REPORT R87-3

INTEGRATION OF ENGINEERING MODELS IN COMPUTER-AIDED PRELIMINARY DESIGN

FTL COPY, DON'T REMOVE
33-412, MIT 02139

RONNIE M. LAJOIE

JANUARY 1987

MIT

DEPARTMENT OF AERONAUTICS & ASTRONAUTICS

FLIGHT TRANSPORTATION LABORATORY
Cambridge, Mass. 02139

FLIGHT TRANSPORTATION LABORATORY REPORT R87-3

INTEGRATION OF ENGINEERING MODELS IN COMPUTER-AIDED PRELIMINARY DESIGN

(Formally "Multi-valued Output and External Code Interface Capability")

by

RONNIE M. LAJOIE
B. S., Boston University
(1984)

© Massachusetts Institute of Technology, 1987

This report reproduces a thesis submitted on January 16, 1987, 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.

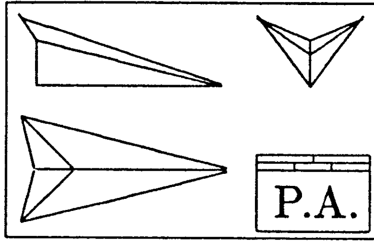
INTEGRATION OF ENGINEERING MODELS
IN COMPUTER-AIDED PRELIMINARY DESIGN

by

RONNIE M. LAJOIE

Abstract

The problems of the integration of engineering models in computer-aided preliminary design are reviewed. This paper details the research, development, and testing of modifications to *Paper Airplane*, a LISP-based computer program, designed to address these integration problems. *Paper Airplane* integrates engineering models by treating them like a set of simultaneous non-linear functions and numerically solving for them as such. The original version of *Paper Airplane* could only handle engineering models represented by single equations and simple LISP functions; that is, multiple-input single-output (MISO) functions. The modifications to *Paper Airplane* were to allow it to handle engineering models represented as complex LISP functions and external computer programs as well; that is, multiple-input multiple-output (MIMO) functions. The research was divided into three tasks: (1) to get *Paper Airplane* to communicate with an external computer program (without changing the computer program), (2) to get *Paper Airplane* to numerically solve a non-linear MIMO function, and (3) to get *Paper Airplane* to numerically solve a set of simultaneous non-linear functions made up of MISO and MIMO functions.



Acknowledgments

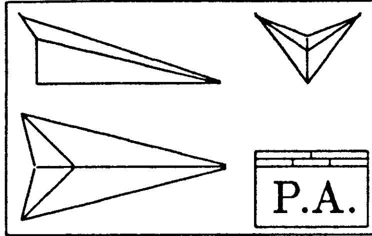
I would like to acknowledge the contributions to this work of the following individuals, and express to them my sincere gratitude and appreciation:

Dr. Antonio Elias, my first thesis advisor, for starting this wonderful project and for giving me the enthusiasm to keep it going; Dr. Robert (Bob) Simpson, my second thesis advisor, for picking up the pieces when Antonio had to leave and for never complaining; Dr. Alan Mitchell, for guiding from afar and for keeping the research properly focused; Boeing Aerospace, for funding this research and for giving me one good year of real experience; Dr. Miller and Dr. Dudley, M.I.T. professors, for providing information invaluable to this research; Dr. John Pararas and Lyman Hazelton, for knowing how to do everything I couldn't, and for explaining it to me so that I could; Mark Kolb and Thanasis (Tom) Trikas, my colleagues, for their advice and assistance on everything from research and homework to an *occasional* game of ROGUE or UNIVERSE; the creators of LaTeX and DEC-SPELL, for allowing me to sharpen my writing skills; Robert (Bob) Bruen, for tolerating me and my wild cans of Diet Coke; Dave, Jim, Mick, Michele, and Randy, my college chums, and Joanna, Barbara, Ruthanne, and Clark, for those occasional bits of social life I could rarely afford.

I would especially like to thank Dr. Marc Whitlow, my roommate, for putting up with me and my stupid jokes, and for laughing at them. Your company has restored my faith in roommates.

Finally, I would like to thank my family, for their love and support throughout this year and a half, and for their understanding that research and thesis writing takes precedence over visits and Christmas shopping.

Paper Airplane logo designed by Mark Kolb.



Contents

	Abstract	**ii**
	Acknowledgments	**iii**
1	**Introduction**	**1**
2	**The Paper Airplane Code**	**7**
2.1	The Definition of Paper Airplane	7
2.2	The Terminology of Paper Airplane	8
2.3	The Usage of Paper Airplane	11
2.4	The MISO Design Set Benchmark	15
2.5	The Code of Paper Airplane	18
3	**The External Code Interface**	**21**
3.1	Research	21
3.2	Development	24
3.3	Testing	25
4	**MIMO Design Functions**	**30**
4.1	Simple MIMO Capability	30
4.1.1	Research	30
4.1.2	Development	32
4.1.3	Testing	34
4.2	General MIMO Capability	34
4.2.1	Research	34
4.2.2	Development	37
4.2.3	Testing	38
5	**The Numerical Solvers**	**44**
5.1	The Design Function Solver	44
5.2	The Loop Solver	48
5.3	Thoughts on Numerical Methods	57

6 The Aerospaceplane Design Test **59**
6.1 Research 59
6.2 Development 63
6.3 Testing 68

7 Summary and Conclusions **77**
7.1 Summary 77
7.2 Conclusions 79

A The Test Design Sets **81**
A.1 The MISO Design Set 81
A.2 The XCODE Design Set 83
A.3 The MIMO Design Set 84

B MISO Design Set Source File **85**

C MISO Design Set Solution **92**

D The "RUN-PROGRAM" Code **98**

E Vector Equation Solving Code **100**

F MIMO Design Set Solution **102**

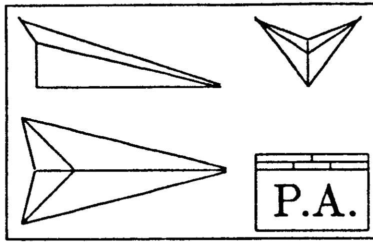
G The NASP Design Set **109**

H NASP Aerodynamics Code **115**

I NASP Performance Code **125**

Glossary **136**

Bibliography **143**



List of Figures

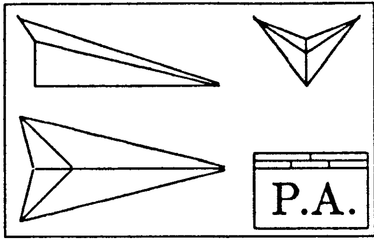
2.1 Example design variable and design function declarations. 12
2.2 Example tableau as it first appears. 13
2.3 Example tableau after design set processing. 16
2.4 The initial design point of the MISO Design Set benchmark. 17
2.5 The computational agenda to find the MISO Design Set benchmark. . . . 19
2.6 The final design point of the MISO Design Set benchmark. 20

3.1 Data passing technique used by Boeing's Preliminary Design Tool. 23
3.2 The SYS$CREPRC definition macro and test function. 24
3.3 The modified design function "DF-4." 26
3.4 Example display of Design Function Exerciser after processing. 28

4.1 The agenda for the solution to the design path of the MIMO Design Set. . 35
4.2 The initial design point for the "inverse engineering" problem. 40
4.3 The agenda for the solution to the "inverse engineering" problem. 42
4.4 The final design point for the "inverse engineering" problem. 43

5.1 The Newton-Raphson technique for locating zeros. 45
5.2 Comparison of methods with solution near the lower limit. 47
5.3 Comparison of methods with solution near the upper limit. 49
5.4 The logarithmic distribution method for solving loops. 51
5.5 Comparison of methods for linear loop branches. 53
5.6 Branches of drag coefficient versus gross take-off weight. 54
5.7 Branches of minimum landing weight versus cruise velocity. 55
5.8 Comparison of methods for non-linear loop branches. 56

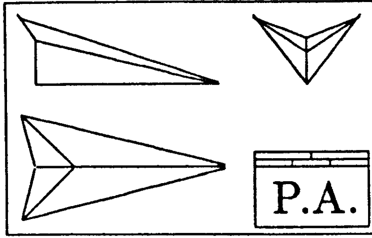
6.1 Geometry of the aerospaceplane. 61
6.2 Drag Coefficient vs. Lift Coefficient. 65
6.3 Lift-to-Drag Ratio vs. Lift Coefficient (150000 ft). 66
6.4 Computational agenda for Second Stage testing. 70
6.5 Computational agenda for Third Stage testing. 76



List of Tables

A.1 Design Variables comprising the MISO Design Set. 82

G.1 Geometry Design Variables comprising the NASP Design Set. 110
G.2 Other Design Variables comprising the NASP Design Set. 111



Chapter 1

Introduction

In the manufacturing environment, after a product is built, it is run through a series of tests: structural tests, acoustic tests, performance tests, thermal tests, safety tests, and many others; this is called "Product Testing." In the engineering environment, *before* a product is built, it is run through the same tests to decide whether or not the product *should* be built; this is called "Preliminary Design." Preliminary design is what takes an idea and possibly turns it into a blueprint for a product.

Representation of the Idea

Since the product is only an idea during preliminary design, an alternative representation of it must be found. This representation comes in the form of a mathematical model. A mathematical model of a simple metal screw, for example, must contain information on its geometric properties, its structural properties, its thermal properties and its electrical properties. Attach this screw to a metal plate and the mathematical model must not only include the aforementioned properties of both the screw and plate, but also the interaction of those properties between the two. Attach this plate to an avionics box and the mathematical model becomes very complex. Attach this avionics box to the cockpit of a commercial jetliner and the mathematical model becomes extremely complex.

To simplify the mathematical model, it is separated into many groups of components, or sub-systems, using a hierarchy similar to the one followed above. To simplify the mathematical model even more, each component model is further separated into groups according to its properties. Instead of one large and extremely complex mathematical

model, preliminary design thus deals with many small and simpler sub-models. These sub-models are commonly referred to as **engineering models**, since these are the types of mathematical models an engineer usually deals with. Dividing the mathematical model into many engineering models also has the advantage that some engineering models of the idea may be common to other ideas already transformed into products.

The engineering models, depending upon on their own level of complexity, are physically represented by single equations, by sets of equations, and by computer programs. They are stored on magnetic tape and hard disk, in textbooks and notebooks, and on scraps of paper and piles of computer print-out.

The Problems with Engineering Models

In an ideal engineering environment,

1. The engineering models of an idea would be available in several different layers of complexity, ranging from a conceptual level to an advanced level of design.

2. At each level of design, there would be engineering models of that level's complexity to account for all parts of the proposed product and all of their properties. (Even though a structural model of an aircraft wing at the preliminary design level rarely includes the rivets and joint connections, the model should nevertheless account for them, even if it means merely adding some structural efficiency factor.)

3. The information on the proposed product would be stored in one secure central location and referenced by all the engineering models involved. This would insure that, for example, all engineering models requiring geometry information would acquire the *same* geometry information.

In the real engineering environment, however,

1. The engineering models of an idea are not always available in several different layers of complexity. For example, thermal models at the conceptual design level usually do not exist and their properties are usually ignored until the idea enters advanced preliminary design.

2. At each level of design, there are not always engineering models of that level's complexity to account for all parts of the proposed product and all of their properties. Instead, the missing engineering models and the information they contain are

ignored (as mentioned above) or engineering models from more complex levels of design are substituted for the nonexistent simpler ones. This could be worse than ignorance since it brings unnecessary detail into the design at that level. It also can lock the design prematurely before all the degrees of freedom that a simple level of design has to offer are analyzed.

3. The information on the proposed product is scattered all over a company. The information resides in the company's main computer, in engineers' personal computer files, on notepads on engineers' desks, and on blueprints on drafters' tables. The time delay in acquiring needed information often results in an engineering model using assumed, and often conflicting, information. Wrong information can propagate throughout the design before it is finally detected and, expensively, corrected.

Part of the problem has been the enormity of the task. To correct Problems 1 and 2, more engineering models would have to be created; however, this would add to Problem 3. Making sure that many engineering models, scattered throughout a large company, never have conflicting information is an impossible job for a human being. Adding more human beings to the job requires one more human being above them to make sure that *they* communicate with each other. Most companies cannot afford either the manpower, money, or time to do this, thus business continues as usual.

A Computerized Solution

With the advent of smart computers and even smarter computer programmers, there may finally be a cost-effective means to monitor and handle the information engineering models require and produce and to insure that the design never has conflicting information.

Recent developments in computer technology have created engineering workstations, powerful cousins of personal computers. These new computers give the engineer the power of a large computer on a desktop. This, of course, means nothing if the engineer still has to write down the results or make a hard-copy to pass along design information. Other recent developments in computer technology and in communications technology have created very high speed networks that, once linked to several computers, provide instant communication between them. Even now, networks linking computers speed data across the country in a matter of minutes when it used to take days by mail and even by person.

Linking together engineering workstations is one thing, linking together engineers and their engineering models is another. The former is a matter of computer hardware; the latter, of computer software. The purpose of computer software is to do the same thing a human could do only faster, and repeatedly without adding mistakes. High-level computer programming languages, especially the object-oriented ones, now have the capability to monitor and handle design information between engineering models quickly and accurately. Computer databases now have the capability to house all the design information in one secure location plus allow for fast information storage and retrieval. A computer-based engineering model information sharing system (CEMISS) is now a cost-effective prospect to the engineering community.

The Paper Airplane Project

The *Paper Airplane* Project is the first one of its kind to apply this CEMISS idea to aerospace engineering. Although other computer programs have been developed that can share information between similar engineering models, they have been limited to certain types of products, such as general aviation aircraft [5] and naval airships [11], or to certain types of properties, such as NASTRAN[1] and MATRIX_X.[2] Neither group could pass information on to any random engineering model. The goal of the *Paper Airplane* Project is to do just that.

The project was begun by Dr. Antonio Elias [3], former professor of Aeronautics and Astronautics at the Massachusetts Institute of Technology, in 1981 with a LISP-based code that could solve a simple system of design equations. Prof. Elias, working together with Mark Kolb ([7] and [8]), then a Master's candidate, later made *Paper Airplane* a user-friendly interactive test-bed for general systems of linear and non-linear design equations. Although *Paper Airplane* was designed with aerospace engineering in mind, it was not designed for aerospace engineering applications only and therefore could be used for *any* systems design work.

Paper Airplane thus linked engineering models, as long as each model was represented by a single equation, a multiple-input single-output (MISO) function. Given a complete set of these equations and the parameters they related, *Paper Airplane* could find nu-

[1]NASTRAN is a finite-element modeling and analysis program for dealing with the structural and thermal properties of a component.

[2]MATRIX_X is a mathematical modeling and analysis program for dealing with the dynamics and control properties of a component.

merical solutions whereby values assigned to the parameters would allow them to satisfy all the equations.

Modifications to Paper Airplane

Although *Paper Airplane* worked well with equations, it could not work at all with computer programs, and thus it did nothing to help the engineer having engineering models represented by computer programs. Without such a capability, *Paper Airplane* would remain an academic research tool and never graduate to a professional engineering tool. The author joined the project to solve this problem, as he was returning to school after working towards similar goals for Boeing Aerospace.

Simply, the objective was to make *Paper Airplane* work with computer programs, or more generally, with any multiple-input multiple-output (MIMO) function. The research and development was scheduled for the entire year of 1986. The author joined the project in August 1985 and spent the rest of the year getting familiar with M.I.T., the *Paper Airplane* Project, its staff, its history, and the *Paper Airplane* code itself. During this period the plan of action for the research was laid out.

The research was initially divided into two parts:

1. Development of an external code interface capability to allow *Paper Airplane* to pass information to and from a computer program as easily as passing information to and from an internally-defined equation.

2. Development of a MIMO design function capability to allow *Paper Airplane* to solve a multiple-input multiple-output function as easily as solving an equation or other multiple-input single-output function.

The author's research at Boeing had already yielded a satisfactory method to solving Part 1; indeed, a solution was found that required no internal changes to the *Paper Airplane* code. Part 2 was a different matter completely. It required as many hours of thought as hours of programming changes to the *Paper Airplane* code. The success of the MIMO design function capability modification was due in a large part to the advice received from Prof. Elias [4], Mark Kolb [9], and mathematics professor Richard Dudley [2], and to the information found in a text on numerical methods [1].

One major outcome of this research has been the author's deeper appreciation of the power and potential of mathematics applied to aerospace engineering.

Outline to this Paper

The rest of this paper gives the details of the research that was required, and of some of the research that was not, but was worth doing nevertheless.

Chapter 2 gives the reader an overview of *Paper Airplane*, and establishes the benchmark to which all modifications to *Paper Airplane* were compared.

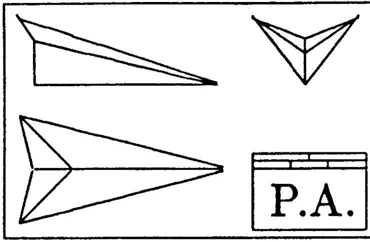
Chapter 3 details the research, development, and testing of the external code interface capability modification to *Paper Airplane*.

Chapter 4 details the research, development, and testing of the MIMO design function capability modification to *Paper Airplane*, which included the creation of the *Paper Airplane* MIMO Solver.

Chapter 5 details the modifications made to *Paper Airplane*'s other Numerical Solvers, the Design Function Solver and the Loop Solver, to improve their efficiency.

Chapter 6 details the creation and use of the NASP Design Set for the preliminary design of a national aerospaceplane, which served as a major test of *Paper Airplane*'s modified capabilities.

Chapter 7 gives the reader a summary of the research and presents conclusions.



Chapter 2

The Paper Airplane Code

This chapter describes *Paper Airplane*, a computer program modified under this research. The information that follows was taken from the *Paper Airplane User's Manual* [10], written by the author as part of his research into the *Paper Airplane* Project. This information has been updated to include the results of this research; therefore this chapter is really a summary of the final product. New terms will appear in **boldface** where they are defined. These terms also appear in the glossary at the end of this paper.

2.1 The Definition of Paper Airplane

Paper Airplane is a computer-aided Preliminary Design Tool (PDT); that is, a computer program designed to aid an engineer in the design of an aircraft or any other system capable of being described by a set of scalar parameters. Superficially, *Paper Airplane* is a "simultaneous calculator"; that is, a calculator capable of determining the values of a set of parameters satisfying a set of linear and non-linear simultaneous functions. In this sense, *Paper Airplane* might be viewed as an "engineer's spreadsheet" program, similar to TK!Solver [16]. But while TK!Solver can only handle simple algebraic expressions, *Paper Airplane* can also handle complex multiple-input single-output (MISO) functions such as numerical integrators, and complex multiple-input multiple-output (MIMO) functions such as computer programs.

The *Paper Airplane* Project has focused on creating powerful computer abstractions capable of representing many different elements of an engineer's design knowledge. The long-term goal of this research is to develop a *symbolic computational environment* con-

taining all the elements that an aircraft or systems designer normally manipulates — in his or her head, up until now.

2.2 The Terminology of Paper Airplane

In order to explain how *Paper Airplane* works, some basic terms need to be defined first. (Author's note: Most of the terms below were defined long before the author joined the project. Thus for the sake of continuity in *Paper Airplane* documentation, the author did not change the terms themselves, but hopefully made their definitions more clear.)

design variable: is a scalar parameter, such as Vehicle Length or Vehicle Weight, whose value uniquely determine part of the configuration of an aircraft, spacecraft, or any other system. A design variable has a number of attributes associated with it, such as its value, its dimensions, its order of magnitude, and the limits of its value.

design function: is a relationship between design variables. A design function can range in complexity from a simple algebraic equation to a very large and complex computer program.

design set: is a set of certain design functions and the design variables those functions relate towards the goal of solving a particular design problem.

source file: is a computer file containing the information on all of the design variables and design functions to be loaded internally into a *Paper Airplane* design set.

loading: is a COMMON LISP term for reading and evaluating LISP code from a file into main memory.

variable tableau: is a spreadsheet of information on the design set arranged on a computer screen. This information includes a list of design variables and their current values, units, and states.

variable state: is the condition of the value of a design variable. Variable states come in the following three varieties, which are assigned to design variables according to their initial letter.

Initialized-value state: This indicates a design variable that has been given a known value by the user. A design variable obtains state I whenever the user changes its value, or when the user **freezes** it. I-state design variables, officially designated as **base variables**, will be referred to simply as **knowns**.

Guessed-value state: This indicates a design variable that has been given a trial value by the user. A design variable obtains state G whenever the user **floats** it. G-state design variables, officially designated as **derived variables**, will be referred to simply as **unknowns**.

Computed-value state: This indicates a design variable that *had* been given a trial value by the user, and was later given a known value by *Paper Airplane*. A design variable obtains state C only when the user **processes** the design set; and then only if *Paper Airplane* can find a solution which satisfies all the design functions in the user's design set.

design point: is the values and states of all the design variables in a design set at any stage in the design process.

design path: is the selection of certain design variables as knowns and the rest as unknowns; thereby setting up some implied path, or sequence of design functions, for *Paper Airplane* to follow once values are provided for the design variables.

computational agenda: is the actual path, or sequence of design functions, to be evaluated to find the values of the unknowns once given the initialized values for the knowns and the guess values for the unknowns. The computational agenda is also called the **computational path**. The computational agenda consists of a forced path and loops.

forced path: is a sequence of perfectly constrained design functions, each of which can be solved individually, but sequentially. The path is called "forced" since there is no alternative but to solve the design functions in this sequence in order to compute the values of their unknowns.

loop: is a sequence of perfectly constrained design functions, each of which computes values required by other design functions in a closed loop. Loops are solved by guessing the value of a **forcing variable** to compute two independent values of a **loop variable**. When the two values converge, the values of all the unknowns involved can be found.

Agenda Building

Given a design path of design variables selected as knowns and unknowns, *Paper Airplane* assembles its methodology for solution into a computational agenda. This process is commonly referred to as **agenda building**. (For complete details on agenda building, see [8].) The key to understanding agenda building is that it only involves the knowns, the unknowns, and the design functions that use them. No design function is evaluated and no numbers are produced. Each design function is merely examined to find out what kind of design variables (knowns or unknowns) go in, and what kind come out.

An iterative search is performed to find the design function with the least amount of unknowns, whether they are going in or coming out. At any time, if the number of unknowns of the design function is less than the number of values it computes, that overconstrained design function is discarded and the unknowns involved are labeled "inconsistent."

If the number of unknowns of the design function equals the number of values it computes, however, the design function can be solved for using the forward computation method, the reverse computation method, or (for MIMO design functions) a combination of both. The design function is then labeled "used" and is placed as an agenda entry into the forced path of the computational agenda. An **agenda entry** is merely the design function and the unknowns to be solved for by it. The states of the unknowns are then changed to C and the design variables are then treated as knowns. In this manner, a design function that was initially underconstrained can become perfectly constrained because of the solution of another design function.

When the search returns a design function whose number of unknowns is greater than the number of values it computes (i.e., an underconstrained design function), the forced path construction is ended and the loop construction begins. The choice of forcing variable for a possible loop is the unknown most common to the remaining unused design functions. The state of the forcing variable is temporarily set such that the search will treat it is a known. As long as the search keeps returning perfectly constrained design functions, a forced path of **preliminary entries** will be constructed. This construction stops when an overconstrained or an underconstrained design function is returned.

If an overconstrained design function is returned, it is checked to see if it has a computed unknown common to one of the design functions in the preliminary entries. If

it does, then there are two design functions that can independently compute the value of the same unknown, then called the loop variable; thus the loop can be closed. If it doesn't however, then the overconstrained design function is discarded and its unknowns are labeled "inconsistent." On the other hand, if an underconstrained design function is returned, the loop can never be closed; thus a new forcing variable must be chosen.

The preliminary entries of any closed loop are organized into an initial path, two branches, and a final path. The **initial path** is a sequence of perfectly constrained design functions whose computed unknowns are required by both branches. The **branches** are two independent sequences of perfectly constrained design functions for computing the value of the loop variable. Lastly, the **final path** is a sequence of perfectly constrained design functions whose unknowns can be solved for once the loop has converged.

Agenda building continues until all the design functions are used or discarded, or until a loop construction failure occurs, when all possible forcing variables have been tried to construct a loop and have failed. If all the design functions are used, *Paper Airplane* should be then able to compute a unique numerical solution for any initial design point obeying the design path. If design functions have been discarded, however, any numerical solution found will have inconsistencies. On the other hand, if a loop construction failure occurs, the computational agenda will be incomplete, and *Paper Airplane* will only be able to find a partial numerical solution to any initial design path.

2.3 The Usage of Paper Airplane

A user of *Paper Airplane* must first gather the engineering knowledge he or she will need and represent it as equations, functions, and/or computer programs. (A **function** is an internal piece of code written in COMMON LISP; whereas a **computer program** is an external piece of code usually not written in COMMON LISP.) Next, that information must be coded as design variables and design functions into a source file; the format to be followed is shown in Figure 2.1. The user then starts up *Paper Airplane* and loads the source file into it.

Figure 2.2 shows the spreadsheet-like tableau a user may see once the source file is loaded into a design set. The columns are for the design variable names, states, current values, and current units. The name of the tableau appears at the top since a design set can have more than one tableau (to better organize design set information).

The user selects the design path by changing design variable states to the best of his

```
(pa-defvar WING_REFERENCE_AREA
           :category (geometry wing)
           :documentation "The Reference Area of the Wing."
           :TeX-name "$S_{ref}$"
           :order-of-magnitude 261.0
           :lower-value 220.0
           :upper-value 300.0
           :dimensions "12"
           :default-units "ft2")


(pa-defun DF-1
          :category weights
          :computed-variables (GROSS_TAKE-OFF_WEIGHT "lbf")
          :input-variables ((PAYLOAD_WEIGHT "lbf")
                            (FUEL_WEIGHT "lbf")
                            (EMPTY_WEIGHT_FRACTION ""))
          :function-body (/ (+ PAYLOAD_WEIGHT FUEL_WEIGHT)
                            (- 1 EMPTY_WEIGHT_FRACTION))
          :TeX-name "$W_{gto} {} = {} {{W_p + W_f} \\over {1 - f_e}}$"
          :documentation "Gross Take-off Weight Equation.")
```

Figure 2.1: Example design variable and design function declarations.

```
                                CRUISE

-->RANGE                    G  3000.0000 sm
   CRUISE_VELOCITY          G   565.0000 sm hr-1
   TSFC                     G     0.8000 lb lbf-1 hr-1
   LIFT-TO-DRAG_RATIO       G    15.0000
   GROSS_TAKE-OFF_WEIGHT    G 15000.0000 lbf
   MIN_LANDING_WEIGHT       G 11000.0000 lbf
   TIME_ON_RESERVES         G     0.7500 hr







(PF1->Process 2->Float 3->Freeze 4->Exit)  Value:
```

Figure 2.2: Example tableau as it first appears.

or her knowledge. As long as the number of unknowns equals the total number of values computed by all the design functions, *Paper Airplane* should then be able to build a computational agenda to solve for all the unknowns. If the number of unknowns are less than this total, as in an **overconstrained** problem, *Paper Airplane* would eventually come across an overconstrained design function it could not solve for; and if the number of unknowns are greater than this total, as in an **underconstrained** problem, *Paper Airplane* would eventually come across an underconstrained design function it could not solve for or a loop it could not close, and thus not solve either.

Once the design path is selected, the user then provides initialized values for the knowns and guess values for the unknowns to define the **initial design point** of the design set. The user then instructs *Paper Airplane* to process the design set to find the true values for the unknowns.

Paper Airplane processes the design set in combinations of three techniques:

1. It may evaluate a design function *directly*, if all output values of the design function are unknown, and all input values are known. This one-time single-function evaluation is called **forward computation.**

2. It may *invert* a design function, if a number of output values of the design function are known, and the *same* number of input values are unknown. *Paper Airplane* will attempt to *numerically* invert that design function by repeatedly evaluating it in order to find the values of the unknowns. Almost always, this will be successful, and the values of the unknowns will be obtained. This iterative single-function evaluation is called **reverse computation.**

3. It may *iterate* a set of design functions in a loop, if those design functions form a closed loop containing several interdependent unknowns. *Paper Airplane* will repeatedly guess values for a chosen forcing variable until the two computed values of a loop variable converge; thereby stabilizing the values of all the unknowns of the design functions in the loop. This iterative multiple-function evaluation is called **loop computation.**

After *Paper Airplane* builds the computational agenda to solve the design path, it then uses numerical methods to find a numerical solution to the initial design point. If a solution is found, *Paper Airplane* will update the values of the unknowns and make them computed knowns, as shown in Figure 2.3. (Note that *Paper Airplane* also informs the user when values obtained in the solution fall outside the recommended limits.)

Now that the user has one solution, design variable values can be changed to form new initial design points to find more solutions as part a trade study or design optimization. (Automatic trade study and optimization features have yet to be incorporated into *Paper Airplane*; however, they are matters under research.) Design variable states can also be changed to form new design paths to find new computational agendas and lead to new types of solutions, trade studies, and optimizations.

2.4 The MISO Design Set Benchmark

In order to test the future modifications to *Paper Airplane*, a benchmark needed to be established. The design set used for this purpose was to have only MISO design functions with none of them calling external codes. This **MISO Design Set** already existed as the conceptual aircraft design set used for a tutorial in the *Paper Airplane User's Manual* [10].

Appendix A contains a list of the 17 design variables and 7 design functions comprising the MISO Design Set. Appendix B contains the complete listing of the MISO Design Set source file loaded by *Paper Airplane*. The benchmark test case would be for *Paper Airplane* to compute the weights and aerodynamic properties of an aircraft given its geometry and performance properties.

A design path was chosen so that the design set was perfectly constrained; that is, so that the number of unknowns equaled the total number of values computed by all the design functions; and then values were provided. The initial design point is shown in Figure 2.4 which contains a list of the design variables, their states, and their values.

The design set was then processed. Appendix C contains the complete listing of the documentation produced by *Paper Airplane* as it first built the computational agenda to find the solution to the chosen design path and then found the numerical solution to the initial design point. Figure 2.5 shows a brief summary of the computational agenda and Figure 2.6 shows the **final design point**, the states and values of the design variables after processing has been completed. (Note that the G-states of the unknowns have become C-states.) This numerical solution established the benchmark to which all other initial tests were compared.

```
                              CRUISE

   RANGE                    I  3000.0000 sm
-->CRUISE_VELOCITY          I   565.0000 sm hr-1
   TSFC                     I     0.8000 lb lbf-1 hr-1
   LIFT-TO-DRAG_RATIO       C    15.0923
   GROSS_TAKE-OFF_WEIGHT    C 20279.2698 lbf        above suggested upper value
   MIN_LANDING_WEIGHT       C 15304.4805 lbf        above suggested upper value
   TIME_ON_RESERVES         I     1.0000 hr


(PF1->Process 2->Float 3->Freeze 4->Exit)  Value:
Building agenda ...  Agenda construction completed.
Processing forced path ...
Processing GROSS_TAKE-OFF_WEIGHT/DRAG_COEFFICIENT loop ....
```

Figure 2.3: Example tableau after design set processing.

```
                         LIST OF INTERNED VARIABLES

F T      O.O.M.      WT  ST   VARIABLE NAME      CURRENT VALUE          INCOMP'S
------------------------------------------------------------------------------
  T      8.0000          I  ASPECT_RATIO            8.0000 sm
  T    565.0000          I  CRUISE_VELOCITY       565.0000 sm hr-1
  T 13300.0000           G  CRUISE_WEIGHT       13300.0000 lbf
  T      0.0200          G  DRAG_COEFFICIENT        0.0200
  T      0.6000          I  EMPTY_WEIGHT_FRACT      0.5500
  T   4000.0000          G  FUEL_WEIGHT          4000.0000 lbf
  T 15000.0000           G  GROSS_TAKE-OFF_WEI  15000.0000 lbf
  T     15.0000          G  LIFT-TO-DRAG_RATIO     15.0000
  T      0.3000          G  LIFT_COEFFICIENT        0.3000
  T 11000.0000           G  MIN_LANDING_WEIGHT  11000.0000 lbf
  T      0.8000          I  OSWALD_EFFICIENCY       0.8000
  T   2200.0000          I  PAYLOAD_WEIGHT       2200.0000 lbf
  T   3000.0000          I  RANGE                3000.0000 sm
  T      0.7500          I  TIME_ON_RESERVES        0.7500 hr
  T      0.8000          I  TSFC                    0.8000 lb lbf-1 hr-1
  T    261.0000          I  WING_REFERENCE_ARE    250.0000 ft2
  T      0.0150          I  ZERO-LIFT_DRAG_COE      0.0150

--Pause--
```

Figure 2.4: The initial design point of the MISO Design Set benchmark.

2.5 The Code of Paper Airplane

Paper Airplane is written in NIL, an object-oriented dialect of COMMON LISP, a computer programming language best suited for handling information not necessarily numerical. In this way, design variables, design functions, design sets, and computational agendas can be as easily represented as numerical arrays are represented in other computer programming languages such as FORTRAN and PASCAL.

One special type of object used frequently in *Paper Airplane* programming is the **flavor**, a powerful abstraction that allows for information storage and retrieval and data communications, all in a hierarchical structure.

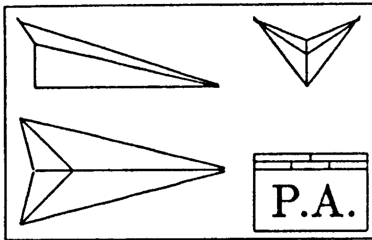
This author welcomed the opportunity to expand his computer programming skills by using such a language to solve the MIMO design function and external code interface capability problems. In fact, systems written in object-oriented computer programming languages are the best candidates for the computer-based engineering model information-sharing systems now required by the engineering community.

```
                          AGENDA for design LASER

    AGENDA ENTRY       DESIGN VARIABLE    COMPUTED BY DESIGN FUN   DIRECTION
-------------------- ------------------ ------------------------ -----------

Forced Path          <NO ENTRIES>

Loop 1: Initial Path GROSS_TAKE-OFF_WEI DF-1                       FORWARD
Loop 1: Initial Path MIN_LANDING_WEIGHT DF-3                       FORWARD
Loop 1: Initial Path LIFT-TO-DRAG_RATIO DF-4                      *REVERSE*
Loop 1: Initial Path CRUISE_WEIGHT      DF-2                       FORWARD
Loop 1: Initial Path LIFT_COEFFICIENT   DF-6                       FORWARD

Loop 1: Branch 1     DRAG_COEFFICIENT   DF-7                       FORWARD

Loop 1: Branch 2     DRAG_COEFFICIENT   DF-5                      *REVERSE*

Loop 1: Final Path   <NO ENTRIES>



--Pause--
```

Figure 2.5: The computational agenda to find the MISO Design Set benchmark.

```
                        LIST OF INTERNED VARIABLES

F T      O.O.M.     WT  ST   VARIABLE NAME      CURRENT VALUE         INCOMP'S
------------------------------------------------------------------------------
  T      8.0000          I  ASPECT_RATIO           8.0000 sm
  T    565.0000          I  CRUISE_VELOCITY      565.0000 sm hr-1
+ T  13300.0000          C  CRUISE_WEIGHT      14666.2540 lbf
+ T      0.0200          C  DRAG_COEFFICIENT       0.0177
  T      0.6000          I  EMPTY_WEIGHT_FRACT     0.5500
  T   4000.0000          C  FUEL_WEIGHT         5176.2540 lbf
+ T  15000.0000          C  GROSS_TAKE-OFF_WEI 16391.6713 lbf
+ T     15.0000          C  LIFT-TO-DRAG_RATIO    13.1132
+ T      0.3000          C  LIFT_COEFFICIENT       0.2315
+ T  11000.0000          C  MIN_LANDING_WEIGHT 11856.0726 lbf
  T      0.8000          I  OSWALD_EFFICIENCY      0.8000
  T   2200.0000          I  PAYLOAD_WEIGHT      2200.0000 lbf
+ T   3000.0000          I  RANGE               3000.0000 sm
  T      0.7500          I  TIME_ON_RESERVES       0.7500 hr
  T      0.8000          I  TSFC                   0.8000 lb lbf-1 hr-1
  T    261.0000          I  WING_REFERENCE_ARE   250.0000 ft2
  T      0.0150          I  ZERO-LIFT_DRAG_COE     0.0150

--Pause--
```

Figure 2.6: The final design point of the MISO Design Set benchmark.



