The Design and Implementation of a GUI-Based Control Allocation Toolbox in the MATLAB® Environment

by

Michelle L. Glaze

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Aerospace Engineering

Approved:

Dr. Wayne Durham, chair

Dr. Mark Anderson

Dr. Frederick Lutze

December, 1998

Blacksburg, Virginia

Key Words: Control Allocation, Matlab, GUI

Copyright 1998 Michelle L. Glaze

THE DESIGN AND IMPLEMENTATION OF A GUI-BASED CONTROL ALLOCATION TOOLBOX IN THE MATLAB® ENVIRONMENT

by

Michelle Lynn Glaze

Chairman: Dr. Wayne Durham

Aerospace and Ocean Engineering Department

(ABSTRACT)

Control Allocation addresses the problem of the management of multiple, redundant control effectors. Generally speaking, control allocation is any method that is used to determine how the controls of a system should be positioned to achieve some desired effect. An infinite number of allocation methods exist, from the straight-forward direct allocation technique, to the daisy chaining approach, to the computationally simple generalized inverse method. Because different methods have advantages and disadvantages with respect to others, the determination of the "optimal" control allocation method is left to the system designer. The many tradeoffs that are addressed during control system design, of which control allocation is an integral part, dictate the need for a reliable, computer-based design tool. The Control Allocation Toolbox for MATLAB® satisfies such a need by providing the designer with a means of testing/comparing the validity of certain allocation methods under prescribed conditions. The issues involved in the development and implementation of the Control Allocation Toolbox are discussed.

This work was conducted under NASA research grant NAG-1-1449 supervised by John Foster of the NASA Langley Research Center.

Acknowledgments

I would like to express my gratitude and appreciation to all those who helped to contribute to this research. For serving as my advisory committee, and providing their knowledge and support throughout, I thank Dr. Wayne Durham, Dr. Mark Anderson, and Dr. Frederick Lutze. I thank Dr. Ken Bordignon and John Bolling, former Virginia Tech graduates, for their previous work and well-written documentation on control allocation as it is the foundation for the work herein. I am also grateful to my parents, Larry and Judy Glaze, my sister, Kerri, and my boyfriend, Michael, for their unending support and faith in my abilities.

Michelle L. Glaze

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

One of the primary considerations in control system design is positioning the control effectors to achieve some desired effect. In the case of aerospace vehicles, the desired effect is the generation of body-axis moments (i.e., roll, pitch, yaw) in response to pilot inputs. Classically, an aircraft was assumed to have three primary control effectors, each responsible for independently controlling one of the three rotational degrees of freedom; the ailerons for roll, the rudder for yaw, and the elevator for pitch. Modern aircraft, however, often utilize these classical effectors in combination with one or more advanced effectors, e.g., flaps, canards, and thrust vectoring, to control the same types of rotational motion. Tactical aircraft provide relevant examples; the F-15 ACTIVE operates with 12 effectors, the F-18 HARV with 10, and the F-16 with 5 independent control effectors.

The addition of advanced, redundant control effectors, though offering increased aircraft maneuverability, inherently increases the complexity of control system design. In mathematical terms, most modern aircraft are represented as under-determined systems because there are more unknowns (i.e., control effectors) than there are equations (i.e., rotational degrees of freedom). Unlike the uniquely determined classical system, the under-determined system has an infinite number of possible solutions; for the case considered here, there exist an infinite number of possible control configurations for the same set of desired moments. With so many possible combinations, there is no longer an obvious answer to the question of how to allocate controls in response to prescribed demands.

To effectively address the problem of managing multiple, redundant control effectors, a control allocator is introduced to the system. Generally speaking, control

allocation is considered as any method that can be used to determine how the controls of a system should be positioned to achieve some desired effect. As discussed in Ref. 1, a diverse group of allocation methods exist, from the straightforward direct allocation scheme, to the daisy chaining approach, to the computationally simple generalized inverse algorithms. Each algorithm has advantages and disadvantages with respect to others; it is the designer who determines which is the most suitable in a given situation.

In the interest of determining "optimal" performance, control allocation has been the subject of much research in the past, and is the foundation for the work presented in this thesis. Unlike previous theoretical research in this area, however, the main focus here is the comparative value of the allocation results in assessing "optimal" method performance.

1.1 Background

With the introduction of redundant control effectors to the classical aircraft, control allocation has become an integral part of control system design. The purpose of the control allocator is to, given a set of desired moments, $\mathbf{m_d}$, determine the set of control effector deflections, $\mathbf{u}$, that satisfy the vector equation:

$$\mathbf{m_d} = B\mathbf{u} \tag{1.1}$$

where B is a matrix that represents the effectiveness of the control effectors in generating body-axis moments (i.e., control effectiveness). The effectiveness is assumed to be linear, and is calculated as the slope of the moment vs. control deflection curve. This slope can be defined in any way desired, e.g., slope at the origin, slope at an arbitrary operating point, or secant slope to the limits of control deflection, as discussed in Ref. 2.

Although extensive in its utility, control allocation does have certain limitations. One such limitation is a direct result of the physical limits imposed on the aircraft hardware. Of the many possible control configurations satisfying equation (1.1) there exist those that are physically impossible to achieve because of control effector position

and rate limits. In these instances, control allocation is restricted to commanding only those deflections that are within the operational scope of the control effectors (i.e., the admissible set of controls). Another limitation of control allocation results from the linear approximation in (1.1). This approximation does not account for the non-linear effects introduced by changes in aircraft flight condition over time; thus, allocation must be restricted to very small time intervals. Although other limitations exist, as discussed in Ref. 1, these two in particular are addressed in this research.

Despite its limitations, control allocation remains a necessary part of control system design. If the controls of a system can be allocated in such a manner as to provide desired system performance while minimizing system weight and complexity, then clear gains in life-cycle cost may be achieved. Consider, for example, the case of an aircraft with 10 control effectors that are allocated using some particular method. If the effectors can be allocated in some different way such that 8 effectors provide the same performance, then the use of 10 effectors introduces unnecessary weight and complexity into the overall design. The determination of an "optimal" control allocation method to satisfy particular performance goals is therefore critical in aircraft control system design.

1.2 Research Objectives

The many tradeoffs inherent in control system design dictate the need for a computer-based design tool. Such a tool should provide the designer with a means of testing/comparing the validity of various control allocation methods under prescribed conditions. This tool should present both graphical and numerical results as to which method is most effective in the overall system design. The primary objective of this research was the design and implementation of such a tool. More specifically, developing a reliable, platform-independent, user-friendly toolbox in the MATLAB® environment that could easily and effectively be used to compare control allocation methods.

Previous work at Virginia Tech resulted in the development of the **C**ontrol **A**llocation **T**oolbox, simply referred to as CAT. Originally designed by Dr. Wayne

Durham and Dr. Ken Bordignon in support of their common research interests, this computer-based tool is very effective in testing/comparing a wide variety of allocation methods. Certain aspects of its design, however, prevent CAT from being used by anyone outside of the Virginia Tech Dynamics and Control research community. Of these, the most important is the fact that the Control Allocation Toolbox is platform dependent; it was written in Pascal for use on a Macintosh computer. Being used only for in-house research until now, CAT does not provide appropriate user instruction during operation, or in an off-line user's guide. Consequently, a trial-and-error approach must be adopted which can, more often than not, cause software failure without warning or justification. The MATLAB® Control Allocation Toolbox presented in this thesis attempts to correct for the deficiencies in the original **C**ontrol **A**llocation **T**oolbox design architecture, while continuing to provide high-quality, comparative allocation results.

A secondary aim of this research indirectly related to the first, is the establishment of a metric in the evaluation and comparison of allocation methods that determined "optimal" performance. From equation (1.1) and the associated background, it is understood that for a given linear effectiveness matrix, *B*, and control effector position limits, $\mathbf{u}_{min}$ and $\mathbf{u}_{max}$, there exists a set of maximum attainable performance. Expanding on this knowledge in a subsequent section of this text results in the desired performance metric.

1.3 Comments to the Reader

The majority of the research presented herein deals with the implementation issues associated with the development of a graphical user interface in the MATLAB® environment. Therefore, any further theoretical discussions will focus on providing the necessary background to clearly understand the implementation of various control allocation methods. The theoretical background provides the framework for Chapter 2, where each allocation method implemented in the toolbox is discussed in detail. The discussion therein is not limited to theory; it includes the allocation algorithms, and the actual method of implementation within the MATLAB® environment.

The reader who is solely interested in designing a graphical user interface for MATLAB® should skip directly to Chapter 3, Chapter 4, and Appendix I. Chapter 3 presents an overall picture of what a graphical user interface is trying to accomplish, in terms of appearance and overall usefulness. It then delves into a more detailed description of how to utilize MATLAB® in creating and operating such a GUI-based tool. In Chapter 4, the author reflects on problems encountered during the design and implementation process. Appendix I contains the various M-files that were written to support toolbox functions. The code has been documented to facilitate the possibility of future revision. It should be noted, that some of the code is internal to the graphical user interface, and has not been included as part of Appendix I.

To fulfill the aims of this thesis, a User's Guide for the MATLAB® Control Allocation Toolbox was written, and is included as Appendix II. This guide takes the user on a tour of the toolbox, providing him/her with all of the knowledge necessary to successfully operate the toolbox and understand the results presented. It also demonstrates the utility of the toolbox through an illustrative example using linearized data from an F-18 HARV aerodynamic database.

CHAPTER 2

Control Allocation Methods

The desire to increase control power has given rise to the use of multiple, redundant control effectors on most modern aircraft. The use of these effectors, although providing greater maneuverability and better performance, increases the overall complexity of control system design. This complexity arises from the fact that aircraft now have more control effectors to position than rotational degrees of freedom to control. Therefore, there exist an infinite number of possible control configurations to satisfy the moments demanded. The problem of control allocation then, is to determine how to position a set of control effectors to achieve a specific objective.

It has been established that there exist an infinite number of possible allocation methods for any aircraft that has more control effectors than degrees of freedom. However, only three such methods are examined in this thesis as they are directly relevant to the goals set forth previously; direct allocation, pseudo-inverse allocation, and weighted pseudo-inverse allocation. The numerous other allocation methods are accounted for nonetheless, through the introduction of a "general allocation method" which utilizes a convex hull algorithm to generate comparative results.

It is the purpose of this chapter to provide a greater understanding of the allocation methods evaluated in the MATLAB® Control Allocation Toolbox. The underlying theory of each method, including that for the "general allocation method", will be presented in preparation for subsequent discussions. The control allocation algorithms are then presented and explained to provide a foundation for the resulting implementation. After establishing the algorithmic particulars, the actual MATLAB® implementations are investigated with frequent reference to the code presented in Appendix I.

2.1 Direct Allocation

Simply defined, direct allocation is a solution to the constrained control allocation problem. To comprehend the capability afforded by direct allocation it is imperative to thoroughly understand the nature of the problem it solves. The constrained control allocation problem is unique in that it considers a set of pre-determined constraints in its solution. The addition of these constraints limits the scope of the method such that the solution achieved is unique. The basic knowledge required to understand the problem, and its solution is presented here. For a more detailed theoretical discussion on the underlying theory, consult Ref. 1, 3, and 4.

2.1.1 Theoretical Background

As in the past, the redundant control effectors on modern aircraft are primarily considered as moment generators. The effectors are also considered independent of one another in their effects, and individually linear in their motion. Considering this, it may be concluded that each control has some linear effectiveness in generating the three aircraft body-axis moments. With this information, a 3x*m* control effectiveness matrix (*B*) can be defined for the constrained control allocation problem. The rows of this matrix correspond to the rolling, pitching, and yawing moments, respectively, while the columns represent the *m* redundant control effectors. To satisfy the requirement of independent control effectors, and hence determine a unique solution, every 3x3 partition of the effectiveness matrix must be nonsingular (i.e., have a nonzero determinant).

In addition to the previous considerations, it is generally understood that every aircraft control effector has restrictions on its range of motion. Imposed in the form of minimum and maximum position/deflection limits, such restrictions are imparted due to physical limitations of the effector geometry, or some aerodynamic constraint. Suppose the control effectors and their respective position limits are mapped to some *m*-dimensional control space. Within this space there exists some subset of control combinations that will never violate any of the constraints. This set of combinations is

referred to as the subset of admissible controls, and is represented in the constrained control allocation problem as Ω.

By definition, the control effectiveness matrix directly relates the control effectors to the moments they command. As such, it can be used with the subset of admissible controls to completely determine all moments that are attainable within the pre-determined control limits. This set of moments, represented as Φ in the control allocation problem, is referred to as the attainable moment subset or AMS. The 3x*m* *B* matrix essentially acts as a linear mapping between the *m*-dimensional Ω and the 3-dimensional attainable moment subset, Φ.

The final component of the constrained control allocation problem is the vector of desired moments, $\mathbf{m_d}$ (with magnitude $|\mathbf{m_d}|$ and direction $\hat{\mathbf{m}}$), associated with a particular aircraft maneuver. Having completely defined its constituents, the constrained control allocation problem may be defined as follows: given the control effectiveness, *B*, the subset of admissible controls, Ω, and some desired moment vector, $\mathbf{m_d}$, determine the control combination in Ω that generates a moment vector in the direction of $\hat{\mathbf{m}}$ with the largest possible magnitude $|\mathbf{m_d}|$.

In solving the constrained control allocation problem, direct allocation utilizes the maximum capabilities of the aircraft control effectors in generating specific moment demands. Thus, the resulting control combination is the most efficient possible solution, yielding the maximum moments available by the given set of effectors. Similarly, the attainable moment subset (AMS) and its associated volume are the maximum possible for this method. Effectively establishing the performance metric that was the secondary aim of this research, the volume of the direct allocation AMS will be used to measure the relative performance of other allocation methods.

Subsequent sections will explain the direct allocation algorithm used to solve the constrained control allocation problem, and will provide insight into its implementation in the MATLAB® environment for use in the Control Allocation Toolbox.

2.1.2 Solution Approach

There are essentially two parts to solving the constrained control allocation problem using direct allocation; the first is calculating the attainable moment subset, the second is computing the allocated control vector. The theory behind this solution approach is discussed in the references cited earlier, and will not be reproduced here except as needed to support later implementation issues. The algorithm for direct allocation is presented below. Each paragraph with an *italicized* leading sentence represents a new step in the algorithm.

Using the control effectiveness matrix, B, and the subset of admissible controls, Ω, determine the AMS boundary, ∂(Φ). Though there are several methods that accomplish this task, only one is introduced here. Relatively speaking, the chosen method is simple in that it only requires the inversion of a 2x2 matrix and subsequent matrix-vector multiplication to effectively describe a facet in 3-dimensional moment space. Initially, the method pairs up control effectors from the admissible control subset, Ω, by taking 2x2 nonsingular partitions of the *B* matrix. These control effectors generate parallel facets in Ω which contain ∂(Ω). Applying the linear transformation (1.1), the parallel facets in Ω are mapped to moment space, remaining parallel in Φ. According to Ref. 5, there will be 2^{m-2} such faces in Φ but only two will lie on ∂(Φ).

Effectively determining which of the parallel facets in Φ lie on ∂(Φ) requires two steps. The first involves rotating the moment axis (i.e., C_l, C_m, or C_n) not represented by the 2x2 partition of *B* such that it parallels the perpendicular to the facets in question. The alignment is achieved by applying a pure rotation transformation to *B* such that the columns of the resulting matrix that correspond to the pair of facet-defining control effectors (i.e., those represented by the 2x2 partition of *B*) are equal to zero. The resulting 2x3 under-determined system of equations formed by these columns is used subsequently.

The second step in determining ∂(Φ) requires calculating the control combinations that yield the minimum and maximum moments along the newly aligned moment axis. Solving the previously-mentioned 2x3 system of equations, as described in Ref. 5, yields

a row vector with *m*-2 nonzero entries (i.e., provided every 3x3 submatrix of *B* has full rank); those corresponding to the pair of facet-defining controls are zero. The sign of each nonzero entry is used to determine what values of control effector deflection provide the minimum and maximum moments along the new direction; a positive value indicates maximum deflection, a negative value indicates minimum deflection. Coupling these results with the four possible deflection configurations for the facet-defining controls effectively describes the four vertices of a facet in moment space. This two step procedure is applied to every possible 2x2 partition of the original *B* matrix, resulting in the completely defined AMS boundary, $\partial(\Phi)$. By definition, everything inside $\partial(\Phi)$ is Φ, and is used to provide a means of comparison between allocation methods; its volume is the performance metric for determining "optimal" method performance.

