequent iterations can again use the *loop variable*'s current value, which will have been calculated during the previous iteration. As long as the input values are not changed too much from one processing run to another,[16] using the value from previous computations should be sufficiently close to the correct output value for the *loop variable* to assure success of the fixed-point method. Thus, it is the "previous value" of the *loop variable* which is used to

[15]The next section, Section 2.6, suggests how this procedure might be modified should the computational path exhibit instabilities.

[16]Here, it is assumed that the designer/user will make gradual changes in the input values to the design set.

commence iterative processing of the computational loops.[17]

## 2.6 Recovery from Instabilities

As discussed in Section 2.1, instabilities in the computational path can lead to divergence in the iterative values computed for design variables. In order to compute the actual values for these design variables (as determined by the set of simultaneous equations which characterize the design set), however, a means must be found for changing the computational path in such a way that convergence of these values will result.

### 2.6.1 Satisfying the Convergence Criterion

As indicated on page 16 of Section 2.1, convergence of the fixed-point method depends on the derivatives of the two functions to which the method is being applied. It would thus appear that a simple calculation of the derivatives of the appropriate design functions just before processing the loop would indicate whether or not the loop calculations will be convergent or divergent. Based upon this determination, the loop may be restructured if necessary. Note that stability may not be determined until loop processing has begun; although prediction of convergence at the time of loop construction would be preferable (in order to avoid building divergent loops in the first place), calculation of the derivatives upon which the prediction is based requires that the values of those design variables which are computed in the forced path and in previous computational loops be calculated. These calculated values are needed because evaluation of the derivatives requires accurate knowledge of the values of all the design variables associated with the design functions.

The fixed-point convergence criterion was applied to the processing of the computational loops described above as follows:

1. Prior to processing a computational loop, the derivative of the *loop variable* with respect to the entry variable of the *looping entry* (which may be referred to as the *looping variable*) is calculated, using the entry function of the *looping entry* (the *looping function*), and the entry function of the *closing entry.* The former derivative is referred to as the *looping derivative*, while the latter is referred to as the *closing derivative.*

2. If the *closing derivative* has a smaller absolute value than the *looping derivative*, then the convergence criterion is satisfied, and no alteration of the computational loop is

[17]Recall that, prior to any calculations or value assignments made by the designer/user, the value of a design variable defaults to its order of magnitude value.

necessary. The iteration (i.e., a single pass through the entries of the computational loop) may immediately proceed.

3. If the *closing derivative* has a larger absolute value than the *looping derivative*, then the convergence criterion is not satisfied, and the computational loop must be rearranged, in order to promote convergence.

How is this rearrangement to be effected? Well, it should first be observed that if the *loop variable* and the *looping variable* were to be exchanged, the new derivatives would just be the inverses of the original derivatives:

$$\frac{dx_{\text{loop}}}{dx_{\text{looping}}} = \frac{1}{\frac{dx_{\text{looping}}}{dx_{\text{loop}}}}$$

And since inverting both sides of an inequality reverses the direction of the inequality (i.e., if $a > b$, then $\frac{1}{a} < \frac{1}{b}$), it follows that exchanging the *loop variable* and the *looping variable* will result in satisfaction of the fixed-point convergence criterion for the computational loop. This observation may be added to the statement of the fixed-point method convergence criterion given above (see page 16), yielding the amended convergence criterion,

$$\begin{array}{l} \left|\left(\frac{dy}{dx};f\right)\right| > \left|\left(\frac{dy}{dx};g\right)\right| \implies \left|\left(\frac{dx}{dy};f\right)\right| < \left|\left(\frac{dx}{dy};g\right)\right| \implies \begin{cases} f \longrightarrow x \\ g \longrightarrow y \end{cases} \\ \left|\left(\frac{dy}{dx};f\right)\right| < \left|\left(\frac{dy}{dx};g\right)\right| \implies \left|\left(\frac{dx}{dy};f\right)\right| > \left|\left(\frac{dx}{dy};g\right)\right| \implies \begin{cases} f \longrightarrow y \\ g \longrightarrow x \end{cases} \end{array}$$

Thus, the final step in the loop "pre-processing" procedure outlined above may be re-written

3. If the *closing derivative* has a larger absolute value than the *looping derivative*, then the convergence criterion is not satisfied. Convergence may be established by making the following rearrangement:

   (a) Set the *entry variable* of the *looping entry* to be the *entry variable* of the *closing entry* (i.e., the *loop variable*).

   (b) Set the *entry variable* of the *closing entry* to be the original *entry variable* of the *looping entry* (i.e., the *looping variable*).

Note that in making this exchange, the order in which the design functions are processed is unchanged (since their locations in the computational agenda are not changed). Only the computational directions of these two functions are altered, since the design variables whose values they are to compute have been switched. By effectively reversing the roles of these

two variables, the convergence criterion may be met, and processing of the agenda entries of the computational loop may proceed.

To close this section, perhaps some elucidation of the role and definition of the *looping entry*, is in order, to help clarify the above statements. First of all, the *looping entry* is found by searching backwards through the agenda entries of the computational loop after the *closing entry* has been constructed. In accordance with the definition given on page 31, the *looping entry* will be the last entry in the loop which calculates one of the C-variables of the *closing entry and* has the *loop variable* as one of the associated variables of its entry function. Defining the *looping entry* in this manner ensures that

- its entry function is a function involving both the *looping variable* and the *loop variable*, as will be the entry function of the *closing entry*; and
- that this function will occur rather late in the loop (in fact, it should typically be the entry which immediately precedes the *closing entry*).

This first condition guarantees that the derivative calculation and the entry variable exchange indicated above will be possible. The second condition is intended to minimize any problems in the processing of the loop which might result from making the exchange.

### 2.6.2 Implementation Problems

After the algorithms and data structures discussed above had been written and interfaced with the existing *Paper Airplane* code, testing of this new code with the design set presented in the Appendix revealed serious problems: convergence, for the case tested, was impossible to obtain. The program had no problem in constructing the computational agenda according to the guidelines presented above, with computing the derivatives needed to predict convergence, or with performing the exchange of entry variables for the *closing entry* and the *looping entry* when necessary. However, when the algorithms above were followed, convergence did not result.

The failure of the implementation of the fixed-point method described in this chapter is attributed to the following problem: the fixed-point method, as indicated in Figure 2.1 on page 15, is essentially a technique for solving two equations in two unknowns, and this implementation incorrectly identifies the two equations to be solved. In this implementation, the (iterative) calculation of the *loop variable* in the *closing entry* and the *looping variable* in the *looping entry* are treated as the two equations in two unknowns. The derivatives on which the convergence criterion is founded are *actually* those of two "composite functions" based on the chain of design functions represented by the agenda entries of the computational loop.

(For instance, there is one "composite function" based upon the entries from the loop's *first entry* to its *looping entry*, and another "composite function" following the chain of entries from just after the *looping entry* up to the *closing entry*.) Because the procedure outlined above looks only at the derivatives of one design function from each of the two "composite functions," the convergence prediction which is made is invalid.

Furthermore, because the values of the design variables will change during loop processing (after all, that is why loop processing occurs), the values of the associated variables—and, therefore, the derivatives—of the design functions of the *closing entry* and the *looping entry* will be different from those in effect prior to loop processing; i.e., when the derivative calculations for the convergence prediction were made. This observation further invalidates the convergence prediction made prior to loop processing. And while the problem discussed in the preceding paragraph may be overcome by re-structuring the representation of computational loops to facilitate calculation of "composite derivatives,"[18] this second problem represents a fundamental shortcoming of the fixed-point method: the convergence prediction, which must be made prior to computational loop processing, depends on the results of this loop processing. More simply stated, the fixed-point method has the problem of trying to calculate the derivative of a function when the values of that function's variables are not known.[19]

Of course, there are means for overcoming this problem, as well. For instance, the loop processing could take place before the convergence prediction is made. Then the values of all the associated design variables would be available, and the ("composite") derivative calculation could follow rather straightforwardly. However, this delayed derivative calculation could hardly be called a convergence *prediction*. Indeed, if loop processing is to precede any kind of divergence prevention, the derivative calculation isn't really necessary: a simple *observation* of whether or not the values of the design variables are converging may be made. Based on this observation, the computational loop would be rearranged, if necessary. In addition, restoration of the values of the design variables to the values which the design variables had prior to the divergent iteration(s) may also be in order, since it is unlikely that the values calculated during divergent iterations will be sufficiently close to the solution values for the fixed-point method to succeed.

However, it would be preferable to have a method for solving computational loops which would not need to rely on backtracking and possibly wasteful computations. For this reason,

[18]Indeed, one such restructuring is presented in Chapter 3.

[19]This problem presents a major difficulty in the implementation presented here, where derivatives for just two design functions are needed. If, as suggested, the derivative of a "composite function" were required, the seriousness of this problem would be greatly compounded.

investigation of the fixed-point method was dropped in favor of research into other techniques. The next method to be investigated, the so-called "simultaneous Newton-Raphson" method, will be discussed in the next chapter.

### 2.6.3 Interactions between Multiple Loops

Investigation of the fixed-point method was dropped due to the problems in establishing a reliable means for predicting convergence of the calculated design variable values (as discussed in the preceding section). Other problems with this implementation of the fixed-point method were also anticipated, concerning the possible interactions between multiple loops. The following discussion of these anticipated problems is included here primarily for historical completeness, and also because the problems connected with the possible need for multiple loops will also have some relevance to the discussion of other methods to follow in subsequent chapters.

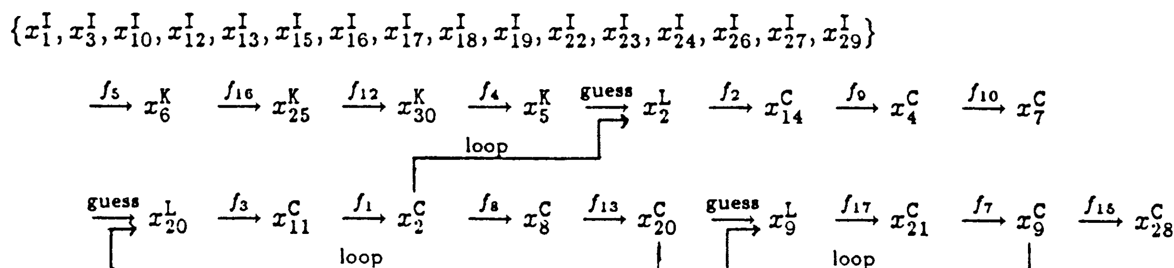
In the implementation of the fixed-point method described here, there is a definite order in which the individual loops must be processed, since, when building the agenda, decisions concerning construction of a new loop (such as loop variable selection) depend upon what entries have already been included in the agenda. Thus, in considering a specific loop, it is appropriate to speak of **outer loops**, which precede the given loop in the agenda, and **inner loops**, which follow the given loop in the agenda. To achieve convergence of a single loop (as discussed above), the entry variables of certain entries should be exchanged. However, such an exchange within a single loop is only feasible when the loop-closing function for that loop—i.e., the function used to compute a value for the loop-variable of that loop—is an entry within that loop (and does not, therefore, occur in some inner loop of that loop). A loop which contains its own loop-closing function is said to be a **nested loop**. Those loops which do not contain their own loop-closing functions are referred to as **tangled loops**,[20] since trying to rearrange such a loop may require at least partial re-ordering of those loops which are inner loops of the tangled loop.

To illustrate these concepts of nested and tangled loops, a second, larger example design set is introduced. This design set has seventeen design functions and thirty design variables, sixteen of which have been initialized. There are thus fourteen free variables, indicating that, as long as there are no local incompatibilities (see the footnote on page 27), there will be three overconstraining functions. Without going through any further analysis, Figure 2.3 is presented as the pictorial representation of an agenda for such a design set. (Note in this

[20] Construction of such tangled loops is explicitly allowed in item 3(b) of the description of the loop-building heuristics on page 23.

figure that the loop headers include an indication of whether the loop is nested or tangled.)

The agenda in Figure 2.3 has three computational loops, for the *loop variables* $x_2$, $x_{20}$, and $x_9$; each loop header has a pointer to the corresponding closing entry. The $x_2$ loop is the outermost loop, and is tangled, since the closing entry for this $x_2$ loop actually occurs in the $x_{20}$ loop. The two inner loops of this $x_2$ loop are both nested loops, since their closing entries occur within their own loops. Following the treatment on page 27, the sequence of operations comprising this loop may be written as




Because the computational scheme suggested here assumes values for the *loop variables*, calculates new values for the remaining variables, and then replaces the assumed values, it follows that divergence of the values of the *loop variables* will typically cause divergence of the values of the other variables computed as part of the divergent loops. It is suspected that by examining the set of all variables which exhibit divergence (both *loop variables* and others), it should be possible to determine which parts of the computational path are unstable. Once the unstable loops are identified, an appropriate rearrangement may be effected, though further research would be required to determine the correct rearrangement for various types of multiple-loop interactions.[21]

Finally, a pair of heuristic guidelines is proposed for analyzing unstable computational paths when multiple loops are present, and they follow directly from the implementation described above:

- When divergent entries are detected among nested loops, it is the outermost loop containing divergent entries which is probably responsible for the instability.
- When divergent entries are detected among tangled loops, it is the innermost loop containing divergent entries which is likely to be responsible for the instability.

These rules follow from the fact that divergence of the *loop variables* is transmitted to the other variables in the computational path, and it is upon computation of values for these

[21] As yet, only nested loops have been examined in any detail. It is likely that rearrangement of a tangled loop would require modifications to its inner loops, with these modifications perhaps depending on the nature of the inner loops (i.e., whether the inner loops are themselves nested or tangled).

