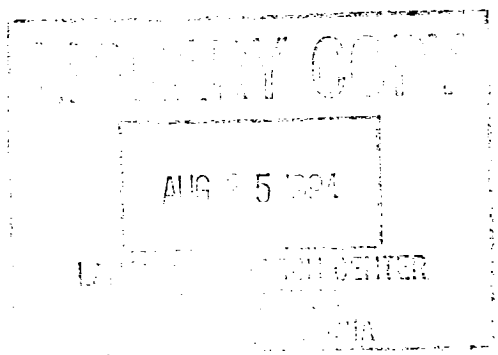
NASA Contractor Report-194960

Advanced Information Processing System: Hosting of Advanced Guidance, Navigation and Control Algorithms on AIPS Using ASTER

Richard Brenner, Jaynarayan H. Lala, Gail A. Nagle, Andrei Schor, John Turkovich
THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565
August 1994



National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

NASA Contractor Report-194960

Advanced Information Processing System: Hosting of Advanced Guidance, Navigation and Control Algorithms on AIPS Using ASTER

Richard Brenner, Jaynarayan H. Lala, Gail A. Nagle, Andrei Schor, John Turkovich
THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565
August 1994

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

ASTER is a registered trademark of the Charles Stark Draper Laboratory, Inc.

Ada is a registered trademark of the Ada Joint Program Office (JPO).
KMS is a registered trademark of Knowledge Systems Incorporated.
MATLAB is a registered trademark of The MathWorks, Inc.
OpenLook is a registered trademark of UNIX System Laboratories, Inc.
UNIX is a registered trademark of UNIX System Laboratories, Inc.
X-Window System is a registered trademark and product of the Massachusetts Institute of Technology.
Sun Workstation is a registered trademark of Sun Microsystems, Inc.
Ada is a registered trademark of the United States Government, Ada Joint Program Office.
FrameMaker is a registered trademark of Frame Technology Corporation.
PostScript is a registered trademark of Adobe Systems Incorporated.

TABLE OF CONTENTS

Title	Page
List of Illustrations	v
List of Tables	vii
1.0 INTRODUCTION	1-1
2.0 BACKGROUND	2-1
2.1 FENOC	2-1
2.2 ASTER	2-2
2.3 Advanced Information Processing System (AIPS)	2-4
3.0 GN&C ALGORITHMS AND VEHICLE SIMULATION	3-1
3.1 Guidance	3-2
3.1.1 Guidance Algorithm Formulation	3-2
3.1.2 MATLAB Scripts	3-7
3.1.2.1 MATLAB Scripts Adaptation and Rewrite	3-7
3.1.2.2 New FENOC MATLAB Scripts Dependency Diagram	3-8
3.1.3 Interface Definition	3-9
3.1.4 Implementation Considerations	3-9
3.2 Vehicle Simulation	3-10
3.2.1 Vehicle Dynamics Model	3-10
3.2.2 Environment Model	3-13
3.2.3 Sensor Model	3-13
3.2.4 Interface Definition	3-14
3.2.5 Numerical Integration	3-14
3.2.6 Implementation Considerations	3-16
3.3 Control	3-16
3.3.1 Control Scheme	3-16
3.3.2 Analysis	3-17
3.3.3 Interface Definition	3-18
3.3.4 Implementation Considerations	3-18
3.4 Navigation	3-18
3.4.1 Sensor Conditioning	3-18
3.4.2 Analysis	3-19
3.4.3 Interface Definition	3-19
3.4.4 Implementation Considerations	3-20
3.5 ASTER Specification	3-20

4.0 ASTER 4-1
4.1 Technical Background..... 4-1
4.2 Algebraic Transform Engine Overview 4-2
4.2.1 Overall Structure..... 4-2
4.2.2 General Operation 4-4
4.3 ATE Imlementation and Design Trade-Offs 4-4
4.3.1 The MATLAB Language Specializer..... 4-5
4.3.2 Declarations in ASTER Style MATLAB..... 4-10
4.3.3 The ATE Installation Mechanism 4-10
4.4 Experience..... 4-11
4.4.1 Converting MATLAB Scripts to ASTER Style..... 4-11
4.4.2 Converting ATE Representation to ASTER Block Diagram; Representation 4-12

5.0 HOSTING OF AGN&C ON AIPS 5-1
5.1 Interprocess Communication Protocol for AGN&C 5-1
5.2 Task Scheduling..... 5-6
5.3 Demonstration Hardware Configuration..... 5-8
5.4 Graphical User Interface..... 5-9
5.5 Integration and Testing..... 5-12

6.0 SUMMARY AND CONCLUSIONS..... 6-1
6.1 Summary 6-1
6.2 Future Work 6-2

7.0 REFERENCES..... 7-1

APPENDICES
A. JACOBIAN: An Excerpt of Original Martin Marietta MATLAB Script..... A-1
B. JACOBIAN: An Excerpt of Revised MATLAB Script for ASTER B-1
C. ASTER Style MATLAB Guidelines C-1

LIST OF ILLUSTRATIONS

Figure	Title	Page
1.1	Architecture for the Unified Demonstration of FENOC, ASTER, FTPP, and AP Network	1-4
2.1	Architecture of the ASTER Automatic Programming Subsystem	2-4
3.1	High Level View of a GN&C System	3-1
3.2	Simplified Flight Problem	3-2
3.3	Geometry of the Vehicle Dynamics Model	3-11
3.4	Attitude Control Block Diagram	3-16
3.5	Block Diagram Definition: System-View	3-21
3.6	Block Diagram Definition: Operator_Interface	3-22
3.7	Block Diagram Definition: Command_Mission	3-23
3.8	Block Diagram Definition: Signal_Conditioning	3-24
3.9	Block Diagram Definition: Navigation	3-25
3.10	Block Diagram Definition: Guidance	3-26
3.11	Block Diagram Definition: Initial-Trajectory	3-27
3.12	Block Diagram Definition: FENOC-Algorithm	3-28
3.13	Block Diagram Definition: Control	3-29
3.14	Block Diagram Definition: Pitch Control	3-30
3.15	Block Diagram Definition: Vehicle Model	3-31
3.16	Block Diagram Definition: Vehicle-Sensor-Models	3-32
3.17	Block Diagram Definition: Environment-Model	3-33
4.1	The Architecture of the ATE	4-3
5.1	Communication Interfaces of the Distributed AGN&C Application	5-5
5.2	Sim and Control Execution Time Line	5-6
5.3	Scheduling Paradigm of the FENOC Computation and Communication Tasks	5-7
5.4	Hardware Configuration for the AGN&C Demonstration System	5-8
5.5	The AGN&C Graphical User Interface	5-9
5.6	Launch Parameters	5-11
5.7	Vehicle Altitude vs Time	5-13
5.8	Vehicle Range vs Time	5-14
5.9	Vehicle Altitude vs Vehicle Range	5-14
5.10	Vehicle Y Velocity vs Time	5-15
5.11	FENOC Trajectory	5-15
5.12	Actual Pitch Angle and Commanded Pitch Angle vs Time	5-16
5.13	Delta Pitch Angle vs Time	5-16

LIST OF TABLES

Table	Title	Page
4.1	A Comparison of ASTER and MATLAB Working Paradigms	4-6
5.1	The AGN&C Execution Sequence	5-3
5.2	Distributed AGN&C Message Specifications	5-5
5.3	AGN&C Development Approach	5-19

1.0 INTRODUCTION

There is an increasing interest on the part of NASA and DOD to modernize the country's capabilities to launch payloads into space. The current suite of launch vehicles dates back several decades in the technological sophistication. To compete in the world markets for the space launch business the US must develop new launch vehicles that possess two main attributes: low cost and high dependability. The goal of the NASA/DOD Advanced Launch System [1] was to place a payload in low earth orbit at $300 per pound which is about an order of magnitude lower than the current costs and to do so with very high reliability and availability. A number of technologies have been developed over the past few years that can help achieve these ambitious goals.

The primary objective of this project was to demonstrate a unified application of a diverse but inter-related set of technologies for the space launch vehicles, in particular, and for mission- and/or safety-critical applications, in general.

An important cost factor in current launch systems is the large amount of mission preparation required for every launch. Generally, the entire trajectory must be custom designed for each mission, depending on the payload weight and environmental and mission constraints. This preparation requires long lead times before launch and makes current planning systems rather inflexible to last minute changes in launch conditions. As a matter of fact, launches have been delayed both due to higher than expected winds at high altitudes as well as lower than forecast and planned wind conditions. Use of an automated mission planner, especially one with an in-flight trajectory redesign capability, could make a significant contribution to reducing launch costs. A Finite Element Numerical Optimal Control (FENOC) law was, therefore, selected as the application for this demonstration. FENOC is the result of collaboration among academia, industry and government represented by the Georgia Institute of Technology, the Martin Marietta Corporation and the NASA Langley Research Center, respectively. FENOC is intended to determine guidance trajectories in real-time, is computationally intensive, and lends itself to parallel processing. FENOC specifications are available in a form that represents how guidance engineers would like to communicate their functional designs to software engineers. Martin Marietta designed and developed their application of FENOC in MATLAB™ which is an application design and analysis language and environment.

Another contributor to the cost of space launches is the development of high quality software. The traditional methods of designing, developing and testing software are labor-intensive and error prone. Tremendous effort is expended in testing, simulating and, in general, validating software for mission- and/or life-critical space operations. ASTER™ (Automatic Software Technology for Engineering Reliability) is a system focused on automatic software development at the Charles Stark Draper Laboratory with

the intended goal of producing very high quality software at a low cost. This technology program has three key features: a software development process that is viewed from the perspective of application engineers, a collection of technologies that allow automation of that new development process, and an automatic programming subsystem that is a tool that can be used by GN&C application engineers to specify functional designs and automatically generate Ada® code, C code and documentation. One of the goals of the current project was to provide a MATLAB interface for ASTER so that a guidance engineer can directly produce flight software in Ada corresponding to the MATLAB specification of FENOC without going through the intermediate process of either explaining the algorithm design to a software engineer or re-specifying the algorithm using block diagrams.

Once FENOC has been implemented in Ada, the next challenge is to provide a hardware platform to execute the code in real-time with a high degree of dependability. The flight computer must have sufficient throughput to process the sensor data and compute a new trajectory in real-time under nominal conditions, i.e., when all hardware components are operational. It should also be able to execute the FENOC algorithm correctly in the presence of failed components. The design, development and validation of fault tolerant computers for mission- and/or safety-critical applications has been an expensive proposition. Development of cost-effective validated fault tolerant architectures can contribute to the reduction of launch vehicle costs as well as increase the dependability of launch services, in effect, further reducing the life-cycle cost. Under the Advanced Information Processing System (AIPS) program, sponsored by NASA, a knowledge base has been created which will allow achievement of validated fault tolerant distributed computer system architectures, suitable for a broad range of applications [2]. Among the components of this knowledge base are hardware and software building blocks. The hardware building blocks include fault tolerant computers of varying levels of redundancy and throughput. The software building blocks include real-time operating systems and redundancy management software. One of these fault tolerant computers, specifically the Fault Tolerant Parallel Processor (FTPP) [3, 4], was selected to host the ASTER-produced Ada code for FENOC.

In addition to the fault tolerant computers, one also requires a dependable means of communicating information between the computers and between the I/O devices such as sensors and actuators and computers. Concurrent to the current program, a joint NASA LaRC and SDIO program at the Draper Laboratory has been investigating the use of authentication protocols (AP) for reliable communications [5]. The goal of the AP program is to use digital signatures to sign messages on the network such that the receiver can authenticate the signature and verify the correctness of the message. Some of the attributes of the AP network include capability to provide communication between sites of varying redundancy level without jeopardizing the more reliable sites; maximum use

of existing industry and military standard network topology, protocols, and physical media; support of heterogeneous computational platforms (workstations, embedded computers), operating systems (UNIX, LynxOS, Ada Run Time System), and programming languages (C, Ada); support of interoperability of heterogeneous network topologies, protocols, and physical media such as Ethernet, FDDI, ATM, Mil-Std 1553, etc. It was decided to use the AP network to interconnect the various computational nodes that are required for the demonstration of the ASTER-produced FENOC Ada code on AIPS FTPP building blocks.

Figure 1.1 shows the initial overall architecture of the intended demonstration. A guidance engineer conceptualizes a trajectory-generation algorithm and optimizes it using MATLAB. The engineer then inputs the optimized algorithm into ASTER using the MATLAB script. ASTER produces the corresponding Ada code and documentation automatically without further human intervention. The Ada code is compiled, linked and hosted on the target flight computer, the Fault Tolerant Parallel Process (FTPP). (The eventual goal is for ASTER to automatically produce parallel code. However, this was not within the scope of the current program). The launch vehicle control as well as navigation functions are executed by another fault tolerant processor. A Sun workstation simulates the launch vehicle dynamics, i.e., the model. Another Sun workstation acts as the operator's console. It displays the vehicle state as the mission progresses such as vehicle altitude, downrange, horizontal and vertical velocity, etc. It also accepts operator commands to set the launch parameters such as final orbital altitude and velocity, to excite wind gusts, etc. Yet another Sun workstation simulates the environment such as wind, turbulence, gravity, etc.