The final step is allocating the control effectors to achieve the maximum attainable moment. Provided the desired moment vector, $\mathbf{m_d}$, is known, each facet on $\partial(\Phi)$ is tested to determine if $\mathbf{m_d}$ is pointing in its direction. This is achieved by solving the relationship:

$$\begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix} = F^{-1}\mathbf{m_d} \tag{2.1}$$

where c_1, c_2, and c_3 are the constants being determined, and F is a nonsingular matrix whose columns define the geometry of the current facet (i.e., the first column is a vector from the AMS origin to a chosen vertex, the second and third columns are edges from the chosen vertex to two other facet vertices). The constants in (2.1) must satisfy the following requirements for the current facet to be used for allocation:

$$c_1 > 0, \quad 0 \le c_2 \le 1, \quad 0 \le c_3 \le 1 \tag{2.2}$$

Whereupon the conditions in (2.2) are satisfied, allocation continues. Each of the *m*-2 effectors in **u** that do not define the current facet are allocated as:

$$U_i = c_1 u_i \ , \qquad i = 3,m \tag{2.3}$$

where U_i is the value of the ith allocated effector, c_1 is the scaling constant from (2.1) and u_i is either the minimum or maximum deflection limit of the current control effector depending on the results from step 2. The remaining pair of facet-defining control effectors, u_1 and u_2, are allocated similarly according to:

$$\begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = c_1 \begin{Bmatrix} u_{1_{min}} \\ u_{2_{min}} \end{Bmatrix} + \begin{bmatrix} c_2 & 0 \\ 0 & c_3 \end{bmatrix} \begin{Bmatrix} u_{1_{max}} - u_{1_{min}} \\ u_{2_{max}} - u_{2_{min}} \end{Bmatrix} \tag{2.4}$$

where U_1 and U_2 are the allocated pair of facet-defining effectors, c_1, c_2, and c_3 are the scaling constants resulting from (2.1); and the remaining variables are the minimum/maximum positions of u_1 and u_2 as labeled. To complete the allocation process, the attained moments, $\mathbf{m}_{\mathbf{atn}}$, are calculated:

$$\mathbf{m}_{\mathbf{atn}} = B\boldsymbol{U} \tag{2.5}$$

where B is the control effectiveness matrix and $\boldsymbol{U}$ is the vector of allocated control effectors including the results of (2.4). If the attained moments are determined to lie beyond the AMS boundary, the allocated control vector is scaled such that the attained moments will lie on the AMS boundary.

Thus concludes the direct allocation algorithm. The attainable moment subset (AMS) was computed by determining its boundary, the control effectors were allocated accordingly, and the attained moments were calculated as a result. The algorithm above was coded in MATLAB® for later use in the Control Allocation Toolbox. The various parts of the implementation are discussed in the next section.

2.1.3 MATLAB® Implementation

The MATLAB® programming language is relatively easy to understand and apply. As such, the code written to implement the direct allocation algorithm is included in Appendix I, and will be referred to throughout this discussion. There are a total of six M-files (i.e., the MATLAB® equivalent of a subroutine) required for the implementation. Brief descriptions of each are provided below in the order they are called during implementation:

1) ***ATTAIN.M***	Main calling routine; determines the AMS and its boundary $\partial(\Phi)$ through an iterative process.
2) ***BEGIN.M***	Loads effectiveness matrix and control position limits; determines number of controls; creates indices to partition *B* matrix into 2x2 sections.
3) ***MATRIX.M***	Determines the facet geometry and sets up matrices that are used for plotting the AMS, determining its volume, etc.
4) ***AMSVOL.M***	Calculates the AMS volume.
5) ***DIRALLO.M***	Actual allocation routine; performs facet search to determine if allocation is possible.
6) ***ALLOCATE.M***	If allocation is deemed possible, determines the vector of allocated control effectors and calculates the set of maximum attainable moments.

Although these descriptions are self-explanatory, each is expanded upon herein to insure proper understanding of the overall implementation. It should be noted that the code in Appendix I contains sections that were not added until the algorithm was included in the graphical user interface. Only those sections relevant to the research presented thus far will be explained; the graphical user interface sections will be discussed in Chapter 3.

Note that the M-file being focused on in each paragraph below is presented in **bold-faced** type.

The main calling function of the direct allocation implementation is ***ATTAIN.M***. As such, it requires the desired moment vector, $\mathbf{m_d}$, as an input. The output is computed by various functions called during implementation, and consists of the allocated control vector, the attained moments, and the AMS volume. Prior to any actual calculation, BEGIN.M is called to determine the control effectiveness matrix, *B*, and the control position limit vectors, $\mathbf{u_{min}}$ and $\mathbf{u_{max}}$. Using this information, ***ATTAIN.M*** begins the iterative process of determining the facets of the attainable moment subset; the remaining calculations are made inside an iterative "while" loop as described below.

To determine each facet, ***ATTAIN.M*** searches for a nonsingular 2x2 partition of the *B* matrix, and inverts it. The moment axis (C_l, C_m, C_n) not represented by this partition is chosen to be aligned with the perpendicular to the undetermined facet. This decided, ***ATTAIN.M*** applies a pure rotation transformation and solves the resulting system of equations to yield a row vector of *m* control effectors; the entries are zero for the two facet-defining controls, and nonzero otherwise. From this, a 1x*m* facet-defining control vector is determined employing the use of a base-3 numbering system; 0 (zero) represents a minimum deflection, 1 a maximum, and 2 represents varying controls. The base-3 vector is sent to the subroutine MATRIX.M where the facet geometry and vertex connections are determined. The vertex connections are sent to AMSVOL.M for associated volume calculations. Finally, ***ATTAIN.M*** calls DIRALLO.M to determine if controls can be allocated using the newly determined facet. The entire process is repeated (i.e., the loop starts over) until every possible partition of the *B* matrix has been used for facet determination.

The function ***BEGIN.M*** is the first function called by the main routine as discussed above. This function takes a 5x*m* matrix (either read from a text file or generated by MATLAB®) and effectively divides it into the 3x*m* control effectiveness matrix, *B*, and the 1x*m* minimum/ maximum position limit vectors. It then determines all possible 2x2 partitions of the *B* matrix, saves the indices (i, j) of the partitions in

individual row vectors, and counts the total number of partitions (i.e., the "while" loop counter in ATTAIN.M). Thus, ***BEGIN.M*** provides all of the required data for the ensuing calculations of the implementation.

The next function of interest is ***MATRIX.M***. The inputs required for calculation include the number of controls, the effectiveness matrix, the base-3 facet-defining control vector resulting from the rotation transformation in ATTAIN.M, the partition index vectors from BEGIN.M, and the minimum/maximum effector position limits. Applying these inputs, ***MATRIX.M*** effectively calculates the coordinates of the four facet vertices in moment space and arranges them to create a matrix of vertex connections. This routine also creates a 3x3 facet geometry matrix (*F* in equation 2.1). The first column represents the coordinates of the defining vertex, and the remaining columns represent vectors from the defining vertex to two other facet vertices (i.e., essentially describe two facet edges).

The fourth function called from the main routine is ***AMSVOL.M***. The sole purpose of this routine is to use the vertex connection matrix determined by MATRIX.M to calculate the volume of the AMS in terms of the new facet. This is accomplished by calculating the volume of the pyramid formed by the facet and the four vectors from the origin of moment space to each vertex of the facet. This elemental volume is returned to ATTAIN.M where it is added to the overall AMS volume.

The time-consuming facet search of the direct allocation algorithm is implemented in ***DIRALLO.M***. The numerous inputs to this function are the number of controls, effectiveness matrix, vectors of partition indices, facet geometry matrix, minimum/maximum position limits, and the desired moment vector. The basic function of this routine is to determine whether or not the current facet can be used to allocate controls. Thus, ***DIRALLO.M*** first calculates the determinant of the facet geometry matrix, *F*. If the resulting determinant is nonzero (i.e. nonsingular), then the three constants in (2.1) are computed and tested against the requirements of (2.2). Provided the conditions are satisfied, ***DIRALLO.M*** calls ALLOCATE.M to proceed with allocation; otherwise, the function terminates returning to ATTAIN.M. If *F* is determined to be

singular, ***DIRALLO.M*** runs through various tests to decide whether allocation is still possible, and proceeds accordingly.

The final routine in the direct allocation implementation is ***ALLOCATE.M***. Called only if allocation is deemed possible, this function effectively determines the vector of allocated control effectors and the associated vector of attained moments. Inputs to ***ALLOCATE.M*** consist of the number of controls, the effectiveness matrix, the partition index vectors, the base-3 facet-defining control vector from ATTAIN.M, the three constants in (2.1), and the minimum/maximum position limits. With this information, ***ALLOCATE.M*** proceeds to allocate controls as described by (2.3) and (2.4), and determines the associated set of maximum attainable moments as described by (2.5).

Thus is the MATLAB® implementation of direct allocation. The implementation, as described above and presented in Appendix I, successfully solves the constrained control allocation problem posed earlier. The implementation effectively determines the AMS for the given set of control effectors, and determines the AMS volume as a metric of comparison with other methods. The implementation then determines whether allocation is possible, allocates accordingly, and when appropriate uses the resulting set of control effectors to determine the maximum attainable moment vector, essentially solving the problem.

2.2 Generalized Inverse Allocation

Generalized inverse allocation encompasses a wide range of solution methods for the problem of positioning redundant control effectors. To completely determine the positions of the allocated effectors, these methods simply multiply the desired moment vector by some pre-determined matrix. Inherently easier and faster than most methods of allocation, generalized inverse methods are more often utilized in modern control systems. A brief discussion of the underlying theory, specifically that pertaining to the pseudo-inverse and weighted pseudo-inverse methods, is presented below. For fundamental knowledge on other generalized inverse methods, consult Ref. 1 and 3.

2.2.1 Theoretical Background

The family of generalized inverses is often used to solve under-determined systems of equations. As applied to control allocation, this requires the solution of the linear approximation in (1.1). To accomplish this, each generalized inverse method constructs a matrix, *P*, such that:

$$I_3 = BP \tag{2.6}$$

where I_3 is the 3x3 identity matrix, and *B* is the control effectiveness matrix as before. Substituting this equation into (1.1) and rearranging appropriately yields the following::

$$\mathbf{u} = P\mathbf{m_d} \tag{2.7}$$

where **u** is the vector of allocated control effectors, and $\mathbf{m_d}$ is the set of desired moments. Equation (2.7) is essentially the defining vector equation for every generalized inverse solution.

The issue then becomes one of specifying *P* to simultaneously satisfy (2.6) and solve (2.7). One of the most common techniques is the pseudo-inverse method. This particular method is simple to solve, and offers the benefit of minimized control energy by effectively minimizing the two-norm of the given set of control effectors. The *P* matrix constructed for this method, appropriately named the pseudo-inverse matrix (i.e., the right pseudo-inverse since the system in question is under-determined), is calculated according to:

$$P = B^T (BB^T)^{-1} \tag{2.8}$$

where *B* is the full rank control effectiveness matrix. Requiring *B* to be full rank prior to allocation insures that BB^T is invertible, and guarantees that there is at least one solution to the control allocation problem.

A popular variation of the pseudo-inverse method is referred to as the weighted pseudo-inverse, and involves the use of an *m*x*m* weighting matrix, *W*. Introducing this matrix into equation (2.8) results in the weighted pseudo-inverse matrix:

$$P = W(BW)^T\left[BW(BW)^T\right]^{-1} \tag{2.9}$$

with *B* defined previously. To offer the interpretation of emphasizing/de-emphasizing the effects of certain controls, *W* is assumed to be a diagonal matrix whose individual weighting coefficients uniquely correspond to a given control effector. The weighting coefficients are frequently computed as the reciprocal of the magnitude of the overall range of motion of the chosen control:

$$W(i,i) = \frac{1}{\left|u_{i\,max} - u_{i\,min}\right|} \tag{2.10}$$

Although worthwhile in terms of computational efficiency, generalized inverses are less impressive in other respects. When unattainable moments are commanded from these particular methods, some of the allocated effectors are truncated due to position saturation. The moment produced as a result of this saturation is often different in magnitude and direction than the desired moment. Though correctable with proper scaling of the control effectors, this moment variation is an undesirable and unpredictable effect. In a similar sense, it is generally true that no generalized inverse solution can achieve the maximum attainable moment subset (AMS) without violating some control effector constraint. Therefore, generalized inverse solutions may not meet certain robustness requirements as successfully as other methods of allocation (e.g., direct allocation).

Seeking to solve the control allocation problem posed by modern aircraft, generalized inverse methods, specifically the pseudo-inverse and weighted pseudo-inverse methods, offer computational benefits over more advanced methods of allocation. They provide simple, closed-form solutions in a comparatively efficient manner.

However, because of their simplicity, generalized inverse methods sacrifice overall capability in comparison to other methods; the solutions provided are often limited due to control effector saturation, and as such will never achieve the maximum attainable moment.

The sections below introduce the standardized generalized inverse allocation algorithm used to solve the control allocation problem. Specific references will be made to both the pseudo-inverse and weighted pseudo-inverse solutions where appropriate. The resulting MATLAB® implementation will also be discussed in detail.

2.2.2 Solution Approach

Unlike the more computationally intense method of direct allocation, generalized inverse methods do not require determination of the attainable moment subset, referred to here as Π or Π_{GI}, prior to allocating controls. However, in the interest of the ensuing MATLAB® implementation, the procedure to completely determine Π_{GI} will be discussed as part of the algorithm. The underlying theory for the procedures presented herein is discussed in considerable detail in Ref. 1. As before, each paragraph with an *italicized* leading sentence represents a new step in the algorithm.

Construct the generalized inverse matrix, P, for the chosen inverse method. This is accomplished by applying equations similar to those given by (2.8) and (2.9). These particular equations construct the *P* matrix for the two methods of interest in this text; the pseudo-inverse and weighted pseudo-inverse methods, respectively. As illustrated, both of these methods require knowledge of the nonsingular control effectiveness matrix, *B*. The weighted pseudo-inverse method also requires knowledge of the *m*x*m* weighting matrix, often diagonal with entries described by (2.10). Regardless of the method chosen, the *P* matrix will have one row for each control effector in **u**, and one column for each body-axis moment (i.e., *P* will be a *m*x3 matrix).

Calculate Π_{GI} for the chosen generalized inverse method as the intersection of the 3-dimensional column space of P with the subset of admissible controls, Ω. To

accomplish this, P is partitioned into two sections. The first section, P_1, is a square 3x3 matrix representing 3 control effectors that will be fixed at either their minimum or maximum position limit. The second section, P_2, is an $(m\text{-}3)$x3 matrix representing the remaining controls. The control effectors in $\mathbf{u}$ are partitioned in a similar manner yielding $\mathbf{u_1}$ and $\mathbf{u_2}$.

The control effectors in $\mathbf{u_1}$ are set to each of the possible combinations of minimum and/or maximum deflection; there are eight possible combinations (i.e., 2^n combinations where $n = 3$ is the number of desired moments). Using the various combinations of $\mathbf{u_1}$, repeatedly solve for the corresponding values of $\mathbf{u_2}$ for all possible combinations of P_1 and P_2, by applying:

$$\mathbf{u}_2 = P_2 P_1^{-1} \mathbf{u}_1 \tag{2.11}$$

The resulting effector positions in $\mathbf{u_2}$ are compared with their respective deflection limits. Provided the effectors in question lie at or between the constraints, then the corresponding vertex of the admissible control subset, Ω, is also a vertex of its intersection with the pseudo-inverse solution (i.e., the column space of P). Each vertex determined in this manner is mapped to moment space using the linear transformation in (1.1), effectively defining the vertices that are in Π_{GI}.