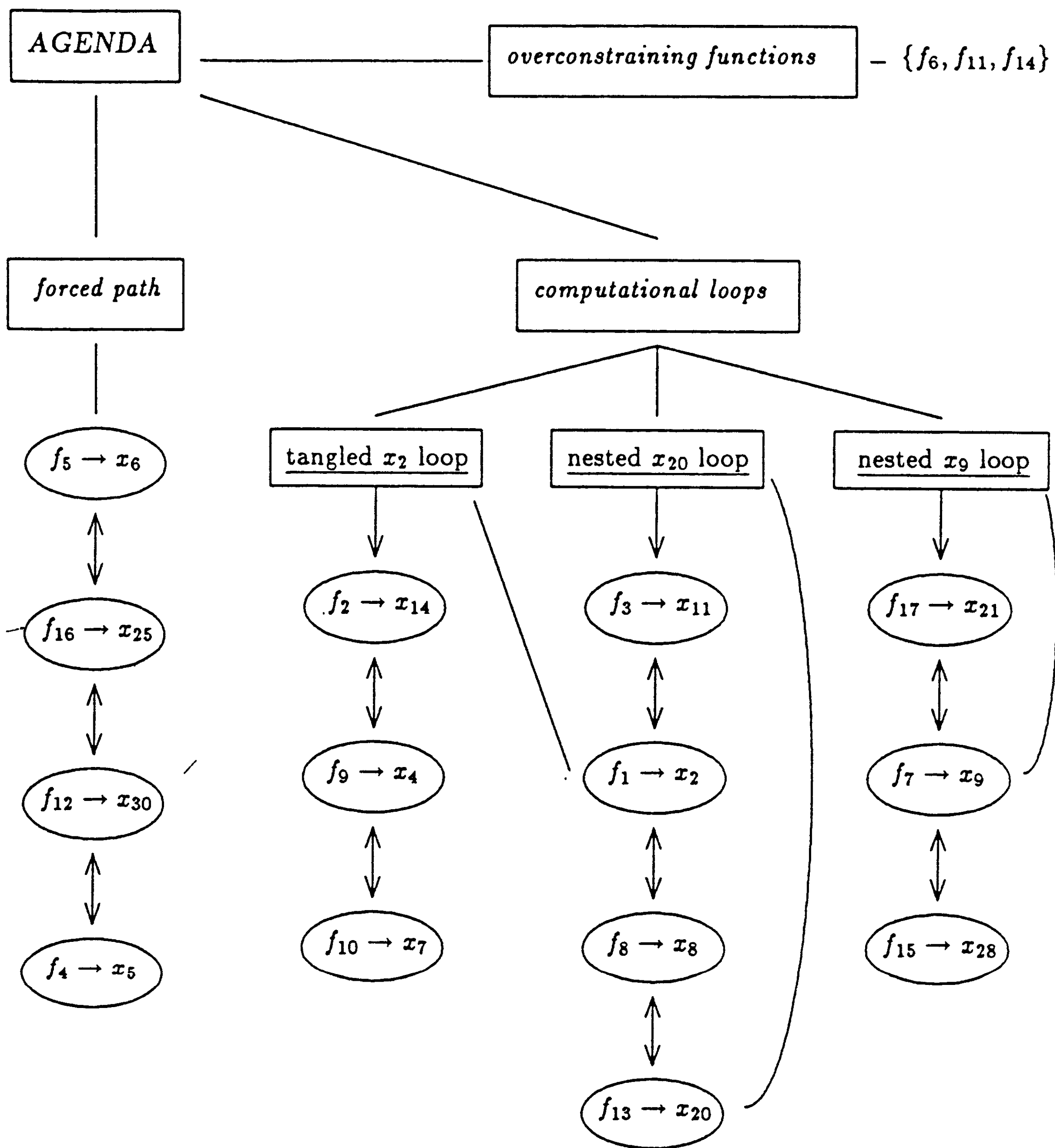



Figure 2.3: A computational agenda structure with nested and tangled loops.

*loop variables* that divergence is introduced. When the loops are nested, divergence of inner loops is inherited from the loop-closing function of the outermost unstable loop.[22] Thus, this outermost unstable loop is likely to be responsible for most (if not all) of the instabilities found in inner nested loops. When the loops are tangled, the loop-closing function of a given loop will occur in an inner loop of that particular loop, and thus divergence is transmitted outwards. In the case of tangled loops, divergence will be transmitted inwards, as well, since processing the entries of inner loops will use the diverging values calculated in outer loops. The case of tangled loops is rather more complicated then, and the heuristic proposed for this case is less trustworthy.

[22] Where "unstable loop" refers to any loop containing divergent entries.

# Chapter 3

# The Simultaneous Newton-Raphson Approach

## 3.1 The Simultaneous Newton-Raphson Method

In Section 2.6.2 of the preceding chapter, it was indicated that one of the primary problems with the fixed-point approach was that it did not take into account the composite nature of the two simultaneous equations to be solved, these equations being based on a chain of design functions in order to calculate consistent values for both the *loop variable* and the *looping variable*. When it was decided that a modified representation for the structure of computational loops was needed in order to reflect this "composite nature," it was also deemed appropriate to abandon the fixed-point approach (due to its other problems) in favor of a new method.

This new technique, an adaptation of the Newton-Raphson method described in Section 1.3.2, is the "simultaneous Newton-Raphson" method, in which assumed values for certain of the derived variables are used to compute *two* calculated values for other derived variables by *two different* computational chains of design functions; unless these two calculated values are sufficiently close (in which case a solution has been found), "composite derivatives" are then calculated in order to extend tangent lines, which are used to update the assumed values. A graphical representation of this procedure is presented in Figure 3.1.

To elaborate, consider, as suggested in Figure 3.1, two design functions, $f$ and $g$, relating the two design variables, $x$ and $y$, neither of which are base variables for the design set which includes these design functions and design variables, such that

$$
\begin{aligned}
y &= f(x) \\
y &= g(x)
\end{aligned}
$$

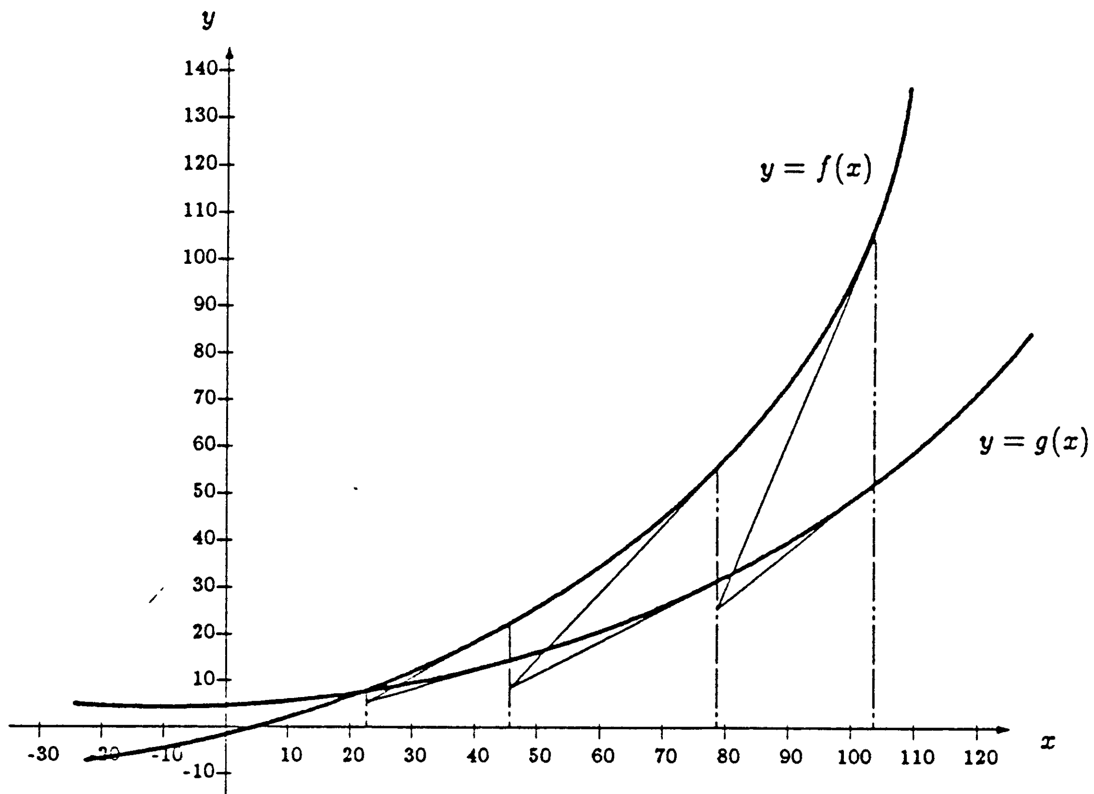



Figure 3.1: The simultaneous Newton-Raphson method for locating intersections.

In a manner similar to the normal Newton-Raphson method (see page 7), the simultaneous Newton-Raphson method, as applied to these two equations in two unknowns, consists of

1. Choosing an initial value for one of the two variables, in this case, $x$, referred to as $x_i$, which is presumed to be near one of the solutions to the pair of simultaneous equations. The variable $x$ is referred to as the *forcing variable*, since selecting a value for this variable will force the other unknown variable to take on values specified by the two governing equations.

2. Computing $f(x_i)$ and $g(x_i)$. These are the two values for $y$ which follow from choosing a value for $x$, the *forcing variable*. The variable $y$ is referred to as the *loop variable*, since it is the role of the computational loop to find a value for the *forcing variable* which yields two equivalent values for the *loop variable*.

3. If $f(x_i) \approx g(x_i)$, then $x_i$ is the solution value for $x$, and $f(x_i)$ (or $g(x_i)$) is the solution value for $y$.

4. If $f(x_i) \neq g(x_i)$, then $f'(x_i)$ and $g'(x_i)$ are calculated, and a new value for $x$ is chosen, $x_{i+1}$, according to where the tangents to $f(x)$ and $g(x)$ at $x_i$ intersect, i.e.,

$$x_{i+1} = x_i - \frac{f(x_i) - g(x_i)}{f'(x_i) - g'(x_i)}$$

and the process is repeated.

An alternative view of this procedure is provided by considering the simultaneous Newton-Raphson method as no more than an application of the normal Newton-Raphson method for finding a zero of the difference between the two simultaneous equations:

$$\begin{array}{rrcl} & y & = & f(x) \\ - & y & = & g(x) \\ \hline & 0 & = & f(x) - g(x) \end{array}$$

implying that the true nature of the simultaneous Newton-Raphson method is use of the normal Newton-Raphson method to find zeroes of the function

$$H(x) = f(x) - g(x)$$

In applying the above procedure to a set of design functions and design variables, it is necessary to identify the *forcing variable*, the *loop variable*, and the two chains of design functions which allow values to be calculated for the *loop variable* based on an assumed value for the *forcing variable*. Discussion of how this may be done is deferred until Section 3.2.2; the next

section will present the modifications to the computational agenda structures described in Sections 2.3-2.4 necessary for implementation of the simultaneous Newton-Raphson method.

Finally, it should be noted that the simultaneous Newton-Raphson method, does not suffer from the problem associated with the fixed-point method of needing to overcome possible divergence of the computed values. As long as the initial assumed value for the *forcing variable* is near the solution value for the *forcing variable* and the two "composite functions" are approximately linear with the *forcing variable* in the neighborhood of the solution point, the procedure by which the value of the *forcing variable* is updated should always yield a value for the *forcing variable* which is successively closer to the solution value.

## 3.2 Implementation of the Simultaneous Newton-Raphson Method

### 3.2.1 Changes to the Computational Agenda Loop Header Structure

Only the computational loop solution procedure is affected by use of the simultaneous Newton-Raphson method instead of the fixed-point method. Thus, only the representation of computational loops, as described in Chapter 2, will require modification. The computational agenda, itself, still consists of a fixed path, a set of computational loops, and a list of overconstraining functions. The role of agenda entries is also unchanged, and therefore the agenda entry representation described in Section 2.4 remains valid.[1] However, the loop header structure (see Section 2.4.2) must be modified to reflect the requirements of the simultaneous Newton-Raphson method. The listing of the loop header components for the fixed-point method is given on page 31; for the simultaneous Newton-Raphson method, the appropriate components for the loop header structure are:

*forcing variable:* the design variable chosen as the *forcing variable* for the loop which follows the header; selecting a value for this variable will provide two different means of computing a value for the *loop variable* (see below).

*other variables:* a list of those design variables which also satisfied the criterion for being chosen as the *forcing variable* at the time the *forcing variable* was actually chosen. (I.e., this attribute records other equally good choices available at the time of *forcing variable* selection.)

*loop variable:* the design variable chosen as the *loop variable* for the loop which follows

[1]It should be pointed out, though, that the "entry divergence flag" attribute of agenda entries (see Section 2.4.1) is no longer needed.

the header; this design variable will also be the entry variable of the final entries of both the *first branch* and the *second branch* (see below).

*preliminary entries:* this attribute points to the first entry in the chain of agenda entries which represents the initial, shared portion of the two different paths which compute values for the *loop variable.*

*first branch:* this attribute points to the first entry in the chain of agenda entries which represents the final, unshared portion of the first of the two paths which compute values for the *loop variable.*

*second branch:* this attribute points to the first entry in the chain of agenda entries which represents the final, unshared portion of the second of the two paths which compute values for the *loop variable.*

*final entries:* this attribute points to the first entry in the chain of agenda entries which will be perfectly constrained *after* consistent values have been established for the *forcing variable* and the *loop variable.*

As indicated in Section 3.1, the simultaneous Newton-Raphson method is basically a procedure for finding a value for the *forcing variable* which produces the same value for the *loop variable* by two different computational paths. The loop header structure described above is intended to reflect this procedure, by providing explicit representation of these two computational paths. First, allowance is made for the two paths to share some common set of calculations under the *preliminary entries* attribute. This permits both paths to include some set of calculations which, based on an assumed value for the *forcing variable*, compute values for certain intermediate variables, values for which are required by both paths in order to calculate a value for the *loop variable.* Then, of course, there are *first branch* and *second branch* attributes for accessing the independent steps in the two paths. Finally, once the loop calculations are completed and consistent values have been found for the *forcing variable*, the *loop variable*, and any intermediate variables, some of the remaining (i.e., unused) design functions may become perfectly constrained, allowing their use for the calculation of any remaining unknown design variables. Since any such calculations will depend upon successful completion of the preceding loop calculations, these entries[2] are included as part of the computational loop, under the *final entries* attribute.

[2]Which represent a kind of "forced path" conditional upon the loop calculations ...