In a flight system, the workstation that simulates the environment would not exist; the operator's console would be replaced by one or more ground links; and the workstation that simulates the vehicle dynamics would be replaced by a set of sensors and actuators of varying redundancy level. However, the in-flight communications requirements are not much different from those of the demonstration system. Figure 1.1 shows the intended communications network to tie all the computers together: an authentication protocols-based dual redundant AIPS network.

The eventual demonstration system turned out to be a little different from the initial configuration, as described in greater detail later in the report. In particular, the navigation and control algorithms were also hosted on a version of the FTPP; the vehicle model and the environmental simulations were hosted on a single workstation; and a linear topology was used for the network.

The overall goal of the current project can now be restated as the unified demonstration of a diverse but inter-related set of technologies that can make a broad range of mission-

and/or safety-critical systems, including space launch vehicles, more cost-effective and dependable. These technologies are: advanced guidance algorithms such as FENOC, tools to produce high quality software such as ASTER, fault tolerant computers such as FTPP, and reliable communications networks such as AP. By replacing the guidance algorithm with an algorithm for a different application, the other enabling technologies, i.e., ASTER, FTPP, AP networks, can be equally applied to these other applications.

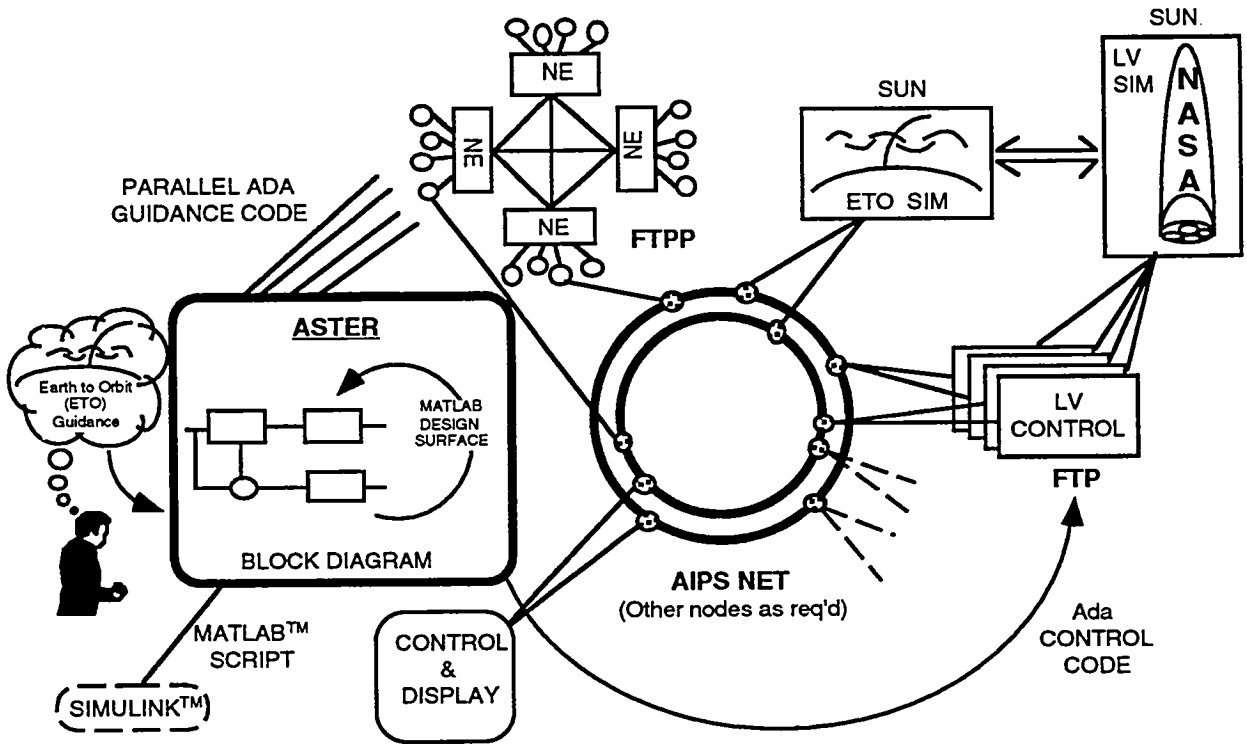


Figure 1.1 Architecture for the Unified Demonstration of FENOC, ASTER, FTPP, and AP Network

The remainder of this report is organized as follows. Section 2 provides an introduction to the main elements of this demonstration: FENOC algorithm, ASTER, and AIPS. Section 3 discusses the Advanced GN&C algorithms (FENOC guidance algorithm and vehicle control and navigation algorithms) in detail. It also describes the simulation of the launch vehicle dynamics. Section 4 describes the Algebraic Transform Engine for ASTER to interface to MATLAB. Section 5 describes the implementation of AGN&C algorithms on AIPS. Section 6 concludes with a summary and thoughts on future work.

Reference 9 is the basis for the FENOC algorithm. Appendix A is an excerpt, called Jacobian, of the FENOC MATLAB script produced by Martin Marietta. Appendix B is the corresponding excerpt of the Draper-modified MATLAB script that was used as input to ASTER. Appendix C is a set of guidelines on constructing MATLAB scripts for ASTER.

2.0 BACKGROUND

2.1 FENOC

Avionics and embedded system applications were reviewed for the purpose of demonstrating automatic generation of code and subsequent execution of this code on an AIPS configuration of fault-tolerant processors. A launch vehicle application using a Finite-Element, Numerical, Optimal Control (FENOC) law was selected as the application for this demonstration. FENOC was selected because it has the following four attributes:

- FENOC is the result of collaboration among academia, industry and government,
- FENOC is intended to determine guidance trajectories in real time,
- FENOC specifications are available, and
- FENOC is computationally intensive and lends itself to parallel processing.

FENOC represents the result of collaboration among academia, industry, and government in the United States. The partners in the FENOC collaborative effort are the Georgia Institute of Technology, Martin Marietta, and the NASA Langley Research Center. The Georgia Institute of Technology provides the analytic concept, theory and analysis for the algorithm. Martin Marietta, Space Systems Division provides the application of this theory to a launch vehicle guidance system. The NASA Langley Research Center provides the coordination and peer review of these efforts. The theoretical and analytical background is published in a number of papers published by Hodges and Bless from the Georgia Institute of Technology [See Ref. 9].

FENOC, for this launch vehicle application, is intended to determine guidance trajectories in real time. FENOC is intended to reside within an onboard, embedded processing system. This approach replaces a ground-based approach where guidance trajectories (typically one) are computed prior to launch in ground-based computers. A trajectory, which is tailored to specific environmental conditions, is loaded aboard the launch vehicle. When environmental conditions are appropriate, launch occurs and the "canned" trajectory is pursued within the control capability of the vehicle. In contrast, with the FENOC algorithm, a vehicle can be launched on demand and in real time compute new commanded trajectories when actual flight deviates from the current commanded trajectory.

FENOC specifications are available in a form that represents how guidance engineers would like to communicate their functional designs to software engineers. Martin Marietta designed and developed their application of FENOC in MATLAB™. MATLAB is an application design and analysis language and environment.

The computational load for FENOC increases exponentially as the desired degree of precision increases. Many of the computations that occur can be performed in parallel. Because of this characteristic and because research had previously been conducted by Draper into automatic generation of parallel code, FENOC was attractive.

Generation of parallel code was not an objective of this effort but selecting an algorithm with this characteristic brings the potential for future evolutionary development that leverages this current work.

FENOC specifications were available for this demonstration. The Georgia Institute of Technology and Martin Marietta applied analytical work to a launch vehicle. The first order approximation of this launch vehicle is a point mass, single stage vehicle undergoing constant thrust in two-dimensional space. This vehicle operates in an environment consisting of a flat earth, constant gravity field with no atmosphere. These characteristics are modelled, simulated and analyzed using MATLAB. The resulting MATLAB scripts were delivered to Draper as a design specification for the FENOC guidance algorithm.

Even though modularized for design and analysis purposes, these MATLAB scripts had to be altered because they included two features that are needed for analysis but not embedded processing. These features are essentially communication and executive code. Communication code implements keyboard inputs, monitor displays and plotting. Executive code expects an analyst to identify when convergence has occurred and stop simulations. The communication and executive MATLAB code was removed in order to incorporate the FENOC functionality into the AIPS embedded system. Also, Draper developed convergence criteria so that the guidance algorithm could automatically terminate without operator intervention.

FENOC is computationally intensive and lends itself to parallel processing.

2.2 ASTER

ASTER (Automatic Software Technology for Engineering Reliability) is a second generation system that resulted from technology programs sponsored by the NASA LaRC and CSDL [6]. These technology programs have three key features:

- a software development process that is viewed from the perspective of application engineers
- a collection of technologies that allow automation of this new development process, and

- an automatic programming system that is a tool which is used by embedded system application engineers to specify functional designs and then automatically generate Ada code, C code and documentation.

ASTER predecessors are CSDL CASE [7] and ALS CASE. CSDL CASE was developed under Draper's Independent Research and Development program and was the basis for ALS CASE. ALS CASE was developed for the Advanced Launch System under the administration of the NASA Langley Research Center. ASTER draws upon experiences gained from designing, developing and applying CSDL CASE and ALS CASE.

All three of these systems view software design, development and maintenance from the viewpoint of application engineers. The approach to code and document generation described in this report maintains consistency among design specifications, code and documentation. The need for prototype implementations is eliminated since embedded code can be produced as rapidly as a prototype with no additional effort. This approach allows application engineers to receive essentially immediate feedback regarding impacts of design changes on code that implement their designs.

During the 1980's, a number of technologies were brought together to create CSDL CASE and ALS CASE. Most of these technologies are outgrowths of knowledge engineering. These technologies include:

- Symbolic processing,
- Functional specification,
- Object-oriented representation,
- Interactive graphics, and
- Engineering workstations

Also during the 1980's, the field of computer-aided software enginering (CASE) had evolved in a direction that differs from the concept behind CSDL CASE and ALS CASE. CSDL CASE and ALS CASE, in concept, focus on the role of application engineers in software development and include the role of software engineers. The general CASE industry, on the other hand, clearly focuses on the role of software engineers but effectively disregards the role of application engineers.

These technologies and the CASE industry were evolving at the same time as CSDL CASE and ALS CASE. As a result, standards, some formal and others ad hoc, were defined. This left CSDL CASE and ALS CASE in a tentative situation.

ASTER was developed to take advantage of the standards and unified support behind tools that adhere to these standards. Secondly, ASTER breaks the perceived association with what has become the traditional set of CASE tools.

ASTER currently resides on SUN workstations. It is designed to be easily portable to general purpose engineering workstations with high resolution, graphics displays. ASTER effectively uses UNIX, X-windows, and OpenLook guidelines. ASTER is designed such that communication protocols and user interface styles can easily be accommodated.

Figure 2.1 illustrates the architecure of ASTER's automatic programming system. This system contains a highly interactive, graphical user interface for entering engineering block diagrams and algebraic expressions, an automatic software designer, an automatic code generator, and an automatic document generator.

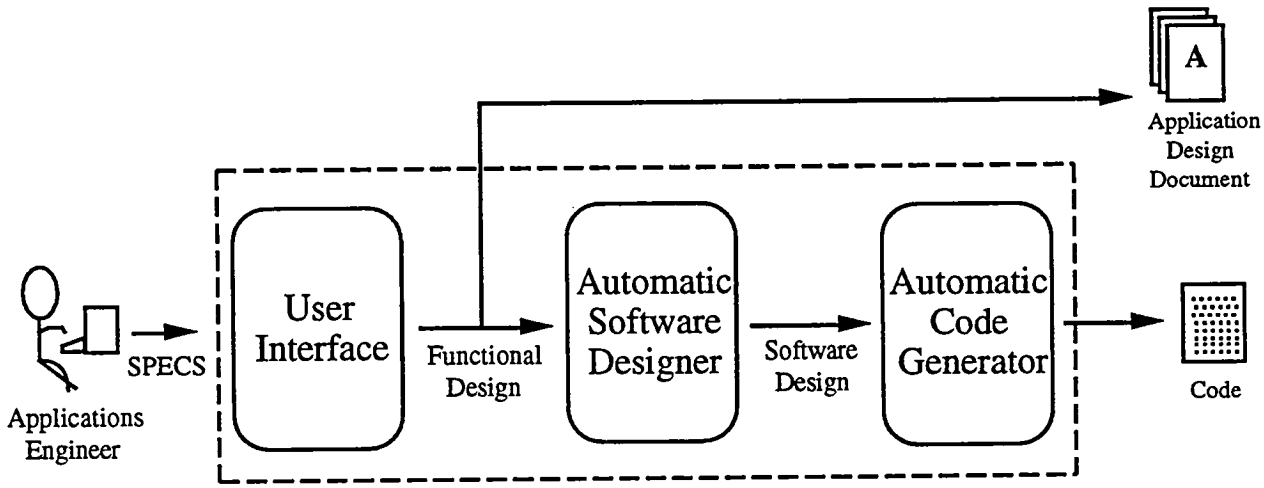


Figure 2.1 Architecture of the ASTER Automatic Programming Subsystem

2.3 Advanced Information Processing System (AIPS)

The goal of the Advanced Information Processing System (AIPS) program, sponsored by NASA and other government agencies, has been to produce the knowledge base necessary to achieve a validated fault tolerant distributed computer system architecture to meet the real-time computational needs of advanced aerospace vehicles. A part of this knowledge base is the demonstration of key AIPS concepts as embodied in hardware and software building blocks. Some of these building blocks were used to host the ASTER-generated code for the AGN&C algorithms and are described in the following.