The various facets of Π_{GI} now need to be defined. Every vertex determined by (2.11) has a base-3 object number associated with it. The object number defines a vertex by representing control effector position as an integer between 0 and 2; a 0 represents a control at its minimum constraint, 1 represents a control at its maximum constraint, and 2 represents a control that is free to vary between constraints. These object numbers are used to determine how the vertices that make up the attainable moment subset, Π, should be connected to define a facet of the generalized inverse solution. For the 3-dimensional problem considered here, two vertices are connected by an edge if they have at least two fixed controls in common, and lie on the same face if they have one or more fixed control

in common. This principle is applied to the vertices to effectively determine the facets that bound the three-dimensional subset, Π_{GI}.

Using the facet geometry, calculate the volume of Π_{GI}. This calculation is accomplished by dividing each facet of Π_{GI} (i.e., some polygon) into triangles. The sides of these triangles are represented as vectors in moment space emanating from the origin. These vectors form the rows of some 3x3 matrix, *A* used to calculate the total volume of Π_{GI}:

$$vol(\Pi_{GI}) = \frac{1}{3!}\sum_{i=1}^{k}\left|Det([A_i])\right| \tag{2.12}$$

where *k* is the total number of facets in Π_{GI}, and *3* is the overall dimension of Π_{GI} (i.e., if Π_{GI} where *n*-dimensional, then the leading term would be 1/*n*! instead).

Using the constructed generalized inverse matrix, P, and the set of desired moments, $\mathbf{m_d}$, allocate controls and calculate the set of attained moments. The allocation is accomplished by applying equation (2.7). The resulting vector of allocated controls is then multiplied by *B* to calculate the attained moments. When allocating with generalized inverse methods, there is the potential for control effector saturation. One way to account for this is to set any over-driven effector to the appropriate saturation value, and calculate the attained moment from the result; this is known as the truncated solution. Another popular approach scales the entire control vector, **u**, such that the effector with the worst violation is scaled down to its respective position limit (preserves moment direction); this is the scaled solution.

This concludes the algorithm for generalized inverse allocation. The generalized inverse matrix, *P*, was constructed based on the method chosen, the attainable moment subset, Π_{GI}, was determined and the associated volume computed, controls were allocated using (2.11), and the attained moments were calculated. The code that implements this algorithm in MATLAB® is discussed below.

2.2.3 MATLAB® Implementation

The generalized inverse algorithm presented above was implemented in MATLAB® for inclusion in the Control Allocation Toolbox. The code written for the implementation is provided in Appendix I, and will be referred to throughout this discussion. As before, there are certain sections of code that were not written until the implementation became part of the toolbox, and as such will be omitted from discussion until Chapter 3. There are four M-files required for implementation of the generalized inverse method, each is briefly described below in the order executed during implementation:

1) ***PSEUDO.M***	The main calling function; constructs the *P* matrix for the chosen method; assigns vertex object numbers; calculates both the truncated and scaled allocated control vectors; computes the attained moments.
2) ***OBJ_SRCH.M***	Determines vertex connections of *P*.
3) ***FSRCH1.M***	Orders vertex connections to determine the facets that bound *P*.
4) ***PVOLUME.M***	Calculates the volume of *P*.

It is evident from the descriptions above that the main implementation issue deals with determining the attainable moment subset, Π_{GI}. Since this is the most difficult part of the implementation, it warrants more discussion here than the less complex allocation sections of the implementation. The immediate discussion focuses on the sections of code that determine Π_{GI} and its associated volume.

To completely define the attainable moment subset, Π_{GI}, the intersection of the admissible control subset, Ω, and the generalized inverse solution must be determined. ***PSEUDO.M*** effectively accomplishes this task by constructing the appropriate *m*x3 generalized inverse matrix, *P*, based on equations (2.7) and (2.8). With this calculation

completed, ***PSEUDO.M*** begins an iterative process to determine the various vertices of Π_{GI}. During this process, the *P* matrix is partitioned into two sections; P_1 is a 3x3 matrix representing controls fixed at one or the other of their deflection limits, P_2 is an (*m*-3)x3 matrix representing the controls that are free to vary between constraints. The vector of control effectors is partitioned similarly into $\mathbf{u_1}$ and $\mathbf{u_2}$.

After appropriate partitioning, the iterative process continues with ***PSEUDO.M*** determining all the possible minimum/maximum control configurations of the effectors in $\mathbf{u_1}$. Each of the resulting configurations is a vertex of the constrained control subset, Ω, and is saved as a column in the matrix U1. This matrix is used to calculate U2 according to equation (2.11), and the resulting control positions are tested against their respective deflection limits. If any of the effectors are determined to be at or between their constraints, then the vertex of Ω (represented by a particular row of U2) is also a vertex of the desired intersection (i.e., of Π_{GI}). As such, a base-3 object number is assigned to the vertex for later use in determining the various facets of Π_{GI}. The process described above is repeated for every possible 3x3 partition of the *P* matrix. Thus, ***PSEUDO.M*** effectively determines the vertices of Π_{GI}. Note that ***PSEUDO.M*** constructs an object number matrix during this iterative process, whose rows represent the object number associated with each vertex of Π_{GI}; in the code of Appendix I this matrix is referred to as u_object.

Having defined the vertices, it remains to determine how they should be connected to effectively define a facet. The function ***OBJ_SRCH.M*** is used as a first step in this regard. This particular routine uses the base-3 object numbers that describe the vertices of Π_{GI} to determine whether they lie on the same face. As noted earlier, this relation is true only if the vertices in question have at least one fixed control in common. To implement this search, ***OBJ_SRCH.M*** uses the MATLAB® `find` command to search the columns of the object number matrix, u_object, for every instance of identical fixed controls (i.e., those columns with entries of 0 or 1). Should the `find` command discover instances of similarity in any one column vector, the appropriate indices are returned. These indices are essentially the vertices of Π_{GI} that lie on a common face. The resulting

vector of facet-defining vertex numbers is presented in numerically increasing order; undesirable for purposes of plotting, and/or calculating the volume of Π_{GI}.

To appropriately order the vertices such that the facets will be well defined, the edges of each facet (i.e., the vertex connections) need to be determined. In terms of object numbers, an edge is defined by two vertices that have two or more fixed controls in common. The function ***FSRCH1.M*** applies this principle to effectively define the facets of Π_{GI}. This routine initially takes the list of facets determined by OBJ_SRCH.M a row at time, and constructs an associated object number matrix using the vertex indices represented there. Using the `find` command, ***FSRCH1.M*** determines which columns of the resulting object number matrix contain fixed controls. If the `find` command uncovers any instances of similarity, the column indices are returned, and are saved to a list of indices. This list is essentially a record of the facet edges; the indices in the list represent facet-defining vertices that share two fixed controls. This procedure, as well as the one below, is repeated for every facet in the original list obtained from OBJ_SRCH.M.

Using the determined list of edges, ***FSRCH1.M*** attempts to connect them in such a way that will completely define the facet. The connections are determined by performing various searches on the list of indices (i.e., applying the `find` command). The result of these searches is a list of ordered indices that describe the desired edge connections. When applied to the original list of facets, the edge connections re-order the vertices such that Π_{GI} is represented by well-defined facets. Thus, the attainable moment subset, Π_{GI}, has been completely determined in 3-space.

To provide the comparative performance metric for the generalized inverse solution, ***PVOLUME.M*** computes the volume of Π_{GI}. Using an iterative process to account for every facet of Π_{GI}, this routine divides each facet into triangles. The triangles are determined from the previously-defined facet geometry, and as such are described by vectors emanating from the origin of moment space. Each triangle is used to construct the *A* matrix in equation (2.12) prior to applying the equation for volume calculation.

Successfully completing the discussion on Π_{GI}, the main purpose of generalized inverse methods (i.e., control allocation) may now be addressed. The main calling function, ***PSEUDO.M***, is responsible for this computation. Having constructed the generalized inverse *P* matrix earlier in the implementation, the relationship in (2.7) is applied. This relationship directly determines the vector of allocated controls; the implementation presented in ***PSEUDO.M*** accounts for the possibility of control effector saturation by calculating both the truncated and scaled solutions (discussed previously). The vector of attained moments corresponding to these solutions, $\mathbf{m}_{atn}$, is computed by applying the linear approximation of (1.1), re-written here as:

$$\mathbf{m}_{atn} = B\mathbf{u} \tag{2.13}$$

where **u** is either the truncated or scaled allocated control vector.

Thus ends the discussion on the MATLAB® implementation of generalized inverse allocation. The implementation as described above, is presented in Appendix I. The implementation effectively allocates controls according to the desired method, and accounts for the possibility of control saturation by providing multiple solutions. Using these solutions, the implementation calculates the vector of attained moments. Finally, although not an integral part of the allocation process, the implementation effectively determines the attainable moment subset, Π_{GI}, as the intersection of the admissible control subset, Ω, with the generalized inverse solution.

2.3 General Allocation

It is impractical to discuss, much less implement every possible control allocation method. This fact, in a sense, limits the overall scope of a comparative tool such as the MATLAB® Control Allocation Toolbox. To overcome such limitations, a general allocation method is introduced. Although not a specific method in itself, general allocation utilizes the output of any chosen allocation method to generate comparative

results (i.e., the attainable moment subset and associated volume). Thus, the various allocation methods not explicitly discussed in this text will be referred to as General Methods, and as such will be comparatively evaluated using general allocation.

To warrant comparison in the toolbox, the implementation of any of the various General Methods is expected to allocate controls for a suitably large number of specific demands with special consideration given to the control effector deflection limits. Accordingly, the moments attained are a subset of the attainable moments for the given General Method, referred to here as Π_{CH}. The graphical representation of Π_{CH} associated with any of the General Methods is approximated by fitting a 3-dimensional convex hull to the given set of points (i.e., the vertices of the attainable moment subset). It should be noted here, that since the convex hull algorithm generates the outermost boundary of the set of attainable moments, then if the actual Π_{CH} is not convex then the resulting approximation would be optimistic.

The sections below will highlight the theory of convex hulls, and present the hull-generating algorithm selected for the general allocation method. The implementation of this algorithm within the MATLAB® environment will also be examined.

2.3.1 Convex Hull Algorithm

The convex hull of a set of points is the smallest convex set that contains the whole of the original set. A fundamental application in the realm of mathematics and computational geometry, the convex hull is a constant subject of development. While there are many methods that can be easily applied to 2-dimensional problems, such as the gift-wrapping and divide/conquer techniques discussed in Ref. 6, there are very few that can efficiently be applied when considering higher-dimensional problems. Since the representation of the attainable moment subset in 3-dimensional moment space is the focus of this discussion, an efficient *n*-dimensional (i.e., $n > 2$) convex hull algorithm must be utilized. Such an *n*-dimensional algorithm was found to exist in the form of the Quickhull Algorithm for Convex Hulls. The specifics of the underlying theory for this particular algorithm are discussed in greater detail in Ref. 7, and are only highlighted here

to provide the basic knowledge required to understand the subsequent MATLAB® implementation.

The Quickhull Algorithm for Convex Hulls was written as a joint effort between the University of Minnesota Geometry Center and Princeton University. The foundation of the Quickhull algorithm is the *n*-dimensional Beneath-Beyond algorithm described in Ref. 7. Considered as a randomized incremental algorithm, the Beneath-Beyond method performs multiple searches to determine which facets are "visible" from a randomly chosen point in the original data set; "visible" in the sense of being beneath the point in question. As facets are added to the simplex (i.e., convex hull), previous constructions are saved for use in later searches. Hence, although relatively simple to implement, this family of algorithms is rather inefficient in terms of storage requirements and computational efficiency.

In an attempt to correct for the inefficiencies of typical randomized algorithms, the Quickhull algorithm uses the furthest point (i.e., in terms of distance from the origin) of the original data set as a basis for initial facet construction. As each facet is created, this algorithm automatically tests the vertices to ensure that they lie on the hull boundary prior to adding the facet to the simplex. Unlike the randomized algorithms which store previously constructed hulls, the Quickhull algorithm stores an unprocessed set of outside points for each facet on the current hull (i.e., the convex hull is updated each time a new point is processed) to be used in subsequent searches. According to Ref. 7, one final benefit of the Quickhull algorithm is that it handles floating-point operations with minimal error.

The Quickhull Algorithm for Convex Hulls, by efficiently computing the *n*-dimensional convex hull of a set of points, serves the purposes presented herein. The implementation of this particular algorithm has already been completed using the C programming language. The executable program, appropriately named Qhull, was written by C.B. Barber and H.T. Huhdanpaa of the University of Minnesota Geometry Center. The implementation of this already executable code in the MATLAB® environment is explained below.

2.3.2 MATLAB® Implementation

Prior to discussing the implementation issues associated with incorporating the executable Qhull program into the MATLAB® environment, its overall purpose should be understood. The original C code, consisting of seven subroutine modules, was downloaded from the web at http://www.geom.umn.edu. Though modified from its original form to make it platform-independent, the overall functionality of Qhull (i.e., the seven subroutine modules) has remained unchanged. Brief descriptions of each subroutine module are presented below:

1) ***qhull.c***	Contains all of the top level routines to initialize the calculation, incrementally construct a hull, and determine outside sets of points for each facet.
2) ***poly.c***	Responsible for creating the initial simplex from a list of vertices, and constructing facets during implementation.
3) ***geom.c***	Calculates distances to determine "furthest" point, and calculates facet normals.
4) ***mem.c***	Contains the memory management routines.
5) ***set.c***	Provides all the mathematical manipulation routines of adding, appending, and/or deleting points from a prescribed data set.
6) ***io.c***	Deals with the various input and output functions available in Qhull.
7) ***globals.c***	Declares all the global variables used by Qhull during implementation.

These routines describe the basic interactions during execution of Qhull. They will not be affected when Qhull is converted into a MATLAB® function.

To convert any C or Fortran program into a callable MATLAB® function, the MEX-file is introduced. These files are dynamically linked subroutines that MATLAB®

automatically loads and executes, effectively treating C and Fortran programs as "built-in" functions. Exact details on writing and creating callable MEX-files are presented with supporting examples in Ref. 8; they were very useful in creating the MEX-file version of Qhull .

Prior to creating the Qhull MEX-file, a main program was written to act as an application interface between the C compiler and the MATLAB® interpreter. In this respect, the main program contained a section of C code for the compiler to run, and an analogous section of MEX-file code for the interpreter to run. The main program written as this application interface, ***mat_qhull.c***, is included in Appendix I, and is briefly discussed here. The C section of the main program calls ***qhull.c*** (i.e., the top level routine in Qhull) with the set of points to be fitted with a convex hull, and the size of the double array containing the points. The corresponding MEX-file section initially defines the "mexFunction" to be called from MATLAB® with the appropriate number of input and output arguments. The MEX-file code proceeds to determine the dimensions of the input matrix, define the matrix as a MATLAB® variable, and call the newly created "mexFunction" (appropriately named ***mat_qhull*** after its C counterpart) to perform the desired convex hull calculations. Once complete, the MEX-file code translates the results such that they are displayed as a matrix in MATLAB®.

After creating and debugging the main program, the Qhull code was made into a MATLAB® MEX-file using the relatively simple process described in Ref. 8. The input to the new MATLAB® function is the set of moments obtained from an off-line implementation of some General Method, and the output is the ordered list of vertices defining the facets of the convex hull (i.e., the attainable moment subset, Π_{CH}). Having explained the MEX-file generation of Qhull, it now remains to explain the overall implementation of general allocation within MATLAB®. There are only two routines required for implementation, one to determine the convex hull and one to calculate the resulting hull volume. The code for each routine presented is provided in Appendix I. As before, the sections of code written explicitly for the graphical user interface will be omitted from discussion until Chapter 3.

The main calling function for general allocation is the M-file ***CVXAMS.M***. This routine essentially reads in a pre-determined *k*x3 set of data from a text file, where *k* is the number of moments obtained from the off-line implementation of some General Method. The resulting matrix is used as the input to the Qhull MEX-file ***convx***; it was renamed for purposes of notation. As established previously, ***convx*** returns the ordered list of facets to ***CVXAMS.M*** where it is sent to CVXVOL.M for volume calculation.