Chapter 3

The External Code Interface

This chapter describes the research, development, and testing of the incorporation of the external code interface capability modification into *Paper Airplane*.

3.1 Research

An **external code interface** from one program to another allows both programs to share information without the need for a human to manipulate input and output files. The external code interface of *Paper Airplane*, a LISP-based code, allows it pass input information to a non-LISP-based external code (**XCODE** for short), execute the code, then retrieve the output information — without user intervention. This capability greatly expands the domain of information accessible to *Paper Airplane*.

The "New" Implementation of LISP

The first step in getting *Paper Airplane* to being able to execute external codes was in getting a NIL function to execute a computer operating system command on a Digital Equipment Corporation VAX/750 running VAX/VMS Version 3.7.

NIL, for "New Implementation of LISP," was actually an *old* implementation of LISP and had recently become a dead language. Fortunately, NIL did have much in common with COMMON LISP and the *Paper Airplane* programmers had taken much care to insure that COMMON LISP was used wherever and whenever possible. One major exception is *Paper Airplane*'s use of flavors, currently not part of COMMON LISP, but part of NIL and ZetaLISP, another COMMON LISP dialect (used by the *LISP Machine*

and the Texas Instruments *Explorer*). In fact, over 99.9% of the current code is compatible with the *Explorer*.

NIL was developed by computer scientists at the Massachusetts Institute of Technology; as such, it was not supported by the Digital Equipment Corporation and thus they did not provide a library of internal system functions, such as the ones easily called by VAX FORTRAN and VAX PASCAL.

This problem was identified and partially combated in 1983 by Dr. John Pararas [15], a Flight Transportation Laboratory researcher, for his research into the improvement of air traffic control systems. The NIL system communication package he created allows NIL to execute VAX system functions but not VAX library functions. Because of this, the external code interface research was steered away from using LIB$SPAWN, a VAX library function that creates a monitored sub-process, and towards SYS$CREPRC, a VAX system function that creates an unmonitored independent process.

The Influence of Preliminary Design Tool

While working for Boeing Aerospace, the author was involved in the Preliminary Design Tool Development Project, a project very similar to the *Paper Airplane* Project. One of the major issues addressed was how information would be passed to and from external codes. It was decided early on that the external codes themselves would not be modified in any way. This sacredness arose from several issues:

1. Users might no longer have trusted the results of a modified code.
2. The source code might not have been accessible, as was the case for most licensed software.
3. The task of modifying every code would have been enormous.

To pass information to and from an external code therefore became a matter of connecting to its standard input and output (I/O) channels. For most computer programs, these come in two forms, file I/O and terminal I/O. Since terminal I/O can be easily diverted to file I/O on most computer systems, it was decided to use ASCII data files for all information passing. This technique, which would be adopted for use in creating *Paper Airplane*'s external code interface, is shown in Figure 3.1.

The technique shown can be explained as follows. Data is passed from the user to the system where it is then formatted into one or more input files via the external code's **preprocessor**. The system then executes the external code. The external code reads the

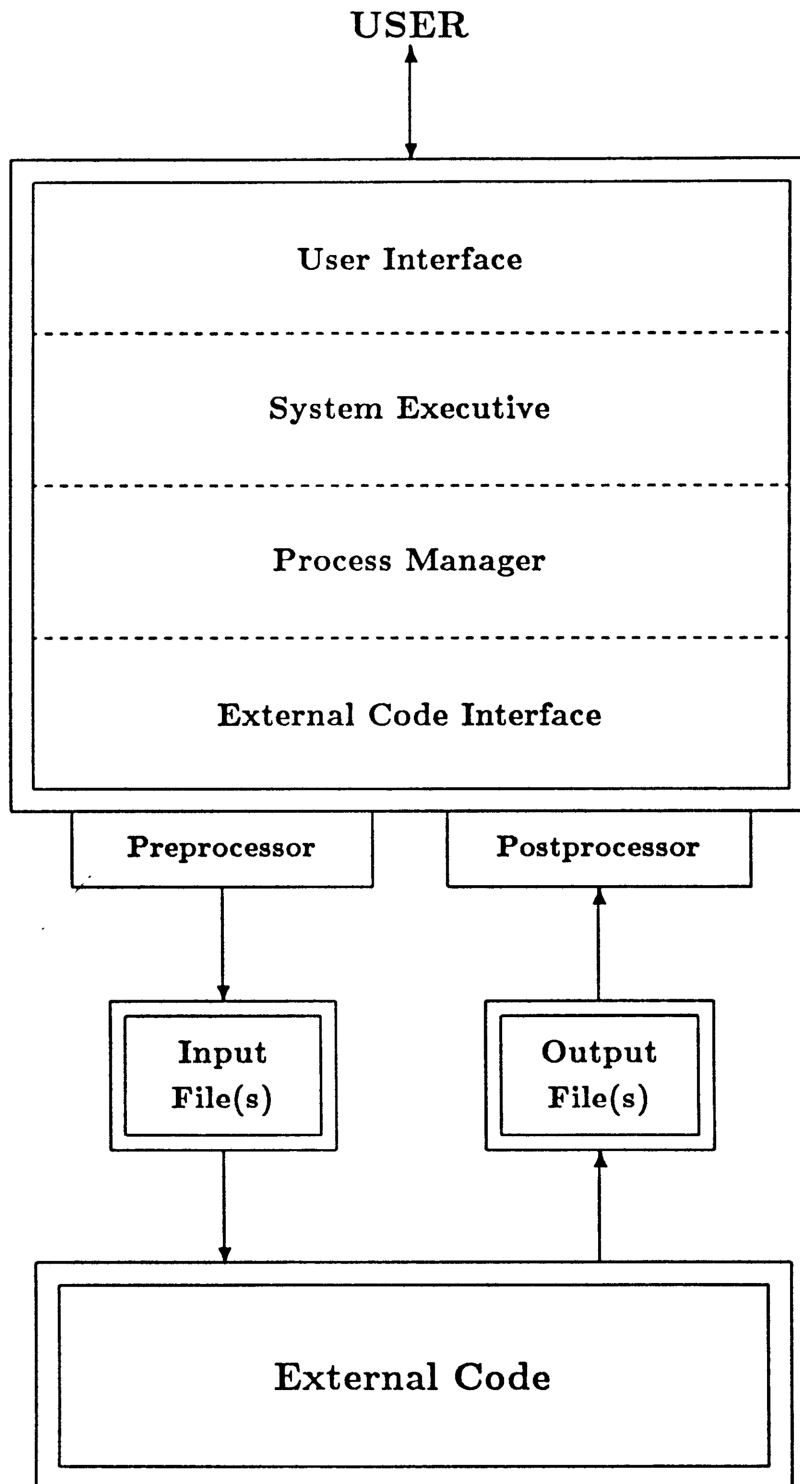


Figure 3.1: Data passing technique used by Boeing's Preliminary Design Tool.

```
(defsyscall ($creprc sys$creprc)
  (pidadr :out :bits)                 ; Address of longword of PID
  (image  :in  :string :required)     ; Maximum 63 characters COMMAND
  (input  :in  :string)               ; Maximum 63 characters SYS$INPUT FILE
  (output :in  :string)               ; Maximum 63 characters SYS$OUTPUT FILE
  (error  :in  :string)               ; Maximum 63 characters SYS$ERROR FILE
  (prvadr :in  :bits)                 ; Address of 64-bit mask PRIVILEGES
  (quota  :in  :bits)                 ; Address of list of QUOTA values
  (prcnam :in  :string)               ; Maximum 15 characters PROCESS NAME
  (baspri :in  :long)                 ; Value of BASE PRIORITY 0-31
  (uic    :in  :bits)                 ; Value of UIC 0-31 (0=subprocess)
  (mbxunt :in  :word)                 ; Mailbox Unit number
  (stsflg :in  :long))                ; 32-bit status flag


(defun execute-program (executable infile outfile errfile process)
  (delete-file errfile nil)
    ($creprc image  executable
             input  infile
             output outfile
             error  errfile
             prcnam process
             baspri 4))
```

Figure 3.2: The SYS$CREPRC definition macro and test function.

data from the input file(s) and writes its data to one or more output files. The system then reads the results from the output file(s) via the external code's **postprocessor** and presents them to the user.

3.2 Development

To develop the NIL version of the Preliminary Design Tool's external code interface, the author had to first develop a COMMON LISP function that could create a new process on the system. As mentioned in the Research section, because of the limitations of NIL, only VAX system functions could be called.

Figure 3.2 shows the NIL macro call function needed to define the SYS$CREPRC command, plus "EXECUTE-PROGRAM," a COMMON LISP function to test it. Several iterations were necessary in order to get all the "`:bits`" and "`:word`" declarations in the right places; but eventually the tests proved successful.

Once tested and ready for use, the "EXECUTE-PROGRAM" function needed to be em-

bedded into another function, "RUN-PROGRAM," that could write the input files, run the program, somehow find out when the program terminated, and then read the output files. Since the preprocessing and postprocessing of an external code's information would be specific to each code, these would have to be functions just called by "RUN-PROGRAM." The main problem, then, was in determining a method for finding out when the program terminated, since the SYS$CREPRC command would only initiate the program and not wait for it to complete.

A COMMON LISP function called "PROBE-FILE" will return "true" if it can open a file and "false" if it cannot. Since a process can write to SYS$OUTPUT at any time, a file receiving SYS$OUTPUT will always be locked as long as the process is running; and a locked file cannot be opened by any other process. When the SYS$OUTPUT-directed file is unlocked, therefore, the process has terminated. By continuously calling "PROBE-FILE" on the SYS$OUTPUT-directed file, "RUN-PROGRAM" would immediately know when the program had terminated. This kind of monitoring wastes CPU needed by the executing program, however.

A NIL function called "SLEEP" puts the process to sleep for a number of seconds. If the average execution time of the program could be computed, *Paper Airplane* could be put to sleep for 80% of it, then begin monitoring and sleeping at intervals of 5% up to some predetermined overtime allowance limit.

The final version of "RUN-PROGRAM," shown in Appendix D, combined all these techniques into a user-friendly function that can be called by *Paper Airplane* from a user's design function. "RUN-PROGRAM" calls the external code's preprocessor, executes the external code, waits for it for to terminate, then calls the external code's postprocessor.

3.3 Testing

To test the external code interface capability, the the Bréguet Range Equation was turned into an extremely short FORTRAN program, as shown in Appendix A, and the MISO Design Set design function "DF-4" was modified to call it, as shown in Figure 3.3. This created what will be called the **XCODE Design Set**.

Since *Paper Airplane* allows the user to switch between several design sets during a session, the original MISO Design Set was loaded in addition to the XCODE Design Set so that processing time could be compared. In each run, the initial design point was set up exactly as shown in Figure 2.4. The agenda for the solution to the corresponding design

```
(pa-defun DF-4
          :category (performance cruise)
          :computed-variable (RANGE "sm")
          :input-variables ((CRUISE_VELOCITY "sm hr-1")
                            (TSFC "lb lbf-1 hr-1")
                            (LIFT-TO-DRAG_RATIO "")
                            (GROSS_TAKE-OFF_WEIGHT "lbf")
                            (MIN_LANDING_WEIGHT "lbf"))
          :inversion-intervals 6
          :function-body
          (progn
            (range-preprocessor CRUISE_VELOCITY TSFC LIFT-TO-DRAG_RATIO
                                GROSS_TAKE-OFF_WEIGHT MIN_LANDING_WEIGHT)
            (run-program :program-name       'range
                         :program-directory  "sys$user:[ftl.rml.pa.xcode]"
                         :preprocessor       nil
                         :postprocessor      nil
                         :average-run-time   1
                         :overtime-allowance 50)
            (range-postprocessor))
          :TeX-name "$R {} = {} {V_{Cr} \\over {\\rm TSFC}} {L \\over D} \\log
                     \\bigl({W_{gto} \\over W_{l_{\\rm min}}} \\bigr)$"
          :documentation "Breguet Range Equation.")

(defun range-preprocessor (W_CR TSFC L/D W_GTO W_MIN)
  (delete-file "rml$xcod:range.in" nil)
  (delete-file "rml$xcod:range.out" nil)
  (let ((infile (open "rml$xcod:range.in" 'out)))
    (unwind-protect
        (progn
          (format infile "~f~%~f~%~f~%~f~%~f~%" W_CR TSFC L/D W_GTO W_MIN))
      (close infile))))

(defun range-postprocessor ()
  (let ((RANGE nil))
    (with-open-file (outfile "rml$xcod:range.out")
       (setq RANGE (read outfile))
    RANGE)))
```

Figure 3.3: The modified design function "DF-4."

path, as is shown in Figure 2.5, placed the design function "DF-4" in a loop. The agenda also chose the reverse computation method to numerically invert the design function each time it would be evaluated. In a sense, therefore, this would be a worst-case analysis of the external code interface, since the external code would be run again and again in an inversion loop within an iteration loop.

The MISO Design Set was solved in 90 seconds; the XCODE Design Set, however, was solved in a shocking 54 *minutes*. The success of being able to communicate with an external code and still converge exactly to the benchmark was overshadowed by the failure of being able to do so quickly. Disturbed by this result, the MISO and XCODE design functions were examined more closely using the *Paper Airplane* Design Function Exerciser, a feature modified by the author to allow a single design function to be processed in either forward or reverse directions (see Figure 3.4). This would give the time to invert the function once and thus eliminate the outer loop.

The MISO design function was processed in 4 seconds (including screen-refresh); the XCODE design function, in 78. The latter was again examined more closely and was found to execute once every 4 seconds; thus each design function was evaluated 18 times. Thinking that the 0.1% convergence criteria was responsible, it was relaxed to 1.0%, and then to 10% — the processing time dropped only to 74 seconds. Clearly something was wrong.

Modifications to the Numeric Processor

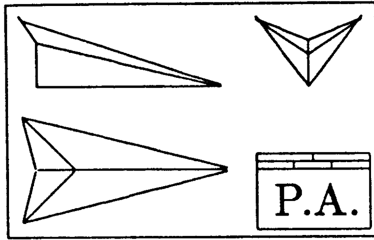
It turned out that nothing was "wrong," just inefficient. After examining the ways in which *Paper Airplane* numerically solved both single design functions (via the Design Function Solver) and loops of them (via the Loop Solver), it was found that both methods could be improved upon which could reduce processing time for all design functions, whether calling external codes or not (see Chapter 5). What was surprising was the magnitude of the improvements. By modifying the Design Function Solver, total processing time was reduced by a factor greater than 4. By modifying the Loop Solver, total processing time was further reduced by a factor of 6, making the total reduction a factor of 25.

With the modifications in place, the MISO Design Set was solved almost instantaneously, and the XCODE Design Set in about 2 minutes, a much more reasonable time frame. The external code interface was then both successful in getting data to and from an external code and doing so quickly. Also, by adding the external code interface capa-

```
                                                                1/1
                     Function DF-4 calculating 1 variable

    Variable Name      Test State/Value  System State/Value  Current Units
  -----------------   ----------------  ------------------  ---------------
   CRUISE_VELOCITY     I        565.0    I          565.0    sm hr-1
   TSFC                I          0.8    I            0.8    lb lbf-1 hr-1
-->LIFT-TO-DRAG_RATI   C   13.6956960    G           15.0
   GROSS_TAKE-OFF_WE   I      15000.0    G        15000.0    lbf
   MIN_LANDING_WEIGH   I      11000.0    G        11000.0    lbf
   RANGE               I       3000.0    I         3000.0    sm




(PF1->Process 2->Float 3->Freeze 4->Exit)  Value:
Function DF-4 has 5 base variables and 1 derived variable.
The function path is perfectly constrained.
Processing function DF-4 ...
```

Figure 3.4: Example display of Design Function Exerciser after processing.

bility to *Paper Airplane*, inefficiencies in the Numerical Solvers that would have otherwise gone unnoticed had been corrected making *Paper Airplane*, in general, 25 times more efficient at solving problems.



Chapter 4

MIMO Design Functions

This chapter describes the research, development, and testing of the incorporation of the MIMO design function capability modification into *Paper Airplane*. This task was divided into two sub-tasks: the creation of a simple MIMO design function capability and the creation of the general MIMO design function capability.

4.1 Simple MIMO Capability

This section details the research, development, and testing of the simple MIMO design function capability.

4.1.1 Research

Simple MIMO capability would be one in which a multiple-input multiple-output (MIMO) design function would be solved for in the same manner as for a multiple-input single-output (MISO) design function. Specifically, this would mean one of two cases:

1. **0UI-0KO**: (zero unknown inputs, zero known outputs). In this case, no input value to the design function is unknown and no output value is known (i.e., all input values are known and all output values are unknown). For either MISO or MIMO design function, the solution here is to use the forward computation method.

2. **1UI-1KO**: (one unknown input, one known output). In this case, one input value to the design function is unknown and one output value is known (i.e., all but one input values are known and all but one output values are unknown). For either

MISO or MIMO design function, the solution here is to use the reverse computation method.

Since a MISO design function only computes one output value, these are all of its cases. Odd cases such as 0UI-1KO and 2UI-0KO cannot be solved by the forward or reverse computation method since they are overconstrained and underconstrained, respectively. An overconstrained design function cannot be solved for at all; an underconstrained design function can be solved for, however, by the loop computation method if other design functions are found that also involve the same unknowns.

For the general case of mUI-nKO, therefore, the MISO version of *Paper Airplane* could solve for cases with $n \leq 1$. Cases with $n > m$ would be overconstrained and cases with $n < m$ would be underconstrained. Simple MIMO capability would thus solve for the set of cases with $n = m \leq 1$. General MIMO capability, discussed in the next section, would solve for the set of cases with $n = m \geq 2$.

The reason why n must equal m for a perfectly constrained function is found in mathematics. As long as the total number of unknowns in a functional relationship equals the number of its output values, at least one solution can be found. If U is the total number of unknowns, then $n = m \leq 1$ can also be written as $U = 1$, and $n = m \geq 2$ as $U > 1$.

Since the simple MIMO capability would use the already existing forward and reverse computation methods to solve for its MIMO design functions, the research would concentrate on the incorporation of the MIMO design functions themselves into part of the family of *Paper Airplane* objects. In order to do this, it became necessary to create a new type of design function flavor, since, at that time, it was decided not to alter the structure of the existing MISO design function flavor.

An Introduction to COMMON LISP Flavors

As mentioned in the Chapter 2, a flavor is a powerful LISP object that allows for information storage and retrieval and data communications, all in a hierarchical structure.

A flavor, such as the MISO design function flavor (internally called a "DESIGN-FUNCTION"), consists of many sub-flavors called **mixins**. The role of mixins is to set up a hierarchy whereby higher-ordered flavors, although perhaps very different, could use the same structure of attributes (called **instance variables**) and could call the same flavor-specific functions (called **methods**).

This mouthful can be better explained by an example. Take two higher-ordered flavors, "BOEING-747" and "NORTHROP-F-20." Whatever attributes the two have in common could be assigned to a mixin called "AIRCRAFT." Instance variables of this sub-flavor would include "WING-SPAN," "NOSE-LENGTH," and "AVIONICS-WEIGHT." Attributes not common to both would be flavor-specific. For example, the "BOEING-747" flavor would need an instance variable for "PASSENGER-CAPACITY" while the "NORTHROP-F-20" flavor would need one for "AFTERBURNER-TYPE." Similarly, methods can be divided into common and non-common groups. Common methods would include ":TAKE-OFF," ":COME-TO-HEADING," and ":CHANGE-ELEVATOR-ANGLE." Non-common methods would include ":TURN-ON-NO-SMOKING-LIGHT" for the "BOEING-747" and ":LAUNCH-MISSILE" for the "NORTHROP-F-20."

The "AIRCRAFT" mixin may itself have a mixin called "VEHICLE" with instance variables such as "ENGINE-TYPE" and "FUEL-WEIGHT" and with methods such as ":START-ENGINE" and ":CHECK-FUEL-GAUGE." And "VEHICLE may have a mixin called "OBJECT" with instance variables such as "NAME and "LENGTH" and with methods such as ":MOVE" and ":DESCRIBE-SELF."

This somewhat linear hierarchy is by no means the limit. For example, the "AIRCRAFT" mixin might have other mixins in addition to "VEHICLE" such as "AUTO-PILOT" and "LANDING-GEAR." These mixins better organize the information contained by "AIRCRAFT."

What the user of flavors ends up with, therefore, is a tree-structured hierarchy of information about an object and the tasks it can perform.

4.1.2 Development

Most of the development of the simple MIMO capability involved the creation of a new MIMO design function flavor to be called a "MIMO," and its associated methods. Both flavor structure and methods were "MIMO-ized" versions of those for the MISO design function flavor "DESIGN-FUNCTION." The instance variables of "MIMO," grouped by their mixins, are listed below:

- MIMO-DEFINITION-MIXIN
    1. LISP-NAME
    2. COMPUTED-VARIABLES
    3. COMPUTED-TO-INTERNAL-CONVERSIONS

4. INTERNAL-TO-INPUT-CONVERSIONS

- PA-FUNCTION-DEFINITION-MIXIN
  1. EXPR
  2. PARAMETER-LIST
- LIBRARY-INSTANTIATION
  1. DEFINITION
- PA-OBJECT
  1. NAME
  2. TEX-NAME
  3. DOCUMENTATION

"COMPUTED-VARIABLES" is the list of output design variables while "PARAMETER-LIST" is the list of input design variables. The two "CONVERSIONS" instance variables convert input and output variable values to and from the internal SI-standardized units, respectively. "EXPR" is the body of the design function itself. Both "LISP-NAME" and "NAME" are LISP symbols used for referencing the design function while "TEX-NAME" is a special formatted version of the function name or body for use with LaTeX, the document processing package language with which this paper was prepared.

In addition to instance variables, the flavor "MIMO" also has many methods associated with it. These can be grouped into the following categories:

- Library Incorporation and Information Methods
- Design Function Establishment and Processing Methods
- Design Set Incorporation and Processing Methods

The Library methods pertain to the incorporation of the MIMO design function into the *Paper Airplane* Library, a storehouse of all the user's design variables and design functions. The Design Function methods pertain to the establishment of the MIMO design function as an individual design function and how it is solved for individually (i.e., using the forward and reverse computation methods). Finally, the Design Set methods pertain to the incorporation of the MIMO design function into a design set and how it is solved for in terms of agenda building and numerical processing.

Most of the conversion of the "DESIGN-FUNCTION" methods into "MIMO" methods was of the "copy & edit" type. More hard work than hard thinking was involved; both MISO and MIMO design function flavors were very similar. They were so similar, in fact, that the author finally convinced his colleagues that the two should be merged. This was done between the creation of the simple and general MIMO capabilities.

4.1.3 Testing

To test the simple MIMO capability, the MISO Design Set design functions "DF-6" and "DF-7" were merged to form the MIMO design function "Aerodynamics Package." Another MIMO design function, "Geometry Package" was also added. This created the need for an additional design variable, "WING_SPAN," as shown in Appendix A. This created what will be called the **MIMO Design Set**.

The initial design point was set up almost exactly as shown in Figure 2.4; however, the presence of "WING_SPAN" required a slightly different design path to be selected. To properly constrain the design set, "OSWALD_EFFICIENCY" was selected as an additional, but calculable, unknown. The slightly different computational agenda (a forced path existing where none had before) is shown in Figure 4.1. The rest of the results very closely followed those shown in Appendix C and the computed solution was almost exactly the same.

Paper Airplane then had a limited capability for dealing with MIMO design functions. As a check that both the external code interface and simple MIMO design function capabilities could work together, the design sets were merged (i.e., the MIMO Design Set design function "DF-4" was replaced with that of the XCODE Design Set) and then retested. As expected, the solution was almost exactly the same as that of the MISO Design Set.

4.2 General MIMO Capability

This section details the research, development, and testing of the general MIMO design function capability.

4.2.1 Research

General MIMO capability would be one in which a multiple-input multiple-output (MIMO) design function would *not* be solved for in the same manner as for a multiple-

```
                         AGENDA for design LASER

    AGENDA ENTRY         DESIGN VARIABLE    COMPUTED BY DESIGN FUN   DIRECTION
-------------------- ------------------- ------------------------ -----------


Forced Path           WING_SPAN            Wing Geometry Package    *REVERSE*
Forced Path           OSWALD_EFFICIENCY    Wing Geometry Package    *REVERSE*

Loop 1: Initial Path  FUEL_WEIGHT          DF-1                     *REVERSE*
Loop 1: Initial Path  MIN_LANDING_WEIGHT   DF-3                      FORWARD
Loop 1: Initial Path  LIFT-TO-DRAG_RATIO   DF-4                     *REVERSE*
Loop 1: Initial Path  CRUISE_WEIGHT        DF-2                      FORWARD

Loop 1: Branch 1      DRAG_COEFFICIENT     Aerodynamics Package      FORWARD
Loop 1: Branch 1      LIFT_COEFFICIENT     Aerodynamics Package      FORWARD

Loop 1: Branch 2      DRAG_COEFFICIENT     DF-5                     *REVERSE*

Loop 1: Final Path    <NO ENTRIES>



--Pause--
```

Figure 4.1: The agenda for the solution to the design path of the MIMO Design Set.

input single-output (MISO) design function. Referring back to the previous section, this would mean the general case of mUI-nKO, and specifically the set of cases with $n = m \geq 2$.

With the flavor structure of the MIMO design function already established, the general MIMO capability had merely to add the necessary design function methods to solve for the above stated cases. As this was a problem in numerical methods of mathematics, the author visited the M.I.T. Mathematics Department and conversed with Prof. Richard Dudley [2] who steered the author towards the proper reference material.

Of all the reference material examined, by far the most helpful was the text *Elementary Numerical Analysis: An Algorithmic Approach* by S. D. Conte [1]. On page 217, Conte describes what would become the Vector Newton-Raphson Method of *Paper Airplane*:

> **Algorithm 5.3: Newton's method for a system** Given the system
>
> $$\mathbf{f}(\xi) = \mathbf{0}$$
>
> of n equations and n unknowns, with $\mathbf{f}$ a vector valued function having smooth components, and a first guess $\mathbf{x}^{(0)}$ for a solution of ξ of the system.
>
> For $m = 0, 1, 2, \ldots$, until satisfied, do:
> $\mathbf{x}^{(m+1)} := \mathbf{x}^{(m)} - \mathbf{f}'(\mathbf{x}^{(m)})^{-1}\mathbf{f}(\mathbf{x}^{(m)})$
>
> It can be shown that Newton's method converges to ξ provided $\mathbf{x}^{(0)}$ is close enough to ξ and provided the Jacobian $\mathbf{f}'$ of $\mathbf{f}$ is continuous and $\mathbf{f}'(\xi)$ is invertible. . . .

To solve general cases of MIMO design functions, the number of unknown inputs must equal the number of known outputs. Since the ***known*** inputs would never change and the ***unknown*** outputs could be solved for later, the MIMO could be said to map a vector of n input values, $\mathbf{x}$, into a vector of n output values, $\mathbf{y}$, or more concise, $\mathbf{y} = \mathbf{g}(\mathbf{x})$.

To solve the problem $\mathbf{y}^* = \mathbf{g}(\mathbf{x}^*)$ via Newton's method, the equation would be rearranged to the following

$$\mathbf{f}(\xi) = \mathbf{g}(\mathbf{x}^*) - \mathbf{y}^* = \mathbf{0}$$

where, obviously, $\mathbf{x}^* = \xi$. The problem would then be reduced to continuously solving the vector equation $\mathbf{f}'(\mathbf{x})\Delta\mathbf{x} = \Delta\mathbf{y}$ for $\Delta\mathbf{x}$ so that the guess vector $\mathbf{x}$ could be directed

towards the solution $\mathbf{x}^*$. This would require use of matrix functions, and especially of a matrix inversion algorithm to invert the Jacobian derivative matrix $\mathbf{f}'(\mathbf{x})$.

Although NIL did not contain matrix functions, John Pararas (the Flight Transportation Laboratory researcher who created the NIL system communications package) had solved this problem as well. After getting familiar with the routines in the Pararas matrix functions package, it was decided that they worked well enough to be used by *Paper Airplane* to solve general cases of MIMO design functions.

4.2.2 Development

Some additional methods were written for the then merged MISO/MIMO flavor "DESIGN-FUNCTION" to test the Vector Newton-Raphson Method on MIMO design function "DF-6" which computes C_L and C_D as a function of W_{Cr}, V_{Cr}, ρ, S_{ref}, C_{D_0}, ϵ, and AR. The methods called upon the Pararas matrix package. The test was then to find the values of W_{Cr} and S_{ref} satisfying given values of C_L and C_D.

Although the Jacobian derivative matrix was properly set up, the matrix package failed to invert it. The problem was that the determinant of the Jacobian was very close to zero (about 10^{-13}) and the inversion algorithm could not accurately handle numbers of such magnitude. All other aspects of the matrix package (which included all vector operations) worked perfectly. The problem was then in finding a better matrix inversion algorithm.

Here again Conte's text provided the solution — a technique that yields the solution to the vector equation $A\mathbf{x} = \mathbf{b}$ without requiring a need to compute the inverse of the matrix A. The text explains why on page 166:

> . . . we hasten to point out that *there is usually no good reason for ever calculating the inverse* . . . whenever A^{-1} is needed merely to calculate a vector $A^{-1}\mathbf{b}$ (as in solving $A\mathbf{x} = \mathbf{b}$) or a matrix product $A^{-1}B$, A^{-1} should never be calculated explicitly. Rather, the substitution Algorithm 4.4 should be used to form these products. . . .

According to Conte, there was no need for the author to invert the Jacobian derivative matrix to solve for $\Delta\mathbf{x}$. Instead, Algorithm 4.4 could be used. This algorithm, which involves forward- and back-substitution, was written into the COMMON LISP function "SOLVE-LU-FACTORIZATION" (shown in Appendix E). Another function, "GET-LU-FACTORIZATION" (also shown in Appendix E) was created based upon a FORTRAN

subroutine in the text on computing the LU-factorization of the matrix A. Both functions were successfully tested against the matrix functions of a Hewlett-Packard 15-C calculator. They were then successfully tested on solving the linear vector equation $\mathbf{f}'(\mathbf{x})\Delta\mathbf{x} = \Delta\mathbf{y}$ for the change in the guess vector $\Delta\mathbf{x}$ based upon the error in the target vector $\Delta\mathbf{y}$ and the function Jacobian $\mathbf{f}'(\mathbf{x})$. The guess vector $\mathbf{x}$ could then be steered towards the solution $\mathbf{x}^*$.

The Damped Vector Newton-Raphson Method

During a debugging session, it was decided to incorporate a more fail-safe method than just the Vector Newton-Raphson Method. Called the Damped Vector Newton-Raphson Method, it was designed to keep the search for the solution from diverging by continuously reducing the calculated change in the guess vector $\Delta\mathbf{x}$ until the new error in the target vector $\Delta\mathbf{y}$ was smaller than the previous one. Conte's text describes it on page 219:

> **Algorithm 5.4: Damped Newton's method for a system** Given the system $\mathbf{f}(\xi) = \mathbf{0}$ of n equations and n unknowns, with $\mathbf{f}$ a vector-valued function having smooth component functions, and a first guess $\mathbf{x}^{(0)}$ for a solution ξ of the system.
>
> For $m = 0, 1, 2, \ldots$, until satisfied, do:
>
> $$\mathbf{h} := -\mathbf{f}'(\mathbf{x}^{(m)})^{-1}\mathbf{f}(\mathbf{x}^{(m)})$$
> $$i := \min\left\{j : 0 \le j \le j_{max},\ \|\mathbf{f}(\mathbf{x}^{(m)} + \mathbf{h}/2^j)\|_2 < \|\mathbf{f}(\mathbf{x}^{(m)})\|_2\right\}$$
> $$\mathbf{x}^{(m+1)} := \mathbf{x}^{(m)} + \mathbf{h}/2^i$$
>
> with j_{max} chosen a priori, for example, $j_{max} = 10$.

where, for example, $\| \mathbf{v} \|_2$ would be the two-norm, or magnitude, of vector $\mathbf{v}$. The Damped Vector Newton-Raphson Method was incorporated into *Paper Airplane* and became the *Paper Airplane* MIMO Solver. The general MIMO design function capability was then ready for formal testing.

4.2.3 Testing

Before testing the general MIMO capability on an entire design set, it was decided to use the *Paper Airplane* Function Exerciser and test it on a single MIMO design function.

For this purpose a pure mathematical MIMO design function was created as follows:

$$(x, y, z) = f(a, b, c, d)$$

where a, b, c, and d were the input variables, and x, y, and z were the output variables. x was the sum of the inputs, y was their product, and z was the square root of the sum of their squares.

To test the function, values of the inputs were selected and the corresponding output values were computed. Then, one output variable was set as known, and one input variable as unknown (and its value was changed). *Paper Airplane* used the standard MISO Solver to solve this problem, and the solution was quickly found. Then, two output variables were set as known and two input variables as unknown, and the test was repeated. *Paper Airplane* switched to the MIMO Solver as planned and the solution was quickly found as well. Finally, on the basis of this success, three output variables were set as known, and three input variables as unknown, and the test was repeated. Again, *Paper Airplane* found the solution quickly using the MIMO Solver.

In further tests, guess values for the unknown input variables were set farther apart from their solution values, and the MIMO Solver still converged upon the solution. After going beyond one order of magnitude each way, some tests failed and some succeeded. This should not be taken too seriously, however, since most engineers can guess a solution to a problem to within one order of magnitude. (Most good engineers anyway!)