### 3.2.2 Changes to Computational Agenda Loop Construction

Based on the changes in the representation of computational loops described in the preceding section, the algorithms by which computational loop structures are built must also be altered. As suggested in Section 3.2.1, switching from the fixed-point approach to the simultaneous Newton-Raphson approach towards loop processing will have no effect on the representation of the fixed path and the overconstraining functions, and it will also be the case that switching methods has no effect on the construction of the fixed path or the detection of overconstraining functions. Thus, the comments made in Sections 2.3.1 and 2.3.3 with respect to use of the fixed-point method are also applicable to use of the simultaneous Newton-Raphson method. The only aspect of computational agenda construction which will differ from those discussed in Chapter 2, then, is computational loop construction.

In constructing computational loops for application of the simultaneous Newton-Raphson method, an appropriate means must be found for choosing a *forcing variable* and a *loop variable*, and for determining the two computational paths to which the method will be applied. The algorithm ultimately developed for constructing computational loops for the simultaneous Newton-Raphson method is an adaptation of the algorithm established for use with the fixed-point method, as presented on page 23. The modified algorithm follows these steps:

1. Choose a forcing variable by determining which G-variable occurs most often in the functions which have yet to be assigned to the agenda.

2. Identify those design functions which are perfectly constrained (based upon the assumed value of the forcing variable), and assign them to the *preliminary entries*.

3. Continue adding *preliminary entries* until one of the following conditions are met:

   (a) a design function, referred to as a *forced function*, is found for which values have been calculated for all the associated variables except for one, which happens to be the forcing variable, in which case loop construction should proceed to step 4 for construction of the two branches;

   (b) no more perfectly constrained functions are found, in which case an error has occurred, since loop construction cannot be completed (no branches may be constructed);

   (c) only overconstraining functions remain to be added to the agenda, in which case loop construction terminates in an error, as in step 3(b); or

(d) there are no functions left to add to the agenda, in which case loop construction terminates in an error, as in step 3(b).

4. As long as loop construction has not terminated in an error, loop construction continues with construction of the two branches. At this point, a chain of *preliminary entries* has been constructed, and a *forced function* has been found. The two branches are constructed as follows:

   (a) The *preliminary entries* are searched backwards for an entry which computes a value for one of the C-variables associated with the *forced function.* This entry is referred to as the *forcing entry.*

   (b) The entry variable of the *forcing entry* is chosen as the *loop variable.*

   (c) The *forcing entry* is transferred from the *preliminary entries* to the *first branch.*

   (d) All the agenda entries which followed the *forcing entry* in the *preliminary entries* are transferred to the beginning of the *second branch.*

   (e) An entry using the *forced function* to compute a value for the *loop variable* is added to the end of the *second branch.* (Thus, both branches will end with an entry which calculates a value for the *loop variable.*)

   (f) Finally, both *branches* and the chain of *preliminary entries* are searched in a backwards manner for any entries which calculate variables for which values are *not* needed to calculate values for the *loop variable.* Any such entries are transferred to the beginning of the chain of *final entries.*

5. Again, as long as loop construction has not terminated in an error, loop construction is completed by adding any remaining perfectly constrained design functions to the end of the chain of *final entries* for the loop. When the supply of perfectly constrained design functions is exhausted, one of the following steps is taken:

   (a) if no design functions remain, agenda construction is terminated;

   (b) if only overconstrained design functions remain, loop construction is terminated; or, under any other circumstances,

   (c) a new loop is begun.

The procedure described here basically entails creating a computational loop structure according to the algorithm developed for the fixed-point approach (under the *preliminary entries* attribute of the loop header), and then modifying the loop for application of the simultaneous Newton-Raphson method.

For the purpose of illustration, recall the example design set introduced in Chapter 2. After construction of the forced path, the design set looked like this (see page 22)

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{G}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{G}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{G}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}
$$

There are four design functions yet to be included in the computational agenda, and three G-variables remain. As in Section 2.3.2, it is observed that design variable $x_5$ occurs in all four of the remaining design functions, $x_6$ occurs in three, and $x_7$ occurs in just two. The simultaneous Newton-Raphson *forcing variable* is chosen according to the same criterion as the fixed-point *loop variable* (for exactly the same reasons, see page 23), and therefore $x_5$ is chosen as the first *forcing variable*. Its state is changed from "G" to "F" ("forcing") to reflect its new role, and the design set may thus be written as

$$
\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{F}} &= f_3(x_3^{\mathrm{I}}, x_6^{\mathrm{G}}) \\
x_6^{\mathrm{G}} &= f_4(x_5^{\mathrm{F}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{G}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}
$$

Now that the *forcing variable* has been chosen, actual construction of the computational loop may begin. With the change in state of design variable $x_5$, it is now the case that there is just one G-variable associated with each of the design functions $f_3$, $f_4$, and $f_6$; any one of these three design functions may now be used to commence the *preliminary entries* for the loop. As indicated in Section 2.3.2 on page 25, only one design function may be added to the agenda at a time, in order to avoid incompatibilities. Following the treatment in Section 2.3.2, $f_3$ is chosen for addition to the computational agenda at this stage, and the design set may be

written as

$$
\begin{aligned}
x_3^{\mathtt{I}} &= f_1^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}) \\
x_4^{\mathtt{K}} &= f_2^{\mathtt{U}}(x_1^{\mathtt{K}}, x_2^{\mathtt{I}}, x_3^{\mathtt{I}}) \\
x_5^{\mathtt{F}} &= f_3^{\mathtt{U}}(x_3^{\mathtt{I}}, x_6^{\mathtt{C}}) \\
x_6^{\mathtt{C}} &= f_4(x_5^{\mathtt{F}}, x_1^{\mathtt{K}}) \\
x_6^{\mathtt{C}} &= f_5(x_2^{\mathtt{I}}, x_5^{\mathtt{F}}, x_7^{\mathtt{G}}) \\
x_7^{\mathtt{G}} &= f_6(x_2^{\mathtt{I}}, x_5^{\mathtt{F}}) \\
x_8^{\mathtt{K}} &= f_7^{\mathtt{U}}(x_3^{\mathtt{I}}, x_4^{\mathtt{K}})
\end{aligned}
$$

where design function $f_3$ has been marked as "used," and the state of design variable $x_6$ has changed from "G" to "C" to reflect inclusion in the agenda of an entry for calculating the value of $x_6$ as part of the computational loop under construction.

The result of this last step, however, is that design function $f_4$ is now "overconstrained": it has yet to be added to the agenda, but it has no G-variables left. However, one of the variables associated with this design function happens to be the *forcing variable*, $x_5$, which means that a *forced function* has been detected. Searching backwards through the chain of *preliminary entries* reveals that the entry which was just constructed for computing a value for design variable $x_6$ using design function $f_3$ qualifies as a *forcing entry*,[3] which means that loop construction may continue by building the two *branches*, and need not terminate in an error.

Continuing to follow the algorithm outlined on page 46, $x_6$ is identified as the *loop variable*, and the next step is to transfer the *forcing entry*—the entry using $f_3$ to compute $x_6$—from the *preliminary entries* to the *first branch*. Since no entries follow this entry in the chain of *preliminary entries*, no entries need be transferred to the beginning of the *second branch*. An entry using the *forced function*, $f_4$, to calculate a value for the *loop variable*, $x_6$, is constructed and assigned to the *second branch*. Thus, two means have been established for computing the *loop variable* based on an assumed value for the *forcing variable*. The design set may now

[3]Which is fortunate, since, in this case, this happens to be the *only* entry in the chain of *preliminary entries*!

be written as

$$\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{F}} &= f_3^{\mathrm{U}}(x_3^{\mathrm{I}}, x_6^{\mathrm{L}}) \\
x_6^{\mathrm{L}} &= f_4^{\mathrm{U}}(x_5^{\mathrm{F}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{L}} &= f_5(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}, x_7^{\mathrm{G}}) \\
x_7^{\mathrm{G}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}$$

where design function $f_4$ has been marked as "used," and the state of design variable $x_6$ has been updated from "C" to "L" ("loop") to reflect its new role as *loop variable*.[4]

Two design functions remain, $f_5$ and $f_6$, and both have one G-variable, which happens to be $x_7$ for *both* design functions. Following the treatment in Section 2.3.2 on page 26, $f_5$ is chosen for inclusion in the agenda at this stage. An entry is constructed for computing $x_7$ using design function $f_5$, and this entry is assigned to the *final entries* attribute of the loop header associated with this loop. The design set may now be written as

$$\begin{aligned}
x_3^{\mathrm{I}} &= f_1^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}) \\
x_4^{\mathrm{K}} &= f_2^{\mathrm{U}}(x_1^{\mathrm{K}}, x_2^{\mathrm{I}}, x_3^{\mathrm{I}}) \\
x_5^{\mathrm{F}} &= f_3^{\mathrm{U}}(x_3^{\mathrm{I}}, x_6^{\mathrm{L}}) \\
x_6^{\mathrm{L}} &= f_4^{\mathrm{U}}(x_5^{\mathrm{F}}, x_1^{\mathrm{K}}) \\
x_6^{\mathrm{L}} &= f_5^{\mathrm{U}}(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}, x_7^{\mathrm{C}}) \\
x_7^{\mathrm{C}} &= f_6(x_2^{\mathrm{I}}, x_5^{\mathrm{F}}) \\
x_8^{\mathrm{K}} &= f_7^{\mathrm{U}}(x_3^{\mathrm{I}}, x_4^{\mathrm{K}})
\end{aligned}$$

where $f_5$ has been marked as "used" and $x_7$ has been marked as "computed." Following the treatment on page 27, the sequence of calculations represented by this computational agenda is

$$\{x_2^{\mathrm{I}}, x_3^{\mathrm{I}}\} \xrightarrow{f_1} x_1^{\mathrm{K}} \xrightarrow{f_2} x_4^{\mathrm{K}} \xrightarrow{f_7} x_8^{\mathrm{K}} \xrightarrow{\text{guess}} x_5^{\mathrm{L}} \begin{array}{c} \nearrow \xrightarrow{f_3} x_6^{\mathrm{C}} \searrow \\ \searrow \xrightarrow{f_4} x_6^{\mathrm{C}} \nearrow \end{array} \xrightarrow{f_5} x_7^{\mathrm{C}}$$

In this example, the only function which has yet to be added to the agenda is an overconstrained function; loop construction terminates as indicated in step 5(b) of the loop-building

[4]Note that due to the small size of this design set, it turns out that there are no extraneous entries to be transferred from either the *branches* or the *preliminary entries* to the *final entries*, as per step 4(f) in the simultaneous Newton-Raphson loop-building algorithm presented above (see page 47).

algorithm presented above (see page 47). Agenda construction concludes with addition of design function $f_6$ to the agenda's list of overconstraining functions (see Section 2.3.3). A graphical depiction of the completed agenda is presented in Figure 3.2.

### 3.2.3 Execution of the (Revised) Computational Agenda using the Simultaneous Newton-Raphson Method

Once the appropriate computational agenda structure has been constructed, application of the simultaneous Newton-Raphson method to a single computational loop proceeds as follows:

1. Select an initial value for the *forcing variable.*

2. Process the chain of *preliminary entries* based on the selected value of the *forcing variable.*

3. Calculate two values for the *loop variable* by propagating the results of the *preliminary entries* calculations down the two *branches.*

4. If these two values are sufficiently close to one another, consistent values for the *forcing variable* and the *loop variable* have been determined; a solution has been found and processing of the computational loop is completed by evaluating the *final entries.*

5. If these two values are not sufficiently close to one another, a new value for the *forcing variable* is calculated, according to the formula indicated in step 4 of the outline of the simultaneous Newton-Raphson method given in Section 3.1 (see page 43). Application of this formula requires calculation of the derivatives of the two chains of functions used to compute values for the *loop variable* given a value for the *forcing variable*; these derivatives may be calculated as follows:

   (a) Let $x_f$ be the current test value of the *forcing variable*, and let $y_{lb1}(x_f)$ and $y_{lb2}(x_f)$ be the values calculated for the *loop variable* by propagating this value, $x_f$, down the chain of *preliminary entries* and the *first* and *second branches*, respectively.

   (b) Select a $\Delta x_f$ which is some small fraction of the value, $x_f$.

   (c) Then $y_{lb1}(x_f + \Delta x_f)$ is the value calculated for the *loop variable* following the *first branch* path when $x_f + \Delta x_f$ is chosen as the value of the *forcing variable*, just as, say, $y_{lb2}(x_f - \Delta x_f)$ is the value calculated for the *loop variable* when $x_f - \Delta x_f$ is chosen as the value of the *forcing variable.*

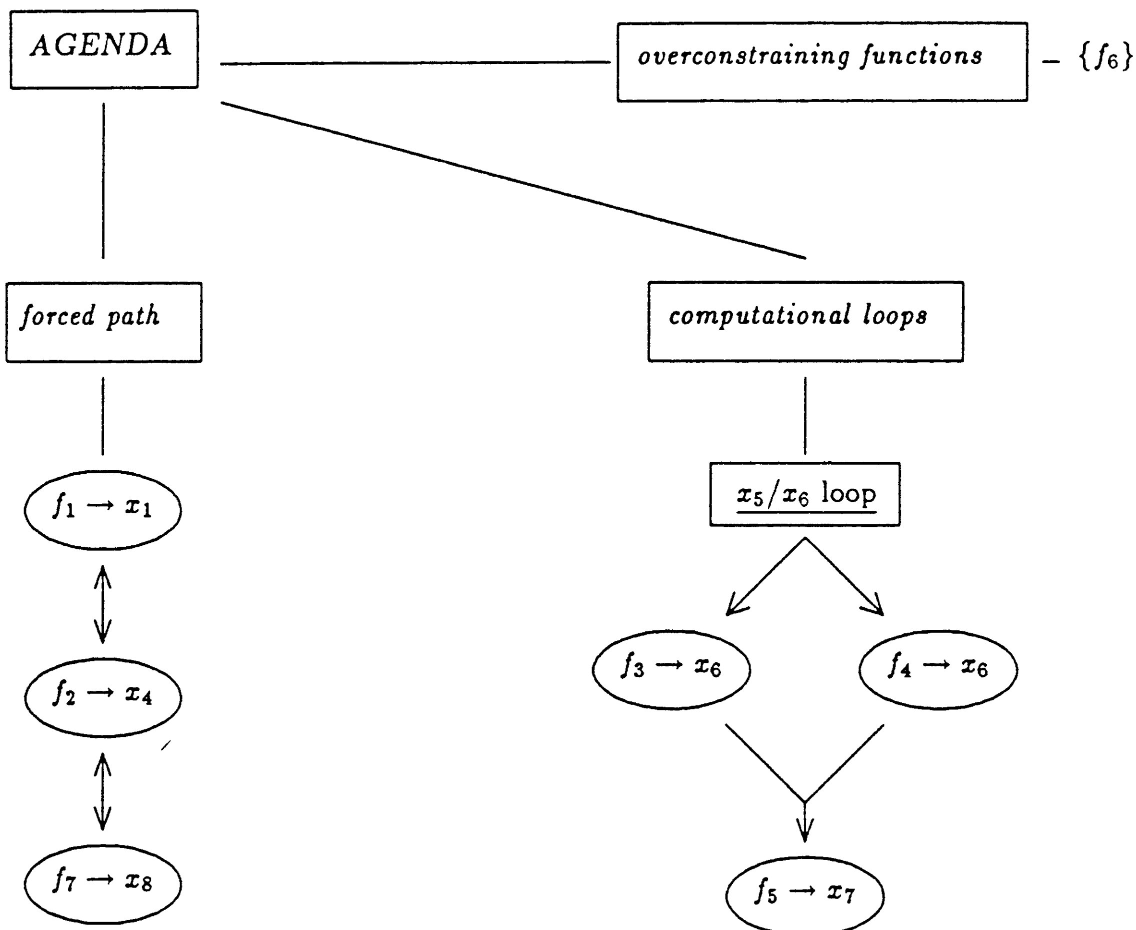



Figure 3.2: The computational agenda structure for the example design set, modified for application of the simultaneous Newton-Raphson method.