The function ***CVXVOL.M*** uses the facet geometry of Π_{CH} to calculate the associated hull volume. This is accomplished by applying the relationship in (2.12). The process is simplified somewhat in that, the facets determined by ***convx*** are already triangles. Thus, the facets do not have to be divided prior to applying equation (2.12).

This completes the discussion of implementing general allocation in the MATLAB® environment. The original Qhull code consisting of seven subroutine modules was modified to make it platform independent, and was then converted into a MATLAB® MEX-file to allow direct execution from within MATLAB®. The new MEX-file uses the set of moments obtained from some General Method to determine an ordered list of facets that define the attainable moment subset, Π_{CH} (i.e., the convex hull). Using the so determined facet geometry, the volume of Π_{CH} is calculated to provide the performance metric of the chosen General Method.

CHAPTER 3

Graphical User Interface Development

The computer technology available today has given impetus to the development of many computer-based design tools in areas such as computational fluid dynamics, trajectory analysis, and structural optimization. These tools utilize input from the designer to complete a specified task in an efficient, reliable manner; essentially acting as an interface between the human and computer counterparts. Whether distinctly answering a design question or presenting comparative results for interpretation, these computer-based design tools greatly simplify the otherwise tedious nature of most design procedures. Thus, the reliability, proficiency, and simplification afforded by these tools has made computer-based design an integral part of the overall design process.

Control allocation is a complex problem in the design of modern aircraft control systems; hence, a prime candidate for computer-based design tools. The design complexity results from the infinite number of allocation methods available to solve a given problem. Therefore, a design tool developed in this regard should attempt to provide the designer with a means of testing/comparing the validity of various allocation methods under prescribed conditions. Ideally, the design tool should provide both graphical and numerical results as to which method would be most effective in the system design. The sections below consider the various elements of developing such a tool in the MATLAB® environment. All developmental discussions assume a working knowledge of the MATLAB® programming language, and a basic knowledge of the MATLAB® GUI design tools introduced in subsequent sections of this text and discussed in instructive detail by Ref. 9.

3.1 Background

The development of a graphical user interface (GUI) is a very involved process, requiring substantial time and patience throughout. The overall development process consists of two parts; the GUI design and the GUI implementation. Ordinarily, the design phase is responsible for establishing an overall layout for the tool (i.e., window placement, color scheme), while the implementation phase is responsible for linking the various interface components with external code (i.e., making it operational). There are instances, however, when the design and implementation phases are used simultaneously to meet a desired goal or satisfy a specified requirement of the design tool (i.e., in the debugging process). Note that in discussing the development processes undertaken to construct the MATLAB® Control Allocation Toolbox, it is assumed that the design and implementation phases are completely independent. Though not the case during development, it is adopted here in the interest of clarity.

The remaining sections of this chapter attempt to provide the fundamental knowledge needed to successfully revise and/or append the design architecture of the MATLAB® Control Allocation Toolbox. The five MATLAB® GUI design tools used throughout the design phase of development will be introduced, and properties of the various GUI components (i.e., figure windows, pushbuttons, pop-up menus) utilized in the toolbox will be discussed. Expanding on these ideas, the design phase will be investigated with particular focus on the appearance and usefulness of the overall design architecture. The implementation phase of development is described in the final sections of this chapter with particular attention given to the allocation routines of Chapter 2. In this respect, the modifications made to the allocation routines will be highlighted, and the external routines written to make the toolbox operational will be examined. The external code referred to throughout these sections is included in Sections 4-5 of Appendix I. Any modifications made to the existing allocation routines are included in the appropriate

sections of Appendix I, and are presented using a stylized font to differentiate them from the original (i.e., algorithmic) sections of code.

3.2 MATLAB® GUI Tools

MATLAB® version 5.0 was chosen as the programming environment for this toolbox because it utilizes a set of GUI building tools that make the overall development process faster and easier than more conventional computer languages. There are five such tools embodied in the MATLAB® **G**raphical **U**ser **I**nterface **D**evelopment **E**nvironment (referred to as Guide); the Guide Control Panel, the Property Editor, the Callback Editor, the Alignment Tool, and the Menu Editor. Each of these tools is used extensively during the development process, and is introduced here in preparation for future discussion. The descriptions provided below are summaries of each design tool, and should not be taken as fundamental knowledge; for further discussion refer to Ref. 9.

3.2.1 Guide Control Panel

The Guide Control Panel, presented in Figure 3.2.1, is the principal tool in the MATLAB® GUI-building environment. This panel, opened by typing `guide` at the MATLAB® command prompt, is divided into three sections; Guide Tools, Guide-Controlled Figure List, and New Object Palette. The Guide Tools push-buttons are used to launch other GUI building tools such as the Property Editor or Callback Editor. The Guide-Controlled Figure List is used for activating and/or controlling the open figure windows. An active figure window is completely operational (i.e., working within its limits), whereas a controlled figure window is in an interactively editable state and must be activated prior to operation. Finally, the push-buttons in the New Object Palette allow the placement of new GUI controls (e.g., pop-up menus, text boxes) in any controlled figure window. Since it is constantly in the center of GUI development, the Guide Control Panel is used extensively during design and implementation.

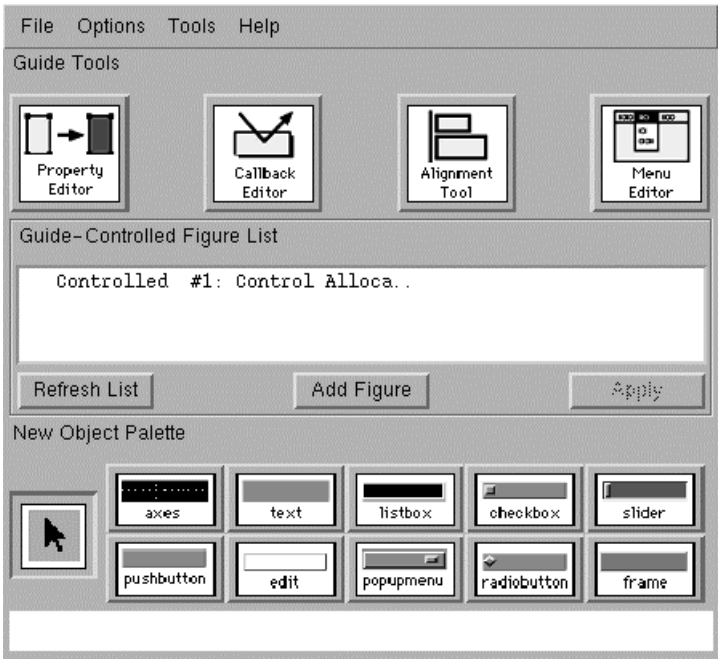

Figure 3.2.1 Guide Control Panel

3.2.2 Property Editor

Every object in MATLAB® has a unique set of properties associated with it, e.g., Position, Tag, Units. The Property Editor, shown in Figure 3.2.2, is used to manipulate these properties during the design phase. The names and types of various objects on a controlled figure are displayed in the Object Browser. Similarly, the complete set of properties associated with an object (i.e., the object highlighted in the Object Browser) are displayed in the Property List. The process of editing a property value involves the use of the Property Field and the Value Field. The chosen property name is entered in the Property Field such that its name is highlighted in the Property List, and its current value is automatically displayed in the Value Field. In most cases, the value is changed by highlighting the current one, and entering a new one in its place. If the property in question has a limited number of possible values associated with it, a pop-up menu appears in the Value Field and one of the available values may be selected as a replacement. Since the Property Editor is relatively easy to use (i.e., once the various

object properties are understood), it is an asset when changing properties of the layout architecture.

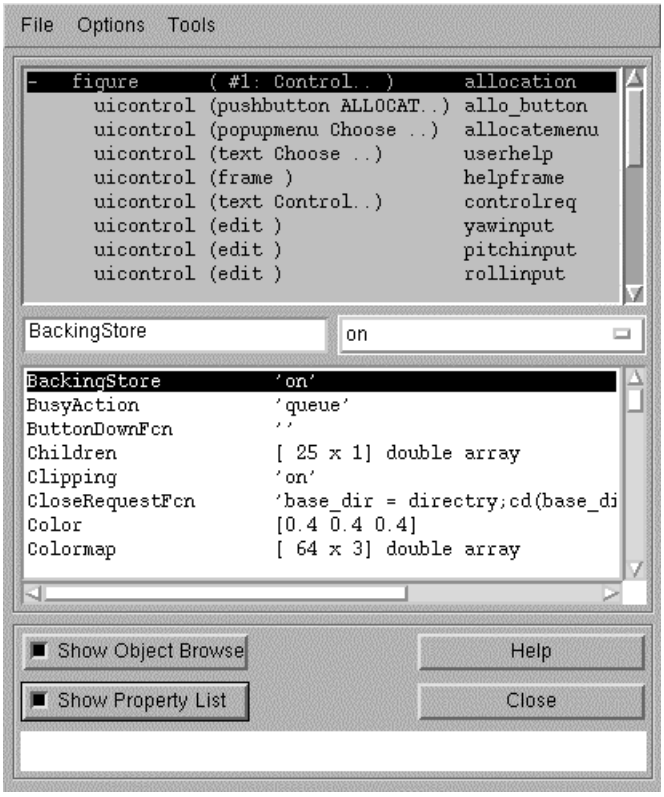

Figure 3.2.2 Guide Property Editor

3.2.3 Callback Editor

Callbacks are essentially M-file code that is executed each time a prescribed action is performed near, on, or over an object. For example, the callback for a push-button is executed when the button is pushed down, while that for a pop-up menu is executed when a valid selection is chosen from the menu list. The Callback Editor, presented in Figure 3.2.3, provides a convenient means of initially writing and then modifying these M-files. The Object Browser, as before, displays a list of the various objects used in a particular figure window (i.e., either active or controlled). The Edit Box in the center of the editor displays the Callback M-file and allows it to be modified. The

'Apply' push-button on the right side of the editor must be pushed before any code modifications take effect. Since it is utilized to write and modify the code that will eventually make the GUI operational, the Callback Editor is only used during the implementation phase of development. It should be noted that any M-files written in the Callback Editor will be internal to the GUI.

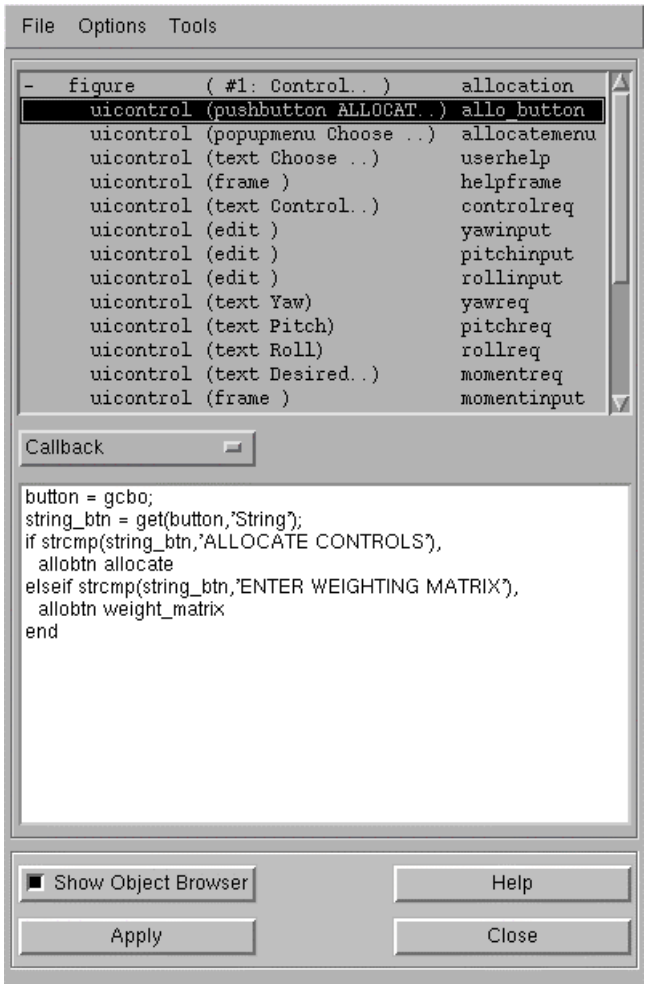

Figure 3.2.3 Guide Callback Editor

3.2.4 Alignment Tool

The Alignment Tool, depicted in Figure 3.2.4, is the fourth tool in the MATLAB® **G**raphical **U**ser **I**nterface **D**evelopment **E**nvironment. This particular tool allows multiple

objects to be positioned relative to one other. There are ten options available in the editor. These options are represented as push-buttons and include right/left horizontal alignment, horizontal/vertical centering, top-to-bottom distribution, right-to-left distribution, etc. The Alignment Tool is similar to both the Property Editor and the Callback Editor in that it utilizes an Object Browser for object selection. As such, objects are positioned by highlighting them in the Object Browser, and depressing the appropriate alignment/distribution push-button. If the alignment results are not acceptable, the Alignment Tool utilizes a 'Revert' push-button to return the selected objects to their previous positions. Since this particular tool is used to provide changes in figure window layout, it is solely used during the design phase of development.

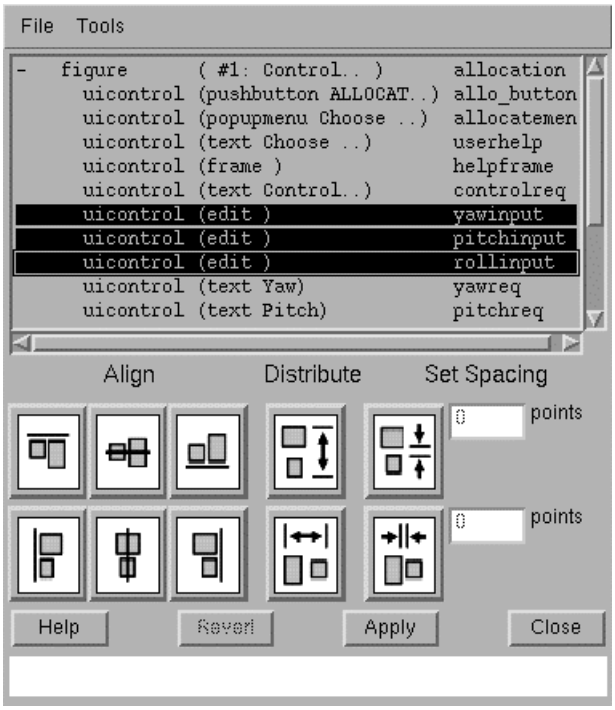

Figure 3.2.4 Guide Alignment Tool

3.2.5 Menu Editor

The Menu Editor is presented in Figure 3.2.5. This tool is used to add and/or edit "user-created" (i.e., not one of the basic system-level menus) pull-down menus during the implementation phase of the development process. The Menu Editor utilizes an Object Browser as the other tools, with the exception that it only displays the menus and associated submenus of a particular figure window. As evidenced in Figure 3.5, each menu has three defining properties; the Label, the Tag, and the Callback. The Label property assigns an external name to the menu that separates it from others when viewed inside a figure window:

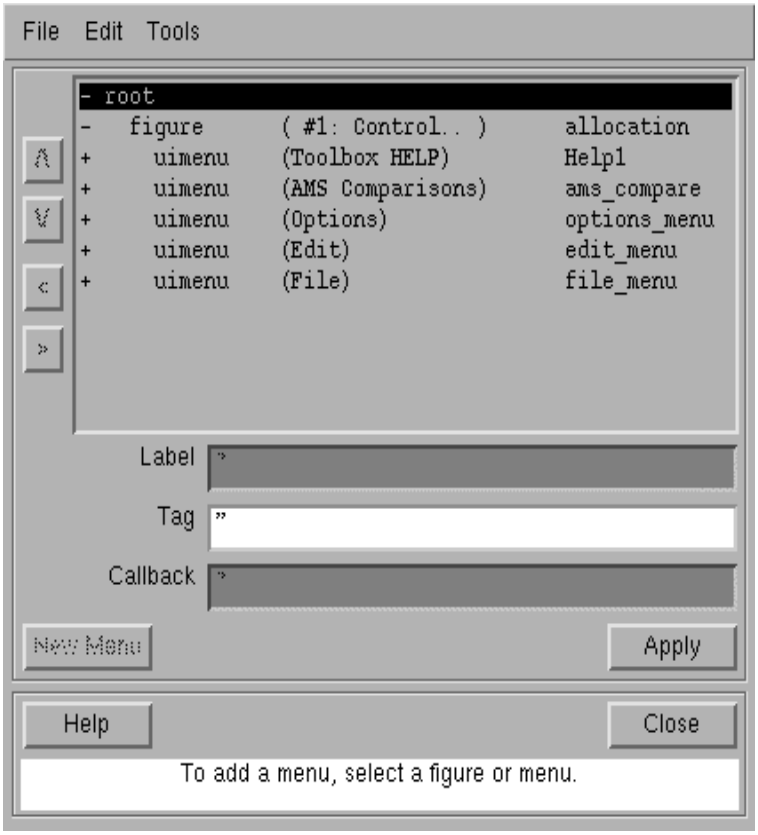

Figure 3.2.5 Guide Menu Editor

The Tag property assigns an internal name to the menu that the computer uses for location purposes during operation. The Callback property, as discussed earlier, is the M-file that is executed when that particular menu item is selected. The Menu Editor utilizes a 'New Menu' button to create a new menu, and four hierarchy buttons to indent and demote menu items as appropriate. Menu additions or changes are not visible in the figure window until the figure has been activated

3.3 GUI controls

Referred to as uicontrols in the MATLAB® programming environment, GUI controls provide the fundamental link between the human and computer counterparts in computer-based design. Each particular uicontrol performs a specified task and has a well-defined set of properties. The MATLAB® **G**raphical **U**ser **I**nterface **D**evelopment **E**nvironment provides ten such controls for use in any interface application including pop-up menus, push-buttons, and text boxes. The sections below introduce the seven uicontrols (i.e., excluding radio buttons, checkboxes, and sliders) implemented in the MATLAB® Control Allocation Toolbox. The purpose of each uicontrol and the most common of its associated defining properties will be highlighted.