When known output values were selected randomly (so that the MIMO Solver could compute the unknown input values), some tests succeeded and some failed. This is understandable because some output value combinations probably had complex input values as the exact solution, which *Paper Airplane* could not find. *Paper Airplane* should have found the closest *real* solution, however; and with that in mind the MIMO Solver was redesigned. More tests were conducted, and in the cases where the exact solutions were complex, *Paper Airplane* successfully found the closest real solutions.

The MIMO Design Set Revisited

A more important practical test of the general MIMO capability would be to solve a true MIMO design path from the MIMO Design Set in Appendix A. In this test, aerodynamic and performance properties of an aircraft would be specified and weights and geometry properties would be computed. This so-called "inverse engineering" problem would decide the potential of *Paper Airplane* as an engineering tool.

```
                          LIST OF INTERNED VARIABLES

F T      O.O.M.      WT  ST   VARIABLE NAME     CURRENT VALUE         INCOMP'S
------------------------------------------------------------------------------
  T      8.0000           G  ASPECT_RATIO           8.0000 sm
  T    565.0000           I  CRUISE_VELOCITY      565.0000 sm hr-1
  T 13300.0000            G  CRUISE_WEIGHT      13300.0000 lbf
  T      0.0200           I  DRAG_COEFFICIENT       0.0180
  T      0.6000           I  EMPTY_WEIGHT_FRACT     0.5500
  T  4000.0000            G  FUEL_WEIGHT         4000.0000 lbf
  T 15000.0000            G  GROSS_TAKE-OFF_WEI 15000.0000 lbf
  T     15.0000           G  LIFT-TO-DRAG_RATIO    15.0000
  T      0.3000           I  LIFT_COEFFICIENT       0.2400
  T 11000.0000            G  MIN_LANDING_WEIGHT 11000.0000 lbf
  T      0.8000           G  OSWALD_EFFICIENCY      0.8000
  T  2200.0000            I  PAYLOAD_WEIGHT      2200.0000 lbf
  T  3000.0000            I  RANGE               3000.0000 sm
  T      0.7500           I  TIME_ON_RESERVES       0.7500 hr
  T      0.8000           I  TSFC                   0.8000 lb lbf-1 hr-1
  T    261.0000           G  WING_REFERENCE_ARE   250.0000 ft2
  T     50.0000           G  WING_SPAN             50.0000 ft
  T      0.0150           I  ZERO-LIFT_DRAG_COE     0.0150
--Pause--
```

Figure 4.2: The initial design point for the "inverse engineering" problem.

The initial design point for this test is shown in Figure 4.2. Appendix F contains a complete listing of the documentation produced by *Paper Airplane* as it successfully built a different computational agenda to find the solution to the chosen design path and then successfully found a numerical solution to the initial design point. Figure 4.3 shows a brief summary of the agenda. Note that the last loop involved two uses of the MIMO Solver. Finally, Figure 4.4 shows the successful results of this "inverse engineering" problem, as wing geometry was computed to match specified aerodynamic properties — albeit on a primitive level.

Again, as a check that both the external code interface and general MIMO design function capabilities could work together, the design sets were merged (i.e., the MIMO Design Set design function "DF-4" was replaced with that of the XCODE Design Set) and then retested. As expected, the solution to the "inverse engineering" problem was converged upon exactly as before. Processing time for the MIMO Design Set was 7 seconds, while processing time for the merged MIMO-XCODE Design Set was 74 seconds, a very reasonable amount of time.

Overall, the MIMO Solver and external code interface performed satisfactorily in all tests conducted thus far. The next major test would be the most practical yet, the design of a national aerospaceplane. Chapter 6 reveals the details of the creation and use of the so called NASP Design Set. But first, Chapter 5, describes the modifications made to *Paper Airplane*'s other Numeric Solvers, the Design Function Solver and the Loop Solver.

AGENDA for design LASER

AGENDA ENTRY	DESIGN VARIABLE	COMPUTED BY DESIGN FUN	DIRECTION
Forced Path	LIFT-TO-DRAG_RATIO	DF-5	FORWARD
Loop 1: Initial Path	FUEL_WEIGHT	DF-1	*REVERSE*
Loop 1: Initial Path	CRUISE_WEIGHT	DF-2	FORWARD
Loop 1: Branch 1	MIN_LANDING_WEIGHT	DF-4	*REVERSE*
Loop 1: Branch 2	MIN_LANDING_WEIGHT	DF-3	FORWARD
Loop 1: Final Path	FUEL_WEIGHT	DF-1	*REVERSE*
Loop 1: Final Path	CRUISE_WEIGHT	DF-2	FORWARD
Loop 2: Initial Path	<NO ENTRIES>		
Loop 2: Branch 1	OSWALD_EFFICIENCY	Aerodynamics Package	*REVERSE*
Loop 2: Branch 1	WING_REFERENCE_ARE	Aerodynamics Package	*REVERSE*
Loop 2: Branch 2	OSWALD_EFFICIENCY	Wing Geometry Package	*REVERSE*
Loop 2: Branch 2	WING_SPAN	Wing Geometry Package	*REVERSE*
Loop 2: Final Path	<NO ENTRIES>		

--Pause--

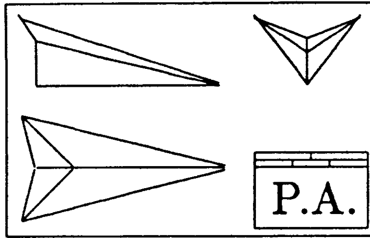
Figure 4.3: The agenda for the solution to the "inverse engineering" problem.

LIST OF INTERNED VARIABLES

F	T	O.O.M.	WT	ST	VARIABLE NAME	CURRENT VALUE	INCOMP'S
	T	8.0000		C	ASPECT_RATIO	7.6518 sm	
	T	565.0000		I	CRUISE_VELOCITY	565.0000 sm hr-1	
	T	13300.0000		C	CRUISE_WEIGHT	14223.0605 lbf	
	T	0.0200		I	DRAG_COEFFICIENT	0.0180	
	T	0.6000		I	EMPTY_WEIGHT_FRACT	0.5500	
	T	4000.0000		C	FUEL_WEIGHT	4941.6203 lbf	
	T	15000.0000		C	GROSS_TAKE-OFF_WEI	15870.2673 lbf	
	T	15.0000		C	LIFT-TO-DRAG_RATIO	13.3333	
	T	0.3000		I	LIFT_COEFFICIENT	0.2400	
	T	11000.0000		C	MIN_LANDING_WEIGHT	11540.2605 lbf	
	T	0.8000		C	OSWALD_EFFICIENCY	0.8000	
	T	2200.0000		I	PAYLOAD_WEIGHT	2200.0000 lbf	
	T	3000.0000		I	RANGE	3000.0000 sm	
	T	0.7500		I	TIME_ON_RESERVES	0.7500 hr	
	T	0.8000		I	TSFC	0.8000 lb lbf-1 hr-1	
	T	261.0000		C	WING_REFERENCE_ARE	233.8814 ft2	
	T	50.0000		C	WING_SPAN	42.3049 ft	
	T	0.0150		I	ZERO-LIFT_DRAG_COE	0.0150	

--Pause--

Figure 4.4: The final design point for the "inverse engineering" problem.



Chapter 5

The Numerical Solvers

This chapter describes the modifications made to the *Paper Airplane* Numerical Solvers, created by Mark Kolb as part of his Master's thesis research [8]. His task was to numerically solve a set of non-linear MISO design functions. Due to this non-linearity, standard linear numerical techniques for solving systems of equations failed to come up with solutions. His research developed non-linear numerical techniques that successfully passed all tests, including those of the author. A method used to solve for individual design functions was incorporated into the Design Function Solver. A second method used to solve for a loop of design functions was incorporated into the Loop Solver. Modifications became necessary when it was discovered that the methods had inefficiencies that caused considerable delay in the processing of design functions calling external codes.

5.1 The Design Function Solver

The method used for the inversion of design functions is a modified Newton-Raphson iteration. This is possible because all but one of the input values are known, thus the problem can be reduced to the single-input single-output equation $y = f(x)$. This type of problem can be easily represented by a two-dimensional curve of y versus x. To solve the problem $y^* = f(x^*)$ via Newton's method (where y^* is the desired value of y and x^* is the value of x that will yield it), the equation would be rearranged to the following:

$$y = g(x) = f(x) - y^*$$

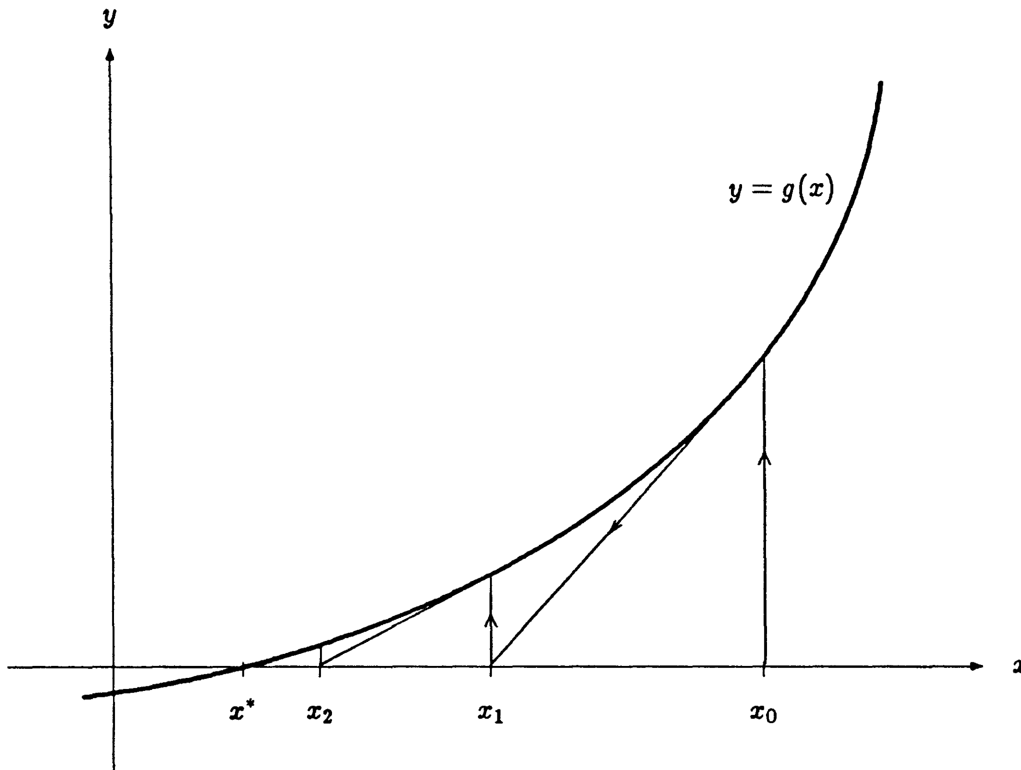


Figure 5.1: The Newton-Raphson technique for locating zeros.

with

$$y = g(x^*) = f(x^*) - y^* = 0$$

Guess values for the unknown x would produce points along the curve of y. The solution x^* would therefore be the intersection of the curve with the x-axis. The function derivative $g'(x)$ would be repeatedly taken to keep directing the next guess for x (starting with x_0) towards x^*. This is shown in Figure 5.1.

According to the original method, before the Newton-Raphson iteration began, an initial search was performed to find a guess value for x computing the point closest to the solution x^*. This was via the Logarithmic-Distribution Method developed for the Loop Solver. The search space was divided logarithmically between the lower and upper value limits of the unknown (the current value of the unknown was also included in the search). The search then proceeded from the lower limit to the upper limit. After the search, the search value x corresponding to the minimum computed value y was chosen to

start the Newton-Raphson iteration. Once the iteration began, the design function was then evaluated three times per iteration, once to compute the point at the guess value and twice more for the calculation of the derivative of the curve at the computed point.

A large initial search space of 15 values was chosen so that the Newton-Raphson iteration would begin at a search value as close to the solution as possible. Unfortunately, this meant that *every* design function would have been evaluated at least 18 times — even if the solution was reentered as the initial guess value. In terms of the XCODE Design Set test from Chapter 3, this meant that the tested design function, which was evaluated 18 times, actually converged after one iteration. The author decided it would be better to relax the initial search and allow the Newton-Raphson method to do its job.

After discussing this with Mark Kolb, it was agreed to relax the number of search values only for design functions calling external codes, but also to add a general check: If any search value x computed a value y within 10% of the desired value y^*, the search would be stopped immediately and that search value x would then be used to begin the Newton-Raphson iteration. It was also agreed to simplify the derivative calculation so that it only required one function evaluation, since the derivative would still be accurate enough to guide the unknown x. By also checking the current value of x first, the minimum number of evaluations would drop from 18 to 1.

With these modifications in place, the XCODE Design Set test was redone. Both MISO and XCODE design functions were again tested with the Design Function Exerciser under this modified technique and both were solved after two iterations following a very brief initial search. The MISO design function was processed in 3 seconds; the XCODE design function, in 17 — a drop of 80% in processing time from its value before the modifications. The MISO and XCODE Design Sets were reset back to their original design paths and then reprocessed. Total processing time was reduced by a factor of 4.5, dropping the MISO time from 90 seconds to 20, and the XCODE time from 54 minutes to 12.

The Search-Outward Method and the Design Function Solver

Despite this success, the fact that design functions calling external codes required a special check bothered the author, because this required adding another instance variable to the "DESIGN-FUNCTION" flavor. So after putting it in, the author quickly decided to take it out.

Instead, a different search method was incorporated. The Search-Outward Method,

Figure 5.2: Comparison of methods with solution near the lower limit.

which was developed for use with the Loop Solver (see next section), replaced the Logarithmic-Distribution Method. The Search-Outward Method searches from the current value of the unknown x towards both upper and lower limits simultaneously, rather than from the lower limit towards the upper limit. Starting at the current value has the benefit of rewarding the user for a good initial guess. It also allows the Design Function Solver to more quickly find the solution to a function inversion inside a loop iteration since the next solution would not vary much from the previous one.

An inversion of the Bréguet Range Equation was chosen to compare the two methods, a sample of which is shown in Figure 5.2. On the x-axis is the choice for the initial guess, W_{gto}, as compared to the actual solution, W^*_{gto}. On the y-axis is the number of design function evaluations, N_{ev}, required to converge upon the solution W^*_{gto}. (This includes the design function evaluations required to perform Newton-Raphson iterations.) The dashed curve (labeled "L-D") shows the number of function evaluations that the Logarithmic-

Distribution Method required to converge upon W^*_{gto} for each initial guess of W_{gto} along the x-axis. The solid curve (labeled "S-O") shows the same for Search-Outward Method.

When the initial guess W_{gto} was chosen randomly (i.e., on average), the Search-Outward Method converged upon the solution W^*_{gto} about as fast as the Logarithmic-Distribution Method did. (The area under each curve gives an indication of this.) On the other hand, when the initial guess W_{gto} approached the solution W^*_{gto}, the Search-Outward Method converged faster.

Since the Logarithmic-Distribution Method always started the search at the same point, the number of design function evaluations N_{ev} changed little outside the immediate vicinity of the solution W^*_{gto}; and since this point was the lower limit, the number of design function evaluations in this outer region rose as the solution tended towards the upper limit. This is shown in Figure 5.3. In fact, with the solution near the upper limit, the Search-Outward Method converged faster than the Logarithmic-Distribution Method for almost all initial guesses W_{gto}. Figure 5.3 also shows that the Logarithmic-Distribution Method required a very good initial guess while the Search-Outward Method required only a fair guess.

The overall modifications to the Design Function Solver allowed to it converge upon the solution to a design function inversion 4.5 times faster than before. While the Search-Outward Method did not improve performance when a random guess was made at the solution to a design function inversion, it did improve performance greatly when a fair guess was made, which would be the case when the design function was part of a loop. The Search-Outward Method also got rid of the need to perform a special check on design functions calling external codes, and thus the need for a new instance variable for the "DESIGN-FUNCTION" flavor.

5.2 The Loop Solver

The technique used for solving a loop of interdependent design functions is an original method. The loop is separated into two paths or branches. Each branch is a sequence of design functions that form a single-input single-output function. A value is guessed for the forcing variable and is fed into both branches, which then compute two values for the loop variable. When the same value is computed by both branches, the loop is solved.

The two branches can be represented as two-dimensional curves with the forcing variable as the ordinate and the loop variable as the abscissa. The solution to the loop is

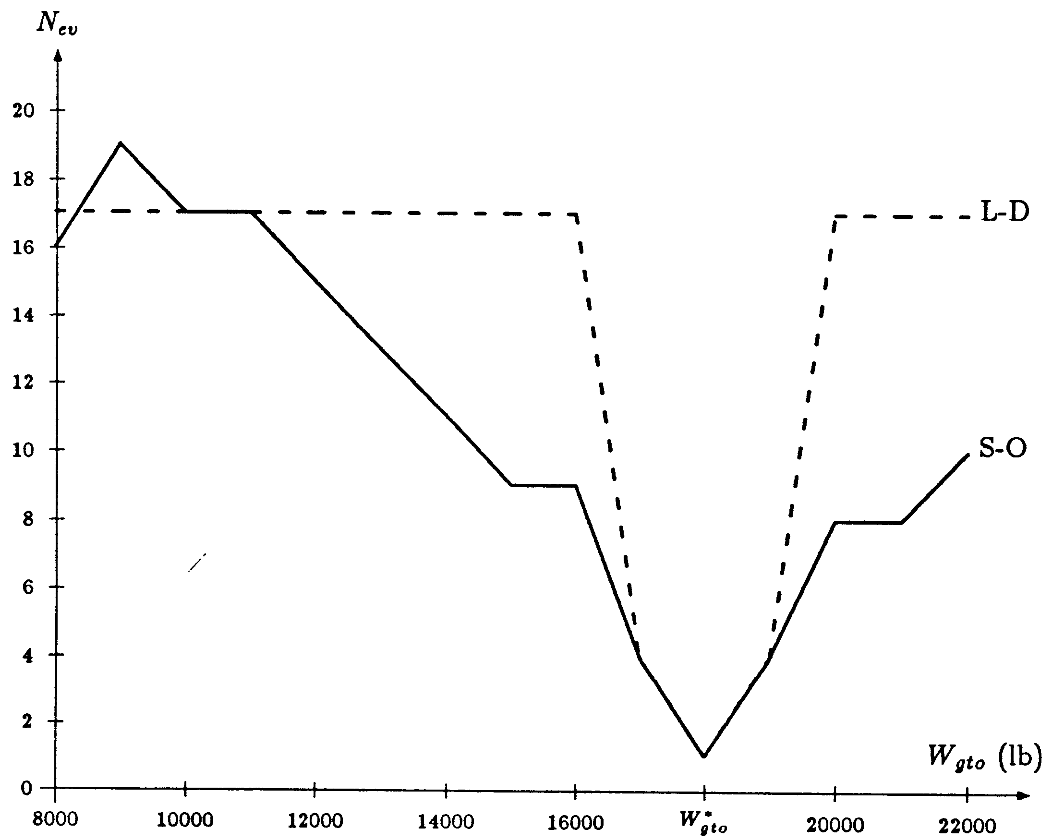


Figure 5.3: Comparison of methods with solution near the upper limit.

then the value of the forcing variable at the intersection of the two curves. This is shown in Figure 5.4, borrowed from [8].

Because the branches are generally nonlinear, a Simultaneous Newton-Raphson Method was dropped in favor of the Logarithmic-Distribution Method, a pure logarithmically distributed search between the lower and upper limits of the forcing variable. After an initial search over 10 values, the value producing the closest points would be used to compute new boundaries to begin a new search over 4 values (including the boundaries). This would be repeated until the points converged. This is also shown in Figure 5.4.

Unlike the problem of the Design Function Solver, the author did not dispute the need for such a wide initial search spread for the Loop Solver; thus the number of search points was not reduced. Also, since a Newton-Raphson iteration was not being performed, there was no need to stop the search whenever the points came within 10% of each other. The problems of the Loop Solver were in the needless reevaluation of the boundary points and in the computationally expensive calculation of the derivatives of the two branches to find out where the new boundaries were.

Since the solution is at an intersection of two curves then, during the search, whenever the successive differences between two computed values change sign, the intersection point, and thus the solution, would have just been crossed over. Since the solution must then lie somewhere between the current search value and the previous one, this automatically would define the new search boundaries. Adding a cross-over detection check would thus get rid of the need to calculate the derivatives of the branches. It would also allow the current search to be terminated as soon as a cross-over was detected, and allow a new search to begin immediately.

Once this modification was made to the Loop Solver code, the MISO and XCODE Design Sets were reset back to their original design paths and then reprocessed. Total processing time was further reduced by a factor of 2, dropping the MISO time from 20 seconds to 10, and the XCODE time from 12 minutes to 6.

The Search-Outward Method and the Loop Solver

Although the cross-over detection check solved the problem of computing new boundaries, it did not solve the problem of reevaluating the boundary points, which was needlessly doubling the computation time for all new searches. Another problem was that the Logarithmic-Distribution Method of searching from the lower limit (or lower boundary) towards the upper limit (or upper boundary) was not very efficient. Assuming the user

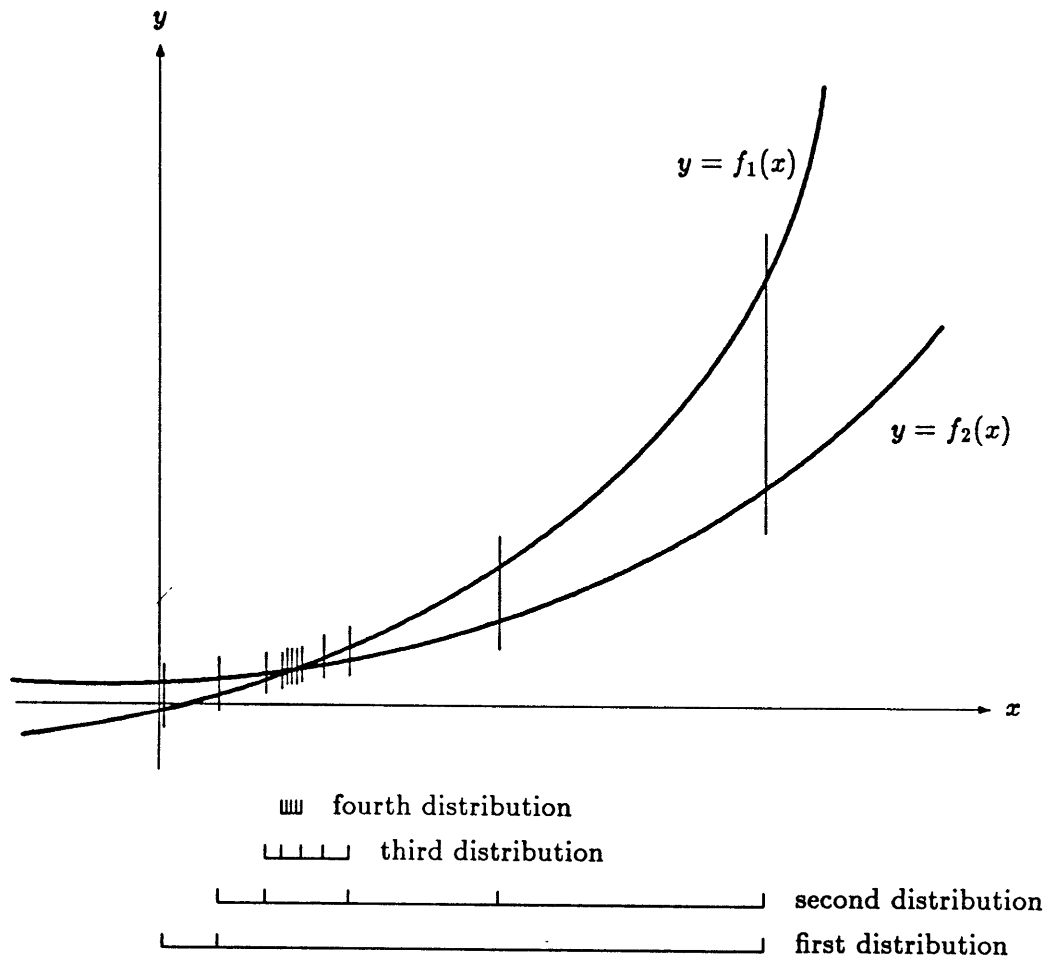


Figure 5.4: The logarithmic distribution method for solving loops.

could provide a fair initial guess at the solution, a better method would be to start the search at that initial guess value and to spread the search outwards towards both upper and lower limits simultaneously.

With this in mind, the Search-Outward Method was created and soon replaced the Logarithmic-Distribution Method of the Loop Solver. For the initial search, "SEARCH-OUTWARD" linearly divides the distance between the upper and lower limits into a region of 10 spaces, which it then centers over the current value of the unknown, the initial guess supplied by the user. The search starts at this initial guess value then alternates proceeding towards the lower and upper limits. The previous search value and computed values for each direction are stored for the cross-over detection check routine.

When a cross-over is detected, the new boundaries and their stored computed values are sent to "SEARCH-BETWEEN" which decides from which boundary the new search should begin. The information is then passed on to "SEARCH-FROM" which searches from one boundary towards the other (without reevaluating the boundary points) until another cross-over is detected to repeat the process. By using "SEARCH-BETWEEN," each new search normally only requires one search value evaluation before proceeding on to a narrower search over one-third the space. In this manner, the number of search value evaluations has dropped from 4 for the Logarithmic-Distribution Method to 1 for the Search-Outward Method.

The MISO Design Set benchmark was chosen to compare the two methods, the results of which is shown in Figure 5.5. On the x-axis is the choice for the initial guess, W_{gto}, as compared to the actual solution, W^*_{gto}. On the y-axis is the number of branch evaluations, N_{ev}, required to converge upon the solution. (One branch evaluation yields computed values for *both* branches.)

The Logarithmic-Distribution Method required an average of 16 branch evaluations to converge upon the solution. In fact, it almost *always* required 16 - 17 branch evaluations because the initial search always started at the lower limit. Reentering the solution still required 8 branch evaluations, and entering guess values very near the solution required, for some unknown reason, more than 18. Meanwhile, the Search-Outward Method required an average of 8 branch evaluations, half of the Logarithmic-Distribution Method's, with less required as the initial guess approached the solution. (Reentering the solution only required 2 branch evaluations.) Figure 5.5 shows that the Search-Outward Method required only a fair guess at the solution for quick convergence, while the Logarithmic-Distribution Method required the exact one.

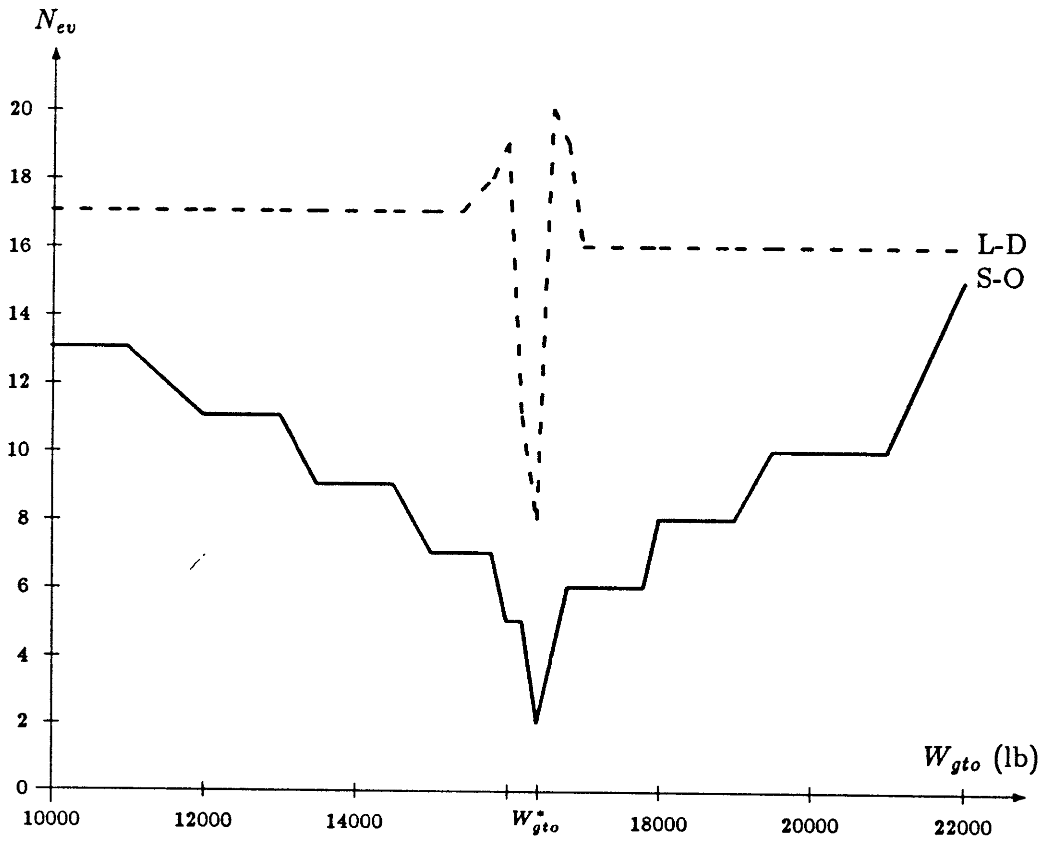


Figure 5.5: Comparison of methods for linear loop branches.

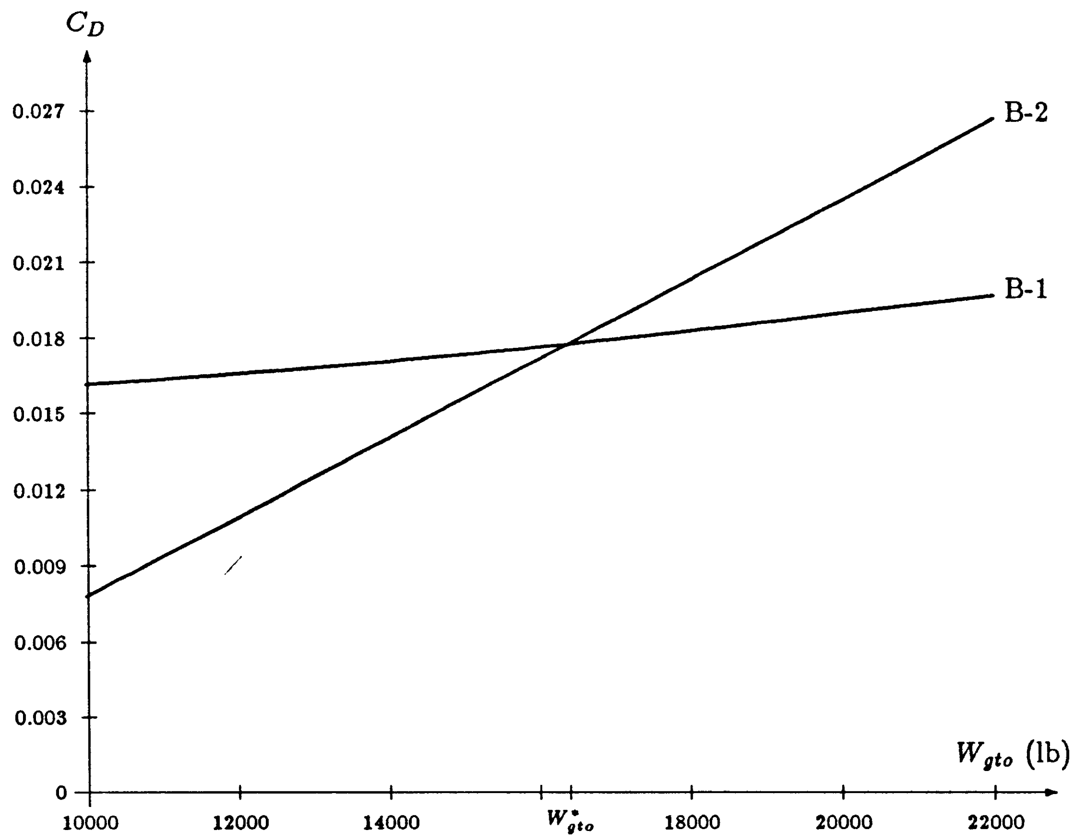


Figure 5.6: Branches of drag coefficient versus gross take-off weight.

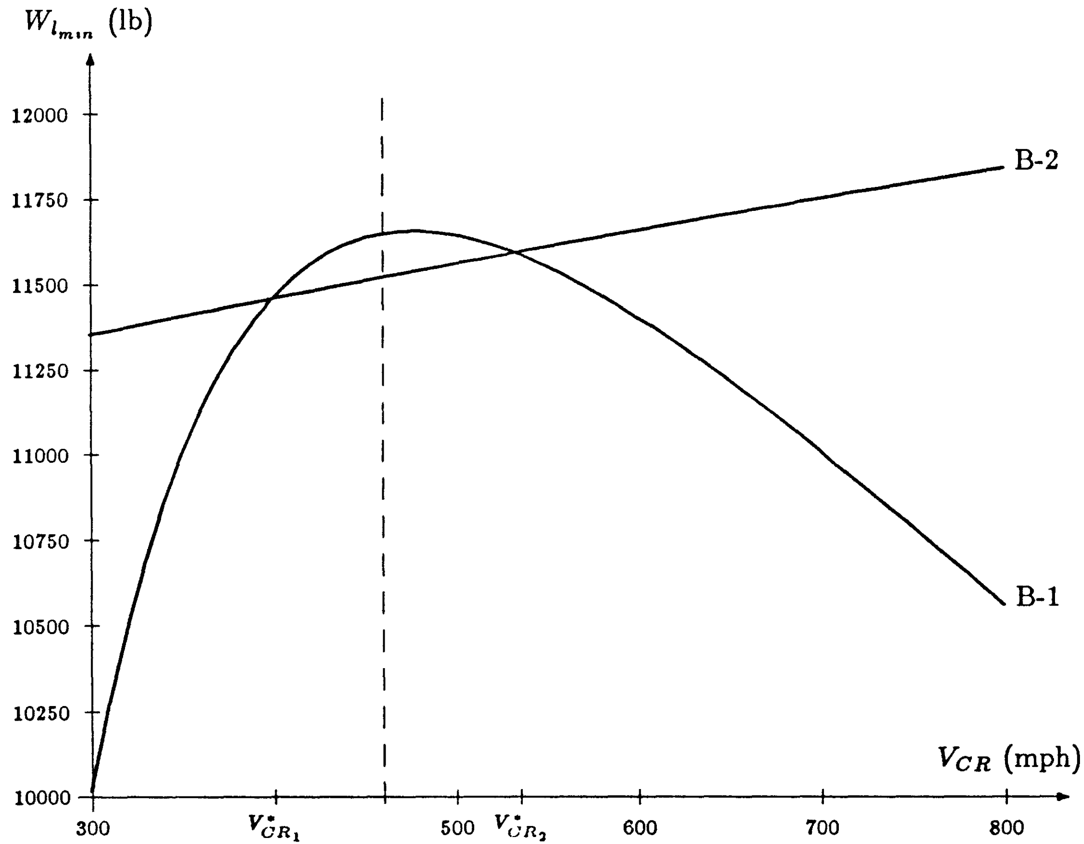


Figure 5.7: Branches of minimum landing weight versus cruise velocity.