(d) The derivative[5] of the *loop variable* with respect to the *forcing variable* following the *first branch* computational path, is, then

$$\left.\frac{d(\text{loop variable})}{d(\text{forcing variable})}\right|_{\text{b1}} = \frac{y_{l\text{b1}}(x_f + \Delta x_f) - y_{l\text{b1}}(x_f - \Delta x_f)}{2\Delta x_f}$$

(e) And the derivative of the *loop variable* with respect to the *forcing variable* following the *second branch* computational path, is, then

$$\left.\frac{d(\text{loop variable})}{d(\text{forcing variable})}\right|_{\text{b2}} = \frac{y_{l\text{b2}}(x_f + \Delta x_f) - y_{l\text{b2}}(x_f - \Delta x_f)}{2\Delta x_f}$$

Based on these derivatives, a new value for the *loop variable* is calculated; steps 2-5 are repeated until consistent values for the *forcing variable* and the *loop variable* are determined.

In this manner, the simultaneous Newton-Raphson method is sequentially applied to each of the computational loops associated with a given computational path.

It is appropriate in this discussion of the execution of the simultaneous Newton-Raphson method to point out a small savings in efficiency provided by use of these procedures. Note that in step 4(f) of the loop-building algorithm described on page 47, any entries which are not required for computation of the *loop variable* are delayed by transferring them to the chain of *final entries*. This speeds the iterative process described in the above procedure, since these entries need only be evaluated *once*: that is, as part of the *final entries*, which are only evaluated once (step 4), instead of as part of either the *preliminary entries* or one of the *branches*, whose entries are evaluated repeatedly (during iterative processing *and* in the calculation of derivatives). The implementation of the fixed-point method described in the preceding chapter required iterative processing over *all* of the entries associated with a computational loop, for the simple reason that the implementation, as suggested earlier, failed to recognize—and therefore take advantage of—the way in which the design functions *combined* to represent the simultaneous equations to be solved.

## 3.3 Problems Associated with the Simultaneous Newton-Raphson Method

As with the fixed-point method, it was observed that, except for design sets using very simple (i.e., linear) design functions, the simultaneous Newton-Raphson method typically

[5] This numerical approximation to the "composite derivative" is based on the second-order numerical approximation discussed in the footnote on page 16.

failed to find the solution point for the derived variables. In this case, however, instead of displaying divergence of the calculated values (as with the fixed-point results), the simultaneous Newton-Raphson method failed by updating the *forcing variable* to values which were outside the range of validity of the design functions which were being used. For instance, design functions which included square root calculations were forced to calculate the square root of a negative number, requiring subsequent entries to use imaginary values where real values were expected. Similarly, reverse solution of design functions (i.e., using the Newton-Raphson method to numerically invert design functions, see Section 1.3.2) was often unable to compute a value for a design function input variable due to the extreme values already assigned to the design function's other associated variables.

It appears that inappropriate selection of initial seed values was responsible for the poor results which were obtained by using the simultaneous Newton-Raphson method. Initial values for the *forcing variable* were chosen by examining a list of points distributed logarithmically between the *forcing variable*'s upper and lower values, selecting the value for which the two corresponding calculated values for the *loop variable* were closest. As indicated on page 44 of Section 3.1, however, the simultaneous Newton-Raphson method requires that this initial value be in some neighborhood around the solution point within which both *branches* exhibit approximately linear behavior. Clearly, because of the rather loose restrictions on design function eligibility used here (see page 6 of Section 1.3.2), the seed value selection procedure just described is inadequate. Chaining a set of potentially non-linear design functions together to create a two-branch computational loop is likely to result in two highly non-linear "composite functions": a combination of non-linear functions must be assumed to be at least as non-linear as its component functions. For the seed value to be in some locally linear region of *both* of the "composite functions," and for this region to also include the solution values, would seem to require a much more refined selection process.

One obvious modification to the selection procedure described above is to search logarithmic distributions between successively closer upper and lower limits until a sufficiently "good" seed value has been found. However, if it is necessary to wait until these limits are *very* close together before the simultaneous Newton-Raphson method may be applied, it hardly seems necessary to use the simultaneous Newton-Raphson method at all: the final search for the correct value of the *forcing variable* can also be carried out by such recursive searching of logarithmic distributions. This search technique may not be as "elegant" as the two methods which have already been discussed, but what the method lacks in elegance it certainly makes up for in reliability. For this reason, investigation of the simultaneous Newton-Raphson method was dropped in favor of the so-called "logarithmic distribution"

method, discussed in the next chapter, Chapter 4.

Finally, it should be mentioned that the simultaneous Newton-Raphson method also exhibited difficulty working with design functions which were multi-valued in one or more inverted directions. (Note that this would affect numerical derivative calculation as well as entry evaluation.) Since this problem was also encountered when using the "logarithmic distribution" method, discussion of the problem is deferred until Section 4.2.2.

- blank page -

# Chapter 4

# The Logarithmic Distribution Approach

## 4.1 The Logarithmic Distribution Method

### 4.1.1 Application of the Logarithmic Distribution Method

As suggested at the end of the preceding chapter, the "logarithmic distribution" method consists of recursively searching progressively smaller intervals of values for the *forcing variable* (referred to as "search intervals") until a value is found for which both *branches* return the same value for the *loop variable*. A graphical representation of this procedure is given in Figure 4.1.

The intervals are searched by subdividing them logarithmically (see Section 4.1.2 below). The *forcing variable* search value which returns the closest pair of values for the *loop variable* (referred to as the "best" search value) is used as one of the two interval limits for the next search. The other limit is determined by calculating the derivatives of the two chains of functions used to compute values for the *loop variable* given a value for the *forcing variable* (by the same procedure as described on page 51 for the simultaneous Newton-Raphson method) to determine on which side of this best search value the tangents to the chains of design functions intersect: if the tangents intersect at a value for the *forcing variable* which is less than the best search value, then the search value which immediately preceded the best search value in the search interval is used as the other limit, otherwise the search value which immediately followed the best search value is used.[1]

[1]Note that it is not really necessary to calculate the derivatives in order to determine on which side of the best search value the search should be continued. Instead, a simple observation of the sign of the difference between the two *branch* values may be used to make this decision: the side on which the sign of this difference changes is the side on which the search should be continued. (Such a sign change results from the intersection of the two curves of *branch values*.) Calculation of the derivatives was used in this implementation because

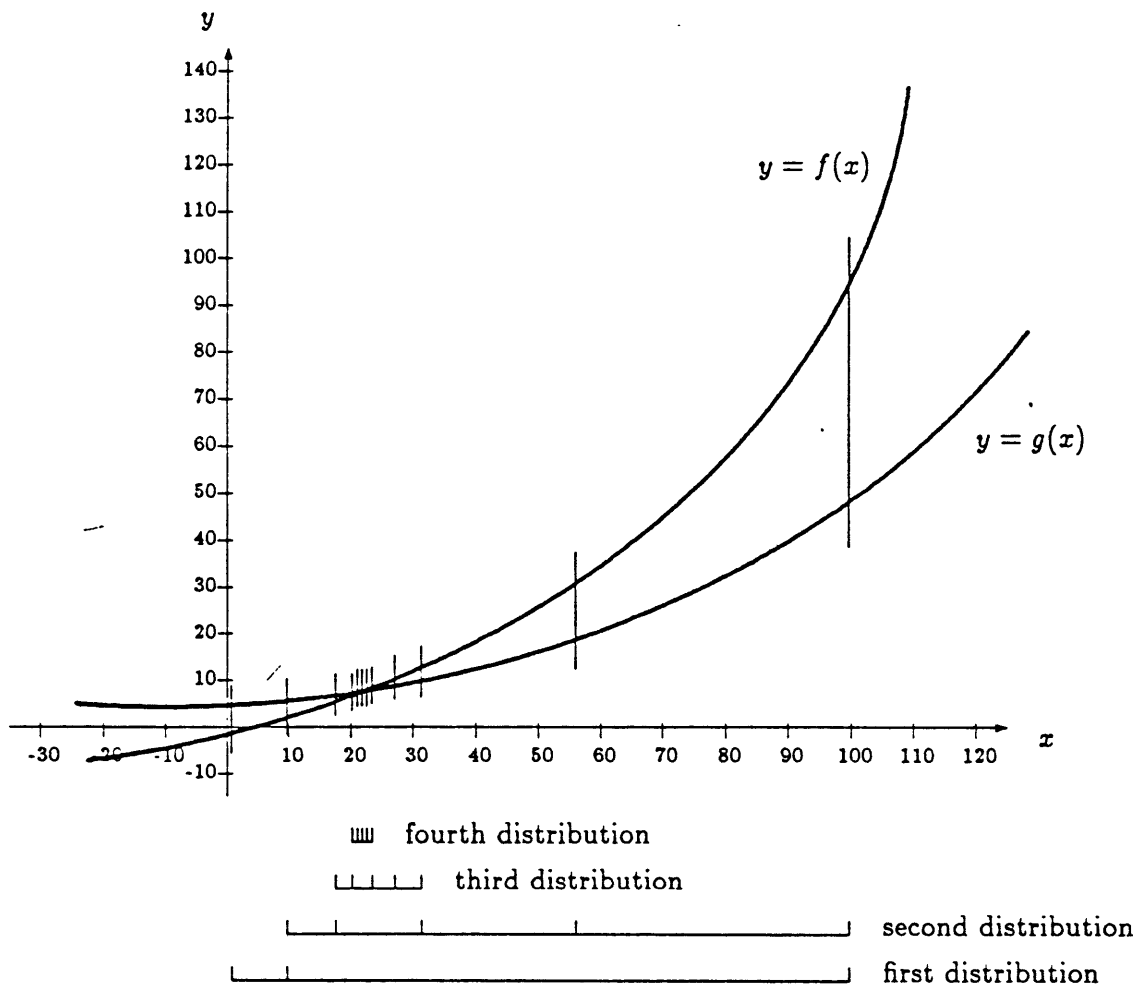



Figure 4.1: The logarithmic distribution method for locating intersections.

The design variable upper and lower values for the *forcing variable* are used as the initial limits on the search interval; subsequent search intervals are chosen by the procedure described in the preceding paragraph. Search continues until one of the following conditions is met:

1. Letting $y_{l_{b1}}$ be the value calculated for the *loop variable* using the *first branch* based on the best search value for the current interval, and $y_{l_{b2}}$ be the value calculated using the *second branch*, a solution has been found when the absolute value of the difference between these two values, divided by the order of magnitude of the *loop variable*, $\mathrm{oom}(y_l)$, is less than some small parameter, $\epsilon$:

$$\left|\frac{y_{l_{b1}} - y_{l_{b2}}}{\mathrm{oom}(y_l)}\right| < \epsilon$$

   In this case, the solution value for the *forcing variable* is the best search value for the current interval, and the solution value for the *loop.variable* is

$$\frac{y_{l_{b1}} + y_{l_{b2}}}{2}$$

2. Letting $x_{f_{\mathrm{upper}}}$ be the value for the *forcing variable* which is the upper limit of the current search interval and $x_{f_{\mathrm{l}}}$ be the value of the lower limit of the current search interval, a solution has been found when the absolute value of the difference between these two values, divided by the order of magnitude of the *forcing variable*, $\mathrm{oom}(x_f)$, is less than some small parameter, $\epsilon$:

$$\left|\frac{x_{f_{\mathrm{upper}}} - x_{f_{\mathrm{l}}}}{\mathrm{oom}(x_f)}\right| < \epsilon$$

   In this case, the solution value for the *forcing variable* is

$$\frac{x_{f_{\mathrm{upper}}} + x_{f_{\mathrm{l}}}}{2}$$

   and the solution value for the *loop variable* is, again,

$$\frac{y_{l_{b1}} + y_{l_{b2}}}{2}$$

3. If the best search value happens to be at one of the endpoints of the current search interval, and if extension of the tangents from this best search value indicates that the other limit for the next search interval should be located outside of the current search

routines for making this calculation were readily available from previous investigation of the simultaneous Newton-Raphson method.

interval, then the current best search value is used as the seed value for application of the simultaneous Newton-Raphson method.[2]

It is by this means that consistent values for the *forcing variable* and the *loop variable* may be determined by application of the logarithmic distribution method.