3.3.1 Figure Windows

Though not explicitly considered one of the MATLAB® uicontrols, figure windows provide a framework for the other controls and consequently deserve some attention. The main function of the figure window is to effectively present the uicontrols, and provide an overview of the task to be accomplished in using them. The most common properties used to define the various figure windows in the MATLAB® Control Allocation Toolbox are presented in Table 3.3.1.

Table 3.3.1 Defining Properties of a Figure Window

Property	***Description***
Color	Sets the figure window background color
MenuBar	Turns system-level menus on/off
Name	Sets the name displayed in the title box
NumberTitle	Turns on/off the figure number in title box
Position	Sets figure position on computer monitor
Resize	Turns figure resizing option on/off
Tag	Internal name that defines the figure
Units	Determines how the figure is sized
UserData	MATLAB® variable for data storage

Note that the 'Color' property referred to in Table 3.3.1 and throughout these sections is most often described by RGB (red, green, blue) color triples; e.g., [1 1 1] represents white, [0 0 0] represents black, and [1 0 0] represents red. Each of the three numbers can be any decimal value from 0 to 1 such that different combinations yield colors outside the MATLAB® standard; e.g., the color triple [0.4 0 0] represents maroon, and [1 0.3 0] represents orange. The 'Position' property is measured from the bottom left corner of the monitor and is represented as the vector [x y width height]. Finally, the 'Units' property for figure windows and their objects is most commonly set to 'points' in the MATLAB® Control Allocation Toolbox; although, there are other options for this property such as 'normalized', 'pixels', and 'inches'.

3.3.2 Axes

The axes uicontrol is responsible for displaying either 2-D or 3-D plots within a figure window in the GUI environment. The axes object is appropriately positioned in a figure window by pushing the `axes` push-button on the Guide Control Panel, positioning the cross-hair cursor over the chosen figure window, and dragging the mouse until the plotbox is the desired size. The various properties associated with the axes uicontrol are presented in Table 3.3.2. Many of its standard properties can be set by using the regular MATLAB® plotting commands such as `hold`, `title`, `xlabel`, `ylabel`,

`view,` etc. Note that if a legend is added to a figure window to describe an axes object (i.e., applying the `legend` command), the resulting legend box is interpreted (by MATLAB®) as an axes object with a different set of properties.

Table 3.3.2 Common Axes Object Properties

Property	***Description***
Color	Sets the background color of the axes
NextPlot	Decides to replace/add to the current axes
Position	Sets the axes position in the figure window
Tag	Internal name that defines the set of axes
View	Sets the viewpoint of the plotbox
XGrid	Turns the x-axis grid lines on/of
YGrid	Turns the y-axis grid lines on/off
ZGrid	Turns the z-axis grid lines on/off
Units	Determines how the axes object is sized

3.3.3 Frames

The frame uicontrol is often used for grouping uicontrols that share a common purpose, e.g., a static text box labeling a pop-up menu. They are also used to enhance the overall appearance of a figure window by creating a layered look similar to that of a matted picture or photograph. Frames are placed in a figure window in the same manner as the axes uicontrol; by clicking the `frame` push-button and dragging the mouse over the desired window. Note that any frame uicontrol must be placed in a figure window prior to the uicontrols that it groups. Otherwise, the frame completely covers the other uicontrols rendering them useless. Table 3.3.3 presents the most commonly used defining properties for a frame. Since it is generally not an interactive uicontrol, the Callback property is not required for proper definition.

Table 3.3.3 Frame Object Properties

Property	***Description***
BackgroundColor	Sets the frame background color
Position	Sets the frame position in the figure window
Tag	Internal name that defines the frame
Units	Determines how the frame object is sized

3.3.4 Pop-up Menus

The pop-up menu uicontrol is used to group various methods that can only be individually applied. For example, in the MATLAB® Control Allocation Toolbox pop-up menus are used for listing the input methods and allocation methods. Pop-up menus are added to a chosen figure window by selecting the `popupmenu` push-button from the Guide Control Panel, and positioning it appropriately with the mouse. This particular uicontrol displays the group of items as a list that "pops up" when the mouse button is clicked over it. There are several required properties that define a pop-up menu; they are presented in Table 3.3.4. Note that all the pop-up menu properties (i.e., whether required or not) are editable using the Guide Property Editor. It is important to remember that this is an interactive uicontrol, and as such requires definition of the Callback property.

Table 3.3.4 Common Pop-up Menu Object Properties

Property	***Description***
BackgroundColor	Sets the pop-up menu background color
Callback	M-file executed when an item is chosen from pop-up
HorizontalAlignment	Sets string horizontal alignment to left/right/center
Max	Maximum number of items in the pop-up
Min	Minimum number of items in the pop-up
Position	Sets the pop-up menu position in the figure window
String	Concatenated string of items displayed in pop-up
Tag	Internal name that defines the pop-up menu
Units	Determines how the pop-up menu object is sized
Value	The item in 'String' to be displayed as a default

3.3.5 Push-buttons

Push-buttons are the most commonly used uicontrols in the MATLAB® Control Allocation Toolbox. This particular uicontrol usually executes a command that directly relates to the word(s) displayed on the button face. For example, a push-button with 'Save' on the button face would most likely execute a command or series of commands to save a file or some other information. Similarly, a push-button labeled 'Continue' might process commands to close unimportant figure windows and open those required for the next step of a procedure. Though similar to the frame uicontrol in that it has very few required properties, the push-button uicontrol differs in that it requires definition of the Callback property prior to implementation. The push-button uicontrols utilized in the previously mentioned toolbox have properties similar to those presented in Table 3.3.5. Be aware that although the push-button is usually assumed to be individually functional (i.e., executing a single callback to get a single result), there are circumstances under which the push-button may become dually functional. There is one such instance of this dual functionality highlighted in a later section of the text.

Table 3.3.5 Common Push-button Object Properties

Property	***Description***
BackgroundColor	Sets the push-button background color
Callback	M-file executed when push-button is pushed down
FontName	Type of font used for the 'String'
FontSize	Sets the size of the chosen font
FontWeight	Sets the font to regular or **bold**
ForegroundColor	Sets the color of the 'String' on the button face
Position	Sets the push-button position in the figure window
String	Becomes the name displayed on the button face
Tag	Internal name that defines the push-button
Units	Determines how the push-button object is sized

3.3.6 Text boxes

There are two types of text box uicontrols, the static text box and the editable text box. Both uicontrols have similar properties with one important exception; the 'String' displayed in the editable text box can be changed by the user during GUI operation. The static text box uicontrol is commonly used for labeling other types of uicontrols. In the MATLAB® Control Allocation Toolbox, for example, static text boxes are used to label the various figure windows, frames containing groups of similar uicontrols, and even editable text boxes. They are also used to display static text in message and/or help boxes. The editable text box, on the other hand, is an interactive uicontrol. It displays text in a similar manner to the static text box, but allows mouse and keyboard input to effectively change the text displayed. Since the properties for both types of text boxes are similar, they are displayed simultaneously in Table 3.3.6. The Guide Control Panel New Object Palette has one button for each text box (`text` for the static text box, `edit` for the editable text box), and that they are properly positioned in a figure window using the mouse.

Table 3.3.6 Some Text Box Object Properties

Property	***Description***
BackgroundColor	Sets the text box background color (usually white)
FontAngle	Used to make the font appear *italicized*
FontName	Type of font used for the text box 'String'
FontSize	Sets the size of the chosen font
FontWeight	Sets the font to regular or **bold**
ForegroundColor	Sets the color of the 'String' in the text box
Position	Sets the text box position in the figure window
String	Becomes the label/value displayed in the text box
Tag	Internal name that defines the text box
Units	Determines how the text box object is sized

3.4 Design Phase

The previous two sections presented the MATLAB® GUI design tools and the various GUI controls, both used extensively throughout the GUI development process. Discussion here focuses on the first phase of development, the design phase. It is during this phase that the functionality of the GUI is assessed, and the complete layout determined. These assessments form an iterative process in the sense that the overall functionality of any design tool is greatly dependent on the ability of the user to understand the design layout. Thus, the two seemingly independent parts of the design phase are instead interdependent. The discussions below describe the purpose of the MATLAB® Control Allocation Toolbox, and examine the procedures taken to determine the final design architecture. The various figure windows of the toolbox will be verbally introduced (there are graphical representations included in the User's Guide presented in Appendix II), and the important features of each window and associated uicontrols will be highlighted. The usefulness of the toolbox is then addressed with special attention given to the overall purpose of the toolbox.

3.4.1 Toolbox Purpose

The MATLAB® Control Allocation Toolbox is to be a relatively platform-independent, computer-based design tool that effectively addresses the problem of control allocation in the design of aircraft control systems. The toolbox is to bestow upon a designer the knowledge necessary to successfully test/compare the validity of certain methods under prescribed conditions. Both graphical and numerical comparisons are to be provided to assist in determining which of a variety of methods is most adroit in a given system. In providing these results, the toolbox will maintain a logical order of progression, and provide assistance when needed to ensure user comprehension and ability. Thus, the MATLAB® Control Allocation Toolbox is to be a user-friendly, reliable, comparative tool for use in the design of control allocation systems.

3.4.2 Relating Purpose to the Design Layout

With the purpose of the toolbox now defined, it remains to determine the layout architecture that will completely fulfill its aims. There are many factors that contribute to this determination including required user inputs, loading and/or saving necessary data during operation, making the various outputs easily understood by the user, and accounting for the possibility of user error. The layout discussed here seeks to account for all of these conditions while effectively accomplishing the purposes of the MATLAB® Control Allocation Toolbox. The various figure windows and uicontrols that describe this graphical user interface will be examined, with particular attention given to how each will be used after the addition of supporting code during the implementation phase of development. Note that each figure window discussed below has a MATLAB®-generated M-file associated with it that contains the defining properties for the window in question and any uicontrols placed upon it during the design process. These generated files are mentioned below, and are displayed in **`bold-faced Courier`** font to distinguish them from the other text. Examples of such MATLAB®-generated files are included in Section 4 of Appendix I.

The first figure window of interest in the MATLAB® Control Allocation Toolbox is the Introduction Page whose properties are defined in the file **`conallo.m`**. This page essentially acts as a welcome mat for the toolbox in that it displays the name of the design tool and provides an evident means of continuation. The figure window utilizes static text boxes and a push-button that were positioned and appropriately defined using the various Guide tools discussed previously. The static text boxes are used to display the toolbox title, the name of the toolbox designer, and the design location. The push-button, however, was designed with implementation in mind. Appropriately labeled ‘Press to Begin’, the push-button will eventually be responsible for determining the current working directory, initializing all of the appropriate temporary files, and officially launching the comparative pages of the toolbox.

Once the ‘Press to Begin’ push-button is used, the Introduction Page will close and the Data Input Page will appear in its place. The primary function of this page, inferred from the figure window title, is gathering input data for later calculations. To accomplish this, the figure window utilizes an editable text box, two static text boxes, two frames, and a pop-up menu. The first frame groups a descriptive static text box with an editable text box that will eventually request the number of controls (as keyboard input) from the user. The second frame groups a labeling static text box with a pop-up menu that will eventually be responsible for determining the method of input, and gathering data in accordance with the selected method. It was also decided that the Data Input Page should provide a means of editing and/or saving the gathered data. To this end, two push-buttons were positioned on the right side of the page; one labeled ‘View/Edit’, and the other ‘Save’. The ‘View/Edit’ push-button will be responsible for opening a second figure window that displays the gathered data. The ‘Save’ push-button will be responsible for determining an appropriate filename, and then writing the edited data to that file. There is one final push-button utilized on the Data Input Page, positioned just below the ‘Save’ button; the ‘Continue’ button. This push-button will close the Data Input Page, and launch the allocation figure window of the toolbox. The generated file containing the figure window and associated uicontrol properties is **`ingui.m`**.

The Edit Data Page is the figure window that will open after the ‘View/Edit’ push-button on the Data Input Page is pressed. This particular figure window is responsible for displaying the gathered input data in tabulated form. Hence, the main uicontrols utilized on this page are the static text box and the editable text box. The static text boxes will be used for applying text labels to the rows and columns of data where appropriate, while the editable text boxes will display the numerical input data and accept modifications to it. Aside from these uicontrols the Edit Data Page also utilizes a push-button; the ‘Done’ button. This push-button will eventually be responsible for saving the edited input data to a file prior to closing the containing figure window. Since the input data displayed by this page will be dependent on the number of controls input, there are

two files that describe the figure window and its uicontrol properties, **`editpg.m`** and **`editbtn.m`**. The first of these is a MATLAB®-generated file that contains the initial configuration of the figure window, whereas the second is an external M-file that was written to account for changes in window sizing and uicontrol properties that result from the variable number of controls.

Pressing the 'Continue' push-button from the data input figure window will close the Data Input Page, and launch the Control Allocation Page in its place. This particular page, whose properties are described by the file **`allopg.m`**, will essentially be the backbone of the toolbox, in that its primary purpose will be performing control allocation. The allocation figure window lists the available allocation methods in a pop-up menu that is appropriately labeled by a static text box; both are grouped by a frame. After the implementation phase, each pop-up menu selection will cause another frame, containing a set of three static text boxes and corresponding editable text boxes, to appear in the figure window. These editable text boxes will be used to determine the set of desired aircraft moments through keyboard input. The allocation figure window also utilizes a push-button that will serve two purposes during operation; it will allocate controls, and it will provide a means of editing the weighting matrix (i.e., a matrix required as an input to one of the allocation methods). When used for control allocation the push-button will be responsible for determining the method selected, checking that all inputs are available, allocating controls, and displaying the numerical results. When used for editing the weighting matrix, however, the push-button will be responsible for calculating a default matrix, opening the Edit Data Page figure window to display the matrix, and saving the edited matrix for later calculations.

Aside from its primary purpose, the Control Allocation Page will also be used to provide links to various other tasks that are either directly related to the results of control allocation, or to general toolbox operation. Pull-down menus will provide direct links to both the graphical and numerical results of the various allocation methods, and will provide the opportunity to print the numerical results to a file. Other links will provide a means of re-editing the input data, to display allocation matrices, and/or to end toolbox

operation. The various pull-down menus and their submenus are listed in Table 3.4.1. Although presented here to illustrate the capability that will be afforded by the Control Allocation Page, the pull-down menu options will not be examined until the implementation phase discussion.