Curious about the shape of the curves of the branches, the author plotted them (as shown in Figure 5.6) and, much to his embarrassment, found them to be two straight lines. Thinking that a more non-linear test case would be more appropriate, a new design path was chosen for the MISO Design Set so that the loop would involve a non-linear branch sequence of design functions. The curves of the branches are shown in Figure 5.7 and the comparison between the two search methods involving these branches is shown in Figure 5.8.

This new problem was so non-linear that it actually had two solutions, as shown by the two intersection points in Figure 5.7. The dashed line in both Figures 5.7 and 5.8 defined the boundary between the two solutions. To the left of the boundary, all initial guesses, V_{CR}, should have fallen towards the first solution, $V^*_{CR_1}$; and to the right, towards the second solution, $V^*_{CR_2}$.

As shown in Figure 5.8, the Logarithmic-Distribution Method required an average

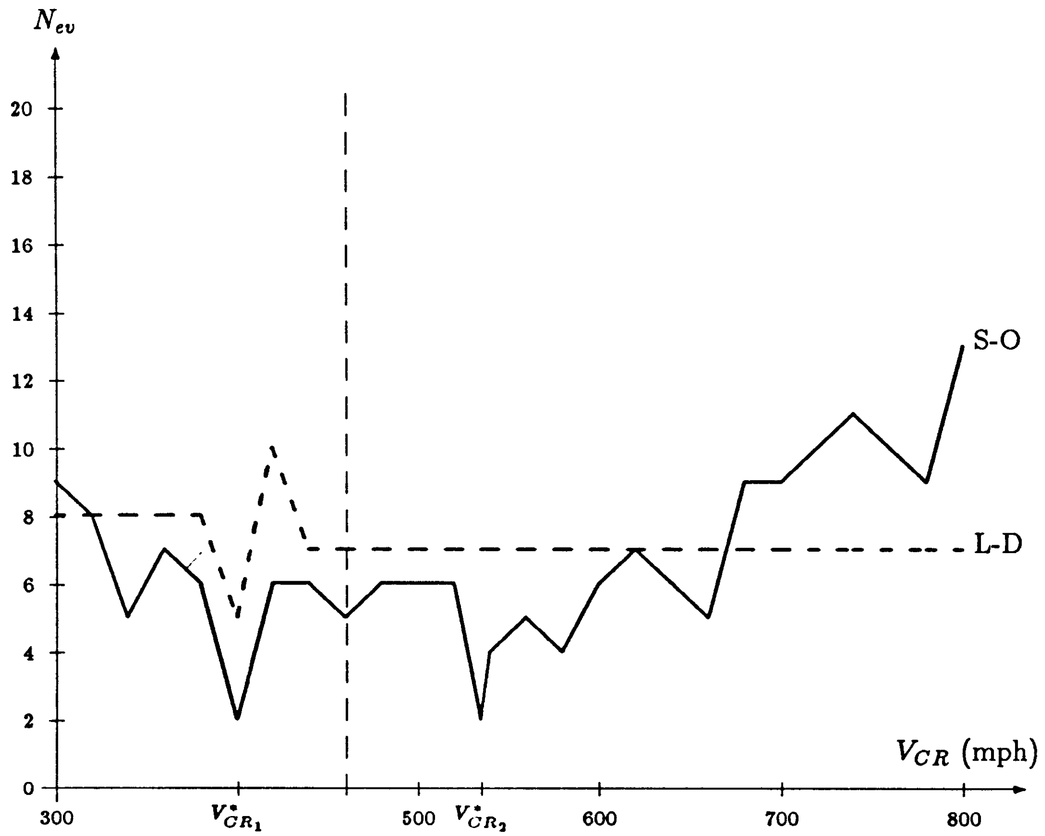


Figure 5.8: Comparison of methods for non-linear loop branches.

of 7 branch evaluations to converge upon a solution; however, since the Logarithmic-Distribution Method always searched from the lower limit, it always converged upon the first solution $V^*_{CR_1}$ — even when the second solution $V^*_{CR_2}$ was entered as the initial guess. Therefore the real average to converging upon the second solution was infinity. This major drawback of the Logarithmic-Distribution Method makes any implementation of the Search-Outward Method worth while. Fortunately, though, the Search-Outward Method performed better than the Logarithmic-Distribution Method. Not only did it converge upon the solution it was supposed to, it also did it with an average of only 5 branch evaluations.

With the Search-Outward Method fully incorporated into the Loop Solver code, the MISO and XCODE Design Sets were reset back to their original design paths and then reprocessed. Total processing time was further reduced by almost a factor of 3, dropping the MISO time from 10 seconds to 3, and the XCODE time from 6 minutes to 2. Overall, the modifications to the Design Function Solver and to the Loop Solver have reduced processing time by a factor of 25.

5.3 Thoughts on Numerical Methods

Throughout the development of the *Paper Airplane* Numerical Solvers (i.e., the Design Function Solver, the Loop Solver, and the MIMO Solver), only two numerical methods really have been applied — searching and taking the derivative.

Searching amounts to nothing more than guessing, taking mathematical potshots in the dark with the hope of miraculously hitting upon the solution. This is the equivalent of trying to find the top of Mount Everest by flying around Tibet in a bomber at 40000 feet (at night) and dropping bombs in order to find out the shortest time to an explosion. The problem with this method is that one would need a *lot* of bombs in order to find the top of Mount Everest in this manner.

Taking the derivative is slightly better, but has its own downfalls. Taking the derivative amounts to nothing more than feeling one's way around mathematical space with the hope of coming across the solution. This is the equivalent of trying to find the top of Mount Everest by driving around Tibet in a snowmobile (at night — during a blizzard) and continuously climbing in order to reach the top. The problem with this method is that the top reached may not be the top of Mount Everest. Indeed, Tibet has plenty of other mountains one might climb by mistake. In mathematics, this is called reaching

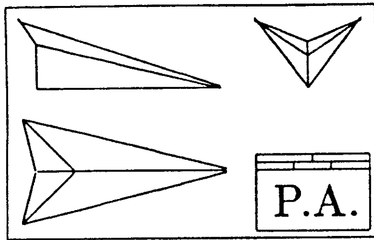
a local maximum. The downfall with this method is that discontinuities, like crevices in a mountainside, can plunge you into a hole from which you can never escape. In mathematics, this is called reaching a local minimum.

What one really needs is the mathematical equivalent of seeing. Anyone standing in Tibet can just take one glance at the landscape and point to the top of Mount Everest. There should be a mathematical equivalent operation to looking up and saying "Oh, there it is."

The author has trivialized the problem somewhat, by using the top of the highest mountain as the solution. A more general solution would be a lost mountain climber. Any Search-and-Rescue operator knows of the difficulty in finding a lost climber; however, he does not drop bombs from 40000 feet nor drives madly about in a snowmobile in order to conduct the search. Instead, he uses a helicopter to get above the mountain and a good pair of eyes to look down over it.

Mathematics needs a numerical method analogous to getting above the problem and looking down over it. Since seeing is a parallel process, the numerical method should involve guessing many values in a region simultaneously, a task best served by a good parallel processor. Since seeing also involves scanning, the visual equivalent of taking the derivative, the numerical method should also take derivatives as well. And since seeing is an intelligent process, the numerical method should be able to intelligently combine guessing with taking the derivative.

The *Paper Airplane* Numerical Solvers do combine guessing with taking the derivative; although how intelligently will only be decided by how well it finds the solutions to problems yet untried. The best piece of information a mathematician could provide the user community at this time is the proper combination of guessing and taking the derivative for solving numerical problems.



Chapter 6

The Aerospaceplane Design Test

This chapter describes the research, development, and testing of the NASP Design Set created to perform a major test of *Paper Airplane*'s new and modified capabilities — the preliminary design of a national aerospaceplane (NASP). The NASP Design Set consisted of 45 design variables and 26 design functions, including MISO design functions, MIMO design functions, design functions calling external codes, and design functions calling internal LISP functions.

6.1 Research

As in the real engineering world, the first order of business was to develop the requirements of the vehicle. It was first decided that the national aerospaceplane to be designed would be a commercial passenger vehicle; therefore, the trajectory would be a simple climb, cruise, and descent. Next, it was decided that the vehicle would cruise at Mach 6 at 150,000 feet and carry 200 passengers 6000 miles.

Since this was to be a practical engineering test of *Paper Airplane*'s new and modified capabilities, the NASP Design Test would require practical engineering design functions. It was therefore decided that complex design functions for the weight, aerodynamic, propulsion, and performance characteristics of the vehicle would be created, and that simpler design functions for other characteristics (such as thermodynamic) would be added later.

The obvious choice for the complex design functions were design functions calling external FORTRAN codes. A quick search and inquiry revealed that no such codes were

available; therefore, they would have to be written from scratch. A search for information led the author to Prof. Rene Miller [13], director of the M.I.T. Space Systems Laboratory. Prof. Miller supplied several papers he had written on the subject of aerospaceplanes plus a former student's Master's thesis from 1967. The thesis, "Aerospaceplane Optimization and Performance Estimation" by James Martin [12], provided the author with a foundation in which to build the NASP Design Set.

Martin's aerospaceplane was not a commercial passenger vehicle, however, but a single-stage-to-orbit cargo transport; therefore, it did not cruise nor descend, but climb continuously — and at speeds much greater than Mach 6. Because of this, it was decided to retain only the vehicle's geometry (shown in Figure 6.1) and its propulsion characteristics (a table of specific impulse as a function of Mach number), and to ignore its aerodynamic characteristics (which concentrated on hypersonic flight), its weight characteristics (which were weight fractions that could not be reverified), and its performance characteristics (which concentrated on climb load factors and thermodynamics).

Aerodynamic Characteristics

All the information needed to define the aerodynamic characteristics of the aerospaceplane was obtained from *Fundamentals of Aircraft Design* by Leland Nicolai [14]. Although Nicolai's text does not cover hypersonic aerodynamics, the supersonic aerodynamics could be extended out to Mach 6. Because the text is written for aircraft designers, the aerodynamics is presented in equations, tables, and graphs. Where possible, the graphs were reduced to equations that could be used by the aerodynamics code. For this and other reasons, it was decided to tailor the aerodynamics code to the geometry at hand (delta wing, sharp leading edges, double-wedge airfoil, etc.) instead of attempting to make a general aerodynamics package.

Weight Characteristics

Nicolai's text was again used to define the weights characteristics of the aerospaceplane. The equations Nicolai provides for computing the weights of aircraft substructures are based upon historical data; since the national aerospaceplane would be a revolutionary design, the equations would therefore not apply. A different historical datum would apply, however. Nicolai's text supplies a graph of the empty weight fraction of a vehicle versus its gross weight at take-off. Although simple, this straight line curve represents

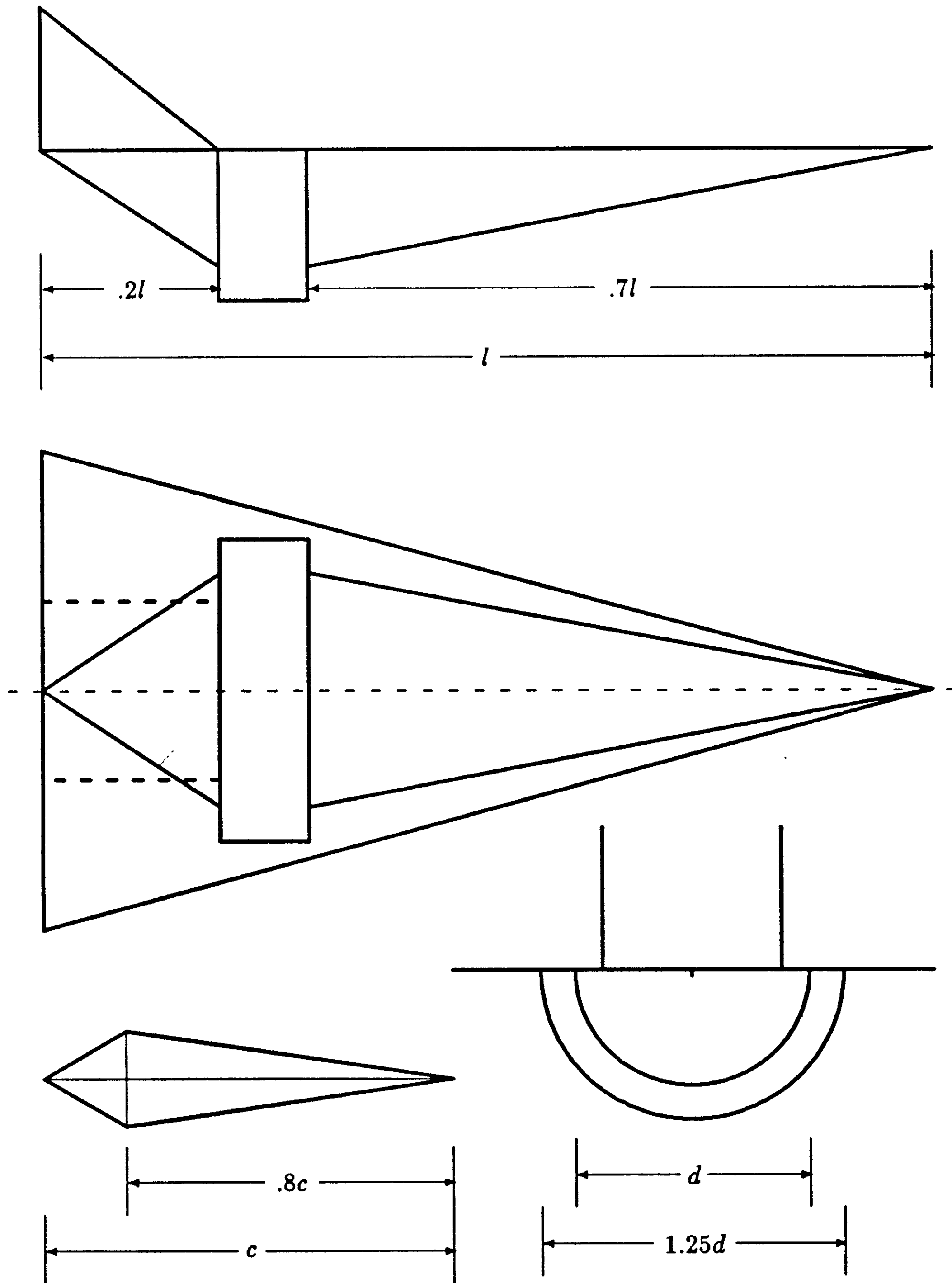


Figure 6.1: Geometry of the aerospaceplane.

the unchanging historical trend of hundreds of aircraft. As this line is not likely to change very much in the near future, it was decided that the an empty weight fraction for the aerospaceplane would be just about as accurate as any weight information other methods could provide.

Assuming that the aerospaceplane would be about the size and weight of a Boeing 747, and using some information in Nicolai on Boeing's canceled SST, the empty weight fraction was set at 0.40 (40%).

Performance Characteristics

The performance characteristics of the aerospaceplane were by far the most difficult ones to define. In fact, the sole reason for continuous delays in creating the NASP Design Set was due to problems in defining precisely what trajectory the aerospaceplane would follow. The "simple" climb, cruise, and descent trajectory was not so simple after all. (It should be pointed out that this problem extends out into the aerospace industry as well.)

Although Nicolai's text was again referenced, it only provided detailed performance information for cruise conditions; which still left climb and descent undefined. Preliminary equations were developed so that the aerospaceplane would climb at a given angle, cruise until it was almost out of fuel, then descend at another angle. The author went to Prof. Miller (one of many visits) to verify the equations. Prof. Miller suggested a boost-glide trajectory instead — a fast climb, a short cruise, and a long glide. Both climb and descent would be at maximum lift-to-drag ratio.

At the same time, the author heard a lecture by a NASA Ames researcher on current aerospaceplane research. He felt that the boost-glide trajectory was a military one, not a commercial one. Once again without a clearly defined trajectory, the defining of the performance characteristics came to a halt. A third opinion to break the tie was not found until a month later during another lecture by a different aerospaceplane researcher, one from Draper Laboratory. Not only did he agree with Prof. Miller on the boost-glide trajectory, he also provided the author with the latest aerospaceplane propulsion data from NASA — specific impulse charts which could be used to update the ones from Martin's thesis.

Propulsion Characteristics

With the new propulsion data from NASA, the specific impulse tables were updated.

Specific impulse, I_{sp}, is not thrust, T, however; but a measure of fuel consumption rate $\dot{m}$ since $T = \dot{m} I_{sp}$. Therefore, the specific impulse could be used to compute the fuel consumption rate only once the thrust was computed. An independent equation for thrust was found in *Mechanics and Thermodynamics of Propulsion* by Hill and Peterson [6]. The equation calculates the maximum thrust through an air-breathing engine; the actual thrust would be less by some throttle factor k:

$$T = k\, T_{max} = k\, \rho V A f g I_{sp}$$

where ρ is the air density, V is the flight velocity, A is the inlet capture area, f is the fuel-to-air ratio, and g is gravity.

Since the inlet capture area could be computed from the geometry, the only unknown was the fuel-to-air ratio. Using a method Hill's text provides to compute the Stoichometric fuel-to-air ratio of aviation fuel, the Stoichometric fuel-to-air ratio of liquid hydrogen, the chosen fuel of the national aerospaceplane, was found to be 0.029. Since the actual fuel-to-air ratio is only slightly less than this number, it was decided to use the Stoichometric fuel-to-air ratio for the actual fuel-to-air ratio.

With this last bit of information, the development of the NASP Design Set could commence.

6.2 Development

The development of the NASP Design Set was in three stages, (1) creation of the aerodynamics code, (2) creation of the performance code, and (3) creation of the rest of the design set.

The Aerodynamics Code

Using the information gleaned from Nicolai, the author wrote a FORTRAN code to compute lift coefficient, drag coefficient, and lift-to-drag ratio as a function of altitude, Mach number, and angle of attack. The code was written for optimum performance. Geometry and geometry-specific information are computed only once, altitude-specific information is computed once for each altitude, Mach-number-specific information is computed once for each Mach number at each altitude, etc. In the angle of attack loop, only lift coefficient, drag coefficient, and lift-to-drag ratio are computed — all other values are already defined. The final version of the aerodynamics code is listed in Appendix H.

The code was compiled for debugging and was stepped through to make sure that all values were being computed correctly. After the code was successfully debugged (a process that did not take long), the code was run and successfully tested. This optimized code computed the aerodynamics for 16 altitudes, 30 Mach numbers, and 30 angles of attack in only 92 seconds — an average of 156 data points per second.

The results were plotted and shown to Prof. Miller and Prof. Simpson, who both validated them. Some of the results are shown in Figures 6.2 and 6.3.

The Performance Code

Using information from Nicolai, Prof. Miller, and the author's past experience, the author wrote a FORTRAN code to integrate the range of an aerospaceplane. The vehicle takes-off and climbs until it reaches cruise altitude. It then continues to accelerate to cruise Mach number and cruises until its fuel supply is down to reserves. It then descends unpowered until it reaches the 50 foot runway altitude, and stops.

Inputs are cruise Mach and altitude, vehicle weight at take-off, fuel and fuel reserves weights, wing reference area, inlet capture area, and fuel-to-air ratio. Other inputs are the specific impulse and aerodynamics tables.

Using the boost-glide trajectory, the vehicle would climb at maximum thrust and at maximum lift-to-drag ratio, cruise under equilibrium conditions, and descend again at maximum lift-to-drag ratio. The code was compiled for debugging and was stepped through to make sure that all values were being computed correctly. Under maximum thrust, the initial acceleration was 200 times that of gravity. The author went back to talk with Prof. Miller.

The problem was that the inlet capture area was too large. To scale it down, it would have to be sized for cruise conditions. The climb trajectory was also wrong. The vehicle should not climb at maximum lift-to-drag ratio, but under steady state conditions at a given climb angle and a given acceleration.

Once two more inputs were added (climb angle and climb acceleration), the code was again compiled and stepped through. The vehicle successfully climbed to cruise altitude, but even during cruise it kept climbing. A control equation modifying equilibrium lift was added to guide the desired climb angle to zero as the cruise altitude was approached. This worked successfully and the vehicle climbed to cruise altitude, leveled off, and cruised until its fuel supply dropped into its reserves. During descent, however, all hell broke loose. The discrete transient switch from equilibrium flight to flying at maximum lift-to-

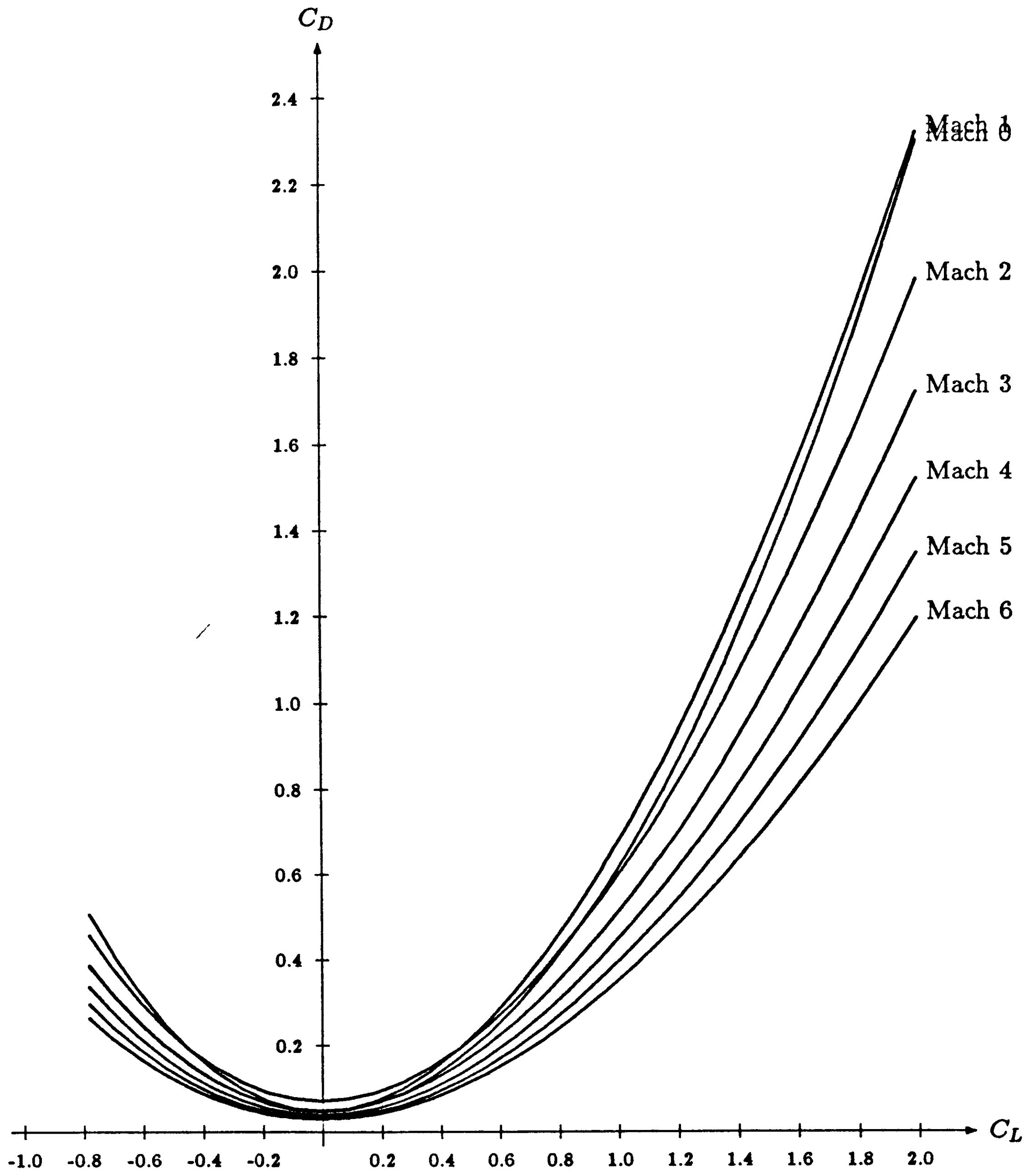


Figure 6.2: Drag Coefficient vs. Lift Coefficient.

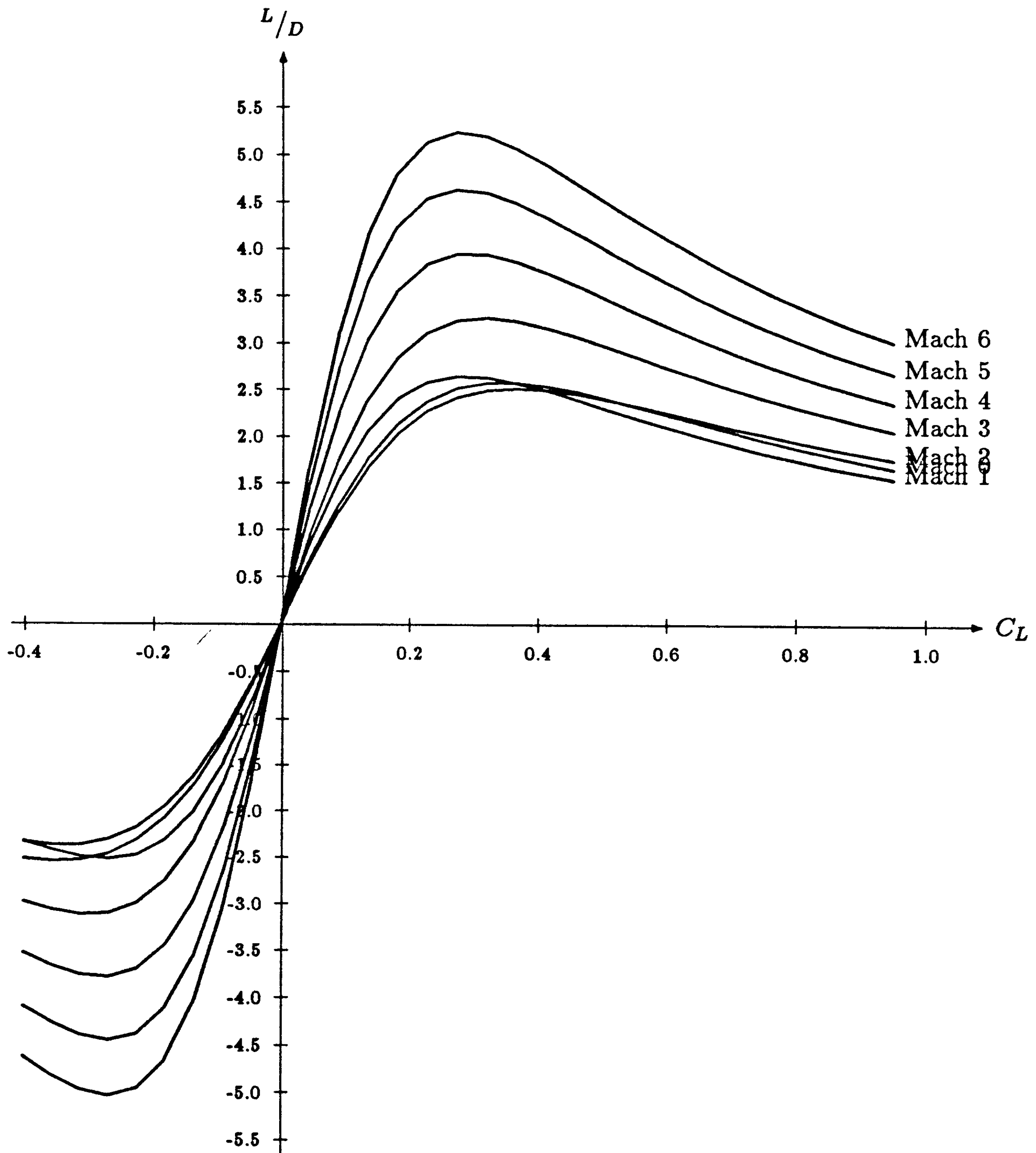


Figure 6.3: Lift-to-Drag Ratio vs. Lift Coefficient (150000 ft).

drag ratio created tremendous phugoid oscillations that sent the vehicle plunging towards the surface then shooting up into the sky. After the phugoid oscillations damped, the vehicle successfully landed at Mach 0.26. Despite the successful landing, the author went back to talk with Prof. Miller.

Prof. Miller explained that the vehicle should always be flying under equilibrium conditions, which are normally designed for maximum lift-to-drag ratio flight. The descent was changed to an equilibrium flight, but the vehicle would not descend. Instead, the angle of attack compensated to keep the vehicle flying level as the Mach number dropped subsonic. Problems in the code allowed the angle of attack to rise to ridiculous values. That corrected, the descent was again modified to fly under equilibrium conditions until the angle of attack reached that for maximum lift-to-drag ratio. At that point, the flight would switch to flying at maximum lift-to-drag ratio, but hopefully would avoid major phugoid problems.

After stepping through the code to correct any mistakes, the code was run from take-off to landing. During descent, the vehicle continued to fly level, while its Mach number fell, until its angle of attack reached that for maximum lift-to-drag ratio. Under maximum lift-to-drag ratio flight, the vehicle descended to the surface with only minor phugoid oscillations and landed, as before, at Mach 0.26.

The final version of the code is listed in Appendix I. Run time to read in all the data and fly out the trajectory was less than two minutes.

The NASP Design Set

With the aerodynamics code and the performance code written, the NASP Design Set could be created around them. The first two design functions written were, obviously, the ones that called the codes. Several others were added to compute the geometry and weights information needed by the codes.

A problem emerged with LISP during the creation of the external codes preprocessors and postprocessors, which format the data for input and retrieve the formatted data from the output, respectively. Due to LISP's poor formatted I/O capabilities, the author was forced to write short FORTRAN codes to perform the required formatting tasks. Instead of the design function calling one external code, it now had to call three. The External Code Interface did not have to be modified in any way, however, since it was already capable to handling unformatted data. The preprocessors and postprocessors still ran quickly enough (a matter of seconds) that their processing time disappeared

when compared to that of the main external code.

During the author's many discussions with Prof. Miller, it became apparent that the aerospaceplane, using Martin's geometry, would not fly 6000 miles; therefore the desired range was reduced to 5000 miles. For thermodynamic reasons, the cruise Mach number was also reduced, to Mach 5. Finally, the cruise altitude was reduced to 120,000 feet because the original altitude of 150,000 feet is only required for the hypersonic aerospaceplanes.

It was decided to keep the geometrical shape of the vehicle and to change only its size. The scaling factors, shown in Figure 6.1, were added as design variables, and the computation of the actual lengths was relegated to design functions.

A cabin was laid out to make sure that it could fit 200 people in the upper fuselage, and all the fuel in the lower fuselage. Formulas were derived to compute the volume in the lower fuselage and a factor of 0.9 was thrown in to allow for structure. Design variables and design functions were then added to compute the distribution of passengers and fuel aboard the vehicle. The weight of the fuel, therefore, depended on the volume avialable to store it.

Finally, several design functions were added to compute the thermodynamic characteristics of the vehicle. Specifically the design functions were to compute the nose radius and wing leading-edge radius required to withstand their respective stagnation temperatures.

Appendix G contains the complete list of the design variables and design functions comprising the NASP Design Set.

6.3 Testing

The testing of the NASP Design Set was performed in three stages. The first stage was to use the *Paper Airplane* Function Exerciser and make sure that all the design functions worked properly. The second stage was to have *Paper Airplane* compute the range of the vehicle given a certain geometry. The third stage was to have *Paper Airplane* compute the geometry of the vehicle to obtain a given range.

Since it was decided to fix the shape of the vehicle, the size could be determined by one design variable; this was chosen to be the wing length. The second stage was then reduced to computing the range of a vehicle corresponding to a given wing length. The third stage was reduced to computing the vehicle's wing length to obtain a given range.

First Stage

This stage yielded the time it would take to evaluate each design function individually. This was most important for the two design functions calling external codes. Most of the simple design functions ran successfully the first time; only a few required minor corrections.

The preprocessors and postprocessors of both the aerodynamics code and the performance code were tested and debugged before the design functions themselves were tested from inside *Paper Airplane*. Therefore, the testing of the design functions calling these external codes went smoothly. The average time to run the aerodynamics code from within *Paper Airplane* was 50 seconds. A program to convert the aerodynamics data to a form usable to the performance code required an addition 30 seconds. The average time to run the performance code from with *Paper Airplane* was 100 seconds.

The results of a inlet sizing design function was used to update the value of the inlet capture area required by the performance code. When the performance code was run, however, the vehicle could not achieve enough thrust to climb to cruise altitude and accelerate to cruise Mach number. The vehicle fell until it accelerated to its cruise Mach number and then leveled off. Then, during descent, it dove straight into the ground at supersonic speed. Needless to say, the range was much less than desired.

The death plunge was a problem with the maximum lift-to-drag ratio algorithm and was corrected. The thrust problem was not as easy to correct. Since the latest version of the code was properly controlling the required thrust, the maximum thrust limit could be raised to keep from overconstraining it. This meant abandoning the concept of sizing the inlet, thus also the inlet sizing design function and the special design variables and design functions associated with it. Despite reservations, this was done. Since an inlet capture area was still required by the performance code, it was equated to the cross-sectional area bounded by the engine diameter. These changes did decrease processing time, however, and allowed the vehicle to fly to its maximum range.