### 4.1.2 Logarithmic Distribution over an Interval

To logarithmically divide an interval between two limits, $a_l$ and $a_u$, into $n$ subdivisions, $a_l$ is successively multiplied by the factor,

$$e^{\frac{(\log a_u - \log a_l)}{n}}$$

until the value of $a_u$ is reached (i.e., $n$ times). Thus, the logarithmic distribution between these two limits may be written

$$\begin{aligned} \mathrm{ld}_n(a_l, a_u) = & \Big\{ a_l e^{\frac{(\log a_u - \log a_l)}{n}},\ a_l e^{\frac{2(\log a_u - \log a_l)}{n}}, \\ & a_l e^{\frac{3(\log a_u - \log a_l)}{n}},\ \ldots, \\ & a_l e^{\frac{(n-1)(\log a_u - \log a_l)}{n}},\ a_u \Big\} \end{aligned}$$

where the $i$th element in the logarithmic distribution is given by

$$a_{i,n} = a_l e^{\frac{i(\log a_u - \log a_l)}{n}}$$

The reason logarithmic distribution was chosen for subdividing the search interval is because, for large intervals, a logarithmic distribution divides the interval according the orders of magnitude included in the interval (i.e., *logarithmically*), while for small intervals, the interval is divided approximately linearly.

To illustrate this point, consider the *forcing variable* search values used in Figure 4.1. In the example presented in this figure, consistent values for $x$ and $y$ are to be calculated by applying the logarithmic distribution method to the pair of simultaneous equations,

$$\begin{aligned} y &= f(x) \\ y &= g(x) \end{aligned}$$

[2]This step is provided primarily for the case in which the *forcing variable*'s upper and lower values were inappropriately chosen, and analysis of the first search interval indicates that the solution value is *outside* this search interval. However, it is likely that the simultaneous Newton-Raphson method will also fail in this case; should this occur, the program notifies the designer/user, suggesting he reevaluate either (1) his assignment of upper and lower values, or (2) his selection of values for the base variables.

(As far as this example is concerned, it doesn't really matter what the functions $f(x)$ and $g(x)$ actually are.) This example uses four-interval logarithmic distributions, and the initial search interval is, rather conveniently, $1 \leq x \leq 10,000$.[3] Thus, the first distribution of values for $x$ which is examined is, of course,

$$\{1,\ 10,\ 100,\ 1,000,\ 10,000\}$$

By examining the slopes of $f(x)$ and $g(x)$, it is determined that the solution value for $x$ lies in the interval $10 \leq x \leq 100$, and so the second distribution of values which is examined is

$$\{10,\ 17.783,\ 31.623,\ 56.234,\ 100\}$$

It is next determined that the solution value for $x$ lies in the interval $17.783 \leq x \leq 31.623$, and therefore the third distribution of values which is examined is

$$\{17.783,\ 20.535,\ 23.714,\ 27.384,\ 31.623\}$$

This distribution is, indeed, approximately linear, as is the final distribution for the interval $20.535 \leq x \leq 23.714$, which is

$$\{20.535,\ 21.287,\ 22.067,\ 22.876,\ 23.714\}$$

Finally, note that the mathematical description of the elements of a logarithmic distribution presented above provides for extension of the logarithmic distribution outside of its given upper and lower limits. As indicated above, the elements of an $n$-interval logarithmic distribution are computed by evaluating the expression

$$a_{i,n} = a_l e^{\frac{i(\log a_u - \log a_l)}{n}}$$

for integral values of $i$ between zero and $n$ (inclusive). By choosing integral values of $i$ which are either less than zero or greater than $n$, the distribution may be extended beyond the prescribed upper and lower limits, $a_l$ and $a_u$. Thus, the reference in the footnote on page 7 to a "twelve-interval logarithmic distribution twice-extended beyond its upper and lower values" is meant to suggest a logarithmic distribution for which $n = 8$ and $i$ varies from $-2$ to 10 (i.e., the distribution is extended twice on each side of the original interval).

In this implementation, extensions to logarithmic distributions were used only in conjunction with the numerical inversion of design functions by means of the Newton-Raphson method. In applying the logarithmic distribution method, it was found that using search

[3] Only a small part of this initial search interval is included in Figure 4.1.

values for the *forcing variable* near either its upper value or lower value generally resulted in what has been termed an "*inversion failure*" (see Section 4.2.1, below), and for this reason it was decided that there would be little value in extending the initial search interval when applying the logarithmic distribution method.

### 4.1.3 Implementation of the Logarithmic Distribution Method

Note that the terminology used above in describing the logarithmic distribution method is the same terminology used in Chapter 3 to describe the simultaneous Newton-Raphson method. This reflects the fact that the logarithmic distribution method developed from investigation of the simultaneous Newton-Raphson method, and is based on the precisely the same representation for computational loops. Thus, the *forcing variable* and the *loop variable* play very similar roles in the two methods. Because of these similarities, the logarithmic distribution method can use the same structures and the same agenda-building algorithms as the simultaneous Newton-Raphson method. The reader is therefore referred to Sections 3.2.1 and 3.2.2 for questions concerning the representation and construction of computational loops for the logarithmic distribution method. Application of the logarithmic distribution method to these computational loop structures is described above, in Section 4.1.1.

## 4.2 Performance of the Logarithmic Distribution Method

### 4.2.1 Extreme Values for the Forcing Variable

In applying the logarithmic distribution method, a wide range of search values is assigned to the *forcing variable* in attempting to find the value which will yield the same value for the *loop variable* in both *branches* of the computational loop. As mentioned in Section 4.1.2, when extreme values—values near either the upper or lower values of the *forcing variable*—are assigned to the *forcing variable*, it is often the case that it is impossible to calculate *any* value for the *loop variable*: in some step which requires numerical inversion of a design function, the Newton-Raphson method fails to return a value for the search variable. Such an occurrence is termed an *inversion failure*.

Section 1.3.2 outlines the mathematical circumstances under which this may happen. However, putting mathematics aside for the moment, it should be pointed out that the occurrence of *inversion failures* is generally associated with attempting to assign some physically unreasonable value to the *forcing variable*. To illustrate this with an example from the design set described in Appendix A, consider a case in which the aircraft payload weight has already been assigned a value of 50,000 lbs. If, in applying the logarithmic distribution

method, a smaller value, say 45,000 lbs, is assigned to the aircraft gross takeoff weight (i.e., the total aircraft weight), then an *inversion failure* is very likely to occur, since, among other things, the cruise fuel fraction will then become negative, perhaps causing either the aircraft range or cruise lift coefficient to become negative, etc.

Since the occurrence of *inversion failures* is to be expected when using the logarithmic distribution method, a means of compensating for this behavior is needed. To this end, the algorithms developed for implementing the logarithmic distribution method were modified to allow for the case when attempting to assign a particular value to the *forcing variable* results in an *inversion failure*: in such an event, the algorithms "forget" that such an attempt was made, and examine only those search values for which *processing errors* were not signalled.

### 4.2.2 Multi-Valued Design Functions

As long as the upper and lower values assigned to design variables are suitably well-chosen, it appears that there is little which could go wrong in applying the logarithmic distribution method, as described in Section 4.1, to a problem in which the numerical solution of a pair of simultaneous—and possibly non-linear—equations is required. However, it must be recalled that the ground rules established in Section 1.3.3 for developing a computational loop solution procedure specifically allow for the possibility that design functions may be multi-valued in some directions; it turns out that the logarithmic distribution method encounters serious difficulties when the inversion of a design function is multi-valued.

To illustrate this point, consider the following design function representation of the Bréguet range equation:[4]

$$x_{\mathrm{cr}} = \left(\frac{1}{SFC}\right) \frac{C_{L\mathrm{cr}}}{C_{D0} + C_{D_{C_L^2}} {C_{L\mathrm{cr}}}^2} M_{\mathrm{cr}} a(h_{\mathrm{cr}}) \log\left(\frac{1}{1-\delta_{\mathrm{f}}}\right)$$

where $x_{\mathrm{cr}}$ is aircraft range, $SFC$ is specific fuel consumption (in cruise), $C_{L\mathrm{cr}}$ is the lift coefficient at cruise, $C_{D0}$ is the zero-lift drag coefficient, $C_{D_{C_L^2}}$ is the lift-square coefficient (from the drag polar), $M_{\mathrm{cr}}$ is the cruise Mach number, $a(h_{\mathrm{cr}})$ is the speed of sound at cruise (which is a function of the cruise altitude, $h_{\mathrm{cr}}$), and $\delta_{\mathrm{f}}$ is the cruise fuel fraction. For the design set described in Appendix A, using the base variables indicated in Section A.3, a computational agenda was developed in which it was necessary to use this design function in an inverted manner. Specifically, this design function was to be used to calculate a value for $C_{L\mathrm{cr}}$ as the *first branch* of a computational loop for which the aircraft gross takeoff weight, $W_{\mathrm{g}}$, was to be the *forcing variable*, and $C_{L\mathrm{cr}}$ was to be the *loop variable* (see Section A.4).

[4]See, for example, Reference [20].

However, it turns out that this version of the Bréguet range equation is multi-valued for $C_{L\mathrm{cr}}$ in this direction; rearranging the above equation indicates that

$$C_{L\mathrm{cr}} = \frac{1}{2C_{D_{C_L^2}} x_{\mathrm{cr}}} \left(\frac{1}{SFC}\right) M_{\mathrm{cr}} a(h_{\mathrm{cr}}) \log\left(\frac{1}{1-\delta_f}\right) \pm \sqrt{\left[\frac{1}{2C_{D_{C_L^2}} x_{\mathrm{cr}}} \left(\frac{1}{SFC}\right) M_{\mathrm{cr}} a(h_{\mathrm{cr}}) \log\left(\frac{1}{1-\delta_f}\right)\right]^2 - 4\frac{C_{D0}}{C_{D_{C_L^2}}}}$$

In terms of the design function inversion method discussed in Section 1.3.2, then, given a value for the range, $R$, there are *two* corresponding values for $C_{L\mathrm{cr}}$, the cruise lift coefficient.

Since the design function inversion procedure will only return one of these two values, and can give no indication that another solution even exists (see Section 1.3.2, page 8), the computational loop solution algorithm must accept whichever value is returned, unquestioningly. Figure 4.2 is a plot of the values for $C_{L\mathrm{cr}}$ (the *loop variable* for this example) which were calculated by the two branches of the computational loop, based on various values for $W_g$, the *forcing variable*.[5] The curve labelled "Branch 1" indicates those values which were computed by the *first branch*, which includes the Bréguet range equation design function described above. Figure 4.2 illustrates the fact that for certain values of $W_g$, one branch of this multi-valued design function was used, while for other values of $W_g$, the other branch of values was used.[6]

The curve in Figure 4.2 labelled "Branch 2" indicates those values which were computed using the *second branch*, in which the value for $C_{L\mathrm{cr}}$, the *loop variable*, was calculated using a design function based on the definition of the lift coefficient, and the requirement that lift balances weight under cruising conditions:

$$C_{L\mathrm{cr}} = \frac{2W_g}{\rho(h_{\mathrm{cr}})S(M_{\mathrm{cr}}a(h_{\mathrm{cr}}))^2}$$

where $\rho(h_{\mathrm{cr}})$ is the atmospheric density at cruise (a function of the cruise altitude), $S$ is the wing area, and $W_g$ is, again, the gross takeoff weight. This particular design function, defined in this manner, did not require inversion, since it was used to compute $C_{L\mathrm{cr}}$, and was therefore single-valued in the direction in which it was used. As seen in Figure 4.2, the loop branch which used this function produced a very smooth distribution of $C_{L\mathrm{cr}}$ vs. $W_g$. Unfortunately, at the value of $W_g$ for which consistent values for $C_{L\mathrm{cr}}$ and $W_g$ could be

[5]The use of the word "deterministic" in the caption of Figure 4.2 will be explain in Section 4.2.3.

[6]Specifically, the lower branch of $C_{L\mathrm{cr}}$ values for the Bréguet range equation design function was used for values of $W_g$ between 148,500 and 155,500 lbs., while the higher branch of $C_{L\mathrm{cr}}$ values was used for all other values of $W_g$ between 145,500 and 160,000 lbs.