The Fault Tolerant Parallel Processor (FTPP) is a high-throughput and a highly dependable computational node [3, 4]. Its major attributes are as follows.

Dependability Attributes

FTPP can tolerate arbitrary component failure modes. It is "Byzantine Resilient". It uses redundant Processing Elements (PEs) for high reliability. The PEs can be organized to provide a triplex or a quadruplex level of redundancy or no redundancy at all, i.e., simplex processing. All three levels of redundancy can co-exist in the same FTPP cluster. Furthermore, the configuration can be changed dynamically to optimize mission reliability and availability. The fault tolerance and system reconfigurations are nearly transparent to the programmer.

Parallelism Attributes

FTPP uses many PEs for high throughput. Cluster C3 can accommodate up to 40 PEs. PEs communicate via message passing. The parallelism is also nearly transparent to the programmer. The system can be reconfigured in real-time to trade throughput for reliability. For example, the 40 processors in C3 can be variously organized as ten quad-redundant virtual processor groups (VGs); or five quad VGs, two triplex VGs and 14 simplexes; or one triplex VG and 37 simplexes; or some other combination of simplexes, triplexes and quads.

Open System Attributes

FTPP is a standards-based architecture. It uses Commercial-Off-The-Shelf (COTS) and Non-Development Items (NDI). For example, any COTS processors, backplanes, power supplies, and I/O boards can be used that the application needs for certain reasons. The FTPP architecture does not impose any additional constraints for fault tolerance or parallelism reasons. Similarly, any programming language and operating system may be used in the FTPP. FTPP also supports heterogeneous resources.

The Authentication Protocols (AP) network [5] was selected to interconnect the FTPP clusters C2 and C3 and the various workstations required for the AGN&C demonstration. Major AP attributes were summarized in Section 1.

3.0 GN&C ALGORITHMS AND VEHICLE SIMULATION

There is an increasing pressure to reduce payload launch costs. Indeed, the budgetary survival of many future civilian and even military space missions hinges on drastically decreasing these costs, along with the cost of operations.

An important cost factor in current launch systems is the large amount of mission preparation required for every launch. Generally, the entire trajectory must be custom designed for each particular mission, depending on the payload weight and environmental and mission constraints. This preparation requires a large amount of lead time before launch and makes current planning systems rather inflexible to last minute changes in launch conditions. In light of this fact, using automated mission planning, especially having an in-flight trajectory redesign capability, could make a significant contribution to reducing launch costs.

In principle, constructing an optimal trajectory algorithm is quite straightforward. The mathematical foundation of such algorithms is solid and there is considerable experience in using them. However, implementing an in-flight, i.e., real-time, guidance capability is considerably more difficult. The algorithm must execute within certain time limitations and must be robust, that is, it should always produce a feasible trajectory. The enormous advances in computational capabilities that can now be provided on-board a launch vehicle make such a capability practical. A high level view of the GN&C system developed for this technology demonstration project is shown in Figure 3.1.

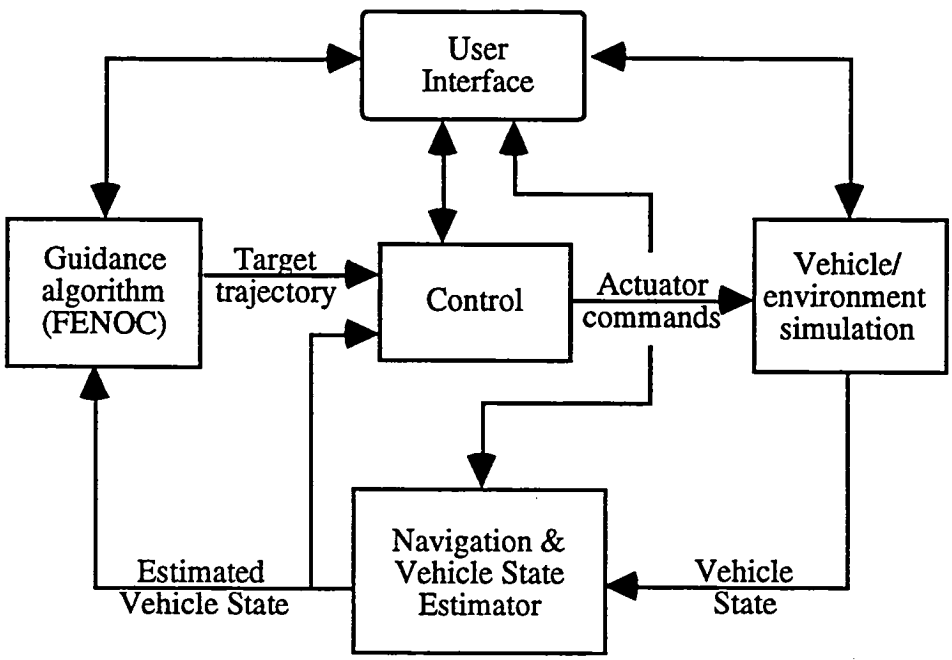


Figure 3.1 High Level View of a GN&C System

The following sections describe the components of this system, the interface definition and various implementation considerations. The primary emphasis is placed on the Guidance algorithm, as it incorporates a number of novel features and serves as a vehicle for the demonstration of ASTER capabilities.

3.1 Guidance

This section introduces the guidance algorithm formulation and its numerical solution. It then discusses the adaptation and rewrite of the original MATLAB scripts to conform to ASTER's algebraic transform constraints. Finally, the interfaces to other modules are defined and specific implementation issues are presented.

3.1.1 Guidance Algorithm Formulation

The optimal vehicle trajectory is generated by the application of the Linear Tangent Guidance (LTG) law. This guidance law is derived from a simplified vehicle flight problem, which assumes a gravitational field constant in magnitude and direction, i.e., the "flat Earth model."

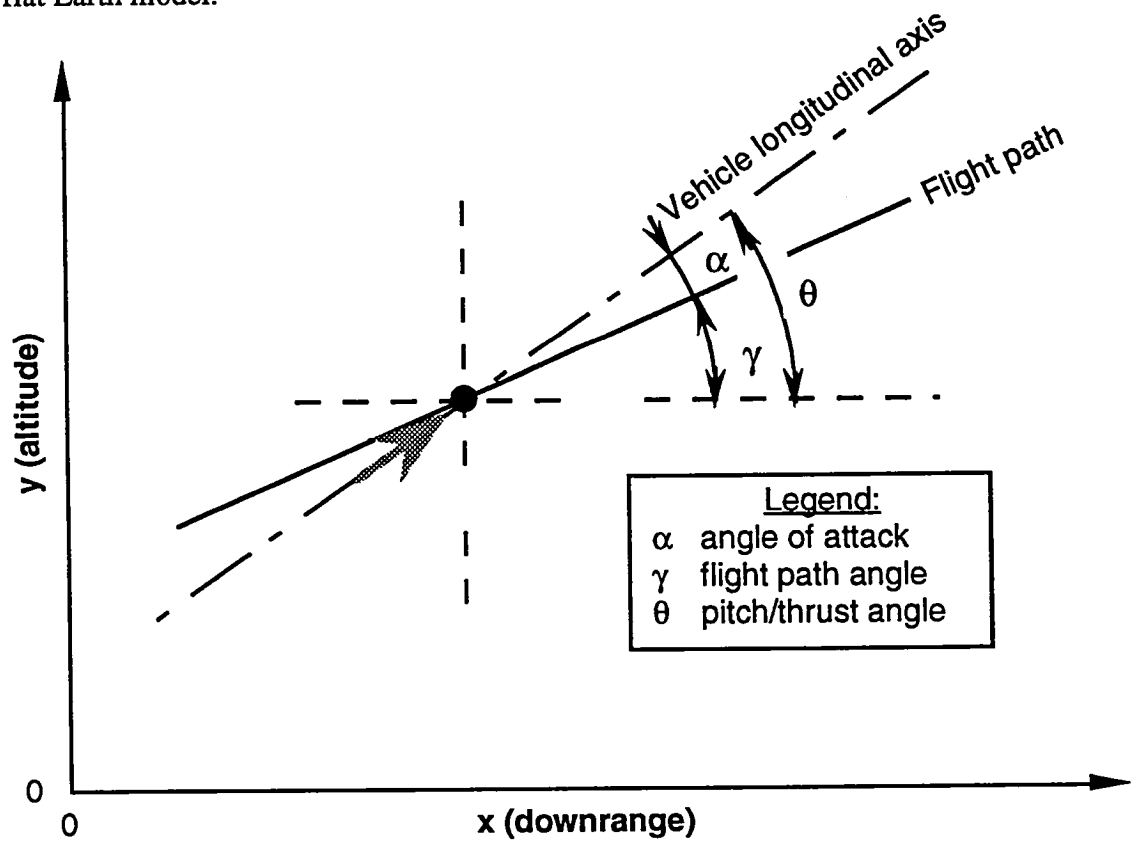


Figure 3.2 Simplified Flight Problem

The assumption is reasonable in practice, if the downrange (to orbital insertion) and the

trajectory altitude are small compared to the Earth's radius. A schematic of the simplified flight problem is shown in Figure 3.2. The vehicle is represented as a point mass, with the thrust axis identical to the longitudinal centerline of the vehicle and the motion taking place in a vertical plane. The thrust acceleration is assumed constant. The differential equations of motion, representing a balance among the inertial, gravitational and thrust forces can be written as:

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x},\mathbf{u},t) \tag{3.1a}$$

where:

$\mathbf{x}$	vehicle state vector
$\mathbf{u}$	vehicle control vector
t	elapsed time

or specifically

$$\frac{d}{dt}\begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} x_3 \\ x_4 \\ \text{thrust\_accel}*\cos\theta \\ \text{thrust\_accel}*\sin\theta - g \end{vmatrix} \tag{3.1b}$$

where:

x_1	vehicle position in the x-direction (m)
x_2	vehicle position in the y-direction (m)
x_3	vehicle velocity in the x-direction (m/s)
x_4	vehicle velocity in the y-direction(m/s)
θ	thrust angle (rad) (note: this is a control command)

The objective of this flight problem is to minimize the fuel consumption during its ascent to orbit. The general form of the cost function is given by:

$$J = \Phi(\mathbf{x}(T),\mathbf{u}(T),T) + \int_0^T L(\mathbf{x}(t),\mathbf{u}(t),t)\, dt \tag{3.2}$$

where:

Φ	terminal component of the cost function
L	integrand of the integral component of the cost function
$\mathbf{u}$	vehicle control vector
T	final time (s)

In this simplified problem, a constant fuel mass flow rate is assumed. There is no terminal cost function and the control vector reduces to the thrust angle θ. The cost function becomes simply

$$J = \int_0^T dt \tag{3.3}$$

Terminal constraints, of the form

$$\Psi(\mathbf{x}(T),\mathbf{u}(T),T) = 0 \tag{3.4}$$

complete the general optimal control problem statement. In the case of the simplified problem at hand, the constraints are used to specify the desired orbit, defined by the orbit altitude, h, and the horizontal orbit insertion velocity, U. The vertical orbit insertion velocity is assumed to be zero. These specific constraints can be expressed as

$$\Psi = \begin{vmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x_1(T) \\ x_2(T) \\ x_3(T) \\ x_4(T) \end{vmatrix} - \begin{vmatrix} h \\ U \\ 0 \end{vmatrix} \tag{3.5}$$

The terminal constraints are adjoined to the basic cost function through the discrete Lagrange multipliers, ν, defined at t = T. The equations of motions can be viewed as equality constraints, to be satisfied at any given instant. They are also adjoined to the base cost function via time-dependent Lagrange multipliers, λ, referred to as costates. The general augmented cost function, with no initial constraints, then becomes:

$$J_a = \Phi(\mathbf{x}(T),\mathbf{u}(T),T) + \int_0^T \left\{ L(\mathbf{x}(t),\mathbf{u}(t),t) + \lambda \cdot [\mathbf{F}(\mathbf{x},\mathbf{u},t) - \dot{\mathbf{x}}] \right\} dt + \nu \cdot \Psi(\mathbf{x}(T),\mathbf{u}(T),T)$$

The general optimal control formulation also uses the Hamiltonian, defined as

$$H = L(\mathbf{x}(t),\mathbf{u}(t),t) + \lambda \cdot \mathbf{F}(\mathbf{x},\mathbf{u},t)$$

which for the simplified problem becomes

$$H = 1 + \lambda_1 x_3 + \lambda_2 x_4 + \lambda_3[\text{thrust\_accel}*\cos\theta\,] + \lambda_4[\text{thrust\_accel}*\sin\theta - g] \tag{3.6}$$

A modified terminal cost is obtained by combining the actual terminal cost and the terminal constraints:

$$\Phi_a = \Phi(\mathbf{x}(T),\mathbf{u}(T),T) + \nu \cdot \Psi(\mathbf{x}(T),\mathbf{u}(T),T)$$

which reduces for the simplified problem to:

$$\Phi_a = \nu_1 [x_2(T) - h] + \nu_2 [x_3(T) - U] + \nu_3 [x_4(T) - 0] \tag{3.7}$$

The general augmented cost function can then be recast in the more compact form:

$$J_a = \Phi_a(\mathbf{x}(T),\mathbf{u}(T),T) + \int_0^T \left\{ L(\mathbf{x}(t),\mathbf{u}(t),t) + \lambda \cdot [\mathbf{F}(\mathbf{x},\mathbf{u},t) - \dot{\mathbf{x}}] \right\} dt \tag{3.8}$$