Second Stage

The design path was chosen so that all the unknown geometry values could be computed once the wing length was specified. Once the geometry was known, the weights and aerodynamics could be determined. Finally, once the geometry, weights, and aerodynamics were known, the range could be computed. This design path should have produced

AGENDA for design NASP

AGENDA ENTRY	DESIGN VARIABLE	COMPUTED BY DESIGN FUN	DIRECTION
Forced Path	AFTERBODY_LENGTH	AFTERBODY-LENGTH-EQUATI	FORWARD
Forced Path	WING_SPAN	WING-SPAN-EQUATION	FORWARD
Forced Path	BODY_DIAMETER	BODY-DIAMETER-EQUATION	FORWARD
Forced Path	FOREBODY_LENGTH	FOREBODY-LENGTH-EQUATIO	FORWARD
Forced Path	TAIL_LENGTH	TAIL-LENGTH-EQUATION	FORWARD
Forced Path	ENGINE_DIAMETER	ENGINE-DIAMETER-EQUATIO	FORWARD
Forced Path	PAYLOAD_WEIGHT	PAYLOAD-WEIGHT-EQUATION	FORWARD
Forced Path	WING_REFERENCE_ARE	REFERENCE-AREA-EQUATION	FORWARD
Forced Path	CRUISE_VELOCITY	CRUISE-VELOCITY-FUNCTIO	FORWARD
Forced Path	CRUISE_AIR_DENSITY	CRUISE-DENSITY-FUNCTION	FORWARD
Forced Path	ENGINE_FRACTION	LENGTH-FRACTION-EQUATIO	FORWARD
Forced Path	ENGINE_LENGTH	ENGINE-LENGTH-EQUATION	FORWARD
Forced Path	FUEL_VOLUME	FUEL-VOLUME-EQUATION	FORWARD
Forced Path	FUEL_WEIGHT	FUEL-WEIGHT-EQUATION	FORWARD
Forced Path	FUEL_RESERVES	FUEL-RESERVES-EQUATION	FORWARD
Forced Path	TAIL_HEIGHT	TAIL-HEIGHT-EQUATION	FORWARD
Forced Path	INLET_CAPTURE_AREA	CAPTURE-AREA-EQUATION	FORWARD
Forced Path	MAXIMUM_LIFT_COEFF	AERODYNAMICS-PROGRAM	FORWARD
Forced Path	PAYLOAD_VOLUME	PAYLOAD-VOLUME-EQUATION	FORWARD
Forced Path	LEADING_EDGE_RADIU	WING-RADIUS-EQUATION	FORWARD
Forced Path	NOSE_RADIUS	NOSE-RADIUS-EQUATION	FORWARD
Forced Path	FUSELAGE_VOLUME	FUSELAGE-VOLUME-EQUATIO	FORWARD
Loop 1: Initial Path	<NO ENTRIES>		
Loop 1: Branch 1	VEHICLE_EMPTY_WEIG	GROSS-WEIGHT-EQUATION	*REVERSE*
Loop 1: Branch 2	VEHICLE_EMPTY_WEIG	EMPTY-WEIGHT-EQUATION	FORWARD
Loop 1: Final Path	TIME_OF_FLIGHT	PERFORMANCE-PROGRAM	FORWARD
Loop 1: Final Path	RANGE	PERFORMANCE-PROGRAM	FORWARD
Loop 1: Final Path	PAYLOAD_WEIGHT_FRA	PAYLOAD-FRACTION-EQUATI	FORWARD

Figure 6.4: Computational agenda for Second Stage testing.

an agenda with one large forced path and no loops; therefore, the processing time would be minimized. The actual computational agenda built by *Paper Airplane* to solve for all the unknowns (shown in Figure 6.4) was just as intended. Except for one small loop, the agenda is one large forced path.

Because of the number of design variables contained in the NASP Design Set, the initial design point is listed below rather than put into a figure.

LIST OF INTERNED VARIABLES

F	T	O.O.M.	WT	ST	VARIABLE NAME	CURRENT VALUE	INCOMP'S
	T	0.2000		I	AFTERBODY_FRACTION	0.2000	
+	T	30.0000		G	AFTERBODY_LENGTH	30.0000 ft	
+	T	40.0000		G	BODY_DIAMETER	40.0000 ft	
	T	7.5000		I	CABIN_HEIGHT	7.5000 ft	
	T	0.5000		I	CLIMB_ACCELERATION	0.5000 g	
	T	20.0000		I	CLIMB_ANGLE	20.0000 deg	
+	T	0.0063		G	CRUISE_AIR_DENSITY	0.0063 kg m-3	
	T	120000.0000		I	CRUISE_ALTITUDE	120000.0000 ft	
	T	5.0000		I	CRUISE_MACH	5.0000	
+	T	5100.0000		G	CRUISE_VELOCITY	5100.0000 ft s-1	
	T	0.4000		I	EMPTY_WEIGHT_FRACT	0.4000	
+	T	50.0000		G	ENGINE_DIAMETER	50.0000 ft	
+	T	0.1000		G	ENGINE_FRACTION	0.1000	
+	T	15.0000		G	ENGINE_LENGTH	15.0000 ft	
	T	0.7000		I	FOREBODY_FRACTION	0.7000	
+	T	105.0000		G	FOREBODY_LENGTH	105.0000 ft	
+	T	3200.0000		G	FUEL_RESERVES	3200.0000 lb	
+	T	15000.0000		G	FUEL_VOLUME	15000.0000 ft3	
+	T	65000.0000		G	FUEL_WEIGHT	65000.0000 lb	
+	T	37700.0000		G	FUSELAGE_VOLUME	37700.0000 ft3	
+	T	1000.0000		G	INLET_CAPTURE_AREA	1000.0000 ft2	
+	T	1.5000		G	LEADING_EDGE_RADIU	1.5000 in	
	T	75.0000		I	LEADING_EDGE_SWEEP	75.0000 deg	
+	T	0.8300		G	MAXIMUM_LIFT_COEFF	0.8300	
+	T	6.0000		G	NOSE_RADIUS	6.0000 in	
	T	2500.0000		I	NOSE_TEMPERATURE	2500.0000 R	
	T	206.0000		I	PASSENGER_CAPACITY	206.0000	
+	T	16500.0000		G	PAYLOAD_VOLUME	16500.0000 ft3	
+	T	47400.0000		G	PAYLOAD_WEIGHT	47400.0000 lb	
+	T	47400.0000		G	PAYLOAD_WEIGHT_FRA	0.3000	
+	T	5000.0000		G	RANGE	5000.0000 sm	
	T	0.6000		I	SEATING_FRACTION	0.6000	
	T	30.0000		I	SEAT_PITCH	30.0000 in	
	T	22.0000		I	SEAT_WIDTH	22.0000 in	
+	T	24.0000		G	TAIL_HEIGHT	24.0000 ft	

```
+ T     30.0000          G  TAIL_LENGTH             30.0000 ft
  T      0.8000          I  THICKNESS_STATION        0.8000

+ T    100.0000          G  TIME_OF_FLIGHT         100.0000 min
+ T 68000.0000           G  VEHICLE_EMPTY_WEIG   68000.0000 lb
+ T 170000.0000          G  VEHICLE_GROSS_WEIG  170000.0000 lb
  T    150.0000          I  WING_LENGTH            150.0000 ft
+ T   6000.0000          G  WING_REFERENCE_ARE    6000.0000 ft2
+ T     80.0000          G  WING_SPAN               80.0000 ft
  T   2500.0000          I  WING_TEMPERATURE      2500.0000 R
  T      0.0300          I  WING_THICKNESS           0.0300
```

The wing length was the default 150 feet. *Paper Airplane* processed the agenda and computed the values for all of the unknowns. Processing time was about three minutes. The value of the most important unknown, the range, was computed to be 2912 miles. The design set was reprocessed using a wing length of 200 feet and, this time, the range came out to be 5105 miles. Once more the design set was reprocessed using a wing length of 190 feet and, this time, the range came out to be 4700 miles. A quick calculation gave the desired wing length to be 197 feet. When the design set was reprocessed using this value, the range came out to be 5007 miles, within 0.2% of the desired value. A summary of the Second Stage testing is tabularized below.

Wing Length (feet)	Range (miles)	Vehicle Gross Weight (pounds)
150	2912	175,200
190	4700	340,400
197	5007	365,000
200	5105	381,900

With the solution found through educated guessing, it was then time to determine if *Paper Airplane* could find the solution by its own methods.

Third Stage

The only modification to the design path was that Range was changed from an unknown to a known, and Wing Length was changed from a known to an unknown. By doing so, however, the entire solution path would be reversed in a classic test of "inverse engineering." This design path would produce one short forced path and one large loop, with at least the performance code being processed in reversed; therefore, the processing time would be maximized. The actual computational agenda built by *Paper Airplane* to

solve for all the unknowns (shown in Figure 6.5) was just as intended. The agenda is one small forced path and one large loop.

The Second Stage final design point corresponding to the wing length of 200 feet was chosen as the Third Stage initial design point for this "inverse engineering" test. The only change was in setting the range to the desired 5000 miles. With the design function calling the performance code being processed in reverse, it was determined that the overall processing time would be much greater than the three minutes processing time from Second Stage testing. Because of this, the allowable numerical error was raised from the default 0.1% to 1.0%.

An error occurred in the performance code while *Paper Airplane* was reverse computing the gross weight of the vehicle to obtained the desired range. It turned out that because the fuel weight was a fixed computed known (its value having been determined by the guessed vehicle size), the lower search values of the gross weight dipped too close to that of the fuel weight and the performance code went unstable. To correct this problem, the lower and upper value limits of the vehicle gross weight design variable were reduced to the values corresponding to the 190-foot wing length vehicle and the 200-foot wing length vehicle, respectively. This was done using the *Paper Airplane* Design Set Editor and did not involve changing the NASP Design Set source file; thus it was a legal, and recommended, move. Also reduced were the lower and upper value limits of the wing length design variable, so that the search for the desired wing length would remain between 190 feet and 200 feet.

The initial design point was reset and the design set was reprocessed. Each search value required ten minutes to compute its branch values. After 35 minutes, a solution was successfully found. The desired wing length to carry the vehicle 5000 miles was 196.7 feet, very close to the educated guess value of 197 feet from Second Stage testing. The final design point of this solution to the NASP "inverse engineering" problem is listed below.

```
                              LIST OF INTERNED VARIABLES

 F T     O.O.M.     WT  ST   VARIABLE NAME       CURRENT VALUE          INCOMP'S
-------------------------------------------------------------------------------
   T     0.2000          I  AFTERBODY_FRACTION     0.2000
 + T    30.0000          C  AFTERBODY_LENGTH      39.3333 ft
 + T    40.0000          C  BODY_DIAMETER         52.6967 ft
   T     7.5000          I  CABIN_HEIGHT           7.5000 ft
   T     0.5000          I  CLIMB_ACCELERATION     0.5000 g
   T    20.0000          I  CLIMB_ANGLE           20.0000 deg
```

```
+ T      0.0063      C  CRUISE_AIR_DENSITY      0.0063 kg m-3
  T 120000.0000      I  CRUISE_ALTITUDE    120000.0000 ft
  T      5.0000      I  CRUISE_MACH             5.0000
+ T   5100.0000      C  CRUISE_VELOCITY      5213.2546 ft s-1
  T      0.4000      I  EMPTY_WEIGHT_FRACT      0.4000
+ T     50.0000      C  ENGINE_DIAMETER        65.8708 ft
+ T      0.1000      C  ENGINE_FRACTION         0.1000
+ T     15.0000      C  ENGINE_LENGTH          19.6667 ft

  T      0.7000      I  FOREBODY_FRACTION       0.7000
+ T    105.0000      C  FOREBODY_LENGTH       137.6667 ft
+ T   3200.0000      C  FUEL_RESERVES        8525.3303 lb
+ T  15000.0000      C  FUEL_VOLUME         39017.5300 ft3
+ T  65000.0000      C  FUEL_WEIGHT        170506.6059 lb
+ T  37700.0000      C  FUSELAGE_VOLUME     85786.1253 ft3
+ T   1000.0000      C  INLET_CAPTURE_AREA   1703.9087 ft2
+ T      1.5000      C  LEADING_EDGE_RADIU      0.7643 in
  T     75.0000      I  LEADING_EDGE_SWEEP     75.0000 deg
+ T      0.8300      C  MAXIMUM_LIFT_COEFF      0.8300
+ T      6.0000      C  NOSE_RADIUS             2.9530 in
  T   2500.0000      I  NOSE_TEMPERATURE     2500.0000 R
  T    206.0000      I  PASSENGER_CAPACITY    206.0000
+ T  16500.0000      C  PAYLOAD_VOLUME      16952.0833 ft3
+ T  47400.0000      C  PAYLOAD_WEIGHT      47380.0000 lb
+ T  47400.0000      C  PAYLOAD_WEIGHT_FRA      0.1307

+ T   5000.0000      I  RANGE                5000.0000 sm
  T      0.6000      I  SEATING_FRACTION        0.6000
  T     30.0000      I  SEAT_PITCH             30.0000 in
  T     22.0000      I  SEAT_WIDTH             22.0000 in
+ T     24.0000      C  TAIL_HEIGHT            31.7701 ft
+ T     30.0000      C  TAIL_LENGTH            39.3333 ft
  T      0.8000      I  THICKNESS_STATION       0.8000
+ T    100.0000      C  TIME_OF_FLIGHT        105.0000 min
+ T  68000.0000      C  VEHICLE_EMPTY_WEIG 145053.1222 lb
+ T 170000.0000      C  VEHICLE_GROSS_WEIG 362632.8055 lb
  T    150.0000      C  WING_LENGTH           196.6667 ft
+ T   6000.0000      C  WING_REFERENCE_ARE  10363.6793 ft2
+ T     80.0000      C  WING_SPAN             105.3933 ft
  T   2500.0000      I  WING_TEMPERATURE     2500.0000 R
  T      0.0300      I  WING_THICKNESS          0.0300
```

A quick analysis verifies the results. The desired vehicle had a wing length of 197 ft and a wing span of 105 ft, given a wing reference area of 10,360 ft^2. With a vehicle gross weight of 362,600 lb, the wing loading was 35 lb/ft^2, a reasonable number. The payload fraction was only 13% of the vehicle gross weight, because of the large need for fuel (almost half the vehicle gross weight). Still, this is not unreasonable. The nose radius and leading edge radius, though small, were as expected. The time of flight to fly 5000 miles

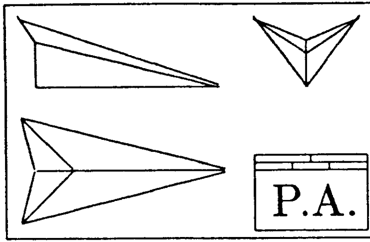
was 105 minutes, which would allow passengers flying west to arrive before the time they left. Although strange, this is one of the major economic problems the aerospace industry is currently studying.

Despite many problems, the Aerospaceplane Design Test was a huge success. A national aerospaceplane was designed to whisk 200 passengers 5000 miles in less than two hours. After three months of intensive research and development, *Paper Airplane* took only 35 minutes to find a solution to the given requirements.

AGENDA for design NASP

AGENDA ENTRY	DESIGN VARIABLE	COMPUTED BY DESIGN FUN	DIRECTION
Forced Path	CRUISE_AIR_DENSITY	CRUISE-DENSITY-FUNCTION	FORWARD
Forced Path	CRUISE_VELOCITY	CRUISE-VELOCITY-FUNCTIO	FORWARD
Forced Path	PAYLOAD_WEIGHT	PAYLOAD-WEIGHT-EQUATION	FORWARD
Forced Path	ENGINE_FRACTION	LENGTH-FRACTION-EQUATIO	FORWARD
Forced Path	PAYLOAD_VOLUME	PAYLOAD-VOLUME-EQUATION	FORWARD
Forced Path	LEADING_EDGE_RADIU	WING-RADIUS-EQUATION	FORWARD
Forced Path	NOSE_RADIUS	NOSE-RADIUS-EQUATION	FORWARD
Loop 1: Initial Path	AFTERBODY_LENGTH	AFTERBODY-LENGTH-EQUATI	FORWARD
Loop 1: Initial Path	WING_SPAN	WING-SPAN-EQUATION	FORWARD
Loop 1: Initial Path	BODY_DIAMETER	BODY-DIAMETER-EQUATION	FORWARD
Loop 1: Initial Path	ENGINE_LENGTH	ENGINE-LENGTH-EQUATION	FORWARD
Loop 1: Initial Path	FOREBODY_LENGTH	FOREBODY-LENGTH-EQUATIO	FORWARD
Loop 1: Initial Path	ENGINE_DIAMETER	ENGINE-DIAMETER-EQUATIO	FORWARD
Loop 1: Initial Path	WING_REFERENCE_ARE	REFERENCE-AREA-EQUATION	FORWARD
Loop 1: Initial Path	TAIL_LENGTH	TAIL-LENGTH-EQUATION	FORWARD
Loop 1: Initial Path	FUEL_VOLUME	FUEL-VOLUME-EQUATION	FORWARD
Loop 1: Initial Path	FUEL_WEIGHT	FUEL-WEIGHT-EQUATION	FORWARD
Loop 1: Initial Path	FUEL_RESERVES	FUEL-RESERVES-EQUATION	FORWARD
Loop 1: Initial Path	TAIL_HEIGHT	TAIL-HEIGHT-EQUATION	FORWARD
Loop 1: Initial Path	INLET_CAPTURE_AREA	CAPTURE-AREA-EQUATION	FORWARD
Loop 1: Initial Path	MAXIMUM_LIFT_COEFF	AERODYNAMICS-PROGRAM	FORWARD
Loop 1: Initial Path	VEHICLE_GROSS_WEIG	PERFORMANCE-PROGRAM	*REVERSE*
Loop 1: Initial Path	TIME_OF_FLIGHT	PERFORMANCE-PROGRAM	*REVERSE*
Loop 1: Branch 1	VEHICLE_EMPTY_WEIG	GROSS-WEIGHT-EQUATION	*REVERSE*
Loop 1: Branch 2	VEHICLE_EMPTY_WEIG	EMPTY-WEIGHT-EQUATION	FORWARD
Loop 1: Final Path	PAYLOAD_WEIGHT_FRA	PAYLOAD-FRACTION-EQUATI	FORWARD
Loop 1: Final Path	FUSELAGE_VOLUME	FUSELAGE-VOLUME-EQUATIO	FORWARD

Figure 6.5: Computational agenda for Third Stage testing.



Chapter 7

Summary and Conclusions

This chapter summarizes the research discussed in the paper, then draws some conclusions about the work done thus far and the work that still needs to be done.

7.1 Summary

In preliminary design, a product idea is represented as a mathematical model; but, because of its extreme complexity, the mathematical model is necessarily broken apart into many sub-models called engineering models.

In an ideal engineering environment, the engineering models would be available in several different layers of complexity, they would account for all parts and properties of the idea at each design level, and their information would be stored in one secure central location. In the real engineering environment, however, enough engineering models just don't exist, and the information of the ones that do exist is scattered all over a company.

Because of advances in computer technology and computer programming, a computer-based engineering model information sharing system (CEMISS) is now a cost-effective prospect to the engineering community. With its recent modifications, *Paper Airplane*, a computer program modified under this research, can now be considered an early prototype of such a CEMISS.

Paper Airplane integrates engineering models by treating them like a system of simultaneous non-linear functions and numerically solving them as such. *Paper Airplane* separates the processing of finding a numerical solution into building a computational agenda and computing the numerical solution. The Agenda Builder examines each func-

tion to find out which inputs and which outputs are known and which are unknown, then figures out the proper sequence in which to evaluate the functions in order to solve for all the unknowns. The Numerical Solvers then follow this agenda to find numerical values for the unknowns once given initialized values for the knowns and guess values for the unknowns.

The original version of *Paper Airplane* could only handle engineering models represented by equations and other MISO functions. The author's research was to expand *Paper Airplane*'s capabilities to handle engineering models represented by computer programs and other MIMO functions. In this way, *Paper Airplane* could graduate from an academic research tool to a professional engineering tool. This research involved the creation of an external code interface capability and a MIMO design function capability.

The external code interface capability was developed based upon research the author performed at Boeing Aerospace. The external codes themselves were not modified, rather *Paper Airplane* would mimic a human user of the code and communicate with it via its standard input and output channels. The external code interface capability was successfully testing on a design set calling an external FORTRAN code. Because of this test, however, several inefficiencies with the Numerical Solvers were discovered and had to be corrected.

The MIMO design function capability was developed based upon research previously done by former M.I.T. professor Antonio Elias, creator of *Paper Airplane*, and by research assistant Mark Kolb, and based upon information obtained from a text on numerical methods. The final version of the MIMO design function capability is a vector form of the Newton-Raphson iteration method. The MIMO design function capability was successfully tested on an "inverse engineering" problem to compute the weights and geometry characteristics of an aircraft given its aerodynamics and performance characteristics.

The modifications required to improve the performance of the Numerical Solvers dealt mainly with the implementation of the methods used, rather than the methods themselves. Even so, total processing time was reduced by a factor of 25.

The design of a national aerospaceplane served as a major and first practical engineering test of *Paper Airplane*'s new and modified capabilities. The 26 design functions included two calling external codes. The first code was a complex aerodynamics program to compute aerodynamics tables for use with the second code, a complex performance program to integrate the range of the vehicle. During testing, *Paper Airplane* success-

fully computed values of range for different vehicle sizes. Then, a range was selected and *Paper Airplane* successfully sized the vehicle to obtain that range. This major test of *Paper Airplane*'s "inverse-engineering" capabilities was performed in only 35 minutes. A problem with LISP did show up during the creation of the design set, however. Because of the poor formatted I/O capabilities of LISP, it became necessary to write short FORTRAN codes to perform the data formatting required of each code's preprocessor and postprocessor. This did not require any change to the *Paper Airplane* External Code Interface, however, since it was successfully tested using unformatted data, as discussed in Chapter 3.

7.2 Conclusions

The tests have proved that *Paper Airplane* can be a valuable tool to the engineer. With the new and modified capabilities added by the author's research, *Paper Airplane* has now graduated from an academic research tool to a professional engineering one. *Paper Airplane* now has the capabilities to perform automatically many of the computer tasks now performed manually by an engineer, such as setting up input files, executing and monitoring codes, and converting output information to data required by another code or by another engineer. *Paper Airplane* also has a user-friendly interface so that the engineer with little programming knowledge can work it as easily as one with expert knowledge. In these two regards, *Paper Airplane* has earned the status of an early prototype of a CEMISS.

To make *Paper Airplane* reach the full status of a CEMISS, however, several more capabilities still need to be added.

1. *Paper Airplane* will require the ability to perform automatic trade studies and optimizations of design variables if it is to be of any value to the modern engineer. *Paper Airplane* already has the ability to compute a "performance function" based upon the weights applied to design variables. What it still needs is a general method to minimize or maximize that performance function and to do it as efficiently as possible.

2. *Paper Airplane* will require a database to house all the design information if it is to properly integrate real engineering models. *Paper Airplane* just doesn't have the memory to store all the information of a design. It needs a link to a database

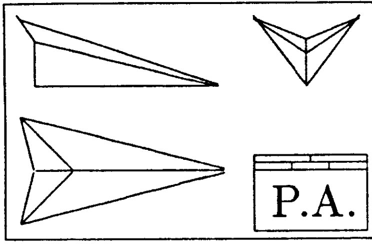
that can store and retrieve design information quickly and efficiently, so that an engineer can acquire any information he or she requires when it is required.

3. *Paper Airplane* will require the ability to handle non-scalar design variables if it is to handle real engineering models. Geometry information is sometimes best handled in a drawing; and tabular information, in a graph. *Paper Airplane* will require the ability to accept design information in this form as well as give it out.

4. *Paper Airplane* will require the ability to communicate with the computers best suited for handling the engineering models. This will require a network interface capability in addition to a modified external code interface.

5. Finally, *Paper Airplane* will require the ability to perform parallel processing of tasks that can be parallel processed. Such an ability would greatly reduced the processing time of numerical searching for the solution well beyond the factor of 25 already accomplished without it.

With all these new abilities, *Paper Airplane* would finally be considered a true CE-MISS. Would this mean that engineers would lose their jobs? No way. Even with all these capabilities, *Paper Airplane* would still be only a computational tool to the modern engineer. The key to finding a good numerical solution quickly is to start with a good intelligent guess at it; such information would still only come from an engineer. An expert system to teach *Paper Airplane* engineering is still a long, long way down the road.



Appendix A

The Test Design Sets

This appendix lists the design variables and design functions used to test the modifications to *Paper Airplane*.

A.1 The MISO Design Set

The MISO Design Set is a set of multiple-input single-output design functions and the design variables they relate. This design set, known to work well with *Paper Airplane* before the modifications were added, served as a foundation for all other design sets, except for the NASP Design Set discussed in Chapter 6.

Table A.1 summarizes the 17 design variables in the MISO Design Set, including its traditional mathematical symbol, its name inside of *Paper Airplane*, and its definition — all of which come straight out of the source file "`MISO.SOU`" listed in Appendix B.

The 7 design functions, which have been labeled simply `DF-1` through `DF-7`, are classical simple relationships which can be found in any book on aircraft design, such as the ones by Torenbeek [17] and Nicolai [14].

`DF-1`: This is an equation for Gross Take-off Weight.

$$W_{gto} = \frac{W_p + W_f}{1 - f_e}$$

`DF-2`: This is a simple equation for Cruise Weight.

$$W_{Cr} = W_{gto} - \frac{2}{3} W_f$$

Symbol	*Paper Airplane* Name	Definition
AR	ASPECT_RATIO	Aspect Ratio of the Wing.
C_D	DRAG_COEFFICIENT	The Drag Coefficient of the aircraft at Cruise.
C_{D_0}	ZERO-LIFT_DRAG_COEFF	The Zero-lift Drag Coefficient of the aircraft.
C_L	LIFT_COEFFICIENT	The Lift Coefficient of the aircraft at Cruise.
ϵ	OSWALD_EFFICIENCY	The Oswald Efficiency of the Wing.
f_e	EMPTY_WEIGHT_FRACTION	The Empty (structural) Weight Fraction of the aircraft.
$^L/_D$	LIFT-TO-DRAG_RATIO	The Lift-to-Drag Ratio of the aircraft at Cruise.
R	RANGE	The Range of the aircraft.
S_{ref}	WING_REFERENCE_AREA	The Reference Area of the Wing.
TSFC	TSFC	The Thrust Specific Fuel Consumption of the Engines.
T_{res}	TIME_ON_RESERVES	The Time Available using the aircraft's Fuel Reserves.
V_{Cr}	CRUISE_VELOCITY	The Velocity of the aircraft at Cruise.
W_{Cr}	CRUISE_WEIGHT	The Weight of the aircraft at Cruise.
W_f	FUEL_WEIGHT	The Weight of the aircraft's Fuel Supply.
W_{gto}	GROSS_TAKE-OFF_WEIGHT	The Gross Weight of the aircraft at Take-off.
$W_{l_{min}}$	MIN_LANDING_WEIGHT	The Minimum Weight of the aircraft at Landing.
W_p	PAYLOAD_WEIGHT	The Weight of the aircraft's Payload.

Table A.1: Design Variables comprising the MISO Design Set.

DF-3: This is an equation for Minimum Landing Weight.

$$W_{l_{min}} = W_{gto} - \left[\frac{R/V_{Cr}}{(R/V_{Cr}) + T_{res}}\right] W_f$$

DF-4: This is a form of the famous Bréguet Range Equation.

$$R = \frac{V_{Cr}}{\text{TSFC}} \frac{L}{D} \log\left(\frac{W_{gto}}{W_{l_{\min}}}\right)$$

DF-5: This is the definition of Lift-to-Drag Ratio.

$$^{L}/_{D} = \frac{C_L}{C_D}$$

DF-6: This is an equation for Lift Coefficient at Cruise.

$$C_L = \frac{W_{Cr}}{1/2 \rho V_{Cr}^2 S_{ref}}$$

DF-7: This is an equation for Drag Coefficient at Cruise.

$$C_D = C_{D_0} + \frac{C_L^2}{\pi A R \epsilon}$$

A.2 The XCODE Design Set

The XCODE Design Set is comprised of the same design variables comprising the MISO Design Set and all but one of the design functions. "DF-4" was replaced with the following short FORTRAN code:

```
program RANGE
c
        real V_CR, TSFC, L_OVER_D, W_GTO, W_MIN, R
c
c       begin
          read(5,*) V_CR, TSFC, L_OVER_D, W_GTO, W_MIN
          R = V_CR / TSFC * L_OVER_D * log( W_GTO / W_MIN )
          write(6,*) R
        end
```

A.3 The MIMO Design Set

The MIMO Design Set is comprised of the same design variables comprising the MISO Design Set plus one: "`WING_SPAN`," the span of the wing from tip to tip, with mathematical symbol b.

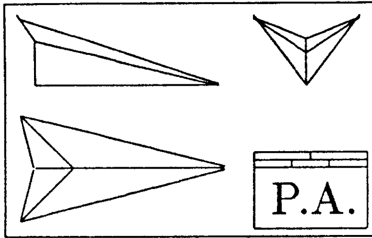
The MIMO Design Set is also comprised of most of the same design functions comprising the MISO Design Set; however MISO design functions "`DF-6`" and "`DF-7`" have been merged to form MIMO design function "`Aerodynamics Package`". Another MIMO design function, "`Geometry Package`" has also been added.

`Aerodynamics Package`: This is the first MIMO, an equation for Lift Coefficient at Cruise and an equation for Drag Coefficient at Cruise.

$$\begin{aligned} C_L &= \frac{W_{Cr}}{1/2\rho V_{Cr}^2 S_{ref}} \\ C_D &= C_{D_0} + \frac{C_L^2}{\pi AR\epsilon} \end{aligned}$$

`Geometry Package`: This is the second MIMO, an equation for Wing Reference Area and a simple equation for Oswald Efficiency.

$$\begin{aligned} S_{ref} &= \frac{b^2}{AR} \\ \epsilon &= 0.8 \end{aligned}$$



Appendix B

MISO Design Set Source File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                        ;;
;;        AM410 Laser Executive Transport Aircraft        ;;
;;                           a                            ;;
;;         design experiment using Paper Airplane         ;;
;;                         and a                          ;;
;;  running example for the Paper Airplane User's Manual  ;;
;;                                                        ;;
;;                 Conceptual Design Level                ;;
;;            Tutorial Design Set -- Number 1             ;;
;;                                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;; CONSTANTS ;;;;;;;;;;;;;;;;;;;;;;;;

(defconstant *AIR_DENSITY* 0.000738)     ; slugs ft-3
(defconstant *PI-D*        3.14159265)   ;


;;;;;;;;;;;;;;;;;;;;; DESIGN VARIABLES ;;;;;;;;;;;;;;;;;;;

(pa-defvar ASPECT_RATIO
           :category (geometry wing)
           :documentation "Aspect Ratio of the Wing."
           :TeX-name "$AR$"
           :order-of-magnitude 8
           :lower-value 5
           :upper-value 20
           :dimensions "")

(pa-defvar DRAG_COEFFICIENT
```

```
            :category aerodynamics
            :documentation "The Drag Coefficient of the aircraft at Cruise."
            :TeX-name "$C_D$"
            :order-of-magnitude 0.02
            :lower-value 0.01
            :upper-value 0.04
            :dimensions "")

(pa-defvar ZERO-LIFT_DRAG_COEFF
            :category aerodynamics
            :documentation "The Zero-lift Drag Coefficient of the aircraft."
            :TeX-name "$C_{D_0}$"
            :order-of-magnitude 0.0150
            :lower-value 0.0100
            :upper-value 0.0300
            :dimensions "")

(pa-defvar LIFT_COEFFICIENT
            :category aerodynamics
            :documentation "The Lift Coefficient of the aircraft at Cruise."
            :TeX-name "$C_L$"
            :order-of-magnitude 0.3
            :lower-value 0.1
            :upper-value 0.5
            :dimensions "")

(pa-defvar OSWALD_EFFICIENCY
            :category (geometry wing)
            :documentation "The Oswald Efficiency of the Wing."
            :TeX-name "$\\epsilon$"
            :order-of-magnitude 0.80
            :lower-value 0.70
            :upper-value 0.90
            :dimensions "")

(pa-defvar EMPTY_WEIGHT_FRACTION
            :category weights
            :documentation "The Empty (structural) Weight Fraction of the aircraft."
            :TeX-name "$f_e$"
            :order-of-magnitude 0.60
            :lower-value 0.55
            :upper-value 0.65
            :dimensions "")

(pa-defvar LIFT-TO-DRAG_RATIO
            :category aerodynamics
            :documentation "The Lift-to-Drag Ratio of the aircraft at Cruise."
            :TeX-name "$^L/_D$"
            :order-of-magnitude 15
            :lower-value 10
```

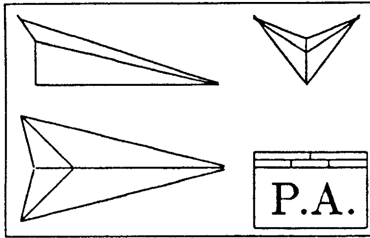
```
            :upper-value 20
            :dimensions "")

(pa-defvar RANGE
            :category (performance cruise)
            :documentation "The Range of the aircraft."
            :TeX-name "$R$"
            :order-of-magnitude 3000
            :lower-value 3000
            :upper-value 3000
            :dimensions "l"
            :default-units "sm")

(pa-defvar WING_REFERENCE_AREA
            :category (geometry wing)
            :documentation "The Reference Area of the Wing."
            :TeX-name "$S_{ref}$"
            :order-of-magnitude 261.0
            :lower-value 220.0
            :upper-value 300.0
            :dimensions "l2"
            :default-units "ft2")

(pa-defvar TSFC
            :category propulsion
            :documentation "Thrust Specific Fuel Consumption of the engines."
            :TeX-name "TSFC"
            :order-of-magnitude 0.8
            :lower-value 0.8
            :upper-value 0.8
            :dimensions "m f-1 t-1"
            :default-units "lb lbf-1 hr-1")

(pa-defvar TIME_ON_RESERVES
            :category (performance cruise)
            :documentation "The Time available using the aircraft's fuel reserves."
            :TeX-name "$T_{res}$"
            :order-of-magnitude 0.75
            :lower-value 0.0
            :upper-value 2.0
            :dimensions "t"
            :default-units "hr")

(pa-defvar CRUISE_VELOCITY
            :category (performance cruise)
            :documentation "The Velocity of the aircraft at Cruise."
            :TeX-name "$V_{Cr}$"
            :order-of-magnitude 565.0
            :lower-value 525.0
            :upper-value 600.0
```

```
           :dimensions "l t-1"
           :default-units "sm hr-1")

(pa-defvar CRUISE_WEIGHT
           :category weights
           :documentation "The Weight of the aircraft at Cruise."
           :TeX-name "$W_{Cr}$"
           :order-of-magnitude 13300
           :lower-value 10000
           :upper-value 20000
           :dimensions "f"
           :default-units "lbf")

(pa-defvar FUEL_WEIGHT
           :category weights
           :documentation "The Weight of the aircraft's Fuel Supply."
           :TeX-name "$W_f$"
           :order-of-magnitude 4000
           :lower-value 2000
           :upper-value 6000
           :dimensions "f"
           :default-units "lbf")

(pa-defvar GROSS_TAKE-OFF_WEIGHT
           :category weights
           :documentation "The Weight of the aircraft at Take-off."
           :TeX-name "$W_{gto}$"
           :order-of-magnitude 15000
           :lower-value 10000
           :upper-value 20000
           :dimensions "f"
           :default-units "lbf")

(pa-defvar MIN_LANDING_WEIGHT
           :category weights
           :documentation "The Minimum Weight of the aircraft at Landing."
           :TeX-name "$W_{l_{\\rm min}}$"
           :order-of-magnitude 11000
           :lower-value 6000
           :upper-value 15000
           :dimensions "f"
           :default-units "lbf")

(pa-defvar PAYLOAD_WEIGHT
           :category weights
           :documentation "The Weight of the aircraft's Payload."
           :TeX-name "$W_p$"
           :order-of-magnitude 2200
           :lower-value 2000
           :upper-value 2400
```

```
           :dimensions "f"
           :default-units "lbf")

;;;;;;;;;;;;;;;;;;;; DESIGN FUNCTIONS ;;;;;;;;;;;;;;;;;;;

(pa-defun DF-1
          :category weights
          :computed-variable (GROSS_TAKE-OFF_WEIGHT "lbf")
          :input-variables ((PAYLOAD_WEIGHT "lbf")
                            (FUEL_WEIGHT "lbf")
                            (EMPTY_WEIGHT_FRACTION ""))
          :function-body (/ (+ PAYLOAD_WEIGHT FUEL_WEIGHT)
                            (- 1 EMPTY_WEIGHT_FRACTION))
          :TeX-name "$W_{gto} {} = {} {{W_p + W_f} \\over {1 - f_e}}$"
          :documentation "Gross Take-off Weight Equation.")