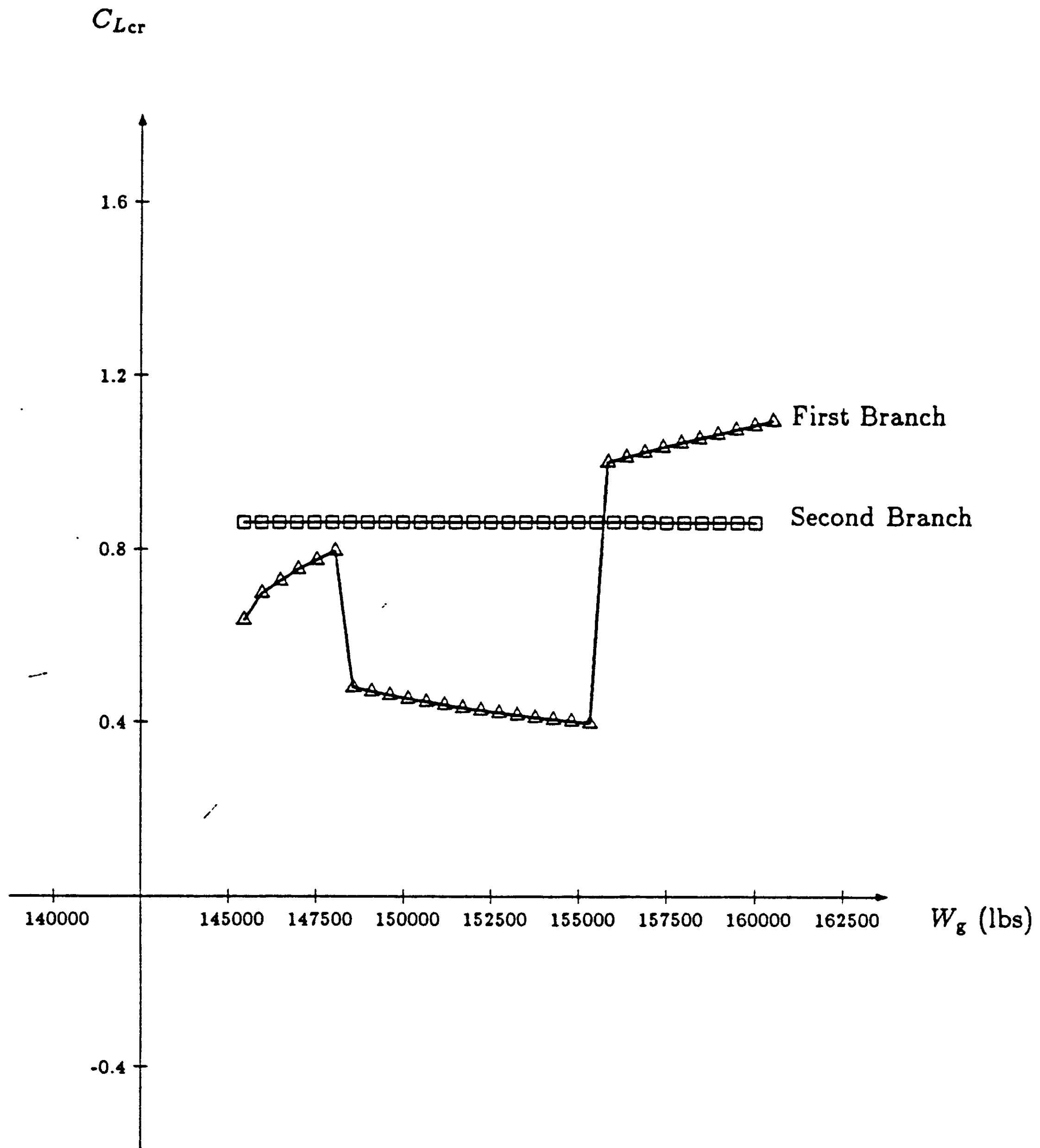



Figure 4.2: Deterministic propagation of forcing values for $W_{\mathrm{g}}$, yielding a pair of computational loop *branch* values for $C_{L\mathrm{cr}}$.

obtained, the value for $C_{L_{cr}}$ returned by the loop's *first branch* was on the incorrect branch of design function values.

Thus, by returning values from the wrong branch of a design function, the logarithmic distribution method can "overlook" the correct solution value for the *forcing variable*, and therefore fail to find consistent values for the *forcing variable* and the *loop variable*.

### 4.2.3 Adapting for Multi-Valued Design Functions

Knowledge that a design function is multi-valued for one of its input variables—i.e., is multi-valued in an inverted direction— can only be derived analytically; there are no systematic numerical techniques for determining such behavior. Since one of the constraints on this research is that only purely numerical tests may be performed on design functions (see page 8 of Section 1.3.3), there is no way to adapt the computational loop solution algorithms to explicitly provide for the case of multi-valued design function inversions. Two measures were taken to modify the algorithms

In the preceding section, it was observed that multi-valued design function behavior is associated with the numerical inversion of design functions, for which the Newton-Raphson method is applied in order to calculate values for design function input variables. As noted in Section 1.3.2, the results of the Newton-Raphson method are highly dependent upon the initial value used to seed the Newton-Raphson iterative search. As further noted in the footnote on page 7, this initial value is chosen by searching through a list which includes

- the order of magnitude of the Newton-Raphson search variable, and
- a twelve-interval logarithmic distribution twice-extended beyond its lower and upper values.

Since the order of magnitude, upper value, and lower value are all constants, it follows that the value which will be chosen as the seed value, given a particular set of values for the other design variables associated with a design function, is completely deterministic.[7] If the values of the other design variables associated with a design function do not change, the same seed value will always be chosen. Since selection of the seed value is, therefore, deterministic, it follows that the solution which the Newton-Raphson method will find is also deterministic. Therefore, given a particular set of values for all other associated variables, numerical design function inversion by the Newton-Raphson method will always yield the same result for the search variable—and will therefore always use the same branch should the design function be

[7] This is the motivation behind the reference to "deterministic propagation of forcing values" in the caption of Figure 4.2.

multi-valued in the inverted direction. If use of some other branch is required for discovery of the solution value, then the solution value cannot be found.

In terms of the example presented in Section 4.2.2, this means that, for a particular set of values for the design set base variables, a given search value for $W_g$ will always yield the same two computational loop *branch* values for $C_{Lcr}$: for a given search value for $W_g$, the Bréguet range equation design function will always use a particular branch of values for $C_{Lcr}$. Thus, the curves presented in Figure 4.2 are reproduced for every run of the loop-processing algorithms; the solution values for $W_g$ and $C_{Lcr}$ can never be found, because the Bréguet range equation design function will always use the wrong $C_{Lcr}$ branch at the solution value for $W_g$.

Two different means for improving the performance of the logarithmic distribution method were investigated, as described in the following two sections. As the reader may have guessed from the repeated use of the word "deterministic" in the above discussion, the technique finally adopted to overcome the difficulties associated with multi-valued inverted design functions was to the introduction of randomness into the procedure.

### Including the Current Value as a Possible Seed Value

As has been stated, no means were available for modifying the loop-processing algorithms to explicitly account for the possibility of multi-valued (inverted) design functions. Since, the complications associated with these multi-valued design functions arise from use of the Newton-Raphson method for the purpose of numerically inverting design functions, it is this procedure which must be modified.

The first modification made to this procedure was to include the Newton-Raphson search variable's current value in the list of possible seed values. A design variable's value will vary as design set processing progresses; during application of the logarithmic distribution method for loop-processing, the current value of the *loop variable* is actually the value calculated for the *loop variable* using the other *branch* of the computational loop. Thus, including this value should increase the likelihood that the design functions in both *branches* of the computational loops will use design function branches for which the values calculated are of similar numerical magnitude. By increasing the likelihood that values calculated using multi-valued design functions will use the "correct" branch, the probability that the solution values for the *loop variable* and the *forcing variable* will be found is improved.

Indeed, inclusion of the current value of the search variable in the list of possible seed values is observed to alter the behavior of design functions to which the logarithmic distribution method is applied. Results show a tendency towards use of the "correct" branch, as

suggested above; however, this tendency does not appear to be strong enough to ***consistently*** produce the possibility of switching to an alternate branch of a multi-valued inverted design function: in test cases of the example problem discussed in Section 4.2.2, the upper branch of $C_{L_{cr}}$ values for the computational loop's *first branch* was used more often than when current values were not included as possible Newton-Raphson seed values, yet the solution point was still not found. For this reason, it was decided that further modification to the Newton-Raphson procedure was required.

### Randomizing the List of Seed Values

As suggested above, the final modification to the Newton-Raphson method was the introduction of randomness into the design function inversion procedure. The most obvious means of introducing randomness into this process is to include random values in the list of possible seed values for the Newton-Raphson method. By making the seed value selection process stochastic, the results of design function inversion—i.e., which branch of a multi-valued function is used—becomes probabilistic, instead of deterministic.

The choice of seed value for the Newton-Raphson design function inversion process was randomized by varying the limits on the logarithmic distribution which is included in the list of possible seed values. These variable limits are obtained by subtracting a random fraction of the difference between the search variable's upper and lower values from the lower value, and adding a random fraction of the same difference to the upper value.[8] With this modification, the list of possible seed values for the Newton-Raphson numerical inversion procedure is altered to include

- the order of magnitude of the search variable,
- the current value of the search variable, and
- a twelve-interval logarithmic distribution twice-extended beyond randomized limits based on the search variable's upper and lower values.

Including this source of randomness is observed to adequately improve the possibility of finding the solution values for the *forcing variable* and the *loop variable* when the design set includes design functions which are multi-valued when inverted. In contrast to all of the

[8]Specifically, the additive and subtractive amounts are obtained by multiplying one tenth of the difference between the search variable's upper and lower values by a random number between 0 (inclusive) and 1 (exclusive), i.e.,

$$\mathrm{rnd}(0,1) \times \frac{x_u - x_l}{10}$$

methods (and adaptations of methods) discussed in preceding sections of this thesis, this final modification to the logarithmic distribution method was, indeed, capable of solving the test problem[9] for consistent values for $W_g$ and $C_{Lcr}$.

[9]See the preceding sections of this chapter, and Appendix A.

- blank page -

# Chapter 5

# Concluding Remarks

## 5.1 Research Results

The task of this research was to develop a workable numerical technique for the solution of simultaneous equations, for the purpose of obtaining solutions to Computer-Aided Preliminary Design problems requiring the construction of computational loops. In working towards this goal, three basic methods for the numerical solution of two equations in two unknowns were examined:

- the fixed-point approach,
- the simultaneous Newton-Raphson approach, and
- the logarithmic distribution approach.

Based on this work, the following conclusions concerning the use of these methods may be drawn:

- All three methods require explicit representation of the steps by which the design problem is to be solved. The development of adequate representations for computational agenda and agenda entry structures is of crucial importance to the implementation of these methods: a major shortcoming of the implementation of the fixed-point method described in this thesis is the failure of this implementation to recognize that the pair of simultaneous equations which are to be solved are generally "composite functions," based upon a sequential chain of design functions.
- The fixed-point method does not guarantee convergence. Convergence of calculated values depends upon the direction in which design functions are used, and, as shown,

convergence cannot be reliably predicted but must, instead, be observed after application of the method has begun. Hence, application of the fixed-point method may result in wasted calculations, and may require backtracking to undo the effects of divergent behavior.

- The implementation of the simultaneous Newton-Raphson method presented here includes a successful representation of the computational procedure involved by identifying the following components of the computational loop structure:
  - the *preliminary entries*,
  - the *first* and *second branches*, and
  - the *final entries*.
- Due to the use of an inadequate procedure for selecting the seed value for the *forcing variable*, however, this implementation of the simultaneous Newton-Raphson method has little success in solving design problems in which the design functions display non-linearity. Because the method relies on local linearity of the "composite functions" represented by the loop *branches*[1], this method is ill-suited to the task at hand. The possible use of non-analytical design functions is explicitly allowed for in the problem statement (see Section 1.3.3).
- The logarithmic distribution method does not rely on the assumed linearity of design functions[2], but instead encounters difficulty when design functions are multi-valued in some inverted direction. This is due to the use of the Newton-Raphson method to numerically invert design functions: there is no numerical test for determining whether or not a function is multi-valued.
- As discussed in Chapter 4, by randomizing the selection of a seed value for the search variable, the Newton-Raphson design function inversion technique may be coerced into sometimes using one of its multi-valued function branches and sometimes another. This, in turn, enables the logarithmic distribution method to sometimes find the solution values for the *forcing variable* and the *loop variable* which could never be found if the process were to remain completely deterministic. Of course, including randomization does not ensure that the logarithmic distribution method will be able to find the solution values the first time—or even the second or third time—it is applied to a particular

[1] As does, it should be added, the fixed-point method ...

[2] Except when calculating the "composite" derivative to determine which interval to search next. (As indicated, however, this derivative calculation is not strictly necessary.)

problem: randomization merely *enables* the solution point to be found in cases where a deterministic approach makes discovery of the solution point impossible. And, indeed, these are precisely the results which are obtained when this "stochastic" version of the logarithmic distribution method is used: given a particular set of values for the base variables, identical runs of the test problem described in the Appendix are observed to sometimes succeed in finding the solution values for $W_g$ and $C_{Lcr}$, and sometimes fail.

Thus, having examined these various methods, it is observed that the logarithmic distribution method fits the requirements established in Chapter 1 for a suitable computational loop solution technique. By applying the logarithmic distribution method, computer-calculated answers to design problems requiring the solution of simultaneous equations may be obtained, while, at the same time

- the method takes advantage of the upper value, lower value, and order of magnitude information which is associated with design variables;
- design functions are only tested numerically; and
- there are no restrictions on the types of operations which may be included in the design functions.

However, the ability to obtain these solutions depends upon introducing randomness into the process whereby design functions are numerically inverted through application of the Newton-Raphson method. Such randomness is required when design functions are multi-valued in an inverted direction, and is introduced by applying a random variation to certain members of the list of possible seed values for the Newton-Raphson process.

## 5.2 Suggestions for Further Investigation

The research underlying the results reported in this thesis have pointed out a number of areas where further investigation would be appropriate. In the following sections, brief discussion of three of these areas is presented.

### 5.2.1 A Hybrid Analytical/Numerical Approach

Because of the limitations of the numerical techniques presented in this thesis, it is this author's opinion that a combination of both numerical and analytical techniques is required to produce a completely flexible Computer-Aided Preliminary Design system. By relying solely on numerical techniques, information which could be used in solving a design problem

is ignored.[3] As indicated above, use of the logarithmic distribution method may require repeated attempts at a particular problem before the solution is found, even when that problem can be solved very straightforwardly by analytical techniques (e.g., algebraic manipulation). To avoid wasted effort, it would seem natural to apply analytical techniques where applicable, reserving numerical techniques for problems involving non-analytic design functions. This approach was taken in the rule-based system developed by Sapossnek [15], in which the control algorithms resorted to numerical techniques when no rules for analytic reduction of the system of design equations could be applied.

However, the use of analytical techniques—i.e., performing symbolic algebra on the design functions—can be a very time-consuming process, requiring search through an extensive set of pattern-matching rules whenever it becomes necessary to develop a computational path.[4] On the other hand, the agenda-building algorithms discussed in this thesis are very efficient, because of the simplified approach taken: design functions are treated as single, indivisible entities, the mathematical nature of the operations performed within them being ignored.

Until an efficient means of performing symbolic algebra is developed, perhaps some sort of hybrid analytical/numerical approach should be examined. For instance, design functions could still be treated as single, entire entities, as described above, but a number of different mathematical representations could be associated with each design function: specifically, one representation for each associated variable, indicating an analytical approach for using the design function to calculate a value for that associated variable, based on the values of the other associated variables. Through the use of symbolic algebra, the designer/user need only be required to supply one form of the design equation—such as, for example, $w = f_1(x; y; z)$—and the system can determine the alternative representations, where possible[5]—namely, for this example, that $x = f_2(w; y; z)$, $y = f_3(w; x; z)$, and $z = f_4(w; x; y)$.[6]

This would obviate the need for numerical inversion of design functions, and would also provide access to knowledge about whether or not a certain representation is multi-valued. In addition, this application of symbolic algebra need be performed only *once*, at the time the design function is first defined. Thus, the inefficiencies associated with performing symbolic algebra need be endured only when a design set is first introduced, and not each time a new

[3] Specifically, information about the types of operations involved in the design functions, and whether or not design functions may be multi-valued, cannot be determined by numerical techniques.