The augmented cost function constitutes the starting point of the variational approach which is the basis of the Weak Hamiltonian Principle. It should be noted that the role played in analytical mechanics by generalized coordinates and momenta is now played by the states and costates in the optimal control theory. A necessary condition for an extremal of J_a is that its first variation be zero. The final time will be assumed to be unknown. A key idea, illustrated for example by Equation (3.7), is to replace strong boundary conditions with weak boundary conditions, through the introduction of Lagrange multipliers. A strong boundary condition is one which specifies the value of the unknown under consideration, in this case the state or the costate vector at the initial and final times. Such an equality boundary condition is transformed into a "weak" boundary condition by adjoining it to the cost function through the introduction of discrete Lagrange multipliers. The detailed derivation of the weak formulation for the latter is presented in the recent paper by Hodges and Bless and will not be repeated here. The final formulation of the weak principle, in a form which does not contain time derivatives of $\mathbf{x}$ and λ, is given by:

$$\int_0^T \left\{ \delta\lambda\cdot\mathbf{x} - \delta\dot{\mathbf{x}}\cdot\lambda + \delta\mathbf{x}\left[\left(\frac{\partial L}{\partial \mathbf{x}}\right) + \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)\lambda\right] + \delta\lambda\cdot\mathbf{F} + \delta\mathbf{u}\left[\left(\frac{\partial L}{\partial \mathbf{u}}\right) + \left(\frac{\partial \mathbf{F}}{\partial \mathbf{u}}\right)\lambda\right] \right\} dt$$
$$+ \delta T\left(L + \lambda\cdot\mathbf{F} + \frac{\partial \Phi_a}{\partial t}\right)_T + (\delta\nu\cdot\Psi)_T + \left(\delta\mathbf{x}\cdot\hat{\lambda}\right)_0^T + \left(\delta\lambda\cdot\hat{\mathbf{x}}\right)_0^T = 0 \tag{3.9}$$

with the "hatted" quantities representing the discrete values of $\mathbf{x}$ and λ at the end points.

This form of the optimal control problem is used as the basis for the finite element discretization. The time interval [0,T] is broken into N elements. Over each element, a dimensionless time, τ, is defined as

$$\tau = (t - t_i) / (t_{i+1} - t_i)$$

The simplest acceptable shape functions are selected. Since no time derivatives of the $\mathbf{x}$ and λ appear above, piecewise constant shape functions will be used for them within each element. To accommodate the existence of derivatives of $\delta\mathbf{x}$ and $\delta\lambda$, i.e., the variations in $\mathbf{x}$ and λ, piecewise linear shape functions are used to represent them. The reader should also note the fact that there are no derivatives in $\mathbf{u}$ or $\delta\mathbf{u}$, again allowing for piecewise constant shape functions. The selected shape functions are summarized below

$$\delta\mathbf{x} = \delta\mathbf{x}_i(1-\tau) + \delta\mathbf{x}_{i+1}\tau$$

$$\delta\lambda = \delta\lambda_i(1-\tau) + \delta\lambda_{i+1}\tau$$

$$\delta\mathbf{u} = \delta\overline{\mathbf{u}_i}$$

$$\mathbf{x}=\begin{Bmatrix}\hat{\mathbf{x}}_i & \text{if } \tau=1\\ \bar{\mathbf{x}}_i & \text{if } 0<\tau<1\\ \hat{\mathbf{x}}_{i+1} & \text{if } \tau=1\end{Bmatrix},\quad \lambda=\begin{Bmatrix}\hat{\lambda}_i & \text{if } \tau=1\\ \bar{\lambda}_i & \text{if } 0<\tau<1\\ \hat{\lambda}_{i+1} & \text{if } \tau=1\end{Bmatrix},\quad \mathbf{u}=\begin{Bmatrix}\hat{\mathbf{u}}_i & \text{if } \tau=1\\ \bar{\mathbf{u}}_i & \text{if } 0<\tau<1\\ \hat{\mathbf{u}}_{i+1} & \text{if } \tau=1\end{Bmatrix}$$

These shape functions are introduced into Eq. (3.9). Carrying out the element quadrature leads to a general algebraic form of the weak Hamiltonian formulation of the optimal control problem, which results into a system of 2n(N+1) + mN + q + 1 nonlinear equations, where n is the number of states, m is the number of controls and q is the number of terminal constraints. There are 2n(N + 2)+ mN + q + 1 unknowns, namely:

- 2nN mean element states and costates,
- mN mean element controls,
- q Lagrange multipliers corresponding to the terminal constraints,
- 1 free final time,
- 4n end points states and costates.

Closure is effected by specifying the initial state vector, $\mathbf{x}_0$, and the final costate vector, λ_f , the latter through the transversality condition

$$\lambda_f=\left(\frac{\partial\Phi_a}{\partial\mathbf{x}}\right)_{t=T}$$

The system of nonlinear equations previously mentioned can be written in the form:

$$\mathbf{f}(\mathbf{z})=\mathbf{0} \tag{3.10}$$

where $\mathbf{z}$ is the composite unknown vector.

Newton-Raphson's is the method of choice for this type of problems. It consists of a succession of linear approximations which will converge to the actual solution, provided a "good" guess is used. Sensitivity to the starting guess is typical of gradient based methods, where robustness is traded for speed. There are other techniques which while slower near the solution, may provide a better initial iteration phase when the guess is poor. For our problem, the costate guess is most challenging. Fortunately, it appears that this simplified flight problem is reasonably insensitive to the initial guess. The solution proceeds recursively as:

$$J(\mathbf{z}_k)\,\Delta\mathbf{z}_k=-\mathbf{f}(\mathbf{z}_k) \tag{3.11}$$

where $\Delta\mathbf{z}_k=\mathbf{z}_{k+1}-\mathbf{z}_k$ and $J(\mathbf{z}_k)$ is the Jacobian matrix of $\mathbf{f}$ evaluated at the k-th iterate. The low order of the shape functions contributes to a very sparse Jacobian, a feature which for larger problems may be used to advantage. The iteration is terminated when one of the following criteria is met:

- the norm of the increment in $\mathbf{z}$ becomes less than a threshold,

- the norm of the vector function **f** becomes less than a threshold,
- the solution diverges, i.e., norms exceed certain thresholds,
- specified maximum number of iterations is exceeded.

It should be noted that nodal values of states and costates can be simply recovered from the element values in light of the shape functions chosen. Once these nodal state and costate values are known, *consistent* nodal control values may be obtained by using the optimality condition

$$\frac{\partial H}{\partial \mathbf{u}} = 0 \tag{3.12}$$

at each nodal point.

3.1.2 MATLAB Scripts

3.1.2.1 MATLAB Scripts Adaptation and Rewrite

ASTER paradigmatic and compatibility requirements imposed extensive changes or complete rewrite of the original scripts. The ASTER-style MATLAB guidelines are presented in detail in Appendix C. In this section, the adaptation work will be summarized, focusing on the salient points.

It should be underscored that throughout this script adaptation work, the objective has been to achieve compatibility with ASTER requirements while maintaining complete compatibility with MATLAB. This has been of paramount importance in allowing step-by-step testing of every modification for agreement with the original scripts.

The first step consisted of extracting from the entire set only those scripts and functions representing the FENOC algorithm proper. The driver, the initialization and a number of display options have been placed in an "outer shell". The FENOC algorithm itself is now driven by a master function, NEWTON, which is called by the driver in MATLAB, and which is transformed by ASTER into Ada code, incorporated into the demonstration computing framework.

The original MATLAB source code contains both script files (similar to the "include" files in other high level languages) and actual function files. For compatibility with the ASTER functional paradigm, all the scripts were transformed into proper functions, with *distinct* inputs and outputs, and with *no global variables.* A number of scripts were further decomposed for additional clarity. Functions containing iterative sections were re-modularized to satisfy the current ASTER requirement of having only one iterative cycle per transform. The body of these iterative transforms must contain only statements which are to be performed repeatedly. Optionally they may contain an initialization for those variables which will be updated within the iterative cycle.

Unlike MATLAB, ASTER is strongly typed. Type declaration statements were added to the new scripts for all the vectors, matrices and arrays involved. In MATLAB, the DECLARE statements merely invoke a "do-nothing" function.

The new scripts have been grouped in a hierarchy consistent with ASTER. This has been accomplished through the use of various nested "folders" (under the Macintosh Operating System) to achieve the proper hierarchical scoping effect.

3.1.2.2 New FENOC MATLAB Scripts Dependency Diagram

Excerpts of the new MATLAB scripts, Jacobian and Jacobian_Iter are included in Appendix B. The corresponding original MATLAB script, Jacobian, is listed in Appendix A. To aid the reader in understanding them, a calling tree diagram for NEWTON is given below.

NEWTON
- **NEWTON_ITER**
  - **FEM**
    - **FEM_ITER**
      - FF
      - Fx
      - Fu
      - L
      - Lx
      - Lu
    - FF
    - L
    - PHIx
    - PHIt
    - PSI
    - LAMBDAF_CALC
    - USOLVEMOD
  - **JACOBIAN**
    - **JACOBIAN_ITER**
      - FF
      - Fx
      - Fu
      - Ft
      - Fxx
      - Fxu
      - Fuu
      - L
      - Lx
      - Lu
      - Lt
      - Lxx
      - Lxu
      - Luu
    - LAMBDAF_CALC
    - PHIx

PHIt
PHIxx
PHItx
PHItt
PHInux
PHInut
PSIx
PSIt
FF
Fx
Ft
Lx
Lt
CONTROL_FOLD
ITER_TERM_TEST
TRAJECTORY
ELEM_ALT_VEL
NODE_ALT_VEL
NODE_ANGLES

3.1.3 Interface Definition

The GUIDANCE module communicates with CONTROL, NAVIGATION and USER INTERFACE as follows:

- From NAVIGATION: estimated horizontal coordinate (downrange), x^{est},
  estimated vertical coordinate (altitude), y^{est},
  estimated horizontal velocity, v_x^{est},
  estimated vertical velocity, v_y^{est},
  elapsed time, $t_{elapsed}$.
- To CONTROL: desired trajectory angle, γ_d, versus altitude.
- From USER: desired trajectory characteristics: final altitude and velocity, i.e., terminal constraints;
  vehicle and environment definition: thrust and gravitational accelerations;
- To User: desired trajectory angle, γ_d, versus altitude.

The estimated vehicle state information received from NAVIGATION is used as initial conditions for the guidance algorithm, which will generate a new trajectory, consistent with these conditions and with the given terminal constraints.

3.1.4 Implementation Considerations

A few implementation remarks are worth mentioning. In the original MATLAB scripts, the size of the Jacobian matrix, J, and of the residual vector , **f**, (see Equation (3.11)) was dependent on the choice of treatment for the final mission time, i.e., fixed (user input) or

free (an additional unknown). Since in ASTER variable dimensioning are not yet supported, in our implementation J and **f** are of constant size, regardless of the type of time boundary condition. This has been accomplished by using a degenerate last equation for the case of the fixed final time, which produces a nul increment for the final time (still handled as an unknown), thus propagating unchanged the initial guess for the final time throughout the iterative process.

Given its intended use in a real time application, the algorithm was set up to always generate at least a feasible solution, if not an optimal one, at each invocation. Generally, the past solution constitutes a very good initial guess for the next iteration cycle. For the first update, a very good initial guess is provided by running the FENOC algorithm off-line with the conditions prevailing at the start of the mission. This way, the iteration converges in only a few cycles.

The trajectory chosen for the demonstration problem is characterized by a final altitude of 400 Km and a final horizontal velocity of 8000 m/s. The final vertical velocity is zero.

3.2 Vehicle Simulation

This section introduces the vehicle simulation assembled for this demonstration. It first describes the models used for the vehicle, the environment and the sensors. The interface to the other modules are defined. Finally, considerations regarding the numerical integration of the mathematical models and specific implementation issues are presented.

3.2.1 Vehicle Dynamics Model

The vehicle model used for simulation adds some complexity to the simple model used in Guidance. It is represented as a rigid body of a certain length, mass and moment of inertia, with a gimbal-mounted thruster at one end. This configuration leads to the appearance of a thrust-induced torque. For simplicity, it is assumed that aerodynamic forces do not induce any torque, i.e., the center of pressure coincides with the center of mass of the vehicle. As before, the motion is taking place in a vertical plane. The thrust acceleration is assumed constant.

This representation leads to six state equations, governing two translations and a rotation, along with their respective rates. A control system is used to maintain the desired vehicle attitude, commanded by the guidance algorithm. A schematic of the geometry used to describe the vehicle motion is shown in Figure 3.3.