(pa-defun DF-2
          :category weights
          :computed-variable (CRUISE_WEIGHT "lbf")
          :input-variables ((GROSS_TAKE-OFF_WEIGHT "lbf")
                            (FUEL_WEIGHT "lbf"))
          :function-body (- GROSS_TAKE-OFF_WEIGHT (/ FUEL_WEIGHT 3.0))
          :TeX-name "$W_{Cr} {} = {} W_{gto} - {2 \\over 3} W_f$"
          :documentation "Cruise Weight Equation.")


(pa-defun DF-3
          :category weights
          :computed-variable (MIN_LANDING_WEIGHT "lbf")
          :input-variables ((GROSS_TAKE-OFF_WEIGHT "lbf")
                            (FUEL_WEIGHT "lbf")
                            (RANGE "sm")
                            (CRUISE_VELOCITY "sm hr-1")
                            (TIME_ON_RESERVES "hr"))
          :function-body (let ((ENDURANCE (/ RANGE CRUISE_VELOCITY)))
                           (- GROSS_TAKE-OFF_WEIGHT
                              (* FUEL_WEIGHT
                                 (/ ENDURANCE
                                    (+ ENDURANCE TIME_ON_RESERVES)))))
          :TeX-name "$W_{l_{min}} {} = {} W_{gto} - W_f
                      \\bigl[{{R/V_{Cr}} \\over
                               {({R/{V_{Cr}})}} + T_{res}}} \\bigr]$"
          :documentation "Minimum Landing Weight Equation.")


(pa-defun DF-4
          :category (performance cruise)
          :computed-variable (RANGE "sm")
          :input-variables ((CRUISE_VELOCITY "sm hr-1")
```

```
                    (TSFC "lb lbf-1 hr-1")
                    (LIFT-TO-DRAG_RATIO "")
                    (GROSS_TAKE-OFF_WEIGHT "lbf")
                    (MIN_LANDING_WEIGHT "lbf"))
      :function-body (* (/ CRUISE_VELOCITY TSFC)
                    LIFT-TO-DRAG_RATIO
                    (log (/ GROSS_TAKE-OFF_WEIGHT MIN_LANDING_WEIGHT)))
      :TeX-name "$R {} = {} {V_{Cr} \\over {\\rm TSFC}} {L \\over D} \\log
                    \\bigl({W_{gto} \\over W_{l_{\\rm min}}} \\bigr)$"
      :documentation "Breguet Range Equation.")


(pa-defun DF-5
      :category aerodynamics
      :computed-variable (LIFT-TO-DRAG_RATIO "")
      :input-variables ((LIFT_COEFFICIENT "")
                    (DRAG_COEFFICIENT ""))
      :function-body (/ LIFT_COEFFICIENT DRAG_COEFFICIENT)
      :TeX-name "$^L/_D {} = {} {C_L \\over C_D}$"
      :documentation "Lift-to-Drag Ratio Equation.")

(pa-defun DF-6
      :category aerodynamics
      :computed-variable (LIFT_COEFFICIENT "")
      :input-variables ((CRUISE_WEIGHT "lbf")
                    (FUEL_WEIGHT "lbf")
                    (CRUISE_VELOCITY "ft s-1")
                    (WING_REFERENCE_AREA "ft2"))
      :function-body (/ CRUISE_WEIGHT
                    (* 0.5 *AIR_DENSITY* CRUISE_VELOCITY
                       CRUISE_VELOCITY WING_REFERENCE_AREA))
      :TeX-name "$C_L {} = {} {W_{Cr}
                        \\over {1/2 \\rho V_{Cr}^2 S_{ref}}}$"
      :documentation "Lift Coefficient Equation.")


(pa-defun DF-7
      :category aerodynamics
      :computed-variable (DRAG_COEFFICIENT "")
      :input-variables ((ZERO-LIFT_DRAG_COEFF "")
                    (LIFT_COEFFICIENT "")
                    (ASPECT_RATIO "")
                    (OSWALD_EFFICIENCY ""))
      :function-body (+ ZERO-LIFT_DRAG_COEFF
                    (/ (* LIFT_COEFFICIENT LIFT_COEFFICIENT)
                       (* *PI-D* ASPECT_RATIO OSWALD_EFFICIENCY)))
      :TeX-name "$C_D {} = {} C_{D_0} + {C_L^2 \\over {\\pi AR \\epsilon}}$"
      :documentation "Drag Coefficient Equation.")
```

```
;;;;;;;;;;;;;;;;;;;;; DESIGN SETS ;;;;;;;;;;;;;;;;;;;;;

(pa-defset laser
           :design-variables
           (aspect_ratio lift_coefficient drag_coefficient lift-to-drag_ratio
                         gross_take-off_weight payload_weight fuel_weight
                         zero-lift_drag_coeff cruise_weight cruise_velocity
                         range tsfc oswald_efficiency min_landing_weight
                         empty_weight_fraction wing_reference_area
                         time_on_reserves)
           :design-functions (df-1 df-2 df-3 df-4 df-5 df-6 df-7)
           :tableaux
           ((aerodynamics
              (WING_REFERENCE_AREA "ft2")
              (ASPECT_RATIO "")
              (OSWALD_EFFICIENCY "")
              (LIFT-TO-DRAG_RATIO "")
              (LIFT_COEFFICIENT "")
              (DRAG_COEFFICIENT "")
              (ZERO-LIFT_DRAG_COEFF ""))
            (cruise
              (RANGE "sm")
              (CRUISE_VELOCITY "sm hr-1")
              (TSFC "lb lbf-1 hr-1")
              (LIFT-TO-DRAG_RATIO "")
              (GROSS_TAKE-OFF_WEIGHT "lbf")
              (MIN_LANDING_WEIGHT "lbf")
              (TIME_ON_RESERVES "hr"))
            (weights
              (GROSS_TAKE-OFF_WEIGHT "lbf")
              (PAYLOAD_WEIGHT "lbf")
              (FUEL_WEIGHT "lbf")
              (MIN_LANDING_WEIGHT "lbf")
              (CRUISE_WEIGHT "lbf")
              (EMPTY_WEIGHT_FRACTION ""))))
```



Appendix C

MISO Design Set Solution

```
                    Processing design set LASER for RML
            on Tuesday the fourth of February, 1986; 2:36:46 pm.


Building new agenda ...

FORCED-PATH CONSTRUCTION--
The function chosen for processing is DF-1,
with two trial-state guess-variables:
          G -- GROSS_TAKE-OFF_WEIGHT
          I -- PAYLOAD_WEIGHT
          G -- FUEL_WEIGHT
          I -- EMPTY_WEIGHT_FRACTION
Equally good functions at this stage in processing are:
          DF-7

FORCED-PATH for LASER's agenda completed.


LOOP CONSTRUCTION-- forcing variable
The variable chosen as the F-variable for this loop is
          GROSS_TAKE-OFF_WEIGHT
which occurs four times in the remaining functions.
Other equally good variables at this stage in processing are:
          FUEL_WEIGHT

LOOP CONSTRUCTION--  top entry
The function chosen for processing is DF-1,
with one trial-state guess-variable:
          F -- GROSS_TAKE-OFF_WEIGHT
          I -- PAYLOAD_WEIGHT
          G -- FUEL_WEIGHT
```

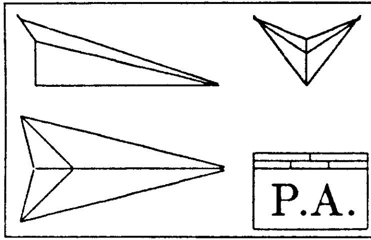
```
          I -- EMPTY_WEIGHT_FRACTION
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-3,
with one trial-state guess-variable:
          G -- MIN_LANDING_WEIGHT
          F -- GROSS_TAKE-OFF_WEIGHT
          C -- FUEL_WEIGHT
          I -- RANGE
          I -- CRUISE_VELOCITY
          I -- TIME_ON_RESERVES
Equally good functions at this stage in processing are:
          DF-2

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-4,
with one trial-state guess-variable:
          I -- RANGE
          I -- CRUISE_VELOCITY
          I -- TSFC
          G -- LIFT-TO-DRAG_RATIO
          F -- GROSS_TAKE-OFF_WEIGHT
          C -- MIN_LANDING_WEIGHT
Equally good functions at this stage in processing are:
          DF-2

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-2,
with one trial-state guess-variable:
          G -- CRUISE_WEIGHT
          F -- GROSS_TAKE-OFF_WEIGHT
          C -- FUEL_WEIGHT
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-6,
with one trial-state guess-variable:
          G -- LIFT_COEFFICIENT
          C -- CRUISE_WEIGHT
          C -- FUEL_WEIGHT
          I -- CRUISE_VELOCITY
          I -- WING_REFERENCE_AREA
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-7,
with one trial-state guess-variable:
          G -- DRAG_COEFFICIENT
          I -- ZERO-LIFT_DRAG_COEFF
```

```
          C -- LIFT_COEFFICIENT
          I -- ASPECT_RATIO
          I -- OSWALD_EFFICIENCY
Equally good functions at this stage in processing are:
          DF-5

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-5,
with zero trial-state guess-variables:
          C -- LIFT-TO-DRAG_RATIO
          C -- LIFT_COEFFICIENT
          C -- DRAG_COEFFICIENT
No other equally good functions.

FORCED FUNCTION detected:  DF-5
LOOPING VARIABLE chosen:  DRAG_COEFFICIENT.   (Forcing variable is GROSS_TAKE-OFF_WEIGHT.)
Transferring entry using DF-7 to calculate DRAG_COEFFICIENT to loop first branch ...
NO subsequent entries to transfer to second branch.

LOOP CONSTRUCTION-- forced function
Closing second branch with function DF-5, computing
loop variable DRAG_COEFFICIENT.  Function variables are:
          C -- LIFT-TO-DRAG_RATIO
          C -- LIFT_COEFFICIENT
          L -- DRAG_COEFFICIENT
No other forced functions.
Reviewing loop branches and preliminary entries ...

LOOPS for LASER's agenda completed.


Beginning forced path computations ...

Forced path computations completed.

Beginning LOOP computation: GROSS_TAKE-OFF_WEIGHT forces DRAG_COEFFICIENT.
   Searching for consistent value of forcing variable ...
   Searching between  10000.00000 lbf and  20000.00000 lbf (zeroth level) ...
     Considering search value:     GROSS_TAKE-OFF_WEIGH    10000.00000 lbf
           First branch value:     DRAG_COEFFICIENT              0.01606
          Second branch value:     DRAG_COEFFICIENT              0.00772
                   Difference:                                   0.00834
     Considering search value:     GROSS_TAKE-OFF_WEIGH    10905.07733 lbf
           First branch value:     DRAG_COEFFICIENT              0.01624
          Second branch value:     DRAG_COEFFICIENT              0.00912
                   Difference:                                   0.00712
     Considering search value:     GROSS_TAKE-OFF_WEIGH    11892.07115 lbf
           First branch value:     DRAG_COEFFICIENT              0.01646
          Second branch value:     DRAG_COEFFICIENT              0.01065
                   Difference:                                   0.00581
```

```
  Considering search value:     GROSS_TAKE-OFF_WEIGH    12968.39555 lbf
        First branch value:     DRAG_COEFFICIENT            0.01671
       Second branch value:     DRAG_COEFFICIENT            0.01232
                Difference:                                 0.00440
  Considering search value:     GROSS_TAKE-OFF_WEIGH    14142.13562 lbf
        First branch value:     DRAG_COEFFICIENT            0.01702
       Second branch value:     DRAG_COEFFICIENT            0.01414
                Difference:                                 0.00287
  Considering search value:     GROSS_TAKE-OFF_WEIGH    15000.00000 lbf
        First branch value:     DRAG_COEFFICIENT            0.01725
       Second branch value:     DRAG_COEFFICIENT            0.01547
                Difference:                                 0.00179
  Considering search value:     GROSS_TAKE-OFF_WEIGH    15422.10825 lbf
        First branch value:     DRAG_COEFFICIENT            0.01737
       Second branch value:     DRAG_COEFFICIENT            0.01614
                Difference:                                 0.00124
  Considering search value:     GROSS_TAKE-OFF_WEIGH    16817.92831 lbf
        First branch value:     DRAG_COEFFICIENT            0.01780
       Second branch value:     DRAG_COEFFICIENT            0.01831
                Difference:                                 0.00051
Determining new search interval ...

Searching between  15422.10825 lbf and  16817.92831 lbf (first level) ...
  Considering search value:     GROSS_TAKE-OFF_WEIGH    15422.10825 lbf
        First branch value:     DRAG_COEFFICIENT            0.01737
       Second branch value:     DRAG_COEFFICIENT            0.01614
                Difference:                                 0.00124
  Considering search value:     GROSS_TAKE-OFF_WEIGH    15874.01052 lbf
        First branch value:     DRAG_COEFFICIENT            0.01751
       Second branch value:     DRAG_COEFFICIENT            0.01685
                Difference:                                 0.00066
  Considering search value:     GROSS_TAKE-OFF_WEIGH    16339.15453 lbf
        First branch value:     DRAG_COEFFICIENT            0.01765
       Second branch value:     DRAG_COEFFICIENT            0.01757
                Difference:                                 0.00008
  Considering search value:     GROSS_TAKE-OFF_WEIGH    16817.92831 lbf
        First branch value:     DRAG_COEFFICIENT            0.01780
       Second branch value:     DRAG_COEFFICIENT            0.01831
                Difference:                                 0.00051
Determining new search interval ...
Searching between  16339.15453 lbf and  16817.92831 lbf (second level) ...
  Considering search value:     GROSS_TAKE-OFF_WEIGH    16339.15453 lbf
        First branch value:     DRAG_COEFFICIENT            0.01765
       Second branch value:     DRAG_COEFFICIENT            0.01757
                Difference:                                 0.00008
  Considering search value:     GROSS_TAKE-OFF_WEIGH    16497.21189 lbf
        First branch value:     DRAG_COEFFICIENT            0.01770
       Second branch value:     DRAG_COEFFICIENT            0.01782
                Difference:                                 0.00012
Determining new search interval ...
```

```
  Searching between  16339.15453 lbf and  16497.21189 lbf (third level) ...
    Considering search value:     GROSS_TAKE-OFF_WEIGH    16339.15453 lbf
          First branch value:     DRAG_COEFFICIENT            0.01765
         Second branch value:     DRAG_COEFFICIENT            0.01757
                  Difference:                                 0.00008
    Considering search value:     GROSS_TAKE-OFF_WEIGH    16391.67134 lbf
          First branch value:     DRAG_COEFFICIENT            0.01767
         Second branch value:     DRAG_COEFFICIENT            0.01767
                  Difference:                                 0.00000
  Logarithmic distribution search has computed consistent value of  16391.67134 lbf
  for forcing variable GROSS_TAKE-OFF_WEIGHT (search level depth of three).

Value chosen for forcing variable, GROSS_TAKE-OFF_WEIGHT:  16391.67134 lbf
  Reverse computing FUEL_WEIGHT using function DF-1, based on
            GROSS_TAKE-OFF_WEIGHT    C      16391.671338 lbf
            PAYLOAD_WEIGHT           I       2200.000000 lbf
            EMPTY_WEIGHT_FRACTION    I          0.550000
    N-R iteration successful (one recursion).
  Assigning 5176.252102294063 lbf to FUEL_WEIGHT.
  Forward computing MIN_LANDING_WEIGHT using function DF-3, based on
            GROSS_TAKE-OFF_WEIGHT    C      16391.671338 lbf
            FUEL_WEIGHT              C       5176.252102 lbf
            RANGE                    I       3000.000000 sm
            CRUISE_VELOCITY          I        565.000000 sm hr-1
            TIME_ON_RESERVES         I          0.750000 hr
  Assigning 11856.0725631462 lbf to MIN_LANDING_WEIGHT.

  Reverse computing LIFT-TO-DRAG_RATIO using function DF-4, based on
            RANGE                    I       3000.000000 sm
            CRUISE_VELOCITY          I        565.000000 sm hr-1
            TSFC                     I          0.800000 lb lbf-1 hr-1
            GROSS_TAKE-OFF_WEIGHT    C      16391.671338 lbf
            MIN_LANDING_WEIGHT       C      11856.072563 lbf
    N-R iteration successful (one recursion).
  Assigning 13.11316031091615  to LIFT-TO-DRAG_RATIO.
  Forward computing CRUISE_WEIGHT using function DF-2, based on
            GROSS_TAKE-OFF_WEIGHT    C      16391.671338 lbf
            FUEL_WEIGHT              C       5176.252102 lbf
  Assigning 14666.2539709999 lbf to CRUISE_WEIGHT.
  Forward computing LIFT_COEFFICIENT using function DF-6, based on
            CRUISE_WEIGHT            C      14666.253971 lbf
            FUEL_WEIGHT              C       5176.252102 lbf
            CRUISE_VELOCITY          I        565.000000 sm hr-1
            WING_REFERENCE_AREA      I        250.000000 ft2
  Assigning 0.231522438115427  to LIFT_COEFFICIENT.
  Forward computing DRAG_COEFFICIENT using function DF-7, based on
            ZERO-LIFT_DRAG_COEFF     I          0.015000
            LIFT_COEFFICIENT         C          0.231522
            ASPECT_RATIO             I          8.000000
            OSWALD_EFFICIENCY        I          0.800000
```

```
  Assigning 1.7665976570380618e-02  to DRAG_COEFFICIENT.
  Reverse computing DRAG_COEFFICIENT using function DF-5, based on
            LIFT-TO-DRAG_RATIO        C         13.113160
            LIFT_COEFFICIENT          C          0.231522
    N-R iteration successful (one recursion).
  Assigning 1.7665976570380618e-02  to DRAG_COEFFICIENT.

GROSS_TAKE-OFF_WEIGHT/DRAG_COEFFICIENT loop computations completed.
```

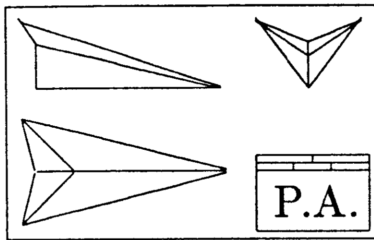


Appendix D

The "RUN-PROGRAM" Code

```
(defun run-program (&key
                    ((:program-name progname) nil)
                    ((:program-directory progdir) nil)
                    ((:file-directory filedir) progdir)
                    (preprocessor (cat-name progname '-preprocessor))
                    (postprocessor (cat-name progname '-postprocessor))
                    (sys$input-file (string-append filedir progname ".in"))
                    (sys$output-file (string-append filedir progname ".out"))
                    (sys$error-file (string-append filedir progname ".err"))
                    ((:monitor-file outfile) sys$output-file)
                    ((:average-run-time runtime) 10)        ; SECONDS
                    ((:overtime-allowance overtime) 20)     ; PERCENT
                    (verbose nil))
  (let* ((steps 20) (maxcount (* (/ (+ 20 overtime) 100) steps)))
    (if preprocessor (funcall preprocessor))
    (when (execute-program
            (string-append progdir progname)            ; EXECUTABLE
            sys$input-file sys$output-file sys$error-file ; FILES
            (string progname))                          ; PROCESS NAME
      (if verbose (format t "~&Executing ~a ..." progname))
      (sleep (* 0.8 runtime))                           ; INITIAL WAIT
      (do ((count 0 (+ count 1)))
          ((or (probe-file outfile) (probe-file sys$error-file)
               (greaterp count maxcount))
           (cond ((greaterp count maxcount) (program-time-overrun))
                 ((probe-file sys$error-file) (program-error sys$error-file))
                 (t (if postprocessor (funcall postprocessor)))))
        (if verbose (format t "~&Waiting for ~a to finish ..." progname))
        (sleep (/ runtime steps))))))                   ; MONITOR & WAIT
```

```
(defun program-error (error-file)
  (with-open-file (errfile error-file)
    (format t "~2&~a" ">>ERROR in program execution.  Contents to follow.")
    (print errfile)
    (do ((errline (readline errfile) (readline errfile)))
        ((null errline))
      (format t "~&~a" errline))
    (print "All done")))


(defun program-time-overrun ()
  (format t "~2&~a" ">>ERROR: program execution surpassed time limit."))
```

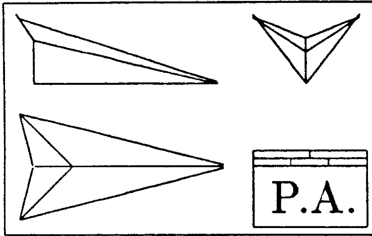


Appendix E

Vector Equation Solving Code

```
(defun solve-LU-factorization (matrix pivot-vector y-vector)
  (let* ((dim (length y-vector)) (N (1- dim)) (sum 0.0)
         (x-vector (make-array dim))
         (pivot-point (aref pivot-vector 0)))
    (setf (aref x-vector 0) (aref y-vector pivot-point))
    (do ((row 1 (1+ row))) ((> row N))                ;;; FORWARD SUBSTITUTION
      (setq sum 0.0)
      (do ((col 0 (1+ col))) ((> col (- row 1)))
        (setq sum (+ sum (* (aref matrix row col) (aref x-vector col)))))
      (setq pivot-point (aref pivot-vector row))
      (setf (aref x-vector row) (- (aref y-vector pivot-point) sum)))
    (setf (aref x-vector N) (/ (aref x-vector N) (aref matrix N N)))
    (do ((row (- N 1) (1- row))) ((< row 0))          ;;; BACKWARD SUBSTITUTION
      (setq sum 0.0)
      (do ((col (+ row 1) (1+ col))) ((> col N))
        (setq sum (+ sum (* (aref matrix row col) (aref x-vector col)))))
      (setf (aref x-vector row)
            (/ (- (aref x-vector row) sum) (aref matrix row row))))
    x-vector))
```

```
(defun get-LU-factorization (matrix)
  (let* ((dim (car (array-dimensions matrix)))
         (N (- dim 1)) (new-col-max 0) (ratio 0)
         (row-max 0) (col-max 0) (pivot-row 0) (temp nil)
         (pivot-vector (make-array dim))
         (row-max-vector (make-array dim)))
    (dotimes (row dim)                              ;;; FIND ROW-MAX-VECTOR
      (setf (aref pivot-vector row) row)
      (setq row-max 0)
      (dotimes (col dim)
        (setq row-max (max row-max (abs (aref matrix row col)))))
      (if (= row-max 0) (return nil)
        (setf (aref row-max-vector row) row-max)))
    (dotimes (K N)                                  ;;; FIND LU-FACTORIZATION
      (setq col-max (/ (abs (aref matrix K K)) (aref row-max-vector K)))
      (setq pivot-row K)
      (do ((row (+ K 1) (1+ row))) ((> row N))
        (setq new-col-max (/ (abs (aref matrix row K))
                             (aref row-max-vector row)))
        (when (greaterp new-col-max col-max)
          (setq col-max new-col-max)
          (setq pivot-row row)))
      (if (= col-max 0) (return nil))
      (when (greaterp pivot-row K)                  ;;; INTERCHANGE INFORMATION
        (setq temp (aref pivot-vector pivot-row))    ;;; SWITCH PIVOT ROWS
        (setf (aref pivot-vector pivot-row) (aref pivot-vector K))
        (setf (aref pivot-vector K) temp)
        (setq temp (aref row-max-vector pivot-row))  ;;; SWITCH ROW-MAX'S
        (setf (aref row-max-vector pivot-row) (aref row-max-vector K))
        (setf (aref row-max-vector K) temp)
        (dotimes (col dim)                           ;;; SWITCH MATRIX ROWS
          (setq temp (aref matrix pivot-row col))
          (setf (aref matrix pivot-row col) (aref matrix K col))
          (setf (aref matrix K col) temp)))
      (do ((row (+ K 1) (1+ row))) ((> row N))     ;;; SIMPLIFY LOWER ROWS
        (setf (aref matrix row K) (/ (aref matrix row K) (aref matrix K K)))
        (setq ratio (aref matrix row K))
        (do ((J (+ K 1) (1+ J))) ((> J N))
          (setf (aref matrix row J)
                (- (aref matrix row J) (* ratio (aref matrix K J)))))))
    (if (zerop (aref matrix N N)) nil pivot-vector)))
```



Appendix F

MIMO Design Set Solution

```
                    Processing design set LASER for RML
          on Thursday the eleventh of September, 1986; 5:37:26 pm.


Building new agenda ...

FORCED-PATH CONSTRUCTION--
The function chosen for processing is DF-5,
with zero extra trial-state guess-variables:
         G -- LIFT-TO-DRAG_RATIO
         I -- LIFT_COEFFICIENT
         I -- DRAG_COEFFICIENT
No other equally good functions.

FORCED-PATH CONSTRUCTION--
The function chosen for processing is DF-1,
with one extra trial-state guess-variable:
         G -- GROSS_TAKE-OFF_WEIGHT
         I -- PAYLOAD_WEIGHT
         G -- FUEL_WEIGHT
         I -- EMPTY_WEIGHT_FRACTION
No other equally good functions.

FORCED-PATH for LATER's agenda completed.


LOOP CONSTRUCTION-- forcing variable
The variable chosen as the F-variable for this loop is
         GROSS_TAKE-OFF_WEIGHT
which occurs four times in the remaining functions.
Other equally good variables at this stage in processing are:
         FUEL_WEIGHT
```

```
        ASPECT_RATIO
        CRUISE_WEIGHT
        MIN_LANDING_WEIGHT
        OSWALD_EFFICIENCY
        WING_REFERENCE_AREA
        WING_SPAN

LOOP CONSTRUCTION--  top entry
The function chosen for processing is DF-1,
with zero extras trial-state guess-variables:
        F -- GROSS_TAKE-OFF_WEIGHT
        I -- PAYLOAD_WEIGHT
        G -- FUEL_WEIGHT
        I -- EMPTY_WEIGHT_FRACTION
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-4,
with zero extra trial-state guess-variables:
        I -- RANGE
        I -- CRUISE_VELOCITY
        I -- TSFC
        K -- LIFT-TO-DRAG_RATIO
        F -- GROSS_TAKE-OFF_WEIGHT
        G -- MIN_LANDING_WEIGHT
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is DF-3,
with minus one extra trial-state guess-variables:
        C -- MIN_LANDING_WEIGHT
        F -- GROSS_TAKE-OFF_WEIGHT
        C -- FUEL_WEIGHT
        I -- RANGE
        I -- CRUISE_VELOCITY
        I -- TIME_ON_RESERVES
No other equally good functions.

FORCED FUNCTION detected:  DF-3

LOOP CONSTRUCTION-- looping variable
The variable chosen as the L-variable for this loop is
        MIN_LANDING_WEIGHT
The forcing variable is GROSS_TAKE-OFF_WEIGHT

Transferring entry using function DF-4
to calculate (MIN_LANDING_WEIGHT) to loop first branch ...
NO subsequent entries to transfer to second branch.

LOOP CONSTRUCTION-- forced function
```

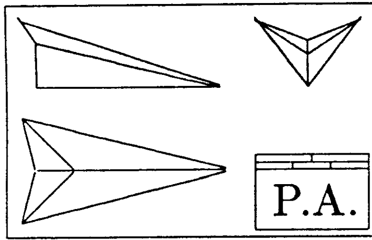
```
Closing second branch with function DF-3, computing
loop variable MIN_LANDING_WEIGHT.  Function variables are:
        L -- MIN_LANDING_WEIGHT
        F -- GROSS_TAKE-OFF_WEIGHT
        C -- FUEL_WEIGHT
        I -- RANGE
        I -- CRUISE_VELOCITY
        I -- TIME_ON_RESERVES
No other forced functions.

Reviewing loop branches and preliminary entries ...
Transferring entry using DF-1 to compute
(FUEL_WEIGHT) to beginning of loop final entries.

LOOP CONSTRUCTION-- final path
The function chosen for processing is DF-2,
with zero trial-state guess-variables:
        G -- CRUISE_WEIGHT
        F -- GROSS_TAKE-OFF_WEIGHT
        C -- FUEL_WEIGHT
No other equally good functions.

LOOP CONSTRUCTION-- final path
The function chosen for processing is Aerodynamics Package,
with one trial-state guess-variable:
        I -- LIFT_COEFFICIENT
        I -- DRAG_COEFFICIENT
        C -- CRUISE_WEIGHT
        I -- CRUISE_VELOCITY
        G -- WING_REFERENCE_AREA
        I -- ZERO-LIFT_DRAG_COEFF
        G -- ASPECT_RATIO
        G -- OSWALD_EFFICIENCY
No other equally good functions.

LOOP CONSTRUCTION completed.

LOOP CONSTRUCTION-- forcing variable
The variable chosen as the F-variable for this loop is
        ASPECT_RATIO
which occurs two times in the remaining functions.
Other equally good variables at this stage in processing are:
        OSWALD_EFFICIENCY
        WING_REFERENCE_AREA
        WING_SPAN

LOOP CONSTRUCTION--  top entry
The function chosen for processing is Aerodynamics Package,
with zero extras trial-state guess-variables:
        I -- LIFT_COEFFICIENT
```

```
         I -- DRAG_COEFFICIENT
         C -- CRUISE_WEIGHT
         I -- CRUISE_VELOCITY
         G -- WING_REFERENCE_AREA
         I -- ZERO-LIFT_DRAG_COEFF
         F -- ASPECT_RATIO
         G -- OSWALD_EFFICIENCY
No other equally good functions.

LOOP CONSTRUCTION-- initial path
The function chosen for processing is Wing Geometry Package,
with minus one extra trial-state guess-variables:
         C -- WING_REFERENCE_AREA
         C -- OSWALD_EFFICIENCY
         F -- ASPECT_RATIO
         G -- WING_SPAN
No other equally good functions.

FORCED FUNCTION detected:  Wing Geometry Package

LOOP CONSTRUCTION-- looping variable
The variable chosen as the L-variable for this loop is
         OSWALD_EFFICIENCY
The forcing variable is ASPECT_RATIO

Transferring entry using function Aerodynamics Package
to calculate (OSWALD_EFFICIENCY WING_REFERENCE_AREA) to loop first branch ...
NO subsequent entries to transfer to second branch.

LOOP CONSTRUCTION-- forced function
Closing second branch with function Wing Geometry Package, computing
loop variable OSWALD_EFFICIENCY.  Function variables are:
         C -- WING_REFERENCE_AREA
         L -- OSWALD_EFFICIENCY
         F -- ASPECT_RATIO
         G -- WING_SPAN
No other forced functions.

Reviewing loop branches and preliminary entries ...

LOOP CONSTRUCTION completed.

LOOPS for LATER's agenda completed.

Agenda construction completed.

Beginning forced path computations ...
   Forward computing (LIFT-TO-DRAG_RATIO) using function DF-5, based on
            LIFT_COEFFICIENT         I          0.240000
            DRAG_COEFFICIENT         I          0.018000
```

```
   Assigning 13.33333333333333  to LIFT-TO-DRAG_RATIO.

Forced path computations completed.

Beginning LOOP computation: GROSS_TAKE-OFF_WEIGHT forces MIN_LANDING_WEIGHT.
   Searching for consistent value of forcing variable ...
   Searching between   10000.0 lbf and   20000.0 lbf (zeroth level) ...
     Considering search value:     GROSS_TAKE-OFF_WEIGH    15000.00000 lbf
           First branch value:     MIN_LANDING_WEIGHT      10906.81660 lbf
          Second branch value:     MIN_LANDING_WEIGHT      11013.14348 lbf
                   Difference:                               106.32688 lbf
     Considering search value:     GROSS_TAKE-OFF_WEIGH    13750.00000 lbf
           First branch value:     MIN_LANDING_WEIGHT       9998.81886 lbf
          Second branch value:     MIN_LANDING_WEIGHT      10256.02410 lbf
                   Difference:                               257.20523 lbf
     Considering search value:     GROSS_TAKE-OFF_WEIGH    16250.00000 lbf
           First branch value:     MIN_LANDING_WEIGHT      11816.73465 lbf
          Second branch value:     MIN_LANDING_WEIGHT      11770.26287 lbf
                   Difference:                                46.47178 lbf
   Determining new search interval ...
   Searching between   15000.0 lbf and   16250.0 lbf (first level) ...
     Considering search value:     GROSS_TAKE-OFF_WEIGH    15833.33333 lbf
           First branch value:     MIN_LANDING_WEIGHT      11513.67062 lbf
          Second branch value:     MIN_LANDING_WEIGHT      11517.88974 lbf
                   Difference:                                 4.21912 lbf
   Search has found consistent value of  15833.33333 lbf for
   forcing variable GROSS_TAKE-OFF_WEIGHT (search level depth of one).