[4] In addition, the kinds of design functions to which symbolic algebra techniques may be applied is rather restricted.

[5] In cases where design functions perform such operations as table-lookups and numerical integration, numerical analysis techniques will still be required.

[6] Note that such a system might also be equipped to determine symbolic representations for the partial derivatives, as well.

computational path must be constructed.

With such an approach, when it becomes necessary to develop a computational path, design functions are treated as single, indivisible entities, and the algorithms presented in this thesis may be applied. In solving design problems, the computational agenda developed in this manner must be analyzed by the numerical techniques discussed herein, though, as indicated, numerical design function inversion will be unnecessary in most cases. This allows multi-valued design functions to be dealt with effectively, since they may be recognized, and a choice may be made as to which branch should be used.[7]

Additionally, knowledge of the linearity of design functions in the directions in which they are to be used by the computational agenda allows the loop-processing algorithms to use the simultaneous Newton-Raphson method instead of the logarithmic distribution method when the behavior of the loop branches is known to be approximately linear. This would be advantageous because the simultaneous Newton-Raphson method converges considerably more quickly than the logarithmic distribution method.

### 5.2.2 Solutions to Larger Systems of Simultaneous Equations

The numerical techniques discussed in this thesis are specifically applicable to sets of functional relationships which may be reduced, by chaining, to systems of two equations in two unknowns. In applying these techniques to problems in Computer-Aided Preliminary Design, the two equations are represented by two chains of sequentially-ordered design functions, and the two unknowns are a carefully-chosen pair of design variables. In all the examples which have been tested so far, the ability of these algorithms to handle only systems which can be reduced to two simultaneous equations has been sufficient. This is probably due to the relative independence of the various sub-disciplines which are applied in the design of aerospace systems, as pointed out in Reference [10].

However, in the design of very complex systems, involving very large numbers of design functions and design variables, it is conceivable that the solution of a system which can be reduced to only three or more simultaneous equations may be necessary. For this reason, research into methods for handling larger sets of simultaneous equations is needed; at the very least, application of the algorithms already discussed in this thesis to problems requiring very large design sets should be attempted, to determine whether or not such capabilities are required.

One possible approach to the solution of larger sets of simultaneous equations is to use

[7]It must be pointed out, however, that the knowledge needed to select which branch should be used cannot be gained by symbolic algebra techniques alone. Two possible numerical approaches to branch selection are suggested in Section 5.2.3.

a nested logarithmic distribution approach. By nesting computational loops, a value can be assumed for the *forcing variable* of an outer loop, based on which the inner loop is solved using the standard logarithmic distribution method. Based on this provisional solution for the inner loop, the two *branch* values for the outer loop's *loop variable* are computed, allowing application of the logarithmic distribution method to the outer loop as well.

### 5.2.3 Recognizing Multi-Valued Functions Numerically

Finally, a point of clarification is in order. Several times in Chapter 4 and in the preceding sections of this chapter, it has been stated that there are no numerical techniques for determining whether or not a function is multi-valued. Actually, what should have been said is that there are no *reliable* numerical techniques for determining whether or not a function is multi-valued. In truth, there is at least one numerical method which could be applied to the numerical inversion of design functions to determine whether or not the inverted form of the design function is multi-valued; as the reader may have already guessed, this technique is based on a modification to the standard Newton-Raphson approach [23].

As discussed in Section 1.3.2, design functions are inverted numerically by applying the Newton-Raphson method to equations of the form, $w = f(x; y; z)$, in order to solve for one of the input variables of the design function—$x$, $y$, or $z$. If a value is sought for design variable $x$, based on known values for $w$, $y$, and $z$, the appropriate value for $x$ is determined by applying the Newton-Raphson method to locate zeroes of the function $F(x; w; y; z) = f(x; y; z) - w$.

To determine whether or not the design function is multi-valued, the following procedure is applied: having found one solution value for $x$, say $x = a_1$, a second solution value is sought by applying the Newton-Raphson method to the modified equation

$$G(x; w; y; z) = \frac{f(x; y; z) - w}{x - a_1}$$

If a second solution is found, say $x = a_2$, a third solution may be sought by applying the Newton-Raphson method to the equation

$$H(x; w; y; z) = \frac{f(x; y; z) - w}{(x - a_1)(x - a_2)}$$

and so on. When a choice must be made as to which value of a multi-valued design function should be used, perhaps a comparison could be made between the possible values, $(a_1; a_2; \ldots)$, and a list of all the values which the design variable has been assigned previously,[8] in order to determine which branch of the multi-valued design function yields an appropriate value

[8]This list could, perhaps, be weighted towards more recent values of the design variable, reflecting the designer/user's growing understanding of the design problem he is working on.

for the design variable. Alternatively, it may often be the case that perhaps not all $a_i$ can lead to a consistent set of values for the remaining unknown design variables, so that the basic requirement that solution values simultaneously satisfy all of the design functions may also be useful in selecting the appropriate branches of multi-valued design functions.

However, because there are a number of circumstances under which the Newton-Raphson method may fail to find a solution value (see Section 1.3.2), failure of this procedure to find multiple values for a design function is *not* a trustworthy indication that the design function is, indeed, single-valued. This is why it has been stated above that there is no "reliable" numerical technique for determining whether or not a design function is multi-valued. Nevertheless, applicability of the technique is worthy of investigation. Since the procedure would be used with design functions—for which information is available about the typical values of the associated variables—its reliability may be enhanced. And, as indicated in Section 5.2.1, further development of both analytical and numerical techniques is mandatory for continued progress in Computer-Aided Preliminary Design.

- blank page -

# Appendix A

# Test Design Set

Details concerning the design set which was used in testing and evaluating the various methods discussed in this thesis are presented in this appendix. Specific reference to this design set is made in Sections 4.2.2, 4.2.3, and 5.1; in addition, the curves plotted in Figure 4.2 are derived from calculations made using this design set.

Included in this appendix are tables describing the design variables and design functions used in the design set, as well as listings of the values assigned to the base variables, the computational agenda constructed for calculating values for the derived variables, and the values calculated for the derived variables by means of the logarithmic distribution method. This design set is identical to the example design set used by Elias in Reference [5]; Tables A.1 and A.2 are based on the text of that document.

## A.1 Design Variables

Table A.1 lists the name of each of the 19 design variables included in the design set, and gives a brief description of the physical significance of each design variable.

## A.2 Design Functions

Table A.2 lists the name of each of the six design functions included in the design set, and gives a brief description of the physical significance of the relationship modelled by the each design function.

## A.3 Base Variables

Table A.3 lists the thirteen design variables which were chosen for use as base variables. The table also lists the upper value, lower value, order of magnitude, and assigned value for

| Design Variable | Description |
|---|---|
| $C_{D_0}$ | Zero-lift drag term of the drag polar. |
| $C_{D_{C_L^2}}$ | Lift-square coefficient of the drag polar. |
| $C_{L_{cr}}$ | Lift coefficient in cruise flight conditions. |
| $C_{L_{max}}$ | Lift coefficient at takeoff. |
| $h_{cr}$ | Cruise altitude. |
| $h_{to}$ | Altitude at which the aircraft is assumed to take off (i.e. the density altitude). |
| $M_{cr}$ | Cruise Mach number. |
| SFC | (Thrust-) specific fuel consumption. |
| S | Reference wing area. |
| (T=W) | Aircraft's thrust-to-weight ratio at takeoff conditions. |
| $(T=W)_e$ | Engines' weight per unit of thrust. |
| $W_g$ | Gross take-off weight. |
| $W_p$ | Weight of the payload. |
| (W=S) | Aircraft's (gross take-off) weight per unit of wing area. |
| $\Delta_{to}$ | Takeoff velocity. |
| $x_{cr}$ | Cruising range. |
| $x_{to}$ | Takeoff distance (ground run). |
| ffi | Fuel used during cruise, as a fraction of the gross take-off weight. |
| ffi | Empty airframe weight (engines excluded) as a fraction of the gross take-off weight. |

Table A.1: Design variables for the test design set.

| Design Function | Description |
|---|---|
| $\Delta_{to} = 1{:}2 \frac{2W_g}{\rho(h_{to}) C_{L_{max}} S}$ | Represents the assumption that the takeoff velocity is 20% faster than the stall speed, equating lift to weight and applying the definition of lift coefficient. |
| $x_{to} = \frac{V_{to}^2}{2g_0(T/W)}$ | Simplified formula for the take-off distance, which assumes a constant acceleration determined by the thrust-to-weight ratio, up to the takeoff velocity. |
| $(W{=}S) = \frac{W_g}{S}$ | Definition of wing loading. |
| $C_{L_{cr}} = \frac{2W_g}{\rho(h_{cr}) S \left(M_{cr} a(h_{cr})\right)^2}$ | Applying the definition of lift coefficient to the cruise condition, again assuming that lift equals weight. |
| $x_{cr} = \left(\frac{1}{SFC}\right) \frac{C_{L_{cr}}}{C_{D_0} + C_{D_{C_L^2}} C_{L_{cr}}^2} M_{cr} a(h_{cr}) \log\left(\frac{1}{1-\delta_f}\right)$ | The Bréguet range equation. |
| $W_g = \frac{W_p}{1-\delta_e-1.25\delta_f-\frac{(T/W)}{(T/W)_e}}$ | This equation represents the requirement that the sum of the component weights must equal the total weight. (Note that since ff represents the *cruise* fuel fraction, it is multiplied by 1.25.) |

Table A.2: Design functions for the test design set.

| Design Variable | Lower Value | Order of Magnitude | Upper Value | Assigned Value | Units |
|---|---|---|---|---|---|
| $C_{D_0}$ | 0.005 | 0.018 | 0.05 | 0.018 | |
| $C_{D_{C_L^2}}$ | 0.005 | 0.0468 | 0.10 | 0.0468 | |
| $C_{L_{max}}$ | 0.50 | 2.20 | 5.00 | 2.20 | |
| $h_{cr}$ | 10,000.0 | 33,000.0 | 70,000.0 | 33,000.0 | ft |
| $h_{to}$ | -1000.0 | 0.01 | 5000.0 | 0.00 | ft |
| $M_{cr}$ | 0.20 | 0.80 | 1.00 | 0.80 | |
| SFC | 0.10 | 0.65 | 2.00 | 0.65 | $hr^{-1}$ |
| (T=W) | 0.100 | 0.30 | 1.0 | 0.30 | |
| $(T=W)_e$ | 2.0 | 5.0 | 20.0 | 5.0 | |
| $W_p$ | 5,000.0 | 20,000.0 | 100,000.0 | 20,000.0 | Kg |
| $x_{cr}$ | 500.0 | 2,085.0 | 12,000.0 | 3,704.0 | Km |
| $x_{to}$ | 500.0 | 2,000.0 | 10,000.0 | 1,828.8 | m |
| ffi | 0.01 | 0.45 | 1.0 | 0.45 | |

Table A.3: Base variables for the test design set.

each of these base variables.

## A.4 Computational Agenda

Table A.4 lists the computational agenda developed for this design set, based upon selection of the indicated base variables.

| Entry Type | Entry Function | Entry Variable |
|---|---|---|
| forced path: | $x_{to} = \frac{V_{to}^2}{2g_0(T/W)}$ | $\Delta_{to}$ |
| preliminary entries: | $\Delta_{to} = 1{:}2\frac{2W_g}{\rho(h_{to})C_{L_{max}}S}$ | S |
| | $W_g = \frac{W_p}{1-\delta_e-1.25\delta_f-\frac{(T/W)}{(T/W)_e}}$ | ffi |
| first branch: | $x_{cr} = \left(\frac{1}{SFC}\right)\frac{C_{L_{cr}}}{C_{D_0}+C_{D_{C_L^2}}C_{L_{cr}}^2}M_{cr}a(h_{cr})\log\left(\frac{1}{1-\delta_f}\right)$ | $C_{L_{cr}}$ |
| second branch: | $C_{L_{cr}} = \frac{2W_g}{\rho(h_{cr})S\left(M_{cr}a(h_{cr})\right)^2}$ | $C_{L_{cr}}$ |
| final entries: | $(W=S) = \frac{W_g}{S}$ | (W=S) |

Table A.4: Computational agenda resulting from the base variables.

| Design Variable | Lower Value | Order of Magnitude | Upper Value | Calculated Value | Units |
|---|---|---|---|---|---|
| $C_{L_{cr}}$ | 0.1 | 0.5 | 4.0 | 0.856 | |
| S | 50.0 | 150.0 | 500.0 | 66.2 | $m^2$ |
| $W_g$ | 5,000.0 | 100,000.0 | 500,000.0 | 150,000 | lb |
| (W=S) | 50.0 | 100.0 | 500.0 | 210 | psf |
| $\Delta_{to}$ | 50.0 | 125.0 | 350.0 | 202 | Kt |
| ffi | 0.05 | 0.25 | 1.0 | 0.157 | |

Table A.5: Derived variables for the test design set.

## A.5 Derived Variables

Table A.5 lists the six design variables which were used as derived variables. The table also lists the upper value, lower value, order of magnitude, and calculated value for each of these derived variables. The calculated values were obtained by applying the logarithmic distribution method to the computational agenda presented in Table A.4.