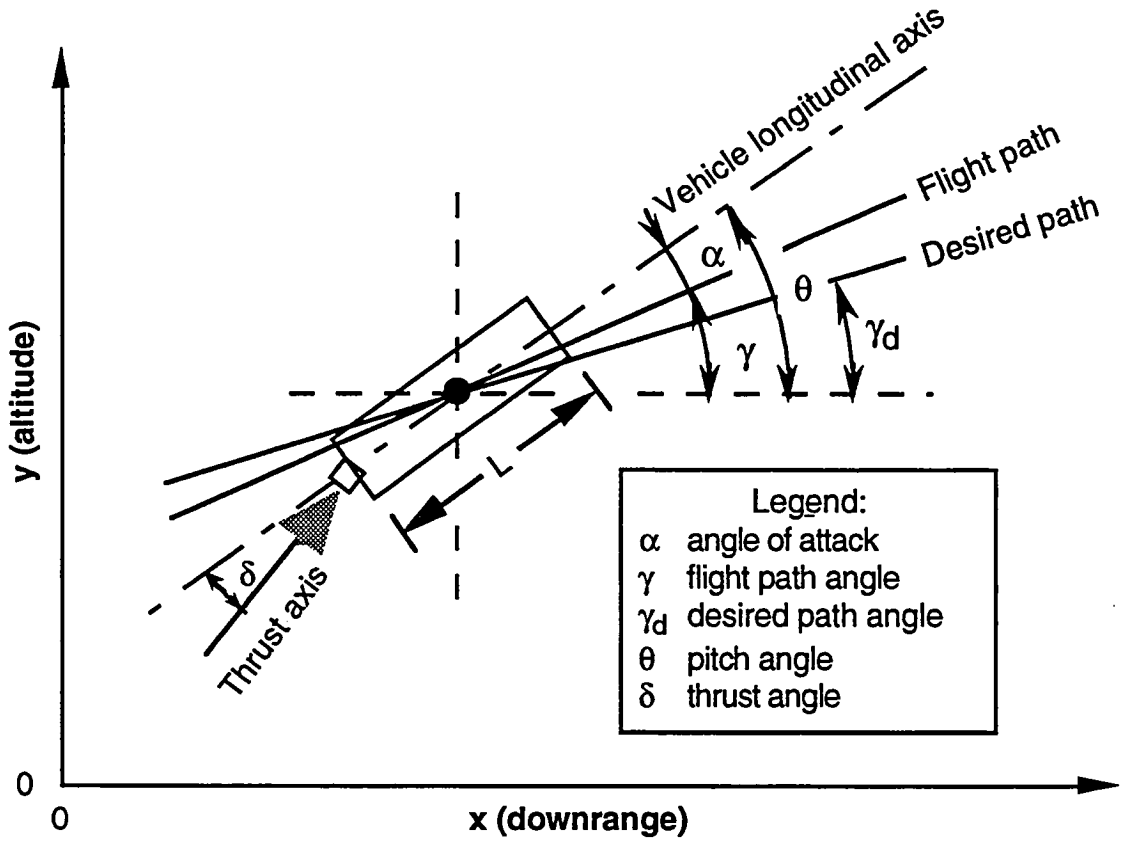


Figure 3.3 Geometry of the Vehicle Dynamics Model

The differential equations of motion, representing a balance among the inertial, gravitational, and thrust forces and torques, can be written as:

$$\frac{d}{dt}\begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{vmatrix} = \begin{vmatrix} x_3 \\ x_4 \\ \text{thrust\_accel}*\cos(x_5+\delta) - \text{dragx/mass} \\ \text{thrust\_accel}*\sin(x_5+\delta) - g - \text{dragy/mass} \\ x_6 \\ -(\text{mass}*\text{thrust\_accel}*(\text{length}/2)\sin(\delta) + \text{dragrot})/\text{inertia} \end{vmatrix} \qquad (3.13a)$$

An alternate, but fully equivalent formulation, can be obtained by using the appropriate trigonometric relationships for the sine and cosine of the sum-of-angles:

$$\frac{d}{dt}\begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{vmatrix} = \begin{vmatrix} x_3 \\ x_4 \\ \text{thrust\_accel}*(\cos x_5 \cos\delta - \sin x_5 \sin\delta) - \text{dragx/mass} \\ \text{thrust\_accel}*(\sin x_5 \cos\delta + \sin\delta \cos x_5) - g - \text{dragy/mass} \\ x_6 \\ -(\text{mass}*\text{thrust\_accel}*(L/2)\sin(\delta) + \text{dragrot})/\text{inertia} \end{vmatrix} \qquad (3.13b)$$

In this form, the transformation of the thrust from a vehicle-based to an inertial coordinate system is readily apparent.

The quantities used in Eqs. (3.13a,b) are defined below:

State variables:

x_1	x-coordinate (m)
x_2	y-coordinate (m)
x_3	x-velocity (m/s)
x_4	y-velocity (m/s)
x_5	pitch angle (θ) (rad)
x_6	angular velocity (ω) (rad/s)

Control input:

δ	thrust angle (rad)

Other quantities:

mass	Vehicle mass (Kg)
L	Vehicle length (m)
g	Gravitational acceleration (m/s^2)
thrust_accel	Thrust acceleration (m/s^2)
T	Final time (s)
dragx	Drag force in the x-direction (N)
dragy	Drag force in the y-direction (N)
dragrot	Drag torque in rotational motion (N m)
inertia	Moment of inertia in the pitch plane(Kg m^2)

The torque due to atmospheric drag is currently neglected, but it can be easily added as a refinement to the model. The drag force is represented simply as:

$$\text{dragx} = Cd_x \rho \, |x_3 - v_{\text{wind, x}}| \, (x_3 - v_{\text{wind, x}})$$

and

$$\text{dragy} = Cd_y \rho \, |x_4 - v_{\text{wind, y}}| \, (x_4 - v_{\text{wind, y}})$$

where $v_{wind,x}$ and $v_{wind,y}$ are components of the wind velocity in the x- and y-directions, respectively, Cd_x and Cd_y are the associated drag coefficients and ρ is the atmospheric density at the current vehicle altitude.

The reader should note that now there is a distinction between the flight path and the vehicle longitudinal axis, giving rise to an angle of attack. The desired path is the trajectory generated by the guidance solution. The control system will respond to deviations between the vehicle pitch and the desired trajectory angle, adjusting the thruster angle such as to drive this deviation to zero.

3.2.2 Environment Model

In this simulation, a rapidly declining density is modeled by assuming an isothermal atmosphere. Neglecting the air motion, the gas momentum equation reduces to the hydrostatic equilibrium equation:

$$\frac{dp}{dh} = -\rho g \tag{3.14}$$

where p is the local pressure and h is the altitude. Assuming the air behaves as a perfect gas and the atmosphere is isothermal (a common simplification) results in the following equation of state, relating p and ρ:

$$\frac{p}{\rho} = \frac{p_o}{\rho_o} \tag{3.15}$$

Differentiating Equation (3.15), substituting it into Equation (3.15) and integrating from zero to the current altitude results in:

$$\rho = \rho_o \exp\left(-\frac{g\rho_o h}{p_o}\right) \tag{3.16}$$

where p_o and ρ_o are reference atmospheric pressure and density. This altitude dependent density is used to calculate the atmospheric drag. To get a feel for the rate of decrease of density, the reader should note that the group $[p_o / g\rho_o]$ is equivalent to a length of about 10,000 meters. Thus, at an altitude of 100 kilometers, the density decreases by a factor of about 20,000.

3.2.3 Sensor Model

The vehicle state is assessed through three sensors:

- Body Mounted Accelerometers, for the accelerations in the x- and y- directions;
- a two degree-of-freedom gyroscope, measuring the pitch angle, and
- a Rate Gyroscope, measuring the pitch angle rate.

Currently, we have implemented a very simple sensor model, whereby a constant bias is added to the *actual* value to produce the *sensed* value. Furthermore, the sensors are assumed mounted at the center of mass of the vehicle to avoid additional frame transformation. It should be mentioned, however, that the highly modular structure of our framework facilitates the introduction of more sophisticated sensor models when desired.

3.2.4 Interface Definition

The SIMULATION module communicates with CONTROL, NAVIGATION and USER INTERFACE as follows:

- From CONTROL: thruster gimbal angle, δ.
- To NAVIGATION: sensed horizontal acceleration, a_x^{sensed},
  sensed vertical acceleration, a_y^{sensed},
  sensed pitch angle, θ^{sensed}
  sensed pitch angle rate, ω^{sensed}.
- From USER: vehicle definition: mass, length, moment of inertia, thrust acceleration and drag coefficients.
  environment definition: gravitational acceleration, wind velocity, reference atmospheric pressure and density.
  integration time step.
- To User: vehicle state vector.

3.2.5 Numerical Integration

There are many integration schemes available for the solution of a system of ordinary equations. For our application, given the fact that the vehicle must be actively controlled for stability with frequent thruster gimbal angle corrections, there was no incentive to resort to a sophisticated integration technique. Instead, an explicit Euler advancing scheme has been selected and implemented. The same scheme is used for both translations and the rotation in the pitch plane. The formulation is shown below for the x-direction translation:

$$x^{n+1} = x^n + \Delta t v_x^n + \frac{\Delta t^2}{2} a_x^n \tag{3.17a}$$

$$v_x^{n+1} = v_x^n + \Delta t a_x^n \tag{3.17b}$$

Note that Equation (3.17a) contains a second order correction which makes the solution slightly more accurate for a trivial additional computation. This scheme is conditionally stable, i.e., the integration time step must be kept below a certain threshold. To examine the scheme's numerical stability, it must be cast in the form:

$$\mathbf{q}^{n+1} = G\,\mathbf{q}^{n} \tag{3.18}$$

where $\mathbf{q}$ is the vector of unknowns and G is the discrete amplification matrix. For stability, the absolute value of the largest eigenvalue of G should be less than 1. To simplify our analysis, we remark from the outset that the rotational motion of the vehicle is characterized by a significantly smaller time constant compared to the two translations. Consequently, it suffices to focus our analysis on the angular momentum portion of the vehicle dynamics model. The equations are first linearized noting simply that $\sin\delta \approx \delta$. Second, the gimbal angle δ is obtained from the control system (see Section 3.3 for notations). With these considerations, the angular momentum equation with control can be written as:

$$\dot{\theta} = \omega \tag{3.19a}$$

$$\dot{\omega} = (K_1 K_2 K_v/J)\,\theta + (K_2 K_v/J)\,\omega \tag{3.19b}$$

after dropping the term containing the guidance command, which, for this analysis, may be assumed constant on the time scale relevant to the control system. Applying the scheme indicated in Equations (3.17a,b), neglecting, for simplicity, the second order term and casting the resulting equations into the standard form, Equation (3.18), results in:

$$\begin{vmatrix}\theta\\ \omega\end{vmatrix}^{n+1} = \begin{vmatrix} \Delta t & 1 \\ 1 + \Delta t(K_2 K_v/J) & \Delta t(K_1 K_2 K_v/J) \end{vmatrix} \begin{vmatrix}\theta\\ \omega\end{vmatrix}^{n} \tag{3.20}$$

After some algebraic manipulations yielding expressions for the eigenvalues of the amplification matrix, the following stability criterion is obtained:

$$\Delta t \le -\frac{4J}{K_2 K_v} \tag{3.21a}$$

or, substituting the appropriate expressions for J and K_v:

$$\Delta t \le -\frac{2L}{3K_2(\text{thrust\_accel})} \tag{3.21b}$$

A few remarks are in order regarding this inequality. It will be shown in the control system analysis that K_2 must be negative. Increasing the length of the vehicle leads to a larger acceptable time step as the rotational inertia increases faster than the arm of the thrust force. Finally, a larger thrust acceleration imposes a more stringent limitation on the time step, as it amplifies the effect of deviations of the thrust axis from the vehicle centerline.

3.2.6 Implementation Considerations

For the demonstration problem a time step consistent with Equation (3.21b) has been used. The other constraints are the actual execution time and the communication lag among various platforms. A time step of 50 msec has been found satisfactory.

The assumed vehicle characteristics are: mass 1000 Kg; length 10 m; maximum thrust acceleration 20 m/s^2; moment of inertia 8333.333 Kg m^2.

3.3 Control

As already mentioned, the vehicle is represented as a rigid body of a certain length, mass and moment of inertia, with a gimbal-mounted thruster at the rear. This configuration leads to the appearance of a thrust-induced torque. The vehicle must be actively controlled for stability with frequent thruster gimbal angle corrections. A control system is used to maintain the desired vehicle attitude, commanded by the guidance algorithm. The selected control scheme and its analysis will be described.

3.3.1 Control Scheme

For this demonstration, a simple double proportional controller was chosen, as we have found it adequate for the application considered. The controller aims to null out deviations in both pitch angle, θ, and pitch angular rate,ω. The block diagram of the control system, which both steers and stabilizes the vehicle, is shown in Figure 3.4.