Table 3.4.1 Pull-down Menus on the Control Allocation Page

Menu	***Submenus***
File	Start Over Quit
Edit	Edit Input Data Edit Weighting Matrix
Options	Display Numerical Results Print PI Matrices Print Numerical Results
AMS Comparisons	Direct/Pseudo-Inverse Direct/Weighted Pseudo Direct/General Allocation All Methods

As illustrated in Table 3.4.1, there will be four AMS comparison pages in the toolbox, each opened when the appropriate submenu item is chosen. The main function of each of these pages will be to provide a graphical comparison between at least two allocation methods, and then allow for manipulation of the results (i.e., changing viewpoints, scaling AMS size). There are various uicontrols utilized to serve these purposes. An axes uicontrol is used to display the graphical comparison, while pull-down menus are used to establish links to the various manipulation techniques. These links provide a means of plotting various moments resulting from allocation, allow for changes in the axes viewpoint, and allow for changes in the axes x- y- and z-axis limits. Once the implementation phase is complete, selecting one of the pull-down menu items will automatically cause an appropriately labeled frame with well-labeled editable text boxes

to appear in the desired figure window. Each frame appearing thus will use a push-button uicontrol to effectively calculate and display the requested result. Each comparison page will also utilize push-buttons to return the axes uicontrol to its original state should the axes properties change, or moments be plotted within the AMS. Finally, each of he first three comparison pages will present the comparison metric (i.e., the AMS volume) for the allocation methods compared by utilizing a frame and several static text boxes; some for labeling purposes, others for displaying the results. Each of the comparison pages has slightly different properties. The typical figure window and associated uicontrol properties associated with a comparison page are presented by the file **`dir_pi.m`**. In the interest of future discussions, note that the 'Visible' property for each of these pages is initially set to 'off' (i.e., when they are opened, they are not visible).

When 'Display Numerical Results' is chosen from the 'Options' pull-down menu on the Control Allocation Page (i.e., after the implementation phase of development), two separate pages will appear. The exclusive purpose of each is to display the numerical allocation results for all implemented methods; particularly the control effector positions after allocation, and the set of attained moments. These figure windows primarily use static text boxes for labeling the rows and columns of data, and editable text boxes for displaying the numerical results. There is an exception, however, in the figure window displaying the control effector positions; there are pop-up menu uicontrols utilized for the allocation methods that compute multiple solutions. These uicontrols will eventually be responsible for displaying the appropriate solution in both numerical output figure windows based on the item selected from the pop-up menu. The figure window and uicontrols for the allocated controls Output Page are described primarily in the external M-file **`num_res.m`** since its size and content will vary based on the number of controls; its base configuration is presented in **`allores.m`**. Similarly, the figure window and uicontrol properties for the attained moments Output Page are mainly presented in **`num_res.m`**, though the base configuration is provided in **`allmom.m`**.

In the interest of being user-friendly, each figure window discussed above has a help page associated with it. The various help pages are accessed through a pull-down

menu on the calling page appropriately labeled ‘Toolbox HELP’. The primary objective of these help pages is to offer user assistance in the absence of understanding. This is accomplished by providing page descriptions, suggesting procedures to follow, and/or explaining the various functions of uicontrols or uimenus associated with a given page. As such, the most common uicontrol utilized by the help figure windows is the static text box though push-buttons are used as well; they are either used to close the current help window, or to open an associated help window. There are ten generated M-files that contain the figure window and uicontrol properties for the Help Pages; a representative example is **`hlpallo.m`** included in Appendix I.

With the layout architecture of the toolbox determined, it remains to discover whether it fulfills the specified purposes. The first purpose is that the toolbox be user-friendly and follow a logical progression. Here the toolbox succeeds since, in most cases, there is only one logical step to take. For times when the progression is not so obvious there are Help Pages to offer guidance. The second purpose is that the toolbox provides reliable, comparative results for a variety of allocation methods. In this, too, the toolbox succeeds. It provides numerical and graphical comparisons for the three methods directly implemented, and it provides a means of graphically comparing any other allocation method by using general allocation as described in Chapter 2.

Thus concludes the design phase of the GUI development process. Each figure window in the MATLAB® Control Allocation Toolbox was introduced through verbal discussion. The important features and associated uicontrols were highlighted with clarification as to what they would be responsible for during toolbox operation (i.e., after completion of the development process). The overall usefulness of the resulting layout was evaluated based on the specific purposes discussed earlier. The desired layout was determined to be an acceptable foundation for the ensuing implementation phase of development.

3.5 Implementation Phase

The implementation of a graphical user interface most importantly requires fundamental knowledge of the subject being addressed. Secondary to this is the requirement that the overall design architecture be well understood. The fundamental knowledge for control allocation is presented in the discussions and algorithms of Chapter 2, and discussions on the overall design architecture were presented previously. Hence, the implementation phase of the GUI development process can be addressed. The discussions below explain the techniques applied to make the MATLAB® Control Allocation Toolbox an operational design tool. Of particular interest are the functions that MATLAB® uses for graphical user interface manipulation (i.e., editing figure window and uicontrol properties during operation); they are used extensively throughout the implementation process. The focus then becomes the modifications that were made to the existing allocation code, and the external M-file code that was written to effectively link all of the components.

3.5.1 Graphical User Interface Manipulation

Considering that the implementation phase of development is responsible for linking the many figure windows and uicontrols, it becomes necessary to be able to locate an object or change its properties during execution. There are several MATLAB® commands used exclusively for these purposes. Each command is briefly explained below following an introduction to MATLAB® Handle Graphics. For more detailed explanations about any of the commands presented here or in subsequent discussions, type ‘help’ at the command prompt followed by the command name.

To properly understand the commands presented in the following discussion, it is imperative to grasp the concept of MATLAB® Handle Graphics. This rather complex idea simplifies down to the fact that every graphical object represented in the MATLAB® environment has a unique identifier associated with it; the identifier is called a graphic handle. These handles are used with the commands below to locate an object, get its

current properties, and modify those properties during GUI operation. For more detailed information on the subject of Handle Graphics, consult Ref. 10.

Two MATLAB® Handle Graphics commands are used specifically to locate figure windows, `gcf` and `gcbf`. The `gcf` command (i.e. get current figure) determines the handle of the current figure window; that is, the window where the results of commands such as plot or `title` would appear. The `gcbf` (i.e., get callback figure) command, on the other hand, determines the handle of the figure window that contains the object whose callback is currently running. For example, if a push-button has been pressed such that its callback is executing, and its figure window needs to be located sometime during this execution, the `gcbf` command would be issued. Note that each time a new figure window is opened, it becomes the current figure and as such will be the handle returned by `gcf`.

To locate objects other than figure windows, the more generalized `findobj` command is used. Although this particular command has many forms, the one used throughout the implementation process discussed here is written as `h = findobj(figure, 'propertyname', value)` where `figure` is the handle of the figure window containing the desired object, `propertyname` is the name of the defining property, `value` is the current value of the defining property, and `h` is the handle returned. To ensure proper object location using this command, the most commonly used `propertyname` is 'Tag', and `value` is then the unique name given to that object during the design phase. Although generally not used for locating figure windows, the `findobj` command can be used in this regard by dropping the `figure` argument.

Once an object has been successfully located (i.e., its handle has been determined), its properties can be determined and/or modified. The two MATLAB® commands that accomplish this are appropriately named `get` and `set`. In its simplest form, `get(object)`, the `get` command is used to provide a listing of all the properties associated with an object and their current values. To determine the value of a specific

object property, the `get` command is appended such that it searches exclusively for that property; `get(object, 'propertyname')`. Similarly, the `set` command has multiple uses; it can be used to modify a single property or a set of properties; `set(object,'propertyname',value,'propertyname',value)`.

Thus concludes the introduction to MATLAB® Handle Graphics and the most commonly applied GUI manipulation techniques. The commands discussed will be used extensively throughout the implementation phase of development, and as such should be understood.

3.5.2 Control Allocation Modifications

The control allocation methods to be included as part of the MATLAB® Control Allocation Toolbox were discussed in Chapter 2. The algorithms and their MATLAB® implementations were highlighted during these discussions. Although the implementations presented in Chapter 2 accurately perform control allocation for a provided set of inputs, they do not provide the necessary allowances for inclusion in the toolbox. Thus, they must be modified prior to inclusion. Of the four allocation methods examined in Chapter 2, only the first three are directly implemented within MATLAB®; the fourth is a dynamically linked MEX-file. As such, the modifications to the directly implemented routines will be minimal in comparison to those for the MEX-file. Reiterated here is the fact that each allocation method implementation is provided in Appendix I. Any external M-files written for these modifications will be presented in **`bold-faced Courier`** to distinguish them from the built-in MATLAB® commands.

The most important modification made to each of the four methods provided a means for calculated results to be written to temporary data files at any point during execution. This particular modification utilizes two MATLAB® commands and an externally-written M-file; `fopen`, `fclose`, and **`filesav.m`**, respectively. The `fopen` command is utilized to open an appropriately named temporary file, and `fclose` is issued to close the file after data has been written to it. The external M-file, **`filesav.m`**, is responsible for writing the data to the temporary file. This routine

determines whether the data type is integer or real, and uses the MATLAB® `fprintf` command to write the data to a tab-delimited text file in the appropriate format. This modification is made in the main calling functions of each method; ATTAIN.M for direct allocation, PSEUDO.M for pseudo-inverse and weighted-pseudo inverse allocation, and CVX_AMS.M for general allocation. The temporary files written include those for the input data, the weighting matrix, the moments that make up each AMS, the facet geometry of each AMS, etc. Note that the temporary files are usually opened at the beginning of a routine, and are not written to until after the desired results have been calculated.

As an indirect result of the previous modification, it was decided that the working directory would need to be changed prior to saving or loading various data files. To allow for this without causing operational errors, an external M-file was written to determine the base MATLAB® directory. This particular file, **`directry.m`**, utilizes the `path` command to determine MATLAB®'s complete path, searches for the first instance of any combination involving the string 'matlab', and sets the base directory to this instance. To change the working directory from its base to some desired value, the MATLAB® `cd` command is applied. After the data file has been successfully loaded or saved, the `cd` command is used again to change the working directory back to its base value. Incorporating this modification effectively avoids operational errors since the working directory is always the base directory unless loading or saving data files. As such, it is applied to every routine that opens temporary data files, loads data from a file, or saves data to a file; specifically the main routine of each allocation method.

When operational, there will be times when data is stored in a toolbox figure window or other uicontrol object (i.e., utilizing the 'UserData' property). Such is the case for the required input data gathered by the Data Input Page. The input data is saved as a matrix in the 'UserData' property of the Control Allocation Page to be used with the allocation routines. As such, the BEGIN.M routine that was originally used to load the required inputs for each directly implemented allocation method was modified to allow for such an occurrence. In BEGIN.M, the `findobj` command is issued to locate the

allocation figure window, and the MATLAB® `get` command is applied to retrieve the matrix stored in 'UserData'. The resulting matrix is partitioned as appropriate into the required inputs for use by the allocation routines.

The remaining modifications pertain only to the general allocation routines discussed in Chapter 2. Since the actual implementation of the general allocation method is a dynamically linked MEX-file, direct modifications were not possible. Consequently, the modifications presented here were made to the general allocation routine, CVX_AMS.M. The most important modification made to this routine allows the user to choose which text file is to be evaluated. The MATLAB® `uigetfile` command is utilized to open a dialog box displaying a list of appropriate files. If a file is selected, the `fopen` and `fscanf` commands are issued to read in the data; otherwise MATLAB® generates a random matrix and uses the `msgbox` command to display a warning box to that effect.

Other modifications made to CVX_AMS.M provided a means of locating the appropriate graphical comparison figure window and its static text box uicontrols. These modifications were necessary so that the graphical and associated numerical results of the general allocation implementation could be presented. The desired graphical comparison figure window and uicontrols are located using the `findobj` command. Once the numerical results have been obtained from the implementation, they are displayed in the static text boxes by applying the `set` command with the static text box 'String' property. Displaying the graphical results requires the introduction of an external M-file, **`plt_cvx.m`**, and use of the `legend` command. As these apply directly to the implementation of the allocation and graphical comparison pages, they will not be discussed here, but will be addressed in the next section.

Though the modifications presented above were not extensive, they were required to ensure proper operation of the control allocation routines in the MATLAB® Control Allocation Toolbox. They provided a means for saving temporary data files during operation, allowed for the working directory to be changed without operational error, allowed the user to select a desired text file for general allocation, and provided a means

of displaying the numerical results obtained by general allocation. The sections below discuss the various external M-files written to link the components of the toolbox.

3.5.3 Making it Operational

The overall purpose of the implementation phase is to turn the layout architecture determined during the design phase into an operational tool. This is usually achieved by introducing computer code to the various objects that make up the layout. In MATLAB®, this computer code is written as M-files both internal (i.e., using an object's 'Callback' property) and external to the tool. This section will explain the external M-file routines written during the implementation phase of development. There are approximately twenty-six of these routines, most called from an internal Callback. To properly relate the routines to the objects they link, the layout architecture will be used as a guide. Thus, each page in the toolbox and the external routines it calls will be addressed according to the order in which they will appear on the screen after implementation. As before, the external routines will be presented below in **`bold-faced Courier`** print to distinguish them from built-in MATLAB® commands. Each external M-file referred to throughout the discussions is presented in Section 5 of Appendix I.

3.5.3.1 Introduction Page

As established during the design phase, the Introduction Page is essentially a welcome mat for the toolbox. Therefore, it is responsible for the initialization of all the temporary files utilized during toolbox operation. The external M-file **`open_tmp.m`** was written in this regard. Executed by the 'Press to Begin' push-button Callback, this routine essentially opens twelve temporary files in the '…\GUI\Temp_Out' directory using the external **`directry.m`** file with the `cd` and `fopen` commands. Once the temporary files are open, a default set of data (i.e., a 1x10 vector with elements of 1000) is written to them using the previously mentioned **`filesave.m`**. The files are closed using `fclose`, and the external routine returns to the push-button Callback which closes the Introduction Page and launches the Data Input Page.

3.5.3.2 Data Input Page

The primary objective of the Data Input Page is gathering the input data required by the three directly implemented allocation methods; the number of controls, the control effectiveness matrix, and the control position limits. The number of controls is input to an editable text box using the keyboard, and the control effectiveness and position limits are either input from a text file or randomly-generated by MATLAB®. The methods of input are selections in a pop-up menu whose Callback executes the external M-file **`userin.m`**. This routine first determines the handles of the text box and the pop-up menu using the `findobj` command. The routine uses the `get` command with these handles to determine the number of controls and the input method selected. Provided the number of controls was entered correctly, **`userin.m`** continues gathering data. If the input method chosen was 'Text File', the `uigetfile` command is issued to provide a list of possible files for selection. Once the desired file is chosen, the M-file opens and reads in the data using `fopen` and `fscanf`. If the input method was 'Random Generation', or if a file was not selected as described above, then **`userin.m`** creates a randomly-generated control effectiveness matrix using the `rand` command, and sets the minimum/ maximum control positions to -1/1, respectively. The required data is put into matrix form and written to a temporary file using **`filesave.m`**. It is also written to the 'UserData' property of the Data Input Page using the `set` command.

Once the required input data has been gathered, the Data Input Page provides a means of editing and/or saving it. There are two push-buttons utilized for this purpose, 'View/Edit' and 'Save', each calling the same external M-file **`buttons.m`**. This particular routine utilizes the MATLAB® `switch` command to determine which section of the external code is to be executed. In the case where 'View/Edit' is pressed, the **`buttons.m`** routine determines the input data gathered on the Data Input Page using the `get` command with the 'UserData' property. Provided the resulting matrix is not empty, the routine opens the Edit Data Page and calls another external M-file **`editbtn.m`** (discussed subsequently) to display the data. However, if the matrix determined using the

`get` command is empty, a warning box is displayed using the `warndlg` command and the routine returns to the push-button Callback. In the case where 'Save' is pressed and the input data matrix is not empty, **`buttons.m`** opens a "Save As" dialog box using the `uiputfile` command, and the input data is written to the appropriate file with `fprintf`. If the input data matrix is empty, a warning box similar to that for 'View/Edit' is displayed using the `warndlg` command.