Value chosen for forcing variable, GROSS_TAKE-OFF_WEIGHT:   15833.33333 lbf
   Reverse computing (FUEL_WEIGHT) using function DF-1, based on
            GROSS_TAKE-OFF_WEIGHT    C      15833.333333 lbf
            PAYLOAD_WEIGHT           I       2200.000000 lbf
            EMPTY_WEIGHT_FRACTION    I          0.550000
     N-R iteration successful (one recursion).
   Assigning 4925.0 lbf to FUEL_WEIGHT.
   Forward computing (CRUISE_WEIGHT) using function DF-2, based on
            GROSS_TAKE-OFF_WEIGHT    C      15833.333333 lbf
            FUEL_WEIGHT              C       4925.000000 lbf
   Assigning 14191.66666666667 lbf to CRUISE_WEIGHT.
   Reverse computing (MIN_LANDING_WEIGHT) using function DF-4, based on
            RANGE                    I       3000.000000 sm
            CRUISE_VELOCITY          I        565.000000 sm hr-1
            TSFC                     I          0.800000 lb lbf-1 hr-1
            LIFT-TO-DRAG_RATIO       K         13.333333
            GROSS_TAKE-OFF_WEIGHT    C      15833.333333 lbf
     N-R iteration successful (two recursions).
   Assigning 11513.62878778228 lbf to MIN_LANDING_WEIGHT.
   Forward computing (MIN_LANDING_WEIGHT) using function DF-3, based on
            GROSS_TAKE-OFF_WEIGHT    C      15833.333333 lbf
            FUEL_WEIGHT              C       4925.000000 lbf
```

```
            RANGE                       I       3000.000000 sm
            CRUISE_VELOCITY             I        565.000000 sm hr-1
            TIME_ON_RESERVES            I          0.750000 hr
   Assigning 11517.88974078131 lbf to MIN_LANDING_WEIGHT.
   Reverse computing (FUEL_WEIGHT) using function DF-1, based on
            GROSS_TAKE-OFF_WEIGHT       C      15833.333333 lbf
            PAYLOAD_WEIGHT              I       2200.000000 lbf
            EMPTY_WEIGHT_FRACTION       I          0.550000
     N-R iteration successful (one recursion).
   Assigning 4925.0 lbf to FUEL_WEIGHT.
   Forward computing (CRUISE_WEIGHT) using function DF-2, based on
            GROSS_TAKE-OFF_WEIGHT       C      15833.333333 lbf
            FUEL_WEIGHT                 C       4925.000000 lbf
   Assigning 14191.66666666667 lbf to CRUISE_WEIGHT.

GROSS_TAKE-OFF_WEIGHT/MIN_LANDING_WEIGHT loop computations completed.

Beginning LOOP computation: ASPECT_RATIO forces OSWALD_EFFICIENCY.
   Searching for consistent value of forcing variable ...
   Searching between       5.0 and      20.0 (zeroth level) ...
     Considering search value:     ASPECT_RATIO                 8.00000
           First branch value:     OSWALD_EFFICIENCY            0.76374
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.03626
     Considering search value:     ASPECT_RATIO                 6.12500
           First branch value:     OSWALD_EFFICIENCY            0.99675
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.19675
   Determining new search interval ...
   Searching between     6.125 and       8.0 (first level) ...
     Considering search value:     ASPECT_RATIO                 7.37500
           First branch value:     OSWALD_EFFICIENCY            0.82797
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.02797
   Determining new search interval ...
   Searching between       8.0 and     7.375 (second level) ...
     Considering search value:     ASPECT_RATIO                 7.58333
           First branch value:     OSWALD_EFFICIENCY            0.80593
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.00593
     Considering search value:     ASPECT_RATIO                 7.79167
           First branch value:     OSWALD_EFFICIENCY            0.78390
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.01610
   Determining new search interval ...
   Searching between 7.5833333 and 7.7916667 (third level) ...
     Considering search value:     ASPECT_RATIO                 7.65278
           First branch value:     OSWALD_EFFICIENCY            0.80000
          Second branch value:     OSWALD_EFFICIENCY            0.80000
                   Difference:                                  0.00000
```

```
   Search has found consistent value of      7.65278  for
   forcing variable ASPECT_RATIO (search level depth of three).

Value chosen for forcing variable, ASPECT_RATIO:       7.65278
   Reverse computing (OSWALD_EFFICIENCY WING_REFERENCE_AREA) using function Aero-
dynamics Package, based on
               LIFT_COEFFICIENT         I          0.240000
               DRAG_COEFFICIENT         I          0.018000
               CRUISE_WEIGHT            C      14191.666667 lbf
               CRUISE_VELOCITY          I        565.000000 sm hr-1
               ZERO-LIFT_DRAG_COEFF     I          0.015000
               ASPECT_RATIO             C          7.652778
     N-R iteration successful (one recursion).
   Assigning 0.8  to OSWALD_EFFICIENCY.
   Assigning 233.3651792638083 ft2 to WING_REFERENCE_AREA.
   Reverse computing (WING_SPAN) using function Wing Geometry Package, based on
               WING_REFERENCE_AREA      C        233.365179 ft2
               ASPECT_RATIO             C          7.652778
     N-R iteration successful (one recursion).
   Assigning 42.26342770724547 ft to WING_SPAN.
   Forward computing (OSWALD_EFFICIENCY) using function Wing Geometry Package, based
on
               ASPECT_RATIO             C          7.652778
               WING_SPAN                C         42.263428 ft
   Assigning 0.8  to OSWALD_EFFICIENCY.

ASPECT_RATIO/OSWALD_EFFICIENCY loop computations completed.
```



Appendix G

The NASP Design Set

This appendix lists the design variables and design functions used to test the modifications to *Paper Airplane* via the preliminary design of a national aerospaceplane.

The NASP Design Set is a set of 26 MISO and MIMO design functions and the 45 design variables they relate. Table G.1 summarizes the 26 geometry design variables in the NASP Design Set and Table G.2 summarizes the other 19 design variables. Included in the tables, for each design variable, are its traditional mathematical symbol, its name inside of *Paper Airplane*, and its definition.

The 26 design functions, which have been given more appropriate names than those of the MISO Design Set, vary from simple geometrical relationships to complex aerodynamics and performance computer programs. Three invaluable sources of information that greatly helped in the formation of these design functions were [12], [13], and [14].

FOREBODY-LENGTH-EQUATION: This is an equation for Forebody Length.

$$l_{FB} = f_{FB}\, l_W$$

AFTERBODY-LENGTH-EQUATION: This is an equation for Afterbody Length.

$$l_{AB} = f_{AB}\, l_W$$

ENGINE-LENGTH-EQUATION: This is an equation for Engine Length.

$$l_E = f_E\, l_W$$

Symbol	*Paper Airplane* Name	Definition
A_E	INLET_CAPTURE_AREA	The Area of Air Captured by the Inlet.
b	WING_SPAN	The largest Span of the Wing.
d_B	BODY_DIAMETER	The Maximum Diameter of the Fuselage.
d_E	ENGINE_DIAMETER	The Diameter of the Engine Shroud Half-cylinder.
f_{AB}	AFTERBODY_FRACTION	The Fractional Length of the Afterbody Half-cone.
f_E	ENGINE_FRACTION	The Fractional Length of the Engine Shroud.
f_{FB}	FOREBODY_FRACTION	The Fractional Length of the Forebody Half-cone.
f_s	SEATING_FRACTION	The Fraction of Floor Space required by Seating.
h_p	CABIN_HEIGHT	The Height of the Ceiling of the Cabin.
h_T	TAIL_HEIGHT	The Height of each Vertical Tail.
l_{AB}	AFTERBODY_LENGTH	The Length of the Fuselage Afterbody Half-cone.
l_E	ENGINE_LENGTH	The Length of the Engine Shroud Half-cylinder.
l_{FB}	FOREBODY_LENGTH	The Length of the Fuselage Forebody Half-cone.
l_p	SEAT_PITCH	The Forward Distance Between Passenger Seats.
l_T	TAIL_LENGTH	The Length of each Vertical Tail.
l_W	WING_LENGTH	The Length of the Wing and of the entire Fuselage.
R_N	NOSE_RADIUS	The Radius of the Nose of the Forebody.
R_W	LEADING_EDGE_RADIUS	The Radius of the Leading Edge of the Wing.
S_{ref}	WING_REFERENCE_AREA	The Reference Area of the Wing.
$(t/c)_{max}$	WING_THICKNESS	The Maximum Wing Thickness-to-Chord Ratio.
V_B	FUSELAGE_VOLUME	The Volume of the vehicle's Fuselage.
V_f	FUEL_VOLUME	The Volume required by the vehicle's Fuel.
V_p	PAYLOAD_VOLUME	The Volume required by the vehicle's Payload.
w_p	SEAT_WIDTH	The Width of a Passenger Seat.
$x_{t/c}$	THICKNESS_STATION	The Station of the Maximum Thickness-to-Chord Ratio.
Δ_{LE}	LEADING_EDGE_SWEEP	The Angular Sweepback of the Wing's Leading Edge.

Table G.1: Geometry Design Variables comprising the NASP Design Set.

Symbol	*Paper Airplane* Name	Definition
$C_{L_{max}}$	MAXIMUM_LIFT_COEFF	The Maximum Lift Coefficient of the vehicle.
f_e	EMPTY_WEIGHT_FRACTION	The Empty Weight Fraction of the vehicle.
f_p	PAYLOAD_WEIGHT_FRACTION	The Payload Weight Fraction of the vehicle.
H_{CR}	CRUISE_ALTITUDE	The Altitude of the vehicle at Cruise.
M_{CR}	CRUISE_MACH	The Mach Number of the vehicle at Cruise.
n_{cl}	CLIMB_ACCELERATION	The Maximum Climb Acceleration of the vehicle.
N_p	PASSENGER_CAPACITY	The Number of Passengers and Crew.
R	RANGE	The Range of the vehicle.
t_f	TIME_OF_FLIGHT	The Time of Flight to achieve the Range.
T_N	NOSE_TEMPERATURE	The Stagnation Temperature on the Nose.
T_W	WING_TEMPERATURE	The Stagnation Temperature on the Leading Edge.
V_{CR}	CRUISE_VELOCITY	The Velocity of the vehicle at Cruise.
W_e	VEHICLE_EMPTY_WEIGHT	The Empty (structual) Weight of the vehicle.
W_f	FUEL_WEIGHT	The Weight of the vehicle's Fuel.
W_{fr}	FUEL_RESERVES	The Weight of the vehicle's Fuel Reserves.
W_{gto}	VEHICLE_GROSS_WEIGHT	The Gross Weight of the vehicle at Take-off.
W_p	PAYLOAD_WEIGHT	The Weight of the vehicle's Payload.
ρ_{CR}	CRUISE_AIR_DENSITY	The Density of the Air at Cruise.
γ_{cl}	CLIMB_ANGLE	The Maximum Climb Angle of the vehicle.

Table G.2: Other Design Variables comprising the NASP Design Set.

LENGTH-FRACTION-EQUATION: This is an equation for Engine Fraction.

$$f_E = 1 - f_{FB} - f_{AB}$$

WING-SPAN-EQUATION: This is an equation for Wing Span.

$$b = 2\, l_W / \tan \Delta_{LE}$$

REFERENCE-AREA-EQUATION: This is an equation for Wing Reference Area.

$$S_{ref} = l_W\, b/2$$

CAPTURE-AREA-EQUATION: This is an equation for Inlet Capture Area.

$$A_E = \frac{\pi}{2} \frac{d_E^2}{4}$$

TAIL-LENGTH-EQUATION: This is an equation for Tail Length.

$$l_t = l_{AB}$$

TAIL-HEIGHT-EQUATION: This is an equation for Tail Height. The tail volume coefficient, C_T, is 0.09:

$$h_T = \frac{2\, C_T\, b\, S_{ref}}{(l_W - l_T) l_T}$$

GROSS-WEIGHT-EQUATION: This is an equation for Vehicle Gross Weight.

$$W_{gto} = W_e + W_p + W_f$$

EMPTY-WEIGHT-EQUATION: This is an equation for Vehicle Empty Weight.

$$W_e = f_e\, W_{gto}$$

PAYLOAD-WEIGHT-EQUATION: This is an equation for Payload Weight. Each passenger weighs 170 lb and has baggage weighing 60 lb.

$$W_p = N_p\, (170\, lb + 60\, lb)$$

PAYLOAD-FRACTION-EQUATION: This is an equation for Payload Weight Fraction.

$$f_p = W_p \,/\, W_{gto}$$

`FUEL-WEIGHT-EQUATION`: This is an equation for Fuel Weight. The fuel density, ρ_f, is that for liquid hydrogen, 4.37 lb/ft^3.

$$W_f = \rho_f V_f$$

`FUSELAGE-VOLUME-EQUATION`: This is an equation for Fuselage Volume.

$$V_B = \frac{\pi}{24} l_{FB}\, d_B^2 + \frac{\pi}{24} l_{AB}\, d_B^2 + \frac{\pi}{8} l_E\, d_B^2$$

`PAYLOAD-VOLUME-EQUATION`: This is an equation for Payload Volume. Each passenger's baggage takes up a volume of 25 ft^3.

$$V_p = N_p\, h_p\, l_p\, w_p / f_s + N_p (25\; ft^3)$$

`BODY-DIAMETER-EQUATION`: This is an equation for Body Diameter.

$$d_B = b/2$$

`ENGINE-DIAMETER-EQUATION`: This is an equation for Engine Diameter.

$$d_E = 1.25\, d_B$$

`CRUISE-DENSITY-FUNCTION`: This is an equation for Cruise Density. The density of air, ρ, varies primarily with altitude.

$$\rho_{CR} = \rho(H_{CR})$$

`CRUISE-VELOCITY-FUNCTION`: This is an equation for Cruise Velocity. The speed of sound, a, varies primarily with altitude.

$$V_{CR} = M_{CR}\, a(H_{CR})$$

`NOSE-RADIUS-EQUATION`: This is an equation for Nose Radius. The emissitivity of titanium, ϵ, is 0.8. The Boltzmann constant, σ, is 0.481×10^{-12} BTU/ft^2-s-°R.

$$R_N = \left(\frac{15}{\epsilon\sigma}\right)^2 \left(\frac{V_{CR}}{1000}\right)^6 \frac{\rho_{CR}}{T_N^8}$$

`WING-RADIUS-EQUATION`: This is an equation for Leading-Edge Radius.

$$R_W = \left(\frac{15}{\epsilon\sigma}\right)^2 \left(\frac{V_{CR}}{1000}\right)^6 \frac{\rho_{CR}}{T_W^8} \cos\Delta_{LE}$$

FUEL-RESERVES-EQUATION: This is an equation for Fuel Reserves. The reserved and trapped fuel is 5% of the fuel weight.

$$W_{fr} = 0.05 W_f$$

FUEL-VOLUME-EQUATION: This is an equation for Fuel Volume. This is the volume of the lower fuselage that can hold fuel.

$$V_f = 0.9 A_B \left[l_E + \frac{d_B - 2h_p}{d_B} \left(\frac{l_{FB}}{3} + \frac{l_{AB}}{3} \right) \right]$$

where

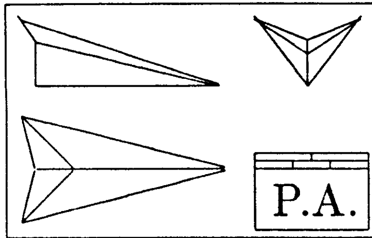
$$A_B = \frac{d_B^2}{4} \cos^{-1} \left(\frac{2h_p}{d_B} \right) - h_p \sqrt{\frac{d_B^2}{4} - h_p^2}$$

AERODYNAMICS-PROGRAM: This calls the Aerodynamics code.

$$C_{L_{max}} = AERO \left((t/c)_{max}, x_{t/c}, b, d_B, l_{FB}, l_{AB}, l_E, d_E, h_T, l_T \right)$$

PERFORMANCE-PROGRAM: This calls the Performance code. The propulsion and aerodynamics data are global variables, not design variables.

$$(R, t_f) = PERF \left(H_{CR}, M_{CR}, \gamma_{cl}, n_{cl}, S_{ref}, A_E, C_{L_{max}}, W_{gto}, W_f, W_{fr} \right)$$



Appendix H

NASP Aerodynamics Code

```
      program NASP_AERODYNAMICS

c   This program computes the aerodynamics of a national aerospaceplane (NASP)
c   in terms of lift coefficient (C_L) as a function of angle of attack (ALPHA)
c   and Mach number (MACH), and in terms of drag coefficient (C_D) as a
c   function of angle of attack, Mach number, and altitude (ALTITUDE).

c   The user provides the necessary geometry information concerning the NASP
c   and the range of altitudes, Mach numbers, and angles of attack to be
c   analyzed and tabulated.

      integer    TAV, FB, AB, ENG, WING, TAIL
      parameter (TAV =  1,                  ! Subscript for the entire TAV
     +           FB =   2,                  ! Subscript for TAV forebody
     +           AB =   3,                  ! Subscript for TAV afterbody
     +           ENG =  4,                  ! Subscript for TAV engine
     +           WING = 5,                  ! Subscript for TAV wing
     +           TAIL = 6,                  ! Subscript for TAV tail
     +           PI =   3.141592654)

      integer
     +     N_ALT,                           ! Number of Altitudes
     +     N_MACH,                          ! Number of Mach Numbers
     +     N_ALPHA,                         ! Number of Angles of Attack
     +     H, M, A, LAST_I                  ! Various indices

      real
     +     ALTITUDE, MIN_ALT, D_ALT,        ! Altitude info (ft)
     +     MACH, MIN_MACH, D_MACH,          ! Mach number info
     +     ALPHA, MIN_ALPHA, D_ALPHA,       ! Angle of Attack info (deg)
     +     ALPHA_RAD,                       ! Angle of Attack (rad)
     +     C_L(0:29,0:29), CL_max, C_D,     ! Lift & Drag Coefficients
```

```
     +      C_L_ALPHA, CLA09, CLA11, CLA13, ! Lift Curve Slopes (rad-1)
     +      C_D_0(6), C_D_0_09, C_D_0_13,   ! Zero-Lift Drag Coeffs
     +      K_SUBSONIC, K_09, K_13, K_TAV,  ! Drag-due-to-Lift Factors
     +      RHO_A_over_MU                   ! Air Density * Speed of Sound
                                            !      / Air Viscosity (ft-1)
      real
     +      D_BODY,                         ! Body Diameter (ft)
     +      LENGTH(6),                      ! Various lengths (ft)
     +      S_WET(6),                       ! Various Wetted Areas (ft2)
     +      S_EXP, S_REF,                   ! Exposed, Reference Area (ft)
     +      MAX_T,                          ! Max. thickness-to-chord ratio
     +      S_MAX,                          ! Max. Cross Sectional Area (ft2)
     +      DELTA_MAX_T,                    ! Sweep of wing at MAX_T
     +      DELTA_LE, DELTA_LE_TAIL,        ! Sweep of wing, tail leading edge
     +      MAC, MAC_TAIL                   ! Mean aero. chord of wing, tail
      real
     +      B_FACTOR,                       ! Wave drag B-Factor
     +      BSFF,                           ! Body Skin Friction Factor
     +      ASFF,                           ! Airfoil Skin Friction Factor
     +      WBLIF,                          ! Wing Body Lift Interference Factor
     +      NLLF,                           ! Non-Linear Lift Factor
     +      CD_ANC,                         ! Afterbody Interference Drag
     +      AR                              ! Wing Aspect Ratio

      common  N_ALT, MIN_ALT, D_ALT, N_MACH, MIN_MACH, D_MACH,
     +        N_ALPHA, MIN_ALPHA, D_ALPHA, C_L, C_L_ALPHA, CLA09,
     +        CLA11, CLA13, C_D_0, C_D_0_09, C_D_0_13, K_SUBSONIC,
     +        K_09, K_13, K_TAV, RHO_A_over_MU, LAST_I,
     +        D_BODY, LENGTH, S_WET, S_EXP, S_REF, MAX_T, S_MAX,
     +        DELTA_MAX_T, DELTA_LE, DELTA_LE_TAIL, MAC, MAC_TAIL,
     +        B_FACTOR, BSFF, ASFF, WBLIF, NLLF, CD_ANC, AR, MACH,
     +        ALTITUDE, CL_max

c     begin
        LAST_I = 1
        call READ_TERMINAL_DATA
        call GET_GEOMETRY_INFO
        write(6,10) CL_max
   10   format(' Maximum lift coefficient is',F12.6,/)

        do H = 0, N_ALT-1
          ALTITUDE = MIN_ALT + D_ALT * H
          write(4,'(/A,F14.1)') ' Altitude (ft) = ', ALTITUDE
          write(6,100) ALTITUDE
  100     format(' Computing aerodynamics at', F12.1, ' ft')
          call COMPUTE_ALTITUDE_INFO

          do M = 0, N_MACH-1
            MACH = MIN_MACH + D_MACH * M
            write(4,'(A,F17.2)') ' Mach Number = ', MACH
```

```
          if ( MACH .eq. 0.0 ) MACH = 0.01
          call COMPUTE_MACH_INFO
          write(4,'(A,F16.5)') ' Lift Curve Slope =', C_L_ALPHA
          write(4,'(A,F18.5)') ' Zero-lift Drag =', C_D_O(TAV)
          write(4,'(A,F9.5)') ' Drag-due-to-Lift Factor =', K_TAV
          write(4,*) '     Alpha      C_L         C_D       L/D'

          do A = 0, N_ALPHA-1
            ALPHA = MIN_ALPHA + D_ALPHA * A
            ALPHA_RAD = ALPHA / 180.0 * PI

c           Compute C_L only for first altitude, but use it for all.

            if ( H .eq. 0 ) then
              if ( MACH .le. 0.9 ) then
                C_L(A,M) = (WBLIF * C_L_ALPHA + NLLF * ALPHA_RAD)
     +               * ALPHA_RAD * S_EXP / S_REF
              else
                C_L(A,M) = WBLIF * C_L_ALPHA * ALPHA_RAD * S_EXP /
     +                    S_REF
              endif
              if ( C_L(A,M) .gt. CL_max ) C_L(A,M) = CL_max
            endif

            C_D = C_D_O(TAV) + K_TAV * C_L(A,M)**2

            write(4,200) ALPHA, C_L(A,M), C_D, C_L(A,M)/C_D
  200       format(F11.1,F11.5,F12.5,F11.3)

          enddo
        enddo
      enddo

      close(unit=3)
      close(unit=4)
    end

    subroutine READ_TERMINAL_DATA

    include 'COMMON.FOR'

    real        MAX_ALT, MAX_MACH, MAX_ALPHA
    character   FILENAME*60

c   begin
      write(6,*) ' Enter the name of the geometry data input file.'
      read(5,'(A)') FILENAME
      open(unit=3, file=FILENAME, readonly, status='OLD')

      write(6,*) ' Enter the name of the aero data output file.'
```

```
      read(5,'(A)') FILENAME
      open(unit=4, file=FILENAME, status='NEW')
      write(4,*) ' ********* TAV AERODYNAMICS TABLES *********'

      write(6,*) ' Enter minimum, maximum, and number of ',
     +           'altitudes to be examined.'
      read(5,*) MIN_ALT, MAX_ALT, N_ALT
      write(4,'(/A,I12)') ' Number of Altitudes =', N_ALT

      if ( N_ALT .eq. 1 ) then
        D_ALT = 0.0
      else
        D_ALT = (MAX_ALT - MIN_ALT) / (N_ALT - 1)
      endif

      write(6,*) ' Enter minimum, maximum, and number of ',
     +           'Mach numbers to be examined.'
      read(5,*) MIN_MACH, MAX_MACH, N_MACH
      write(4,'(A,I9)')  ' Number of Mach Numbers =', N_MACH

      if ( N_MACH .eq. 1 ) then
        D_MACH = 0.0
      else
        D_MACH = (MAX_MACH - MIN_MACH) / (N_MACH - 1)
      endif

      write(6,*) ' Enter minimum, maximum, and number of ',
     +           'angles of attack to be examined.'
      read(5,*) MIN_ALPHA, MAX_ALPHA, N_ALPHA
      write(4,'(A,I5/)') ' Number of Angles of Attack =', N_ALPHA

      if ( N_ALPHA .eq. 1 ) then
        D_ALPHA = 0.0
      else
        D_ALPHA = (MAX_ALPHA - MIN_ALPHA) / (N_ALPHA - 1)
      endif

        return
      end

      subroutine GET_GEOMETRY_INFO

      include 'COMMON.FOR'

      real
     +      AREA(6),                    ! Various planar areas (ft+2)
     +      X_MAX_T,                    ! X-station (x/c) of max. t/c
     +      SPAN,                       ! Wing span (ft)
     +      D_ENG,                      ! Engine section diameter (ft)
     +      H_TAIL,                     ! Height of vertical tails (ft)
```

```
     +      LEN_TO_BODY,                     ! Body length-to-diameter ratio
     +      E                                ! Oswald wing lift efficiency

c     begin
        read(3,'(///,20X,F10.3,/,20X,F10.3)') MAX_T, X_MAX_T
        read(3,'(/,20X,F10.3)')               SPAN
        read(3,'(/,20X,F10.3,/,20X,F10.3)')  D_BODY, LENGTH(FB)
        read(3,'(/,20X,F10.3)')               LENGTH(AB)
        read(3,'(/,20X,F10.3,/,20X,F10.3)')  D_ENG, LENGTH(ENG)
        read(3,'(/,20X,F10.3,/,20X,F10.3)')  H_TAIL, LENGTH(TAIL)

        LENGTH(WING) = LENGTH(FB) + LENGTH(ENG) + LENGTH(AB)
        LENGTH(TAV) =  LENGTH(WING)
        S_MAX =        PI * D_BODY**2 / 8.0
        LEN_TO_BODY =  LENGTH(TAV) / D_BODY

        AREA(FB) =    LENGTH(FB) * D_BODY / 2.0
        AREA(AB) =    LENGTH(AB) * D_BODY / 2.0
        AREA(ENG) =   LENGTH(ENG) * D_ENG
        AREA(WING) =  LENGTH(WING) * SPAN / 2.0
        AREA(TAIL) =  2.0 * LENGTH(TAIL) * H_TAIL / 2.0    ! Two vertical tails
        S_REF =       AREA(WING)
        S_EXP =       S_REF - AREA(FB) - AREA(AB) - AREA(ENG)

        S_WET(FB) =   PI * AREA(FB) *
     +                sqrt( 1.0 + (D_BODY / LENGTH(FB))**2 )
        S_WET(AB) =   PI * AREA(AB) *
     +                sqrt( 1.0 + (D_BODY / LENGTH(AB))**2 )
        S_WET(ENG) =  PI * AREA(ENG)
        S_WET(WING) = S_REF + S_EXP
        S_WET(TAIL) = 2.0 * AREA(TAIL)

        AR =          SPAN**2 / AREA(WING)
        MAC =         2.0 / 3.0 * LENGTH(WING)
        MAC_TAIL =    2.0 / 3.0 * LENGTH(TAIL)
        DELTA_LE =    atan( LENGTH(WING) / (SPAN / 2.0) )
        DELTA_LE_TAIL = atan( LENGTH(TAIL) / H_TAIL )
        DELTA_MAX_T = atan( LENGTH(WING) * (1.0 - X_MAX_T)
     +                       / (SPAN / 2.0))
        BSFF =   1.0 + 60.0 / LEN_TO_BODY**3 + 0.0025 * LEN_TO_BODY
        WBLIF =  1.0 + 12.0 * (D_BODY / SPAN)**2              ! Approximated

c   CL_max and B_FACTOR assume double wedge shaped airfoil.
        CL_max =   0.83
        B_FACTOR = 1.0 / (X_MAX_T * (1.0 - X_MAX_T))

c   E assumes Taper Ratio = 0, AR < 5, DELTA_LE > 60 deg.
        E = 0.98 * (1.0 - (D_BODY / SPAN)**2)

c   ASFF assumes X_MAX_T > 0.3.
```

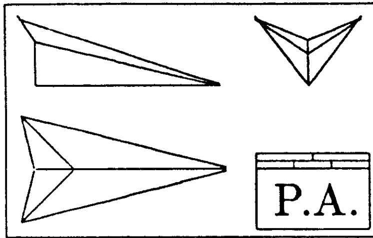
```
        ASFF =   1.0 + 1.2 * MAX_T + 100.0 * MAX_T**4

c  NLLF assumes a delta wing with Taper Ratio = 0.
        NLLF =   0.4 + 11.0 * exp( -1.0 * (AR + 2.0) )          ! Approximated

c  CD_ANC assumes a pointed afterbody.
        CD_ANC = 1.34 / (LENGTH(FB) / LENGTH(AB)) *              ! Approximated
     +           exp( -2.1 * LENGTH(ENG) / LENGTH(AB) ) /
     +           (2.0 * LENGTH(AB) / D_ENG)**2

c  K_SUBSONIC assumes Taper Ratio = 0, sharped-nosed airfoil
        K_SUBSONIC =  1.0 / (PI * AR * E) + 0.16

        CLA09 = 2.0 * PI * AR /
     +          (2.0 + sqrt( 4.0 + AR**2 * (1.0 - 0.9**2)
     +          * (1.0 + tan(DELTA_MAX_T)**2 / (1.0 - 0.9**2)) ))
        CLA11 = 6.6 / tan(DELTA_LE)                                    ! Approx.
        CLA13 = ( 6.0 - 2.5 *                                          ! Approx.
     +          (sqrt(1.3**2 - 1.0) / tan(DELTA_LE) - 0.2) )
     +          / tan(DELTA_LE)

        return
      end


      real function wave_drag( BETA, DELTA )

c     This functions computes the wave drag for sharpe-nosed
c     double wedge shaped airfoils.

      include 'COMMON.FOR'

      real    BETA, DELTA

c     begin
        if ( BETA .lt. tan(DELTA) ) then
          wave_drag = B_FACTOR / tan(DELTA) * MAX_T**2
        else
          wave_drag = B_FACTOR / BETA * MAX_T**2
        endif
        return
      end

      subroutine COMPUTE_ALTITUDE_INFO

      include 'COMMON.FOR'

      real      ALTITUDES(28), RAM_RATIOS(28),
     +          RHO_A_over_MU_ZERO,
     +          RAM_RATIO                          ! at ALT
```

```
      integer   I
      parameter (RHO_A_over_MU_ZERO = 7100235.9)

      real      BETA, CF_BODY, CF_WING, CF_TAIL, LSCF, DELTA_C_D_O,
     +          CCF,  CD_N2, F_A, CD_A, RATIO_N, DELTA_N

      data  ALTITUDES /    0.0,    5000.0,  10000.0,  15000.0,  20000.0,
     +        25000.0,  30000.0,  35000.0,  36089.0,  40000.0,  45000.0,
     +        50000.0,  55000.0,  60000.0,  65000.0,  70000.0,  75000.0,
     +        80000.0,  82021.0,  85000.0,  90000.0,  95000.0, 100000.0,
     +       110000.0, 120000.0, 130000.0, 140000.0, 150000.0 /

      data  RAM_RATIOS / 1.0,    0.8702,    0.7536,    0.6492,    0.5563,
     +       0.4739,    0.4010,    0.3371,    0.3242,    0.2687,    0.2113,
     +       0.16610,  0.13064,  0.10274,  0.08079,  0.06353,  0.04996,
     +       0.03939,  0.03566,  0.03041,  0.02339,  0.018090, 0.014063,
     +       0.008627, 0.005388, 0.003425, 0.002211, 0.001449 /

c     begin
        I = LAST_I
        do while (( ALTITUDE .gt. ALTITUDES(I) ) .and. (I .le. 28))
          I = I + 1
        end do

        if (ALTITUDE .EQ. ALTITUDES(I)) then
          RAM_RATIO = RAM_RATIOS(I)
          LAST_I = I
        else
          RAM_RATIO = exp( alog(RAM_RATIOS(I-1)) +
     +                 (alog(RAM_RATIOS(I)) - alog(RAM_RATIOS(I-1)))
     +                 * (ALTITUDE - ALTITUDES(I-1))
     +                 / (ALTITUDES(I) - ALTITUDES(I-1)) )
          LAST_I = I - 1
        endif

        RHO_A_over_MU = RAM_RATIO * RHO_A_over_MU_ZERO

c       Transonic -- subsonic boundary Mach 0.9

        MACH = 0.9
        CF_BODY = 0.074 / ( RHO_A_over_MU * MACH * LENGTH(TAV) )**0.2
        CF_WING = 0.074 / ( RHO_A_over_MU * MACH * MAC )**0.2
        CF_TAIL = 0.074 / ( RHO_A_over_MU * MACH * MAC_TAIL )**0.2

        LSCF = 0.4 * (1.0 + MACH**2) + 0.7 * cos(DELTA_MAX_T)
        C_D_O(FB) =   CF_BODY * BSFF * S_WET(FB) /   S_REF
        C_D_O(AB) =   CF_BODY * BSFF * S_WET(AB) /   S_REF
        C_D_O(ENG) =  CF_BODY * BSFF * S_WET(ENG) /  S_REF
        C_D_O(WING) = CF_WING * ASFF * S_WET(WING) / S_REF * LSCF
        C_D_O(TAIL) = CF_TAIL * ASFF * S_WET(TAIL) / S_REF
```

```
      DELTA_C_D_O = 0.0012 * 280.0 / S_REF
      C_D_O_09 = C_D_O(FB)  +  C_D_O(AB)  +  C_D_O(ENG) +
     +           C_D_O(WING) + C_D_O(TAIL) + DELTA_C_D_O
      K_09 = K_SUBSONIC

c     Transonic -- supersonic boundary Mach 1.3
c     see subroutine COMPUTE_MACH_INFO for design assumptions and notes.