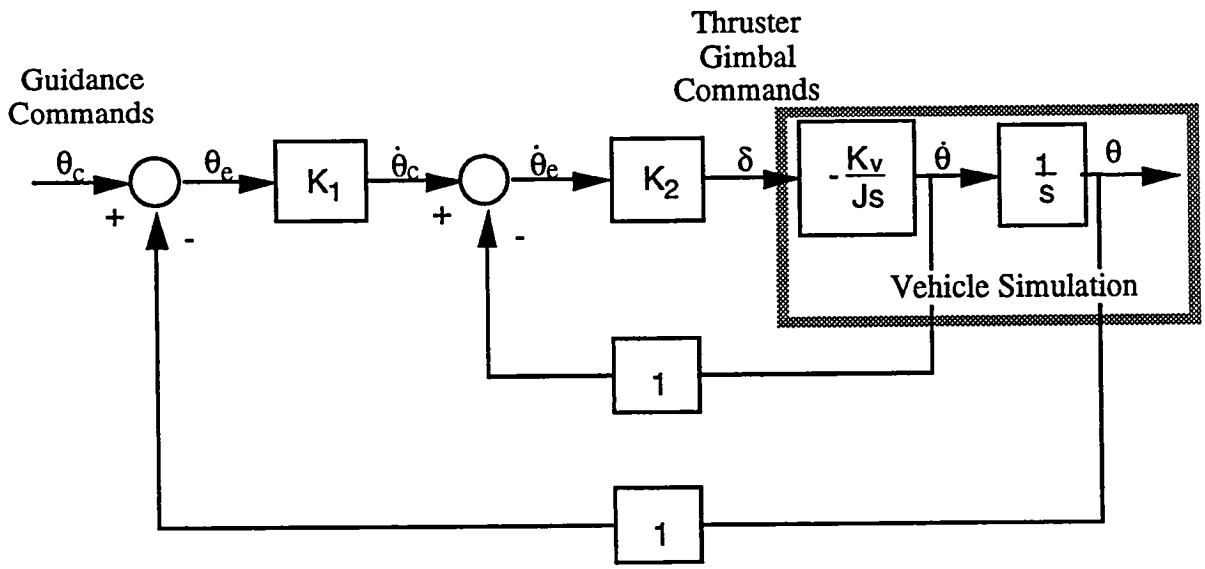


Figure 3.4 Attitude Control Block Diagram

As can be noted, there are two feedback loops, each characterized by a gain factor. These gain factors, K_1 and K_2 , will be determined next, based on stability considerations.

3.3.2 Analysis

The "plant" model is the linearized angular momentum equation, with no drag (consistent with the simulation), written as:

$$\ddot{\theta} = -(K_v/J)\,\delta \tag{3.22}$$

where

$$K_v = (\text{mass})(\text{thrust\_accel})L/2$$

and J is the moment of inertia. Assuming zero initial conditions, the Laplace transform is applied to Equation (3.22). The equivalent transfer function for the "rate" loop is given by:

$$G_{rate}(s) = \frac{-(K_2K_v/J)}{s-(K_2K_v/J)} \tag{3.23}$$

For the pole to be always in the left half-plane, yielding a pure decaying behavior, the condition is simply:

$$K_2 < 0 \tag{3.24}$$

The global transfer function can then be written as:

$$G_{global}(s) = \frac{K_1 G_{rate}(s)}{s + K_1 G_{rate}(s)} = \frac{-\frac{K_1K_2K_v}{J}}{s^2 - \left(\frac{K_2K_v}{J}\right)s - \left(\frac{K_1K_2K_v}{J}\right)} \tag{3.25}$$

Imposing again the requirement of pure decay, it follows that the poles must be both on the negative side of the real axis. Requiring that the discriminant of the denominator be positive (for real roots) leads to the condition:

$$K_1 \le \frac{-K_2K_v}{4J} \tag{3.26}$$

Requiring now that both roots be negative implies that their product be positive, thus K_1 must be positive, given condition (3.24). The gain factors must therefore be selected such as inequality (3.24) and the criteria indicated below

$$0 \le K_1 \le \frac{-K_2K_v}{4J} \tag{3.27}$$

are all satisfied.

Clearly a more extensive control system analysis may be performed, but the foregoing discussion is entirely adequate, enabling us to make a reasonable and robust selection of gain factors.

3.3.3 Interface Definition

The CONTROL module communicates with GUIDANCE, NAVIGATION, SIMULATION and USER INTERFACE as follows:

- From GUIDANCE: desired trajectory angle, γ_d, versus altitude.
- From NAVIGATION: estimated pitch angle, θ^{est},
  estimated pitch angle rate, ω^{est}.
- To SIMULATION: thruster gimbal angle, δ.
- From USER: thruster deflection angle limit, gains for pitch angle and angle rate errors.
- To User: thruster gimbal angle, δ.

The guidance algorithm periodically updates the table containing the desired trajectory angle versus altitude. The control system uses the most up-to-date information available.

3.3.4 Implementation Considerations

A 0.2 rad limit for the gimbal deflection is used. The gains used are: 2.5 s^{-1} for the pitch angle error and -3.333 s for the pitch angle rate error.

The relatively simple, but effective control law is invoked at every simulation integration step. Should a more complex and computationally demanding control algorithm be needed, it may be called less frequently. A more refined analysis would account for the discrete nature of the control process.

3.4 Navigation

The NAVIGATION module receives various sensor outputs and provides an estimate of the vehicle's state, which is then used by the guidance algorithm in updating the desired trajectory and by the control system in generating the proper thruster gimbal actuator commands.

3.4.1 Sensor Conditioning

The NAVIGATION module receives the raw sensors outputs. These signals have to be conditioned, i.e., associated with the proper physical quantity being measured. Currently we assume a one-to-one correspondence, but the modular framework of our technology demonstration permits a straightforward insertion of more realistic conditioning functions.

3.4.2 Analysis

The sensed pitch and pitch rates are turned into estimated values without further processing. In contrast, the sensed translational accelerations are used to infer the vehicle position and velocity. Currently, navigation is assumed to take place solely by "dead reckoning." The sensed accelerations are doubly integrated to obtain an estimate for velocity and position. The equations used are the same as those already described in the Simulation section, except that rather than assemblying the accelerations based on the force balance, "sensed" accelerations are used:

$$x^{est,n+1} = x^{est,n} + \Delta\ tv_x^{est,n} + \frac{\Delta t^2}{2} a_x^{sensed,n} \tag{3.28a}$$

$$v_x^{est,n+1} = v_x^{est,n} + \Delta\, ta_x^{sensed,n} \tag{3.28b}$$

and similarly for the y-direction. Since the computational effort is low, currently the same integration time step as used in Simulation is used here. Obviously, there is no reason to do it more frequently than the Simulation provides new sensed information. On the other hand, a more infrequent update can be used if desired.

3.4.3 Interface Definition

The NAVIGATION module communicates with GUIDANCE, CONTROL, SIMULATION and USER INTERFACE as follows:

- From SIMULATION: sensed horizontal acceleration, a_x^{sensed}
  sensed vertical acceleration, a_y^{sensed} ,
  sensed pitch angle, θ^{sensed}
  sensed pitch angle rate, ω^{sensed}.
- To GUIDANCE: estimated horizontal coordinate (downrange), x^{est},
  estimated vertical coordinate (altitude), y^{est},
  estimated horizontal velocity, v_x^{est} ,
  estimated vertical velocity, v_y^{est} .
- To CONTROL: estimated pitch angle, θ^{est},
  estimated pitch angle rate, ω^{est}.
- From USER: Sensor biases;
- To User: Estimated vehicle state.

In our implementation, the NAVIGATION module also generates the elapsed time and broadcasts it to the other modules.

3.4.4 Implementation Considerations

Currently all sensor biases have been set to zero.

3.5 ASTER Specification

The formulation described in the previous sections constitutes the basis of the specification of the various modules and associated models into ASTER. Figure 3.5 shows the top level block diagram definition of the System as a whole. The main components are shown along with the flow of information among them. These components are then shown individually in the following figures. The Operator Interface (Figure 3.6) is a shell for the Command Mission (Figure 3.7), where all the initializations and default values are provided prior to "launch" and where, after launch, certain parameters can be changed at the user's discretion. Currently the user can specify the desired trajectory prior to launch and input "wind gusts" during the ascent. The specification of the Signal Conditioning module is shown in Figure 3.8. If an actual scaling between the signal and the corresponding physical quantity were modeled, this ASTER transform would incorporate the appropriate function.

The Navigation module is shown next in Figure 3.9. This module generates the estimated vehicle state, used by both the Guidance and the Control modules. The Guidance module, shown in Figure 3.10, includes the FENOC algorithm, the Initial Trajectory and a module, currently only a shell, where the logic for invoking a trajectory update would be placed. The Initial Trajectory, shown in Figure 3.11, simply provides an initial table of angle versus altitude, to be used by the Control right after launch and prior to the completion of the first trajectory update. The FENOC Algorithm transform, shown in Figure 3.12, provides the interface to the actual algorithm, incorporated in NEWTON (see Section 3.1.2). The Guidance_Cmds output is a new table, representing a trajectory updated based on the actual (estimated) vehicle position and velocity.

The Control module is shown in Figure 3.13. The Lookup Pitch Command obtains the desired trajectory angle at the current estimated altitude. The Pitch Control module, shown in Figure 3.14, specifies the control system described in Section 3.3. Finally, the Vehicle Model, the Sensor Models and the Environment Model are shown in Figures 3.15, 3.16 and 3.17, respectively.