The final objective of the Data Input Page is achieved by the 'Continue' button. This push-button executes the same external M-file as the other push-buttons, **`buttons.m`**, though it utilizes a different section of code. The section of interest ensures that the required input data is properly accounted for by using `get` with the 'UserData' property; if it is not accounted for, the `warndlg` command is issued to display a warning box with proper instruction. Provided the required input data is available, the M-file adds a small "delta" to each element of the control effectiveness matrix to guarantee that it remains nonsingular during allocation. Once this is complete, **`buttons.m`** opens the four graphical comparison pages (they are not visible when opened), and calls the direct and pseudo-inverse allocation routines to calculate the respective graphical AMS and associated volume. Using the `findobj` command the routine determines the static text boxes on each graphical comparison page that will be used to display the volume, and uses the `set` command with the 'String' property to display the calculated results. The routine also opens the two numerical output pages for later use (they are not visible when opened) by executing the external M-file **`num_res.m`**, which will be discussed in a subsequent section of this thesis. These tasks complete, **`buttons.m`** effectively closes the Data Input Page and opens the Control Allocation Page where it writes the input data matrix to the 'UserData' property with the `findobj` and `set` commands, respectively.

3.5.3.3 Edit Data Page - Input Data

The Edit Data Page is opened by the Callback of the 'View/Edit' push-button as described above. The same Callback also executes the external M-file, **`editbtn.m`,**

which is responsible for displaying the input data in tabulated format on the Edit Data Page. To accomplish this, the working directory is changed using the `cd` command to allow the temporary input data file to be loaded. Thus, the file is opened using the `fopen` command, and the data therein is assigned to a MATLAB® variable using `fscanf`. The task then becomes creating and appropriately positioning editable text boxes on the Edit Data Page. The editable text boxes are created by the `uicontrol` command called with a defining set of properties; those presented in Table 3.3.6. The 'Position' property in this set is defined by the desired row and column of a particular entry; a `for` loop is used to determine the column (i.e., horizontal) position. Similarly, the 'String' property for each box is set to the appropriate numerical value of an element in either the control effectiveness matrix or the position limit vectors. Note that to display a number in a text box, the `num2str` command must be issued prior to setting the 'String' property. **`Editbtn.m`** then positions the Edit Data Page on the screen according to the total number of controls being used.

Once the data has been successfully displayed, it can be edited using the mouse and numeric keypad. The mouse is double-clicked over the appropriate editable text box, and the new value is typed in using the numeric keypad. To ensure the changes made to the data take effect, the 'Done' push-button is pressed, executing the external M-file **`donebtn.m`**. This routine essentially locates one editable text box at a time using the `findobj` command with 'Position' as the `'propertyname'`, and the position defined for that particular box in **`editbtn.m`** as the `value`. Once located, the 'String' is read from the box using the `get` command, and is then converted to a number using `str2num`. The resulting number is saved as an element in the new input data matrix. When all of the editable text boxes have been located, and the values displayed have been saved as elements in the input data matrix, the matrix is written to the temporary file using the `fopen` command with **`filesav.m`**.

3.5.3.4 Control Allocation Page - Allocation

The two primary objectives of the Control Allocation Page are to successfully perform control allocation, and to provide direct links between the control allocation methods and their results. Since control allocation is the subject of the toolbox, the external routines required for its complete implementation will be examined first. Note that not every aspect of the external routines called from the Control Allocation Page uicontrols or uimenus will be discussed here. However, enough descriptive information will be provided to allow for the possibility of future revision.

To start the process of control allocation, one of the three directly implemented routines is selected from the pop-up menu on the Control Allocation Page. The pop-up menu Callback executes the external function **`method.m`**, which is essentially used to determine the allocation method chosen. Regardless of the method, this routine locates every uicontrol on the Control Allocation Page using the `findobj` command. It then uses the `set` command to turn on (i.e., sets the 'Visible' property to 'on') the frame and static/editable text boxes corresponding to the desired rolling, pitching, and yawing moments. With this complete, **`method.m`** uses the chosen method to determine whether the corresponding graphical comparison pages are open; if they are not, it opens them. Although seemingly method-independent, differences arise in **`method.m`** when weighted pseudo-inverse allocation is chosen. In this case, the routine loads a temporary data file that contains the weighting matrix (i.e., a required input for this particular method). If the data loaded is the original default matrix set up by **`open_tmp.m`**, then **`method.m`** changes the 'ALLOCATE CONTROLS' push-button to 'ENTER WEIGHTING MATRIX'; otherwise, the routine returns to the Callback awaiting selection of another method.

Once a particular method has been chosen from the pop-up menu, allocation is executed by pressing the 'ALLOCATE CONTROLS' push-button. The push-button Callback executes the external M-file, **`allobtn.m`**, which utilizes the MATLAB® `switch` command. This routine is responsible for determining the desired moments,

and allocating controls using the selected method. Regardless of method, the routine locates and determines the values displayed in the desired moment editable text boxes using the `findobj` and `get` commands, respectively. If no data is found, or if the desired moments are at the origin (0,0,0), an error box is displayed using the `errordlg` command. Otherwise, **`allobtn.m`** calls the appropriate allocation routine to allocate controls. The numerical results are sent to the appropriate output pages by calling the external M-file **`num_res.m`** discussed in a later section. For each method, **`allobtn.m`** determines if the appropriate graphical comparison pages are 'Visible' by applying the `findobj` and `get` commands, and comparing the results obtained using the `strcmp` command. If the pages are visible, the respective AMS is plotted, the volume results are displayed in the static text boxes when appropriate, and the legend command is updated to include the new method.

If the weighted pseudo-inverse method is chosen such that the push-button reads 'ENTER WEIGHTING MATRIX', a different section of **`allobtn.m`** is executed. This particular section of code uses the minimum/maximum position limit vectors to compute the default weighting matrix with diagonal entries computed according to (2.10). **`Allobtn.m`** then calls the external M-file **`wt_edit.m`**, discussed in a subsequent section, to display the weighting matrix.

3.5.3.5 Edit Data Page - Weighting Matrix

The Edit Data Page used in this instance is opened by the external M-file **`wt_edit.m`**. The primary objective of this function is to set up an appropriately labeled matrix of editable text boxes that display the entries of the default weighting matrix. The **`wt_edit.m`** routine determines the maximum height of the Edit Data Page based on the number of controls. When complete, the routine determines the vertical position of each static text box used for labeling, and each editable text box used for displaying the default weighting matrix entries. The horizontal position is determined in a similar manner since the weighting matrix is square. The static text boxes are created using the `uicontrol` command with a set of defining properties similar to those in Table 3.3.6. Similarly, the

editable text boxes are created with the 'String' property set to the appropriate numerical entry of the default weighting matrix. Finally, **`wt_edit.m`** positions the Edit Data Page on the screen using the number of controls to determine the final height and width.

To close the Edit Data Page after editing the weighting matrix, the 'Done' push-button is pressed. This action causes the push-button Callback to execute the external M-file **`donebtn.m`,** discussed previously. Although the M-file is the same as that called by the previous Edit Data Page 'Done' button, the section of code executed differs slightly. In this instance **`donebtn.m`** must return the 'ENTER WEIGHTING MATRIX' push-button on the Control Allocation Page to 'ALLOCATE CONTROLS'. Thus, the `findobj` command is used to determine the appropriate push-button handle so that the `set` command can be applied to change its 'String' property. Having changed the string on the button face, **`donebtn.m`** uses the `findobj` and `get` commands to determine each edited entry displayed on the Edit Data Page. The entries are put into matrix form, and the resulting matrix is written to a temporary data file using `fopen`, **`filesav.m`**, and `fclose`.

3.5.3.6 Control Allocation Page - Links to Output

As presented in Table 3.4.1, there are four main menus utilized by the Control Allocation Page, each with submenus. Some of these submenus provide links between the control allocation methods and their respective output, while others perform tasks related directly to the toolbox. Regardless, the submenus all have defined Callbacks. The external M-files executed by the submenu Callbacks are the current subject of discussion. Note that any internal functions of these Callbacks will be briefly highlighted for completeness.

The first menu of interest on the Control Allocation Page is the 'File' menu. Its purpose is to directly affect toolbox operation, evident from the two submenus 'Start Over' and 'Quit'. The Callback of each submenu executes the external M-file **`close_out.m`**. After determining the handles of all the graphical and numerical output pages with the `findobj` command, this routine utilizes the MATLAB® `switch`

command between the ‘close’ and ‘reset’ arguments. Regardless of the argument used, **`close_out.m`** applies the `set` command to change the ‘CloseReq’ property of each figure window. When called with the ‘close’ argument, the ‘CloseReq’ property is set to `closereq,` its default value; this effectively closes each page when the `close all hidden` command is issued. However, when **`close_out.m`** is called with the ‘reset’ argument, the ‘CloseReq’ property is set such that the figure windows become invisible (i.e., its ‘Visible’ property is `set` to ‘off’) when the `close all hidden` command is issued. Thus, the ‘Start Over’ submenu Callback executes **`close_out`** `close` (i.e., the command and argument) to change the ‘CloseReq’ property, issues the `close all hidden` command to close every open output window, executes **`open_tmp.m`** to re-initialize the temporary data files, calls **`close_out`** `reset` to reset the ‘CloseReq’ property, and launches the Data Input Page. Similarly, the ‘Quit’ submenu Callback applies **`close_out`** `close` followed by the `close all hidden` command, effectively returning to the MATLAB® command prompt.

The second menu on the Control Allocation Page is ‘Edit’ with the two submenus ‘Edit Input Data’ and ‘Edit Weighting Matrix’. Neither of the submenu Callbacks execute new external M-files; however, they both have individual internal M-files. The ‘Edit Input Data’ Callback essentially closes all the open figure windows using methods described previously (i.e., executing **`close_out.m`** followed by the `close all hidden` command), and opens the temporary input data file to read in the previous data. It then launches the Edit Data Page with the previous input data displayed, and re-initializes all the temporary data files. When the ‘Done’ button is pushed, **`donebtn.m`** opens the Data Input Page with the number of controls already displayed in the editable text box. The ‘Edit Weighting Matrix’ Callback erases any displayed data obtained using weighted pseudo-inverse allocation (i.e., graphical comparison pages, numerical output). It then opens the temporary weighting matrix file to load in the previous entries, and launches the Edit Data Page with the values appropriately displayed. When the ‘Done’ button is pressed, **`donebtn.m`** executes such that the new weighted pseudo-inverse

AMS is displayed on the appropriate graphical comparison page, and the edited weighting matrix is written to the temporary data file over the previous data.

The next menu of interest is the 'Options' menu, with three submenus. The first of the submenus is 'Display Numerical Results'. The Callback for this particular submenu utilizes the `findobj` command to locate each numerical output page, and `set` to change the 'Visible' property to 'on'. Prior to execution of the 'Display Numerical Results' submenu Callback, the 'Continue' button on the Data Input Page called the external M-file **`num_res.m`**. Mentioned briefly in earlier discussions, this routine is initially used to open and set up the allocated control effector Numerical Output Page. The `uicontrol` command is applied with appropriate defining properties (i.e., 'Position', 'Type', 'BackgroundColor'), to create labeling static text boxes, displaying static text boxes, and pop-up menus for allocation methods with multiple solutions. Each uicontrol created in this regard has a unique 'Tag' associated with it to ease the process of locating and displaying results later. When **`num_res.m`** is called a second time (i.e., from the external M-file **`allobtn.m`**) it determines which allocation method was used for calculations, and uses the `findobj` command to locate the corresponding static text boxes on both Numerical Output Pages. The `set` command is then applied to display the numerical values achieved by the allocation methods; the allocated control effector positions or the attained moments. The **`num_res.m`** routine also allows for the case when one of the desired moments on the Control Allocation Page is changed from its initial value. If this occurs, the routine uses the `findobj` and `set` commands to effectively gray-out (i.e., set the 'BackgroundColor' to [0.7 0.7 0.7]) any static text boxes that do not correspond to the allocation method used for calculation.

The pop-up menus utilized on the allocated controls Numerical Output Page execute the external M-file **`trunc_sc.m`**. This particular M-file is responsible for displaying either the 'Truncated' or 'Scaled' solution to the pseudo-inverse and/or weighted pseudo-inverse allocation methods. The **`trunc_sc.m`** routine determines the method being used, and uses the `findobj` command to locate the appropriate pop-up

menu. Using the obtained graphic handle, **`trunc_sc.m`** applies the `get` command with the 'Value' property to determine the solution to be displayed. The `findobj` command is issued to locate the appropriate static text boxes on the Numerical Output Pages, and the calculated solutions are displayed by applying the `set` command with the 'String' property. Note that prior to displaying the results, **`trunc_sc.m`** converts the numerical solutions to strings using the `num2str` command.

The second submenu of the 'Options' menu is 'Print PI Matrices to Command Window'. The Callback for this submenu executes the external M-file **`disp_mtrx.m`**. This routine utilizes the `switch` function between the arguments 'regular' and 'weighted' to print the appropriate generalized inverse matrices to the MATLAB® command window. Thus, **`disp_mtrx.m`** opens the temporary data file corresponding to the chosen generalized inverse matrix using the `fopen` command, and assigns the data to a variable using `fscanf`. If the data loaded is not the initial default data assigned by **`open_tmp.m`,** the variable is sent to the command window by calling its variable name without using a semi-colon ';' to suppress the output.

The final submenu of the 'Options' menu is 'Print Numerical Results'. This particular submenu has its own submenus; at least one for each allocation method directly implemented in the toolbox. The external M-file executed by each submenu Callback is **`pr_btn.m`**. This routine is responsible for writing the numerical allocation results for each method (i.e., the control position after allocation, and the attained moments) to a user-specified file. Thus, the `uiputfile` command is issued to open a "Save As" dialog box. Once the filename has been entered, the `cd` command is used to change the working directory as appropriate. The **`pr_btn.m`** routine then determines which allocation submenu has been selected, and uses the `fprintf` command to write the numerical results to the specified file.

The final menu on the Control Allocation Page is 'AMS Comparisons'. This particular menu has submenus that open the four graphical comparison pages, effectively displaying the resulting AMS and volume information. The 'Direct/Pseudo-Inverse'

submenu Callback uses the `findobj` command to determine whether Comparison Page #1 is open. If the window is open, the submenu Callback uses the `set` command with the 'Visible' property to make the window visible. If the window is not open, the Callback opens the window and executes the two external M-files **`direct_ams.m`** and **`psplot.m`**. The **`direct_ams.m`** routine uses the `fopen` command and the external M-file **`filesav.m`** to load the temporary data files containing the direct allocation AMS facet geometry. Provided the data loaded is not the default data set by **`open_tmp.m`**, the routine proceeds to plot the AMS. The **`psplot.m`** routine performs similar determinations and calculations using the temporary facet_pi and pi_moment data files.

Similarly, the 'Direct/Weighted Pseudo' submenu Callback determines whether Comparison Page #2 is open.. If the page is not open or the weighted pseudo-inverse method has not been utilized before, the page is opened and the external M-file **`allobtn.m`** is executed to calculate the weighting matrix and display it on the Edit Data Page. When the 'Done' button on the Edit Data Page is pressed, the external M-file **`donebtn.m`** is executed. For the case considered here, the weighted pseudo-inverse matrix is saved to a temporary data file using `fopen` and **`filesav.m`**, and the direct allocation AMS is plotted on the appropriate comparison page by executing the **`direct_ams.m`** routine. The weighted pseudo-inverse AMS is calculated and displayed by calling the **`psplot.m`** routine. If the page is already open, the submenu Callback simply executes **`direct_ams.m`** and **`psplot.m`** to plot the direct allocation AMS and the weighted pseudo-inverse AMS, respectively. Note that in the case of the weighted pseudo-inverse, the **`psplot.m`** routine uses the temporary files facet_wpi and wpi_moment for plotting the AMS.