      MACH = 1.3
      BETA = sqrt( MACH**2 - 1.0 )
      CF_BODY = 0.074 / ( RHO_A_over_MU * MACH * LENGTH(TAV) )**0.2
      CF_WING = 0.074 / ( RHO_A_over_MU * MACH * MAC )**0.2
      CF_TAIL = 0.074 / ( RHO_A_over_MU * MACH * MAC_TAIL )**0.2

      CCF = 1.0 / (1.0 + 0.144 * MACH**2)**0.65
      CD_N2 = ( 0.15 * LENGTH(FB) / (D_BODY * BETA) + 0.6 ) /
     +        ( (LENGTH(FB) / D_BODY)**2 + 0.25 )

      F_A = LENGTH(AB) / D_BODY
      if (BETA .le. F_A) then
        CD_A = (0.505 + 0.29 * tan( PI/2.0 * (1.0 - BETA / F_A) ))
     +         / F_A**2
      else
        CD_A = (0.505 - 0.576 * (1.0 - F_A / BETA)**1.16) / F_A**2
      endif

      C_D_O(FB) = CF_BODY * CCF * S_WET(FB) / S_REF +
     +            CD_N2 * S_MAX / S_REF
      C_D_O(AB) = CF_BODY * CCF * S_WET(AB) / S_REF +
     +            (CD_A + CD_ANC) * S_MAX / S_REF
      C_D_O(ENG) = CF_BODY * CCF * S_WET(ENG) / S_REF
      C_D_O(WING) = CF_WING * CCF * S_WET(WING) / S_REF +
     +              wave_drag( BETA, DELTA_LE ) * S_WET(WING)
     +              / 2.0 / S_REF
      C_D_O(TAIL) = CF_TAIL * CCF * S_WET(TAIL) / S_REF +
     +              wave_drag( BETA, DELTA_LE_TAIL ) * S_WET(TAIL)
     +              / 2.0 / S_REF
      DELTA_C_D_O = 0.0016 * 280.0 / S_REF
      C_D_O_13 = C_D_O(FB)  +  C_D_O(AB)  +  C_D_O(ENG) +
     +           C_D_O(WING) + C_D_O(TAIL) + DELTA_C_D_O

      RATIO_N = 1.16 - 6.26 * ( BETA / tan(DELTA_LE) - 0.43 )**3
      DELTA_N = RATIO_N * (2.0 / (CLA09 + CLA11) - K_SUBSONIC)
      K_13 = 1.0 / CLA13 - DELTA_N

      return
    end

    subroutine COMPUTE_MACH_INFO
```

```
      include 'COMMON.FOR'

      real
     +        BETA,                         ! Mach Number Beta term
     +        CF_BODY, CF_WING, CF_TAIL,    ! Flat Plate Skin Friction Coeff.
     +        LSCF,                         ! Lifting Surface Correlation Factor
     +        DELTA_C_D_O,                  ! Miscellaneous Protuberance Drag
     +        CCF,                          ! Compressibility Correction Factor
     +        CD_N2,                        ! Forebody Wave Drag Factor
     +        F_A,                          ! Afterbody Fineness Ratio
     +        CD_A,                         ! Afterbody Wave Drag Factor
     +        RATIO_N, DELTA_N              ! Leading-edge Suction Parameters

c     begin
        BETA = sqrt( abs( 1.0 - MACH**2 ) )
        CF_BODY = 0.074 / ( RHO_A_over_MU * MACH * LENGTH(TAV) )**0.2
        CF_WING = 0.074 / ( RHO_A_over_MU * MACH * MAC )**0.2
        CF_TAIL = 0.074 / ( RHO_A_over_MU * MACH * MAC_TAIL )**0.2

        if ( MACH .le. 0.9 ) then                        ! SUBSONIC
          C_L_ALPHA = 2.0 * PI * AR / (2.0 +
     +              sqrt( 4.0 + AR**2 * BETA**2 *
     +                    (1.0 + tan(DELTA_MAX_T)**2 / BETA**2) ))

          LSCF = 0.4 * (1.0 + MACH**2) + 0.7 * cos(DELTA_MAX_T)  ! Approximated

          C_D_O(FB) =   CF_BODY * BSFF * S_WET(FB) /   S_REF
          C_D_O(AB) =   CF_BODY * BSFF * S_WET(AB) /   S_REF
          C_D_O(ENG) =  CF_BODY * BSFF * S_WET(ENG) /  S_REF
          C_D_O(WING) = CF_WING * ASFF * S_WET(WING) / S_REF * LSCF
          C_D_O(TAIL) = CF_TAIL * ASFF * S_WET(TAIL) / S_REF
          DELTA_C_D_O = 0.0012 * 280.0 / S_REF
          C_D_O(TAV) = C_D_O(FB)  +  C_D_O(AB)  +  C_D_O(ENG) +
     +               C_D_O(WING) + C_D_O(TAIL) + DELTA_C_D_O
          K_TAV = K_SUBSONIC

        else if ( MACH .ge. 1.3 ) then                   ! SUPERSONIC

c  C_L_ALPHA assumes Taper Ratio = 0.
          C_L_ALPHA = ( 6.0 - 2.5 * (BETA / tan(DELTA_LE) - 0.2) )
     +              / tan(DELTA_LE)                       ! Approx.

          CCF = 1.0 / (1.0 + 0.144 * MACH**2)**0.65

c  CD_N2 assumes a conical forebody.  Approximated.
          CD_N2 = ( 0.15 * LENGTH(FB) / (D_BODY * BETA) + 0.6 ) /
     +          ( (LENGTH(FB) / D_BODY)**2 + 0.25 )

c  CD_A assumes a conical non-truncated afterbody.  Approximated.
          F_A = LENGTH(AB) / D_BODY
```

```
          if (BETA .le. F_A) then
            CD_A = (0.505 + 0.29 * tan( PI/2.0 * (1.0 - BETA / F_A) ))
     +             / F_A**2
          else
            CD_A = (0.505 - 0.576 * (1.0 - F_A / BETA)**1.16) / F_A**2
          endif

          C_D_O(FB) = CF_BODY * CCF * S_WET(FB) / S_REF +
     +                CD_N2 * S_MAX / S_REF
          C_D_O(AB) = CF_BODY * CCF * S_WET(AB) / S_REF +
     +                (CD_A + CD_ANC) * S_MAX / S_REF
          C_D_O(ENG) = CF_BODY * CCF * S_WET(ENG) / S_REF
          C_D_O(WING) = CF_WING * CCF * S_WET(WING) / S_REF +
     +                  wave_drag( BETA, DELTA_LE ) * S_WET(WING)
     +                  / 2.0 / S_REF
          C_D_O(TAIL) = CF_TAIL * CCF * S_WET(TAIL) / S_REF +
     +                  wave_drag( BETA, DELTA_LE_TAIL ) * S_WET(TAIL)
     +                  / 2.0 / S_REF
          DELTA_C_D_O = 0.0016 * 280.0 / S_REF
          C_D_O(TAV) = C_D_O(FB)  +  C_D_O(AB)  +  C_D_O(ENG) +
     +                 C_D_O(WING) + C_D_O(TAIL) + DELTA_C_D_O

c  RATIO_N assumes Taper Ratio = 0, and very small LE radius.  Approximated.
          RATIO_N = 1.16 - 6.26 * ( BETA / tan(DELTA_LE) - 0.43 )**3
          DELTA_N = RATIO_N * (2.0 / (CLA09 + CLA11) - K_SUBSONIC)
          K_TAV = 1.0 / C_L_ALPHA - DELTA_N

        else                                                   ! TRANSONIC
          if ( MACH .le. 1.1 ) then
            C_L_ALPHA = CLA09 + (CLA11 - CLA09) * (MACH - 0.9) /
     +                                             (1.1 - 0.9)
          else
            C_L_ALPHA = CLA13 + (CLA11 - CLA13) * (MACH - 1.3) /
     +                                            (1.1 - 1.3)
          endif

          C_D_O(TAV) = C_D_O_09 + (C_D_O_13 - C_D_O_09) *
     +                            (MACH - 0.9) / (1.3 - 0.9)

          K_TAV = K_09 + (K_13 - K_09) * (MACH - 0.9) / (1.3 - 0.9)
        endif

        return
      end
```



Appendix I

NASP Performance Code

```
      program NASP_PERFORMANCE

c     This program computes the performance of a national aerospaceplane
c     (NASP) by performing a range integration.  The trajectory to be used
c     is a boost-glide-type climb, cruise, and descent, as follows:

c     CLIMB:   Climb at climb angle, GAMMA_cl, under load factor acceleration
c              of n_cl until cruise altitude, Hcr, is reached.

c     CRUISE:  Continue to accelerate to cruise Mach number, Mcr, then cruise
c              at Hcr and Mcr until fuel mass, mf, drops belows reserves, mfr.
c              drops below the FUEL_RESERVES.

c     DESCENT: Turn off engines, and continue to fly level until the angle
c              of attack rises to that for maximum L/D.  Fly a maximum L/D
c              trajectory descent until the altitude, H, drops below the
c              landing flare height, H_50.

      parameter
     +    (go = 9.80665,                  ! Earth 30 deg Gravity (m/s^2)
     +     Ro =  6375400.0,               ! Earth 30 deg Radius (m)
     +     H_50 = 15.24)                  ! Landing Flare Height (m)

      real
     +     et, dt,                        ! Elapsed Time (s)
     +     H, Hcr, R,                     ! Altitudes, Range (m)
     +     M, Mcr, Vx, Vz, V,             ! Mach numbers, Velocities (m/s)
     +     axo, azo, n_cl, g,             ! Accelerations, gravity (m/s^2)
     +     L, D, T, Tmax,                 ! Lift, Drag, Thrust (N)
     +     mv, mf, mfr,                   ! Vehicle Mass, Fuel Mass (kg)
     +     f,                             ! Engine Fuel-to-Air Ratio
     +     Sref,                          ! Wing Reference area (m^2)
```

```
+     Aeng                          ! Engine inlet area (m^2)
 real
+     ALPHA, GAMMA, GAMMA_cl,       ! Angle of Attack, Climb Angle (deg)
+     C_L, CL_max, C_D,             ! Lift and Drag Coefficients
+     q                             ! Dynamic Pressure (Pa)
 real
+     LIFT_COEFF(30,30),            ! Lift Coefficient Aero Matrix
+     DRAG_COEFF(30,30,10),         ! Drag Coefficient Aero Matrix
+     LIFT_TO_DRAG(30,30,10),       ! Lift-to-Drag Ratio Aero Matrix
+     ATTACK_ANGLE(30),             ! Attack Angle for Aero matrices (deg)
+     MACH_AERO(30),                ! Mach Number for Aero matrices
+     ALT_AERO(10),                 ! Altitude for Aero matrices (ft)
+     SPECIFIC_IMPULSE(20), Isp,    ! Engine Specific Impulse (s)
+     MACH_PROP(20),                ! Mach Number for Specific Impulse
+     SPEED_OF_SOUND(28), a,        ! Speed of Sound Table (m/s)
+     AIR_DENSITY(28), p,           ! Atmospheric Density Table (kg/m^3)
+     ALT_INFO(28)                  ! Altitude for above tables (ft)

 integer
+     nALT, nMACHa, nALPHA,         ! Size of Aerodynamics matrices
+     nMACHp,                       ! Size of Propulsion matrices
+     Io, Jo                        ! Saved ALPHA indices

 logical  FLY_EQUILIBRIUM

 common
+     et, H, Hcr, R, M, Mcr, Vx, Vz, V, axo, azo, Isp, GAMMA_cl,
+     Tmax, g, L, D, T, mv, mf, mfr, f, Sref, Aeng, ALPHA, GAMMA,
+     LIFT_COEFF, DRAG_COEFF, LIFT_TO_DRAG, ATTACK_ANGLE, n_cl,
+     MACH_AERO, ALT_AERO, SPECIFIC_IMPULSE, MACH_PROP, CL_max,
+     SPEED_OF_SOUND, AIR_DENSITY, ALT_INFO, a, p, q, C_L, C_D,
+     dt, nALT, nMACHa, nALPHA, nMACHp, Io, Jo, FLY_EQUILIBRIUM

 integer i

 data ALT_INFO /                                                  ! feet
+       0.,    5000.,  10000.,  15000.,  20000.,  25000.,  30000.,
+   35000.,  36089.,  40000.,  45000.,  50000.,  55000.,  60000.,
+   65000.,  70000.,  75000.,  80000.,  82021.,  85000.,  90000.,
+   95000., 100000., 110000., 120000., 130000., 140000., 150000. /

 data SPEED_OF_SOUND /                                            ! m/s
+    340.3,  334.4,  328.4,  322.2,  316.0,  309.7,  303.2,  296.5,
+ 11*295.1,  296.9,  300.0,  303.0,  306.0,  312.0,  317.8,  323.6,
+    329.2,  334.7 /

 data AIR_DENSITY /                                               ! kg/m^3
+  1.2250, 1.0556, 0.9047, 0.7708, 0.6527, 0.5489, 0.4583, 0.3796,
+  0.3639,  0.3016,  0.2372,  0.18645, 0.14663, 0.11532, 0.09069,
+  0.07131, 0.05608, 0.04410, 0.04002, 0.03428, 0.02655, 0.02067,
```

```
     +  0.016170, 0.010040, 0.006344, 0.004076, 0.002658, 0.0017591 /

c      begin
        call INITIALIZE
        call READ_DATA

        open( unit=3, status='NEW', file='PERF.DAT' )

        write(6,10)
        write(3,10)

   10   format(40X,'AEROSPACEPLANE PERFORMANCE FLYOUT'/
     +         40X,'---------------------------------'//
     +         '   TIME   RANGE  ALTITUDE  MACH  ALPHA  GAMMA   FUEL  ',
     +  '  WEIGHT       Q        CL        CD      L/D     Isp    THRUST '/
     +         '  (secs) (s.mi.)  (feet)         (deg)  (deg)    (lb)  ',
     +  '    (lb)      (psf)                              (sec)     (lb)  '/
     +         '  ------  -----  --------  ----  -----  -----  ------ ',
     +  ' --------  -------  ------  --------  -----  -----  --------')

        call WRITE_INFORMATION( 6 )
        call WRITE_INFORMATION( 3 )

        dt = 2
        do while ( H .lt. Hcr )
          i = 0
          do while ((i .lt. 5) .and. ( H .lt. Hcr ))
            call CLIMB
            i = i + 1
          enddo
          call WRITE_INFORMATION( 3 )
        enddo
        call WRITE_INFORMATION( 6 )

        dt = 2
        do while ( mf .gt. mfr )
          i = 0
          do while ((i .lt. 10) .and. ( mf .gt. mfr ))
            call CRUISE
            i = i + 1
          enddo
          call WRITE_INFORMATION( 3 )
        enddo
        call WRITE_INFORMATION( 6 )

        dt = 1
        T  = 0.0
        FLY_EQUILIBRIUM = .true.
        do while ( H .gt. H_50 )
          i = 0
```

```
        do while ((i .lt. 5) .and. ( H .gt. H_50 ))
          call DESCEND
          i = i + 1
        enddo
        call WRITE_INFORMATION( 3 )
      enddo
      call WRITE_INFORMATION( 6 )

      close( unit=3 )
    end

    subroutine INITIALIZE

    include 'COMMON.FOR'

c   begin
      et  =  0.0
      H   =  0.0
      R   =  0.0
      M   =  0.1
      Vx  = 34.0
      Vz  =  0.0
      V   = 34.0
      axo =  0.0
      azo =  0.0
      g   =  go
      GAMMA =  0.0
      C_D = 0.1
      T = 1.0

      return
    end


    subroutine READ_DATA

    include 'COMMON.FOR'

    integer  i, j, k
    logical  SET

c   begin
      open( unit=3, status='OLD', READONLY, file='PERF.IN' )

      read(3,10) Mcr, Hcr, GAMMA_cl, n_cl, Sref, Aeng, CL_max,
     +           mv, mf, mfr, f, nMACHp
  10  format(/,2F10.2,//,2F10.3,//,3F10.4,//,3F10.2,//,F10.6,I10)

      do i = 1, nMACHp
        read(3,20) MACH_PROP(i), SPECIFIC_IMPULSE(i)
```

```
   20     format(2F10.2)
        enddo

        close( unit=3 )
        open( unit=3, status='OLD', READONLY, file='AERO.DAT' )

        read(3,30) nALPHA, nMACHa, nALT
   30   format(/,3I10)

        SET = .false.
        do i = 1, nALPHA
          read(3,*) ATTACK_ANGLE(i)
          if ( (ATTACK_ANGLE(i) .gt. 0.0) .and. (.not. SET) ) then
            SET = .true.
            Io = i + 1
            Jo = i
          endif
        enddo
        do i = 1, nMACHa
          read(3,*) MACH_AERO(i)
        enddo
        do i = 1, nALT
          read(3,*) ALT_AERO(i)
        enddo

        do j = 1, nMACHa
          do i = 1, nALPHA
            read(3,*) LIFT_COEFF(i,j)
          enddo
        enddo

        do k = 1, nALT
          do j = 1, nMACHa
            do i = 1, nALPHA
              read(3,*) DRAG_COEFF(i,j,k), LIFT_TO_DRAG(i,j,k)
            enddo
          enddo
        enddo

        close( unit=3 )
        return
      end

      subroutine WRITE_INFORMATION( iunit )

      include 'COMMON.FOR'

      real   RANGE, ALTITUDE, FUEL, PSF, WEIGHT, LOD, THRUST

c     begin
```

```
        RANGE = R / 1609.344             ! Meters to Standard Miles
        ALTITUDE = H / 0.3048            ! Meters to Feet
        FUEL = mf / 0.45359237           ! Kilograms to Pounds-mass
        WEIGHT = mv / 0.45359237         ! Kilograms to Pounds-mass
        PSF = q / 47.8802590             ! Pascals to Pounds/square-foot
        THRUST = T / 4.4482216152605     ! Newtons to Pounds-force
        LOD = C_L / C_D

        write(iunit,20) et, RANGE, ALTITUDE, M, ALPHA, GAMMA, FUEL,
     +          WEIGHT, PSF, C_L, C_D, LOD, Isp, THRUST
   20   format( F8.0, F7.0, F10.1, F6.2, F7.2, F7.2, F8.0,
     +          F10.1, F9.1, F8.4, F10.6, F7.3, F7.0, F10.1 )
        return
      end


      subroutine CLIMB

      real   GAMMA_d

      include 'COMMON.FOR'

c     begin
        call GET_DATA

        if ( H .lt. H_50 ) then                          ! Take-off Rotation
          C_L = 0.8 * CL_max                             ! requires high C_L
          L = C_L * q * Sref
        else
          L = mv * g * cosd(GAMMA)  -  T * sind(ALPHA)    ! Equilibrium Lift
          GAMMA_d = GAMMA_cl * ( 1.0 - (H / Hcr)**4 )     ! GAMMA desired
          L = L * ( 1.0 + 10.0 * sind(GAMMA_d - GAMMA) )  ! Maintain GAMMA_d
          if ( L .lt. 0.0 ) L = 0.0
          C_L = L / q / Sref
        endif

        call FIND_CL( C_L )

        C_D = table_3D_lookup( 'LIN', DRAG_COEFF,
     +                       'GEO', ATTACK_ANGLE, 30, nALPHA, ALPHA,
     +                       'GEO', MACH_AERO, 30, nMACHa, M,
     +                       'EXP', ALT_AERO, 16, nALT, H/0.3048 )
        D = q * C_D * Sref

        T = ( D  +  mv * g * (sind(GAMMA) + n_cl) ) / cosd(ALPHA)
        if ( T .gt. Tmax ) T = Tmax
        if ( mf .eq. 0.0 ) T = 0.0

        call FLY
        return
```

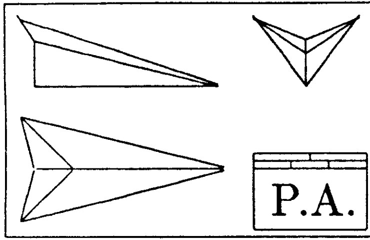
```
      end

      subroutine CRUISE

      include 'COMMON.FOR'
      integer i

c     begin
        call GET_DATA

        L = mv * g * cosd(GAMMA)  -  T * sind(ALPHA)      ! Equilibrium Lift
        L = L * ( 1.0 - 10.0 * sind(GAMMA) )              ! Maintain Level Flight
        if ( L .lt. 0.0 ) L = 0.0
        C_L = L / q / Sref

        if ( C_L .gt. 0.8 * CL_max ) then
          C_L = 0.8 * CL_max
          L = C_L * q * Sref
        endif

        call FIND_CL( C_L )

        C_D = table_3D_lookup( 'LIN', DRAG_COEFF,
     +                    'GEO', ATTACK_ANGLE, 30, nALPHA, ALPHA,
     +                    'GEO', MACH_AERO, 30, nMACHa, M,
     +                    'EXP', ALT_AERO, 16, nALT, H/0.3048 )
        D = C_D * q * Sref

        T = D / cosd(ALPHA)                               ! Equilibrium Thrust
        T = T * ( 1.0 + 3.0 * (Mcr - M) )                 ! Maintain Cruise Mach
        if ( T .gt. Tmax ) T = Tmax
        if ( mf .eq. 0.0 ) T = 0.0

        i = 0
        do while ((i .lt. 2) .and. ( mf .gt. mfr ))
          call FLY
          i = i + 1
        enddo

        return
      end


      subroutine DESCEND

      include 'COMMON.FOR'
      integer i
      real    ALPHA__max_lod

c     begin
```

```
        call GET_DATA

        call FIND_MAX_LOD

        if ( FLY_EQUILIBRIUM ) then
          ALPHA_max_lod = ALPHA
          L = mv * g * cosd(GAMMA)                      ! Equilibrium Lift
          C_L = L / q / Sref
          if ( C_L .gt. 0.8 * CL_max ) then
            C_L = 0.8 * CL_max
            L = C_L * q * Sref
          endif
          call FIND_CL( C_L )
          if ( ALPHA .gt. ALPHA_max_lod ) then
            FLY_EQUILIBRIUM = .false.
            ALPHA = ALPHA_max_lod
            C_L = table_2D_lookup( 'LIN', LIFT_COEFF,
     +                     'GEO', ATTACK_ANGLE, 30, nALPHA, ALPHA,
     +                     'GEO', MACH_AERO, 30, nMACHa, M )
            L = C_L * q * Sref
          endif
        else        ! FLY (L/D)max
          C_L = table_2D_lookup( 'LIN', LIFT_COEFF,
     +                     'GEO', ATTACK_ANGLE, 30, nALPHA, ALPHA,
     +                     'GEO', MACH_AERO, 30, nMACHa, M )
          L = C_L * q * Sref
        endif

        C_D = table_3D_lookup( 'LIN', DRAG_COEFF,
     +                     'GEO', ATTACK_ANGLE, 30, nALPHA, ALPHA,
     +                     'GEO', MACH_AERO, 30, nMACHa, M,
     +                     'EXP', ALT_AERO, 16, nALT, H/0.3048 )
        D = C_D * q * Sref

        i = 0
        do while ((i .lt. 2) .and. ( H .gt. H_50 ))
          call FLY
          i = i + 1
        enddo

        return
      end

      subroutine GET_DATA

      include 'COMMON.FOR'

c     begin
        a = table_1d_lookup( 'LIN', SPEED_OF_SOUND,
     +                     'LIN', ALT_INFO, 28, 28, H/0.3048 )
```

```
        p = table_1d_lookup( 'LIN', AIR_DENSITY,
     +                         'EXP', ALT_INFO, 28, 28, H/0.3048 )
        q = 0.5 * p * V * V
        M = V / a

        if ( T .eq. 0.0 ) then
          Isp = 0.0
        else
          Isp = table_1d_lookup( 'LIN', SPECIFIC_IMPULSE,
     +                             'LIN', MACH_PROP, 20, nMACHp, M )
          Tmax = p * V * Aeng * f * go * Isp
        endif
      end


      subroutine FLY

      include 'COMMON.FOR'

      real  an, at, ax, az, dVx, dVz, dR, dH, dmf

c     begin
        an  = (T * sind(ALPHA)  +  L  -  mv * g * cosd(GAMMA)) / mv
        at  = (T * cosd(ALPHA)  -  D  -  mv * g * sind(GAMMA)) / mv
        ax  = at * cosd(GAMMA)  -  an * sind(GAMMA)
        az  = at * sind(GAMMA)  +  an * cosd(GAMMA)
        if ( (az .lt. 0.0) .and. (H .eq. 0.0) ) az = 0.0

        dVx = (axo + ax) / 2.0 * dt
        dVz = (azo + az) / 2.0 * dt
        axo = ax
        azo = az

        dR  = (Vx + dVx / 2.0) * dt
        dH  = (Vz + dVz / 2.0) * dt

        Vx  = Vx + dVx
        Vz  = Vz + dVz
        V   = sqrt( Vx * Vx + Vz * Vz )
        GAMMA = atand( Vz / Vx )

        H   = H + dH
        if (H .lt. 0.0) H = 0.0
        R   = R + dR * (Ro / (H + Ro))
        g   = go * (Ro / (H + Ro))**2

        if ( T .gt. 0.0 ) then
          dmf = - T / go / Isp * dt
          mf  = mf + dmf
          mv  = mv + dmf
```

```
          if (mf .lt. 0.0) then
            mv = mv - mf
            mf = 0.0
          endif
        endif

        et = et + dt
      end

      subroutine FIND_CL( CL )

      include 'COMMON.FOR'

      real      ANG, CL, CL_L, CL_U, interpolate
      integer   J
      logical   PAST_LIMIT

c     begin
        J = Jo - 2
        CL_U = -9.0
        PAST_LIMIT = .false.

        do while ( (CL_U .lt. CL) .and. (J .lt. nALPHA)
     +             .and. (.not. PAST_LIMIT) )
          ANG = ATTACK_ANGLE( J )
          CL_L = CL_U
          CL_U = table_2D_lookup( 'LIN', LIFT_COEFF,
     +                     'GEO', ATTACK_ANGLE, 30, nALPHA, ANG,
     +                     'GEO', MACH_AERO, 30, nMACHa, M)
          if ( CL_U .eq. CL_L ) PAST_LIMIT = .true.
          J = J + 1
        enddo

        if ( PAST_LIMIT ) then
          ALPHA = ATTACK_ANGLE(J-2)
        else
          ALPHA = interpolate( CL_L, CL_U, ATTACK_ANGLE(J-2),
     +                       ATTACK_ANGLE(J-1), CL, 'GEO', 'LIN' )
        endif

        Jo = J - 2
        return
      end


      subroutine FIND_MAX_LOD

      include 'COMMON.FOR'

      real     LOD, LOD_MAX, ANG
```

```
      integer  I

c     begin
        LOD = 0.1
        LOD_MAX = 0.0
        I = Io - 2
        do while ( LOD .ge. LOD_MAX )
          ANG = ATTACK_ANGLE( I )
          LOD = table_3D_lookup( 'LIN', LIFT_TO_DRAG,
     +                     'GEO', ATTACK_ANGLE, 30, nALPHA, ANG,
     +                     'GEO', MACH_AERO, 30, nMACHa, M,
     +                     'EXP', ALT_AERO, 16, nALT, H/0.3048 )
          if ( LOD .gt. LOD_MAX ) LOD_MAX = LOD
          I = I + 1
        enddo
        ALPHA = ATTACK_ANGLE( I-1 )

        Io = I - 1
        return
      end
```



Glossary

agenda: is the common name for the computational agenda.

agenda building: is the process of determining how design functions will be used to find a solution to a chosen design path of knowns and unknowns. Agenda building does not involve any numerical methods since it does use the values of the design variables, only their states.

agenda entry: is an entry into the computational agenda consisting of a perfectly constrained design function and the unknowns to be solved for using it.

base variable: is the official designation of an I-state design variable, commonly referred to as a "known".

branch: is one of two independent sequences of perfectly constrained design functions for computing the value of the loop variable to solve a loop.

C: is the letter assigned to the state of a design variable whose value has been computed by *Paper Airplane* via processing.

C-state design variable: is a design variable whose value has been computed by *Paper Airplane*.

CEMISS: is a Computer-based Engineering Model Information Sharing System.

computational agenda: is the actual path, or sequence of design functions, to be evaluated to find the values of the unknowns once given the initialized values for the knowns and the guess values for the unknowns. The computational agenda is also called the computational path. The computational agenda consists of a forced path and loops.

computational path: is another name for the computational agenda.

Computed-value state: This indicates a design variable that *had* been given a trial value by the user, and was later given a known value by *Paper Airplane.* A design variable obtains state **C** only when the user processes the design set; and then only if *Paper Airplane* can find a solution which satisfies all the design functions in the user's design set.

computer program: is an external piece of code usually not written in COMMON LISP, such as a FORTRAN or PASCAL program.

derived variable: is the official designation of a **G**-state design variable, commonly referred to as an "unknown".

design function: is a relationship between design variables. A design function can range in complexity from a simple algebraic equation to a very large and complex computer program.

design path: is the selection of certain design variables as knowns and the rest as unknowns; thereby setting up some implied path, or sequence of design functions, for *Paper Airplane* to follow once values are provided for the design variables.

design point: is the values and states of all the design variables in a design set at any stage of the design process.

design set: is a set of certain design functions and the design variables those functions relate towards the goal of solving a particular design problem.

design variable: is a scalar parameter, such as Vehicle Length or Vehicle Weight, whose value uniquely determine part of the configuration of an aircraft, spacecraft, or any other system. A design variable has a number of attributes associated with it, such as its value, its dimensions, its order of magnitude, and the limits of its value.

engineering mode: is a sub-model (reduced model) of a mathematical model describing the structure and properties of an existing or proposed product.

external code interface: from one program to another allows both programs to share information without the need for a human to manipulate input and output files.

final design point: is the numerical solution to the initial design point of a particular design path. Specifically it is the states and values of the design variables after processing has been completed.

final path: is a sequence of perfectly constrained design functions whose unknowns can be solved for once a loop has been solved.

flavor: is a powerful LISP abstraction that allows for information storage and retrieval and data communications, all in a hierarchical structure.

floating: is the act of changing the state of a design variable to G, thereby setting the value of the design variable as guessed at and marking the design variable as an unknown.

forced path: is a sequence of perfectly constrained design functions, each of which can be solved individually, although sequentially. The path is called "forced" since there is no alternative but to solve the design functions in this sequence in order to compute the values of their unknowns.

forcing variable: is the design variable whose value is converged upon during the iteration of a loop. The forcing variable is usually the design variable most common to all the design functions involved.

forward computation: is a one-time single-function evaluation that computes the values of the output unknowns of a single design function by executing the function once using the values of the input knowns.

freezing: is the act of changing the state of a design variable to I, thereby setting the value of the design variable as initialized and marking the design variable as a known.

function: is an internal piece of code written in COMMON LISP.

G: is the letter assigned to the state of a design variable whose value has been guessed at by the user via floating.

G-state design variable: is a design variable whose value has been guessed at by the user.

Guessed-value state: This indicates a design variable that has been given a trial value by the user. A design variable obtains state G whenever the user floats it. G-state

design variables are officially designated as "derived variables" and are commonly referred to simply as "unknowns".

I: is the letter assigned to the state of a design variable whose value has been initialized by the user via freezing.

I/O: Input and output. The data passed to and from a computer program.

I-state design variable: is a design variable whose value has been initialized by the user.

initial design point: is the initial setting of the values of design variables according a particular design path. This consists of initialized values for the chosen knowns and guess values for the chosen unknowns.

initial path: is a sequence of perfectly constrained design functions whose computed unknowns are required by both branches of a loop to solve that loop.

Initialized-value state: This indicates a design variable that has been given a known value by the user. A design variable obtains state **I** whenever the user changes its value, or when the user freezes it. I-state design variables are officially designated as "base variables" and are commonly referred to simply as "knowns".

instance variable: is a parameter that is an element of the structure of a flavor.

known: is the common name for an I-state design variable, officially designated as a "base variable".

loading: is a COMMON LISP term for reading and evaluating LISP code from a file into main memory.

loop: is a sequence of design functions, each of which computes values required by other design functions in a closed loop. Loops are solved by guessing the value of a forcing variable to compute two independent values of a loop variable. When the two values converge, the values of all the unknowns involved can be found.

loop computation: is an iterative multiple-function evaluation that computes the values of all the unknowns of a set of design functions by guessing values of a chosen forcing variable until two independent values of a chosen loop variable converge.

loop variable: is the design variable whose two independently computed values determine the convergence of a loop.

method: is a function that is specifically associated with a flavor.

MIMO: Multiple-Input Multiple-Output. Loosely, a multiple-input multiple-output design function.

MIMO Design Set: is the design set used to test the MIMO design function solving capability enhancement to *Paper Airplane*. Several MISO design functions from the MISO Design Set were merged to form MIMO design functions.

MISO: Multiple-Input Single-Output. Loosely, a multiple-input single-output design function.

MISO Design Set: is the design set that serves as the foundation of all other design sets, except for the NASP Design Set. The MISO Design Set contains 17 design variables and 7 design functions and is used for the conceptual design of aircraft.

mixin: is a flavor that is an element of the structure of another flavor.

NASP Design Set: is the design set used to test the final version of the enhanced *Paper Airplane*. The NASP Design Set is comprised of 12 design variables and 9 design functions, including MISO and MIMO design functions and design functions calling external codes. The NASP Design Set is used for the preliminary design of a national aerospaceplane.

NIL: the New Implementation of LISP, a dialect of COMMON LISP, and the programming language in which *Paper Airplane* is written.

overconstrained: problem is one in which the number of unknowns is less than the number of values computed by all the design functions (i.e., the number of user-specified knowns is greater than that required). This can lead to design variables receiving two or more incompatible values.

Paper Airplane: is the name of the code development at the Massachusetts Institute of Technology to solve systems of linear and/or non-linear functions.

perfectly constrained: problem is one in which the number of unknowns equals the number of values computed by all the design functions (i.e., the number of user-specified knowns is the same as that required). This usually leads to design variables whose values can be exactly determined.

postprocessor: reads output values from the file(s) an external code normally writes to and returns them to the system.

preliminary entries: is the initial sequence of perfectly constrained design functions created while trying to construct a closed loop.

preprocessor: takes input values from the system and writes them out to the file(s) an external code normally reads from.

processing: is the act of instructing *Paper Airplane* to attempt to compute the values of all the unknowns of a design set.

reverse computation: is an iterative single-function evaluation that computes the values of all the unknowns of a single design function by guessing values of the unknown input variables until values of the known output variables converge with their user-specified values.

source file: is a computer file containing the information on all of the design variables and design functions to be loaded internally into a *Paper Airplane* design set.

state: is the common name for variable state.

TAV Design Set: is the former name of the NASP Design Set.

underconstrained: problem is one in which the number of unknowns is greater than the number of values computed by all the design functions (i.e., the number of user-specified knowns is less than that required). This can lead to design variables whose values cannot be exactly determined.

unknown: is the common name for a G-state design variable, officially designated as a "derived variable".

variable state: is the condition of the value of a design variable. Variable states come in the following three varieties, which are assigned to design variables according to their initial letter: Initialized-value state, Guessed-value state, and Computed-value state.

variable tableau: is a spreadsheet of information on the design set arranged on a computer screen. This information includes a list of design variables and their current values, units, and states.

XCODE: is an external code.

XCODE Design Set: is the design set used to test the external code interface capability enhancement to *Paper Airplane*. One design function from the MISO Design Set was modified to call a FORTRAN program to compute its value.

Bibliography

[1] Conte, S. D., and de Boor, Carl, *Elementary Numerical Analysis: An Algorithmic Approach* (Third edition), McGraw-Hill, New York, NY, 1980.

[2] Dudley, Richard M., Dept. of Mathematics, Massachusetts Institute of Technology, personal communication, July 1986.

[3] Elias, Antonio L., "Knowledge Engineering of the Aircraft Design Process," *Knowledge Based Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1985, chapter 6.

[4] Elias, Antonio L., Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, personal communication, September 1985 through August 1986.

[5] 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.

[6] Hill, Philip G., and Peterson, Carl R., *Mechanics and Thermodynamics of Propulsion*, Addison-Wesley, Reading, MA, 1970.

[7] 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.

[8] Kolb, Mark A., "On the Numerical Solution of Simultaneous, Non-linear Equations in Computer-aided Preliminary Design," S.M. Thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, January 1986.

[9] Kolb, Mark A., Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, personal communication, September 1985 through December 1986.

[10] Lajoie, Ronnie M., and Elias, Antonio L., *Paper Airplane User's Manual* (Second Edition), Flight Transportation Laboratory, Massachusetts Institute of Technology, February, 1986.

[11] 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.

[12] Martin, James A., "Aerospaceplane Optimization and Performance Estimation," S.M. Thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, August 1967.

[13] Miller, Rene H., Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, personal communication, June 1986 through December 1986.

[14] Nicolai, Leland M., *Fundamentals of Aircraft Design*, METS, Inc., Dayton, OH, 1975.

[15] Pararas, John, Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, personal communication, September 1985 through December 1986.

[16] Software Arts Inc., *TK!Solver Program Instruction Manual*, Software Arts Inc., Wellesley, MA, 1982.

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