- blank page -

# Appendix B

# Glossary of New Terms

agenda: see computational agenda

agenda entry: the structural representation of a single step in a computational agenda, in which a given design function is used to compute a value for a specified design variable

associated variables: for a given design function, the set consisting of the design function's input variables and its computed variable

atomic operations: the mathematical operations comprising a given algebraic statement (e.g., in the statement $a = b \times c = d + e$, the atomic operations are "×," "=," and "+")

base variables: the design variables of a design set which are assigned fixed initial values, based upon which values are to be calculated for the remaining design variables (cf. derived variables)

branch: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, either the first branch or the second branch of a computational loop; may also refer to one set of values of a multi-valued function

branch derivative: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the derivative of a computational loop's loop variable with respect to its forcing variable, as calculated by applying the chain of agenda entries which includes the preliminary entries and either the first branch or second branch of the computational loop

branch value: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, a value that has been calculated for the loop variable using a

chain of agenda entries which includes the preliminary entries and either the first branch or the second branch

closing derivative: with respect to the fixed-point method, the derivative of the loop variable with respect to the looping variable, as calculated using the the entry function of the closing entry (i.e., the closing function)

closing entry: with respect to the fixed-point method, the agenda entry of a computational loop which is used to compute a value for the loop variable

closing function: with respect to the fixed-point method, the entry function of the closing entry; the closing function is used to compute a new value for the loop variable

computational agenda: the structural representation of a computational path

computational direction: the direction in which a design function is to be used in computing a new value for one of its associated variables: if the design function defined by the relation $w = f(x; y; z)$ is to be used to compute a value for the design variable $w$, its computational direction is "forward"; if this design function is to be used to compute a value for $x$, $y$, or $z$, its computational direction is "reverse," indicating that the design function must be inverted (cf. explicit direction)

computational loop: a sequence of computational steps (see agenda entry) over which iteration is required in order to determine consistent values for the variables calculated within the sequence—the need for one or more computational loops implies that the group of design variables chosen as base variables for a design set requires the solution of a set of simultaneous equations in order to calculate the values of the derived variables

computational path: the sequence of design functions which are applied in order to calculate values for the derived variables of a design set, based on the values which have been assigned to the design set's base variables

computational state: the status of the current value of a design variable—the current value of a design variable may be either initialized, computed, or guessed[1]

computed variable: the design variable which is calculated by a design function when the design function is applied in the manner in which it has originally been defined (e.g.,

[1]In the original version of *Paper Airplane*, design variables' values were either initialized, computed, or unknown. Design variables with unknown values were designated as "free."

the computed variable of the design function defined by the relation $w = f(x; y; z)$ is the design variable $w$)

Computer-Aided Preliminary Design: or CPD, the use of computers to perform the routine computational tasks involved in the preliminary design of engineering systems

C-variable: a design variable whose computational state is "C" (computed); the value of such a design variable has been computed as one step of a computational loop

derived variables: the design variables of a design set which are *not* assigned initial values; the values of these design variables must be calculated using the available design functions and the known values of the base variables

design function: a functional relationship between design variables, expressed in the form $w = f(x; y; z)$ (where $w$, $x$, $y$, and $z$ are design variables, and $f$ is the design function), which may be based on satisfaction of relevant physical constraints, performance requirements, empirical criterion, design variable definitions, etc., for a particular preliminary design problem

design path: a design set and a given choice of base variables; selection of a design path requires the generation of a computational path for determining the values of the derived variables once values have been assigned to the base variables

design set: a collection of design variables and design functions relating those design variables, chosen to represent a specific preliminary design problem

design variable: a parameter representing some physical quantity, dimension, performance measurement, or other numerical feature of a particular preliminary design problem

entry function: the design function which is to be applied as a one step in a computational agenda (cf. agenda entry)

entry variable: the design variable whose value is to be calculated as one step in a computational agenda (cf. agenda entry)

explicit direction: the direction of application of a design function as indicated in the design function's definition (e.g., the explicit direction of a design function defined by the relationship $w = f(x; y; z)$ is to calculate a value for $w$, based on the values of $x$, $y$, and $z$)

final entries: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, those agenda entries of a computational loop over which iteration is not required, but which require successful processing of the earlier agenda entries in the computational loop before they may be processed

first branch: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the first of the two chains of agenda entries which, after processing the preliminary entries may be used to calculate a value for the loop variable, based on a given value for the forcing variable

first entry: the first agenda entry of a computational loop (one of the attributes of a loop header)

forced function: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, a design function detected during construction of a computational loop's preliminary entries for which values are available (either assigned or calculated) for all of its associated variables, except for one associated variable, which happens to be the forcing variable of the current computational loop

forced path: the sequence of computations which may be made using only those design functions which are perfectly constrained by the base variables and any design variables which may be calculated using design functions which are perfectly constrained by the base variables (i.e., design functions which are perfectly constrained by the base variables and any design variables which are calculated as part of the forced path)

forcing entry: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, an agenda entry found in the preliminary entries prior to branch construction, which is used in the preliminary entries to compute a value for one of the C-variables of the forced function

forcing variable: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the design variable associated with a computational loop for which a value is assumed in order to calculate two values for a second design variable (cf. loop variable)

free variable: a design variable whose state is either "F", "G," or "L," indicating that the current value of the design variable may be replaced by a computed value[2]

[2]In the original version of *Paper Airplane*, design variables' values were either initialized, computed, or unknown. Design variables with unknown values were designated as "free."

F-variable: a design variable whose computational state is "F" (see forcing variable)[3]

G-variable: a design variable whose computational state is "G" (guessed); the value of such a design variable has been neither assigned nor computed

incompatibility: the condition resulting when the value which would be calculated for a given design variable using an overconstraining function is sufficiently different from the value already associated with the design variable

inner loop: for a given computational loop, any other computational loop which sequentially follows it in the computational agenda

input variable: any one of the design variables which is a dependent variable of a design function *as it has been defined*; i.e., any one of the design variables for which either an assigned or a computed value is required in order to apply the design function in the manner in which it has originally been defined (e.g., the input variables of the design function defined by the relation $w = f(x; y; z)$ are the design variables x, y, and z)

inversion failure: the condition which is signalled when application of the Newton-Raphson method to numerically invert a design function fails to calculate a value for the chosen search variable

I-variable: a design variable whose computational state is "I" (guessed); the value of such a design variable has been assigned by the designer (cf. base variables)

K-variable: a design variable whose computational state is "K" ("*k*omputed"); the value of such a design variable has been computed as one step of a computational agenda's forced path

local incompatibility: the condition resulting from the occurrence of an overconstraining function when the design set is not itself overconstrained; local incompatibilities arise from a choice of base variables which overconstrains some subset of the group of design functions included in the design set

loop header: a data structure associated with a computational loop, which provides access to the agenda entries which comprise the computational loop, as well as such information about the computational loop as its loop variable and (where applicable) its forcing variable or looping variable

[3] In the original version of *Paper Airplane*, so-called "free" design variables (see preceding footnote) were assigned the computational state "F."

loop variable: with respect to the fixed-point method, the design variable associated with a computational loop whose value is assumed, and updated as a result of calculations based on the assumed value; with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the design variable associated with a computational loop for which two values may be calculated by assuming a value for the computational loop's forcing variable and performing the sequence of computations specified by the preliminary entries and first and second branches of the computational loop

looping derivative: with respect to the fixed-point method, the derivative of the loop variable with respect to the looping variable, as calculated using the entry function of the looping entry (i.e., the looping function)

looping entry: with respect to the fixed-point method, the last agenda entry of a computational loop prior to the closing entry which calculates a value for one of the C-variables associated with the entry function of the closing entry

looping function: with respect to the fixed-point method, the entry function of the looping entry, used to compute the value of the looping variable

looping variable: with respect to the fixed-point method, the entry variable of the looping entry; the looping variable will be one of the C-variables associated with the entry function of the closing entry

lower value: a typical minimum value for a design variable (e.g., in the design of subsonic transport aircraft, the lower value of the structural weight fraction might be 0.2)

L-variable: a design variable whose computational state is "L" (see loop variable)

nested loop: with respect to the fixed-point method, a computational loop which contains its own closing entry

order of magnitude: a typical approximate value for a design variable (e.g., in the design of subsonic transport aircraft, the order of magnitude of the main wing aspect ratio might be 10)

outer loop: for a given computational loop, any other computational loop which sequentially precedes it in the computational agenda

overconstrained: the status of a design set for which the number of available design functions exceeds the number of derived variables (i.e., uninitialized design variables); after

calculating values for all derived variables, some design functions will remain unused (i.e., will not be needed in constructing the computational path)

overconstraining function: a design function for which none of the associated variables are free variables; since the values of all its associated variables have already been either calculated or assigned, an overconstraining function is ineligible for inclusion in the computational path

overconstraint: the condition resulting from the existence of an overconstraining function

perfectly constrained: a design function with only one free variable among its list of associated variables; such a design function is immediately eligible for inclusion in the computational agenda as an entry function for computing the value of its single free variable.

preliminary entries: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the chain of agenda entries in a computational loop which represents the shared portion of the two paths which are used to compute values for the loop variable based on an assumed value for the forcing variable

search interval: with respect to the logarithmic distribution method, a set of logarithmically distributed values to be assigned to the forcing variable of a computational loop as search values

search value: a tentative value which is assigned to the design variable for which a value is sought in applying the Newton-Raphson method to numerically invert a design function; with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, this term may also be applied to a value which is tentatively assigned to a computational loop's forcing variable in attempting to determine consistent values for its forcing variable and loop variable

search variable: the input variable of a design function for which a value is sought in applying the Newton-Raphson method to numerically invert the design function; with respect to simultaneous Newton-Raphson method or the logarithmic distribution method, this term may also be applied to a computational loop's forcing variable

second branch: with respect to the simultaneous Newton-Raphson method or the logarithmic distribution method, the second of the two chains of agenda entries which, after processing the preliminary entries, may be used to calculate a value for the loop variable, based on a given value for the forcing variable

seed value: the first search value assigned to the search variable in applying the Newton-Raphson method

state: see computational state

tangled loop: with respect to the fixed-point method, a computational loop which does not contain its own closing entry

underconstrained: the status of a design set for which the number of derived variables (i.e., uninitialized design variables) exceeds the number of design functions; no means are available for calculating the values of all derived variables

upper value: a typical maximum value for a design variable (e.g., in the design of subsonic transport aircraft, the upper value of the cruise Mach number might be 0.95)

# Bibliography


[1] Bil, C., "CAADIP: Computer-Aided Aircraft Design Interactive Program," Ph. D. Thesis, Department of Aerospace Engineering, Delft University of Technology, May 1981.

[2] Bil, C., "ADAS Command Reference Manual," Department of Aerospace Engineering, Delft University of Technology, April 1985.

[3] Burke, Glenn S., Carrette, George J., and Eliot, Christopher R., "NIL Reference Manual", Report No. TR-311, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1984.

[4] Dixit, C. S., and Patil, T. S., "Multivariate Optimum Design of a Subsonic Jet Passenger Airplane", *Journal of Aircraft*, Vol. 17, No. 6, pp. 429–432, June 1980.

[5] Elias, Antonio L., "Knowledge Engineering of the Aircraft Design Process," in Kowalic, J. S., ed., *Knowledge Based Problem Solving*, Englewood Cliffs, N. J.: Prentice-Hall, 1985, Chapter 6.

[6] Fulton, Robert E., Sobieszczanski, Jaroslaw, and Landrum, Emma Jean, "An Integrated Computer System for Preliminary Design of Advanced Aircraft," AIAA Paper No. 72-796, August 1972.

[7] Greenspan, H. P., and Benney, D. J., *Calculus: an Introduction to Applied Mathematics*, New York: McGraw-Hill Book Company, 1973.

[8] Gregory, Thomas J., and Roberts, Leonard, "An Acceptable Role for Computers in the Aircraft Design Process," in *The Use of Computers as a Design Tool*, AGARD Conference Proceedings No. 280, September 1979.

[9] Hildebrand, F. B., *Introduction to Numerical Analysis*, New York, N. Y.: McGraw-Hill Book Company, Inc., 1956.

[10] Howe, D., "The Application of Computer-Aided Techniques to Project Design," *Aeronautical Journal*, Vol. 83, No. 817, pp. 16–21, January 1979.

[11] Jenkinson, L. R., and Simos, D., "A Computer Program for Assisting in the Preliminary Design of Twin-Engined Propeller-Driven General Aviation Aircraft," *Canadian Aeronautics and Space Journal*, Vol. 30, No. 3, pp. 213–224, September 1984.

[12] Kolb, Mark A., "Problems in the Numerical Solution of Simultaneous Non-linear Equations in Computer-Aided Preliminary Design," Memo No. 85–1, Flight Transportation Laboratory, Massachusetts Institute of Technology, May 1985.

[13] Lancaster, J. W., and Bailey, D. B., "Naval Airship Program for Sizing and Performance (NAPSAP)", *Journal of Aircraft*, Vol. 18, No. 8, pp. 677–682, August 1981.

[14] Mitchell, Allen, Boeing Aerospace Company, personal communication, May 1985.

[15] Sapossnek, Mark, "MARKSYMA: A Parametric Design Tool Based Upon the Analytic and Numeric Solution of Systems of Equations," M. Sc. Thesis, Department of Mechanical Engineering, Rensselaer Polytechnic Institute, December 1985.

[16] Smith, Randall, Lockheed-Georgia Company, personal communication, November 1985

[17] Stallman, Richard, Weinreb, Daniel, and Moon, David, *Lisp Machine Manual* (Sixth Edition), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, June 1984.

[18] Steele Jr., Guy L., "The Definition and Implementation of a Computer Programming Language based on Constraints", Ph. D. Thesis, Department of Electrical Engineering and Computer Sciences, Massachusetts Institute of Technology, July 1980.

[19] Steele Jr., Guy L., *Common LISP: The Language*, Burlington, Ma.: Digital Press, 1984.

[20] Torenbeek, Egbert, *Synthesis of Subsonic Airplane Design*, Delft: Delft University Press, 1982.

[21] Torenbeek, E., "Computer Aided Design as a Discipline in Aeronautical Teaching and Research at the Delft University of Technology," Memorandum M–472, Department of Aerospace Engineering, Delft University of Technology, June 1983.

[22] Wright, Robert L., DeRyder, Dariene D., and Ferebee, Melven J., Jr., "Interactive Systems Design and Synthesis of Future Spacecraft Concepts," NASA Technical Memorandum 86254, June 1985.

[23] *HP–15C Owner's Handbook*, Corvalis, Or.: Hewlett-Packard Company, 1982.

[24] *TK!Solver Program Instruction Manual*, Wellesley, Ma.: Software Arts Inc., 1982.