The third submenu of the 'AMS Comparison' menu is 'Direct/General Allocation'. This particular submenu Callback uses the `findobj` command to locate the graphic handle of Comparison Page #3, and uses the `set` command to make it 'Visible'. It then calls the main general allocation routine CVX_AMS.M. As described in Chapter 2, this M-file calls the convex hull generating MEX-file to determine the facet

geometry of the general allocation solution. Prior to this, however, CVX_AMS.M uses the `uigetfile` command to display a list of text files for evaluation. If a file is selected, it is opened using `fopen`, and assigned to a variable using `fscanf`. If 'Cancel' is selected instead, MATLAB® randomly generates a data set for use with the convex hull code. Once the facet geometry of the convex hull has been determined and saved to a temporary data file using `fopen` and **`filesav.m`**, CVX_AMS.M executes the external M-files **`direct_ams.m`** and **`plt_cvx.m`**. The **`plt_cvx.m`** routine opens and loads the temporary data files that contain the facet geometry and the set of moments used to determine the convex hull. The **`plt_cvx.m`** routine plots the convex hull on the appropriate comparison page within the direct allocation AMS.

The 'All Methods' submenu is used to plot the AMS for each method that has already been evaluated by the toolbox. Each of the plotting routines described above is executed in turn; **`direct_ams.m`**, **`psplot.m`**, CVX_AMS.M with **`plt_cvx.m`**. To differentiate between the four methods presented, a legend is utilized by applying the `legend` command with a vector of colors, and a vector of strings that correspond to the allocation methods. The color and string to be displayed for each method is determined in the respective plotting routine. The color and string vectors are appended as different allocation methods are evaluated. The resulting legend is positioned after creation by using the `findobj` command to locate the legend, and applying the `set` command with the 'Position' property.

Thus concludes the discussion of the various links provided by the four menus of the Control Allocation Page. Each menu and its submenus were introduced, and the appropriate Callbacks were examined with particular attention given to the external M-file routines that were written to complete the implementation of the Control Allocation Page.

3.5.3.7 Graphical Comparison Pages

The four graphical Comparison Pages appear by selecting them, in turn, from the 'AMS Comparison Page' menu on the Control Allocation Page. Once the AMS for each

method has been plotted, the Comparison Pages provide various techniques for manipulating the data. Since each Comparison Page is similar in the techniques employed, only the Direct/Pseudo-Inverse Comparison Page will be discussed.

Most of the data manipulation techniques are available from the 'Options' pull-down menu. The first of these options is 'Plot Moments' with two submenus; one for 'Desired/Attained' moments, the other for 'Facet-defining' moments. The 'Desired/Attained' option is only available if controls have been allocated for the chosen method. The submenu Callback executes the external M-file command, **`plt_atn.m`**, which is used to plot the desired moments against those attained by a particular allocation method. The **`plt_atn.m`** routine opens and loads all of the appropriate temporary data files using `fopen` and `fscanf`, and determines the set of desired moments from the allocation page using the `findobj` and `get` commands. In a similar manner, the routine then locates and determines the set of attained moments from the moments attained Numerical Output Page. Using this information, **`plt_atn.m`** redraws the direct allocation AMS, and plots both the desired and attained moments inside as different-colored vectors from the origin. When complete, **`plt_atn.m`** locates the 'Revert' push-button using the `findobj` command, and makes it 'Visible' using the `set` command. When the 'Revert' button is pressed, its Callback property executes the external M-file **`rvrt_btn.m`** to redraw each AMS in its original form. The `cla` command is used to clear the axes uicontrol, and the appropriate plotting functions (i.e., **`direct_ams.m`** and **`psplot.m`** in this case) are called to redraw each AMS. The 'Revert' push-button is then made invisible by performing the inverse of the previous `set` command.

The second submenu of the 'Plot Moments' option is 'Facet-defining'. For the Direct/Pseudo-Inverse Comparison Page, this is only available for the pseudo-inverse solution. The submenu Callback executes the external M-file **`plot_fac.m`,** a routine responsible for redrawing the direct allocation AMS, and plotting the facet-defining moments for some other allocation solution inside. This is accomplished by first opening and loading the appropriate temporary data files (i.e., those containing direct allocation

facet geometry, and the moments making up the "other method" AMS). Using the data, **`plot_fac.m`** redraws the direct allocation AMS, and plots the facet-defining moments inside as vectors from the origin (0,0,0). As before, **`plot_fac.m`** locates the 'Revert' push-button and makes it visible by using the `set` command.

The second option from the 'Options' menu is 'Change Viewpoint' with the two submenus 'X,Y,Z Coordinates' and 'Azimuth/Elevation'. Both of these methods are responsible for changing the viewpoint of the AMS axes according to some user-defined position. Each of the Callbacks executes the external M-file **`toolmnu.m`**, which utilizes the `switch` command between the arguments 'xyz' and 'azimuth'. When 'xyz' is the chosen argument, a frame and corresponding static/editable text boxes appear requesting that the user enter the x-, y-, and z-components of the desired viewpoint. Similarly, when 'azimuth' is the chosen argument, a frame and corresponding text boxes appear requesting that the desired azimuth and elevation angles be entered.

Each of the frames determined in this manner utilize an 'OK' push-button. When pressed, the push-button Callback executes the external M-file **`okbutn.m`**. This particular function also uses the `switch` command with the calling arguments 'xyz' and 'azimuth'. Using 'xyz', **`okbutn.m`** locates the appropriate editable text boxes using the `findobj` command, and determines the desired numerical components by applying the `get` and `str2num` commands, respectively. The routine then transforms these components into orientation angles using the `cart2sph` command. Once the orientation angles have been computed, **`okbutn.m`** determines the desired set of axes using the `gca` command, and changes the viewpoint using the `set` command with the 'View' property. A similar approach is used by **`okbutn.m`** for the 'azimuth' argument except that the values obtained from the editable text boxes using the `findobj` and `get` commands are the desired orientation angles. Thus, they are used directly in the `set` command with the 'View' property without the need for conversion. Regardless of the calling argument to **`okbutn.m`**, the frames, text boxes, and push-buttons used for gathering data are made invisible during execution. However, the routine also locates a

'Refresh' push-button and makes it visible once the viewpoint is changed. The Callback of this particular push-button sets the viewpoint of the current axes to the default value of [-37.5 30] by applying the `set` command with the 'View' property.

The final option of the 'Options' menu is 'Zoom'. This submenu is responsible for determining a desired scale factor, and scaling the axis limits accordingly. The submenu Callback again executes the external M-file **`toolmnu.m`**, but with the 'zoom' calling argument. As a result, a frame, a push-button, and a static/editable text box appear requesting that the user enter a scale factor. When the 'OK' push-button is pressed, the Callback executes the external M-file **`okbutn.m`** with the calling argument 'zoom'. For the 'zoom' calling argument, the routine finds the current axes uicontrol using the `gca` command, and determines the three axis limits using the `get` command with the 'XLim', 'YLim', and 'ZLim' properties. The desired scale factor is determined by using the `findobj` and `get` commands, respectively. **`Okbutn.m`** uses this scale factor to scale the previously determined axis limits. The new limits are used with the `gca` and `set` commands to resize the current axes uicontrol.

The axes uicontrol on each graphical Comparison Page utilizes its 'ButtonDownFcn' such that when the left mouse button is clicked over the axes, the external M-file **`az_elev.m`** is executed. This particular routine allows for multiple viewpoint changes by simply pointing and clicking with the mouse. The **`az_elev.m`** routine uses the axes uicontrol 'CurrentPoint' property to determine the x-, y-, and z-components of the point clicked on by the mouse. These components are transformed to orientation angles using the `cart2sph` command, and are used with the `gca` and `set` commands to change the axes 'View' property. When **`az_elev.m`** is executed, the 'Refresh' push-button is located and made visible as before.

Thus concludes the discussion of the various data manipulation techniques presented by the graphical comparison pages. The AMS and allocation results can be manipulated through changes in viewpoint, changes in size, and by plotting various moments inside the direct allocation AMS.

The implementation phase of development has been completely addressed through the explanations and descriptions offered by the previous sections. The layout architecture determined during the design phase was used as a framework for presenting the M-files that make the toolbox operational. The control allocation methods were successfully included in toolbox operation, and the various methods of comparison were accounted for. Thus, successful implementation of a graphical user interface is reliant on knowledge; fundamental knowledge of the subject being addressed, and a general understanding of the layout determined during the design phase.

CHAPTER 4

Problems Encountered

Unexpected problems tend to occur during the development of any new product. Regardless of the care taken to avoid them, they are usually responsible for adding undesired time and complexity to the development process. The task of designing and implementing a computer-based design tool is no exception to this otherwise unwritten rule. In the area of computer-based design, most unexpected problems are generally attributed to coding mistakes, or are considered a direct result of misinterpreting the motivations behind development. The discussions presented below demonstrate that this is not always the case. The purpose of this chapter is to present the major design and implementation issues associated with the development of a computer-based tool in the MATLAB® environment.

4.1 Platform-independence

The primary consideration in choosing a programming language for the Control Allocation Toolbox was that it provides for the possibility of platform-independence. Secondary to this was the requirement that the chosen language also support three-dimensional graphics. Since it affords the desired platform and graphics abilities, the MATLAB® programming language was chosen over a more ANSI standard language such as C or C^{++}. Thus, the development of the MATLAB® Control Allocation Toolbox began.

During the design phase of GUI development it was discovered that although MATLAB® M-files are platform-independent, its representation of figure windows and

uicontrols is not. The design phase discussed in Chapter 3 was initially performed on a 32-bit UNIX workstation; hence, the figure windows and uicontrols were positioned accordingly. When the resulting layout was examined on a PC and Macintosh PPC (i.e., to validate platform-independence), the figure windows and uicontrols were visibly misrepresented. From this, it was learned that the 'Units' property of the figure window was set to 'pixels' while that for the uicontrols was 'points'. Thus, to hopefully correct the problem, the 'Units' property of each figure window was modified to match that of the uicontrols (i.e., it was set to 'points'). Though adjusting this property value did improve the previous viewing problems, it did not fully correct them. The only way to fully account for the various visual representations was to modify the original UNIX layout for both the Macintosh and PC platforms. Consequently, the MATLAB® Control Allocation Toolbox is not a truly platform-independent design tool.

During the implementation phase of development, the four control allocation methods to be evaluated in the toolbox were implemented in MATLAB®; described previously in Chapters 2 and 3. Of these, the general allocation method utilizes a dynamically linked subroutine referred to as a MEX-file. Unfortunately, the MEX-file is another platform-dependent part of the MATLAB® programming environment. Thus, three MEX-files had to be created to ensure proper operation in the toolbox; a 32-bit UNIX file with the `mexsg` extension, a Macintosh PowerPC file with the `mex` extension, and a PC file with the `dll` extension.

4.2 MEX-file Generation

With the introduction of the general allocation option to the toolbox, it became necessary to utilize a MATLAB® MEX-file. As established, this particular type of file is platform dependent and as such added complications to the process of implementing it in the MATLAB® Control Allocation Toolbox. The complexity originates from the fact that each platform supports different Fortran and C compilers (i.e., those supported for MEX-

file generation), and some of the compilers may not be those supported by the current version of MATLAB®.

Such a complication was encountered during MEX-file generation for the Macintosh version of the toolbox. The MATLAB® and C compiler versions being used were MATLAB 5.0 and MetroWerks Code Warrior 7, respectively. After much time and frustration, it was concluded that the two versions being used were incompatible for MEX-file generation. It was later discovered that only MetroWerks Code Warrior versions 8, 9, and 10 are supported for MEX-file generation in MATLAB® 5.0. Therefore, a new compiler was procured, and the general allocation MEX-file was successfully generated.

The problems encountered during the design and implementation of the MATLAB® Control Allocation Toolbox are not those commonly associated with computer tool development. The problems discussed resulted directly from the limitations of the MATLAB® programming language. Thus, the desire for a platform-independent computer tool was compromised. Therefore, though operational on 32-bit UNIX, Macintosh, and PC platforms, the resulting design tool is not definitively platform-independent.

CHAPTER 5

Conclusions

The multiple, redundant control effectors utilized on modern aircraft inherently increase the complexity of control system design. This complexity results from the infinite number of possible solutions to the control allocation problem. Simply put, the control allocation problem is one of determining how the control effectors of a given system should be positioned to achieve some specific objective. Of interest here, was the determination of control effector positions that would attain the set of desired moments associated with some aircraft maneuver.

The focus of this research was aimed at developing a user-friendly, platform-independent design tool to successfully address the problem of control allocation. MATLAB 5.0 was chosen as the programming environment since it was initially believed to provide a platform-independent method of designing and implementing a graphical user interface. Three allocation methods were directly implemented in MATLAB® for inclusion in the toolbox; direct allocation, pseudo-inverse allocation, and weighted pseudo-inverse allocation. A fourth allocation option was devised and added (i.e., general allocation) to provide a means of comparing off-line implementations of other methods. The GUI development process was undertaken in two phases, design followed by implementation. The design phase was responsible for establishing the overall layout architecture, while the implementation phase effectively connected the otherwise independent parts of the interface. The end result was the operational MATLAB® Control Allocation Toolbox.

The GUI design tools available in MATLAB® made it an ideal choice for creating a graphical user interface. The tools provided a straightforward approach to both the design and implementation phases of development. During the design phase, they

allowed direct manipulation of the GUI appearance and presented a graphical representation of the resulting layout without having to compile and execute any code. In fact, MATLAB® generated the code for the entire layout. During the implementation phase, the design tools provided a method of independently programming each GUI control in the toolbox. Therefore, MATLAB® provided a simplistic, efficient means of developing a computer-based design tool.

The only disadvantage of choosing MATLAB® as the programming language was the fact that the resulting computer-based tool could not completely satisfy the requirement of platform-independence. Although the code written to implement the graphical user interface was independent, the graphical representation of the resulting layout was not. The introduction of a MEX-file to the GUI code (i.e., to implement the general allocation option of the toolbox) was yet another strike against the possibility of platform-independence since the file type is machine dependent.

The resulting MATLAB® Control Allocation Toolbox addresses the problem of control allocation by providing graphical and numerical results for a variety of control allocation algorithms. To compare the allocation methods, the toolbox employs the use of the comparison metric determined in Chapter 2; the volume of the attainable moment subset. The performance metric effectively determines which method provides the "optimal" solution to a given allocation problem; of the methods discussed, direct allocation was determined to be "optimal". Finally, to satisfy the requirement of user-friendliness, the toolbox incorporates the use of help boxes, and provides direct links to descriptive Help Pages throughout. To illustrate its capabilities, an example problem is included as part of the User's Guide in Appendix II.

REFERENCES

[1.] Bordignon, Kenneth A., *Constrained Control Allocation for Systems with Redundant Control Effectors*, PhD dissertation, Virginia Polytechnic Institute and State University, Blacksburg, VA, December 19, 1996.

[2.] Bolling, John G., *Implementation of Constrained Control Allocation Techniques Using an Aerodynamic Model of an F-15 Aircraft*, MS thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, May, 1997.

[3.] Durham, W.C., "Constrained Control Allocation", *Journal of Guidance, Control, and Dynamics*, Vol.16, No. 4, 1993, pp. 717-725.

[4.] Durham, W.C., "Constrained Control Allocation: Three-Moment Problem", *Journal of Guidance, Control, and Dynamics*, Vol.17, No. 2, 1993, pp. 330-336.

[5.] Durham, W.C., "Attainable Moments for the Constrained Control Allocation Problem", *Journal of Guidance, Control, and Dynamics*, Vol.17, No. 6, 1994, pp.1371-1373.

[6.] Preparata, F.P., and Shamos, M.I., *Computational Geometry: An Introduction*, Springer-Verlag, 1985.

[7.] Barber, B.C. Dobkin, D.P., and Huhdanpaa, H., "The Quickhull Algorithm for Convex Hull", Research Report GCG53, Geometry Center, University of Minnesota, July 1993.

[8.] *MATLAB Application Program Interface Guide*, The Mathworks, Inc., Natick, Massachusetts, December 1996.

[9.] *Building GUIs with MATLAB*, The Mathworks, Inc., Natick, Massachusetts, December 1996.

[10.] *Using MATLAB Graphics*, The Mathworks, Inc., Natick, Massachusetts, December 1996.