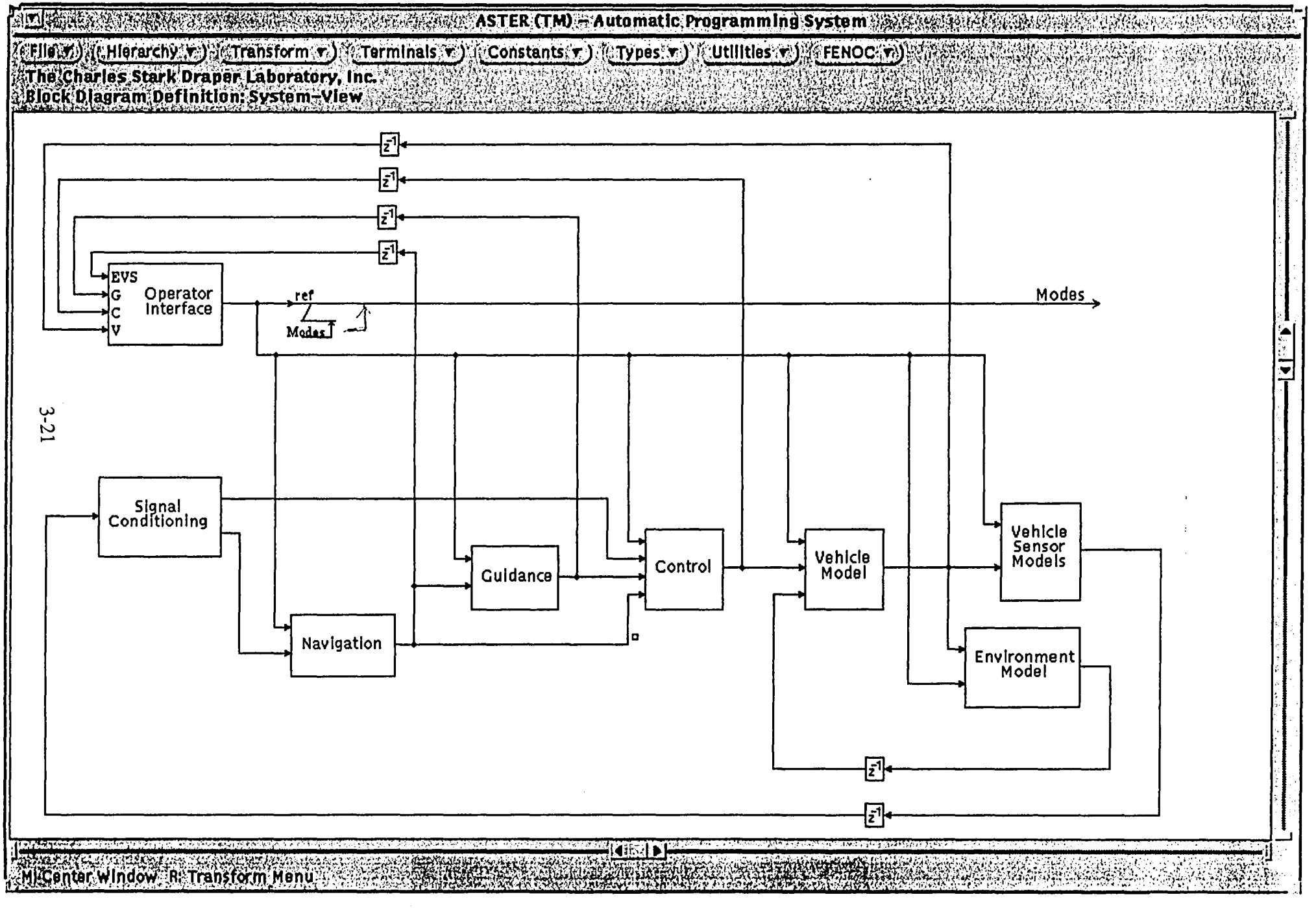


Figure 3.5 Block Diagram Definition: System-View

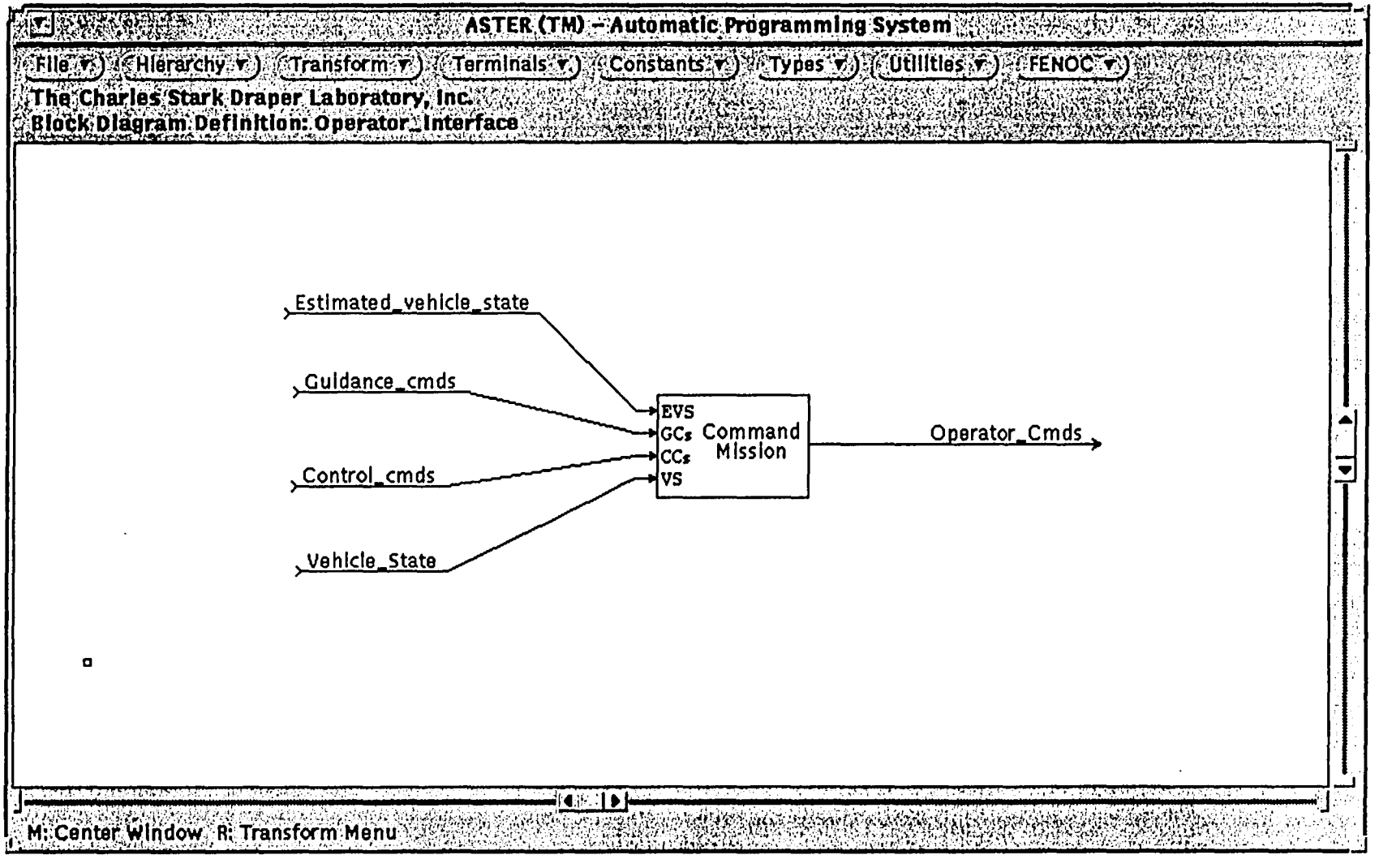


Figure 3.6 Block Diagram Definition: Operator_Interface

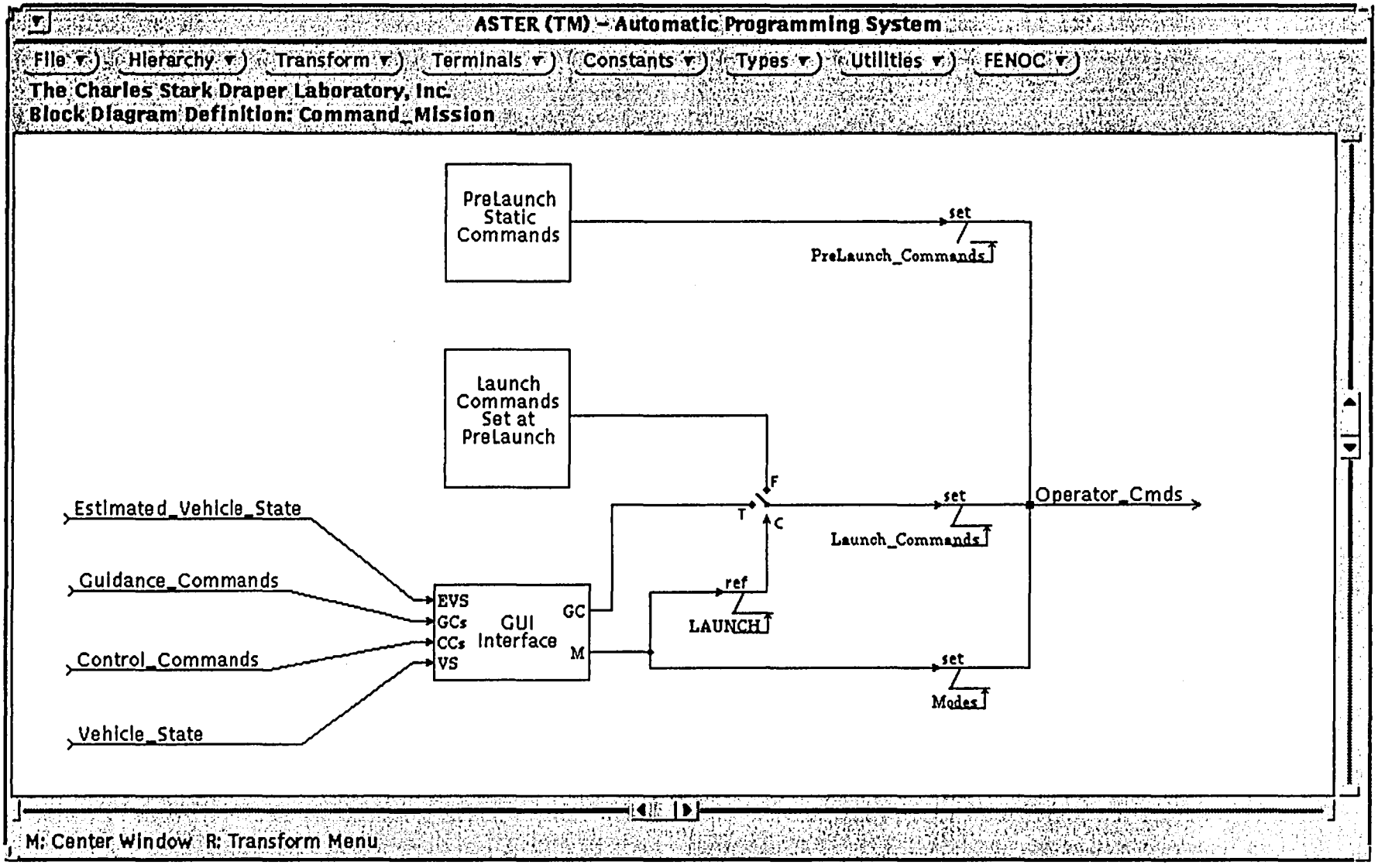


Figure 3.7 Block Diagram Definition: Command_Mission

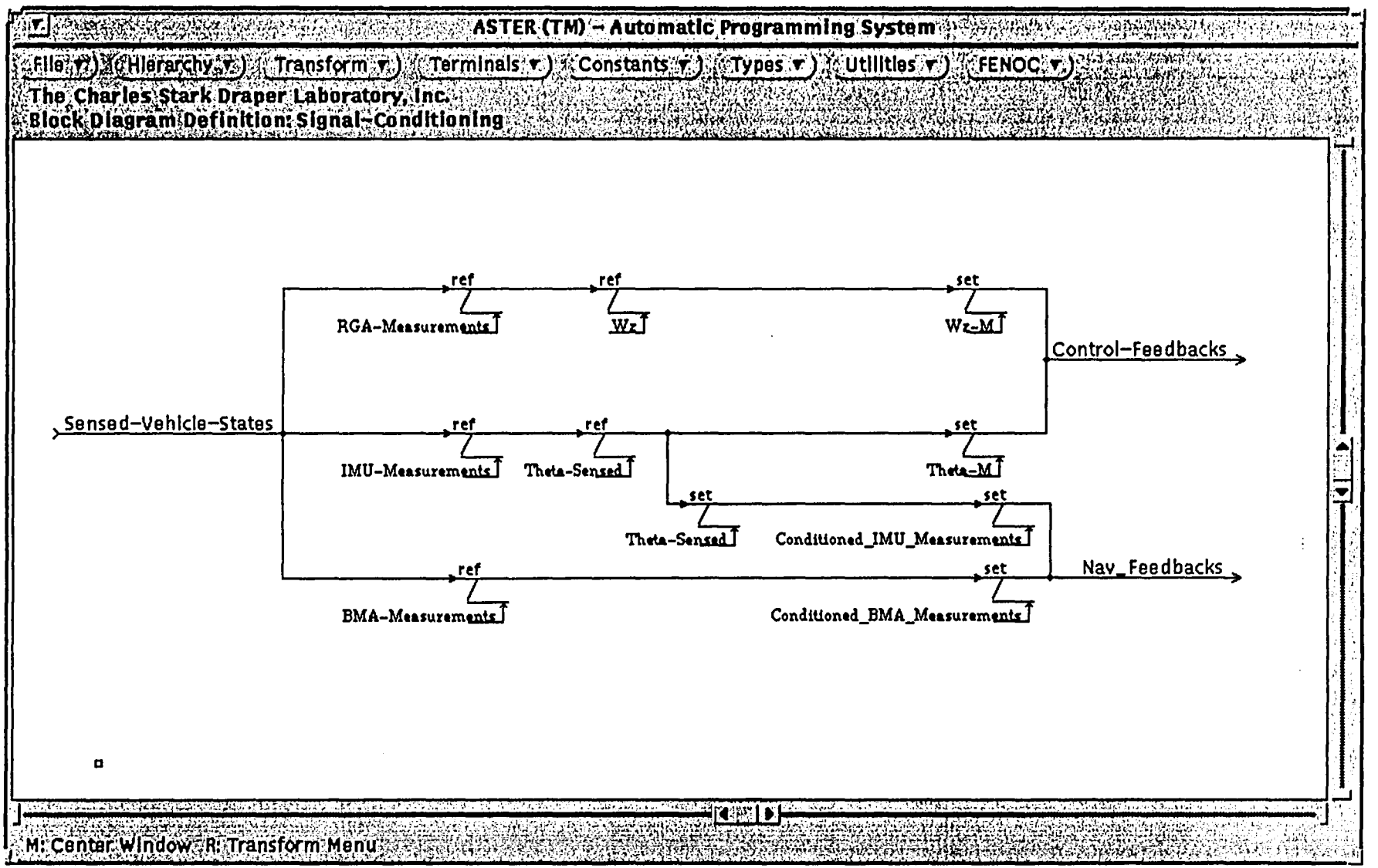


Figure 3.8 Block Diagram Definition: Signal_Conditioning

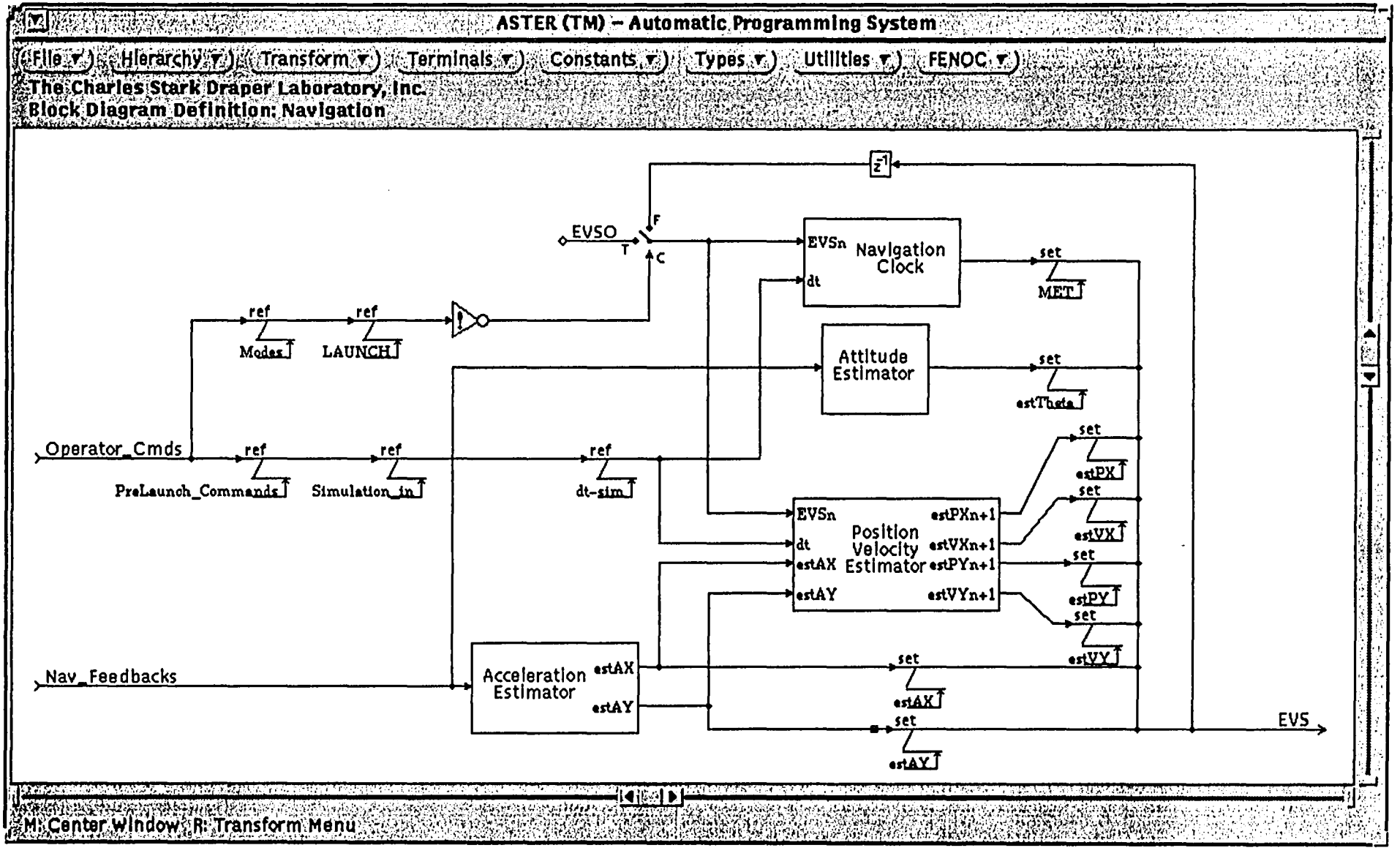


Figure 3.9 Block Diagram Definition: Navigation

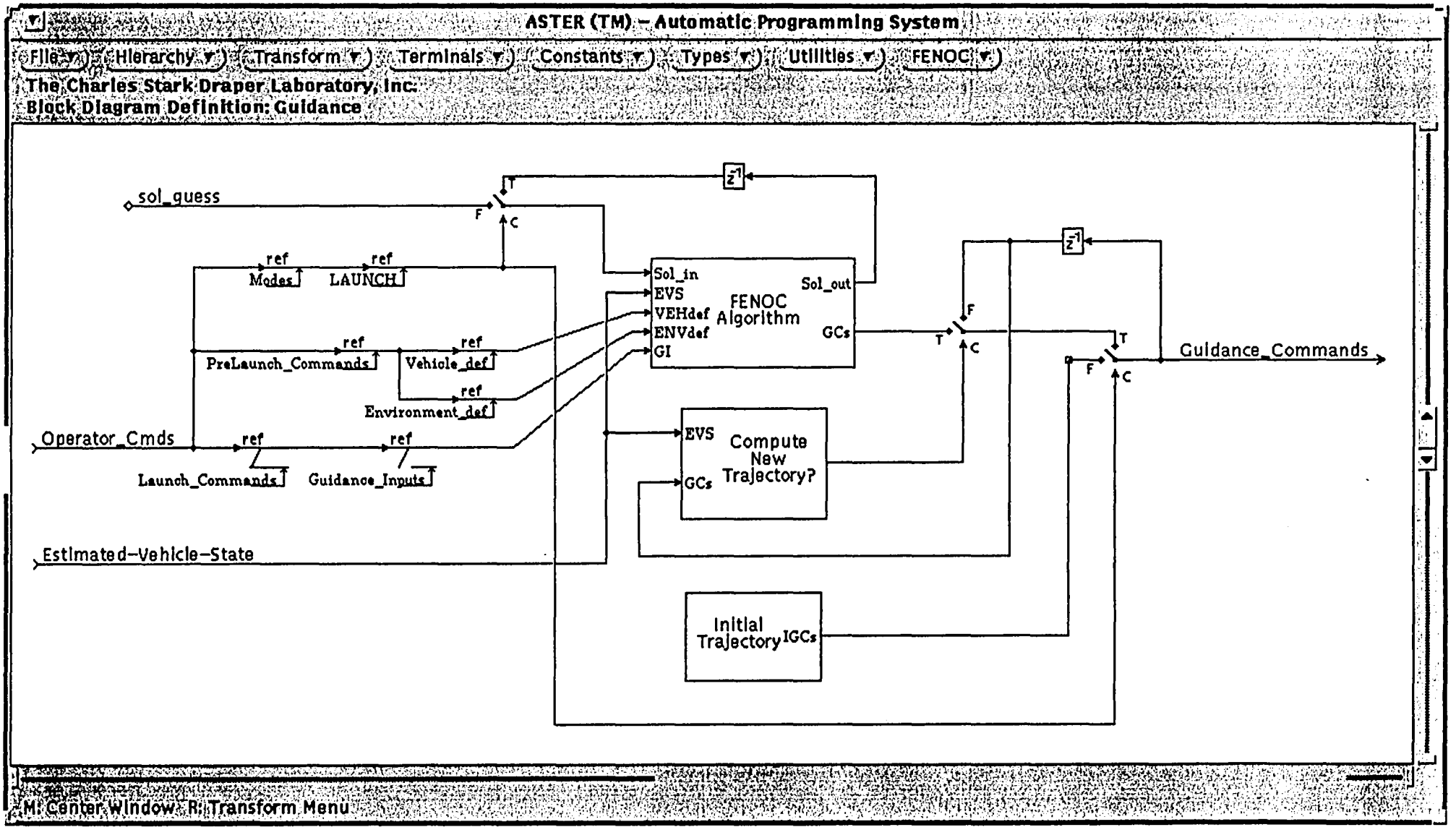


Figure 3.10 Block Diagram Definition: Guidance

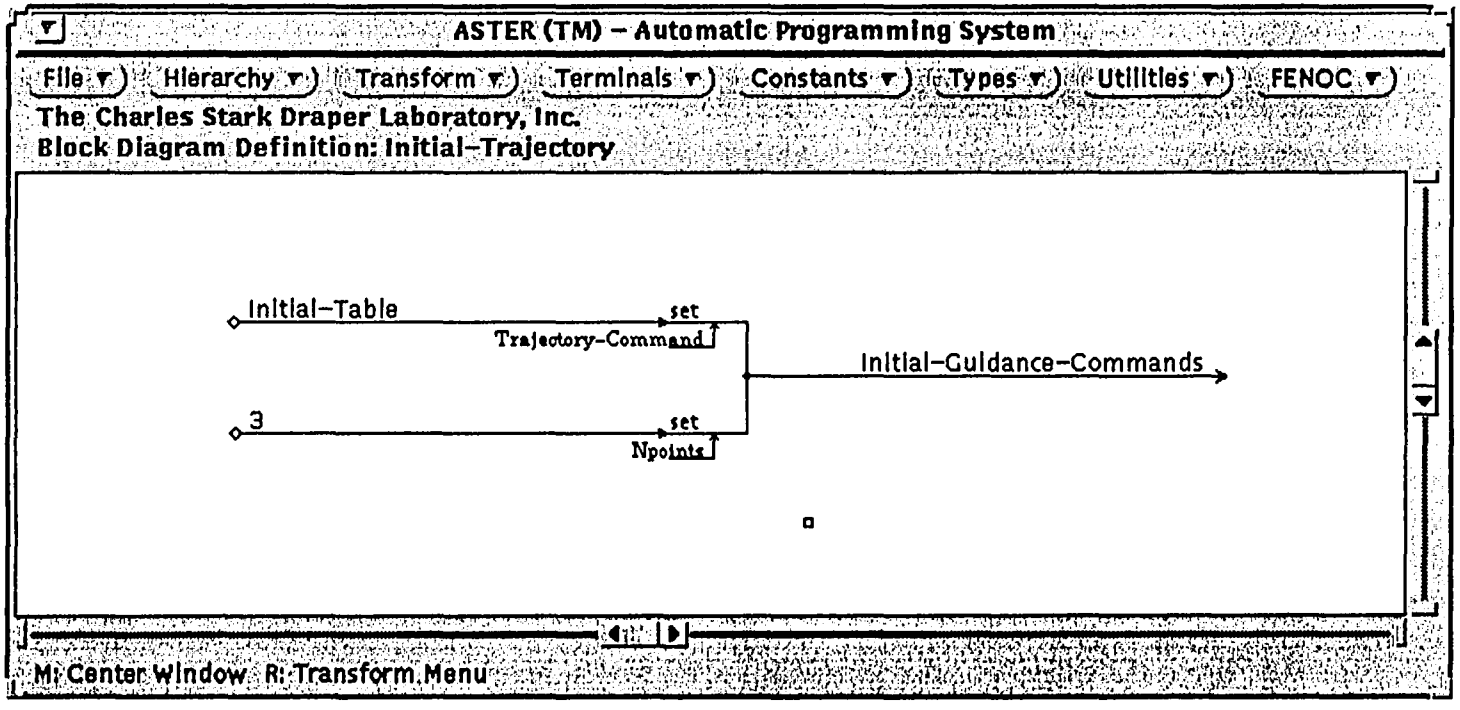
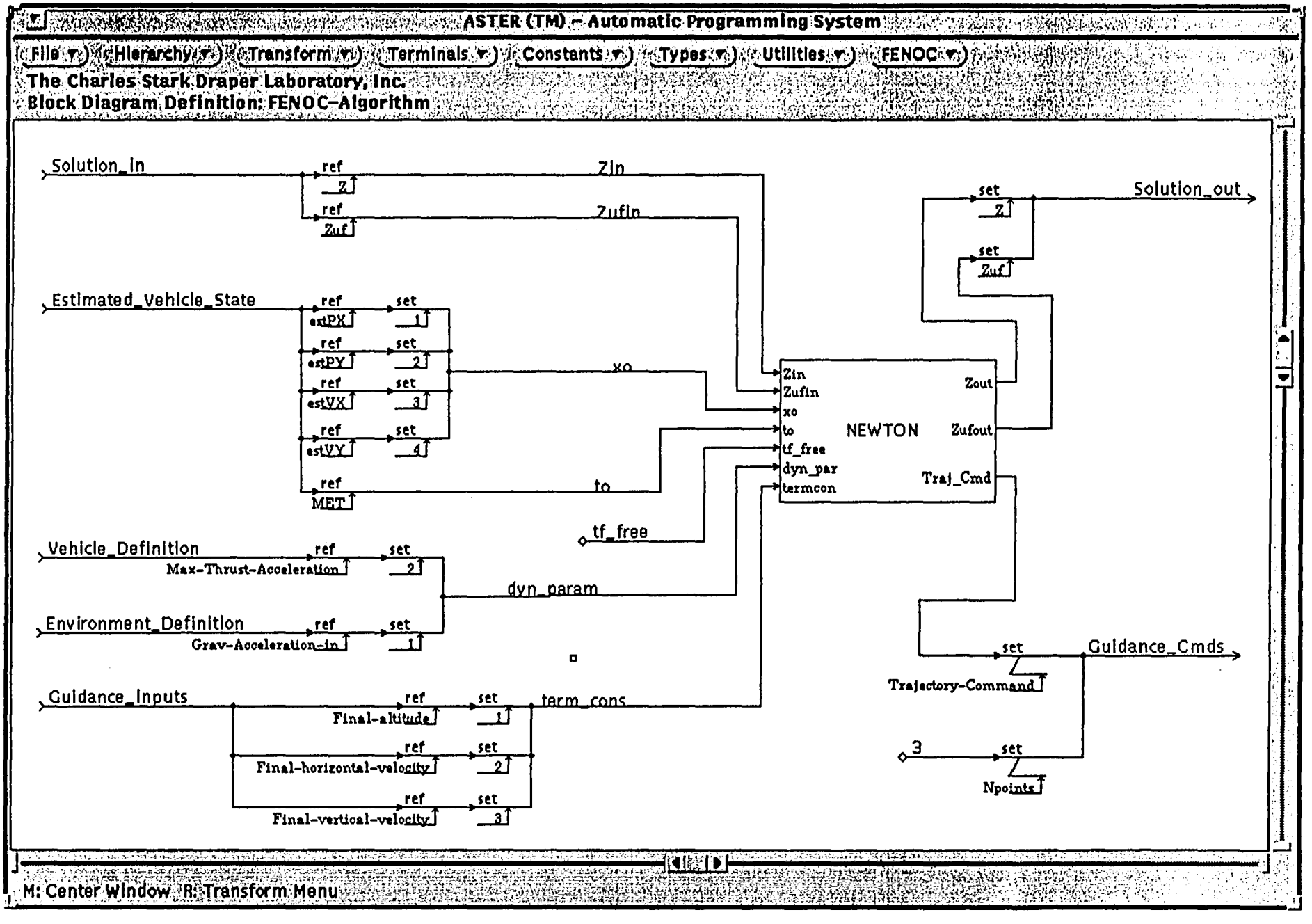


Figure 3.11 Block Diagram Definition: Initial-Trajectory


ASTER (TM) – Automatic Programming System
File
Hierarchy
Transform
Terminals
Constants
Types
Utilities
FENOC
The Charles Stark Draper Laboratory, Inc.
Block Diagram Definition: FENOC-Algorithm
Solution_In
ref Z
ref Zuf
Zin
Zufin
Estimated_Vehicle_State
ref estPX set 1
ref estPY set 2
ref estVX set 3
ref estVY set 4
ref MET
xo
to
tf_free
Vehicle_Definition
ref Max-Thrust-Acceleration set 2
Environment_Definition
ref Grav-Acceleration-in set 1
dyn_param
Guidance_Inputs
ref Final-altitude set 1
ref Final-horizontal-velocity set 2
ref Final-vertical-velocity set 3
term_cons
Zin
Zufin
xo
to
tf_free
dyn_par
termcon
NEWTON
Zout
Zufout
Traj_Cmd
set Z
set Zuf
Solution_out
set Trajectory-Command
Guidance_Cmds
3
set Npoints
M: Center Window R: Transform Menu

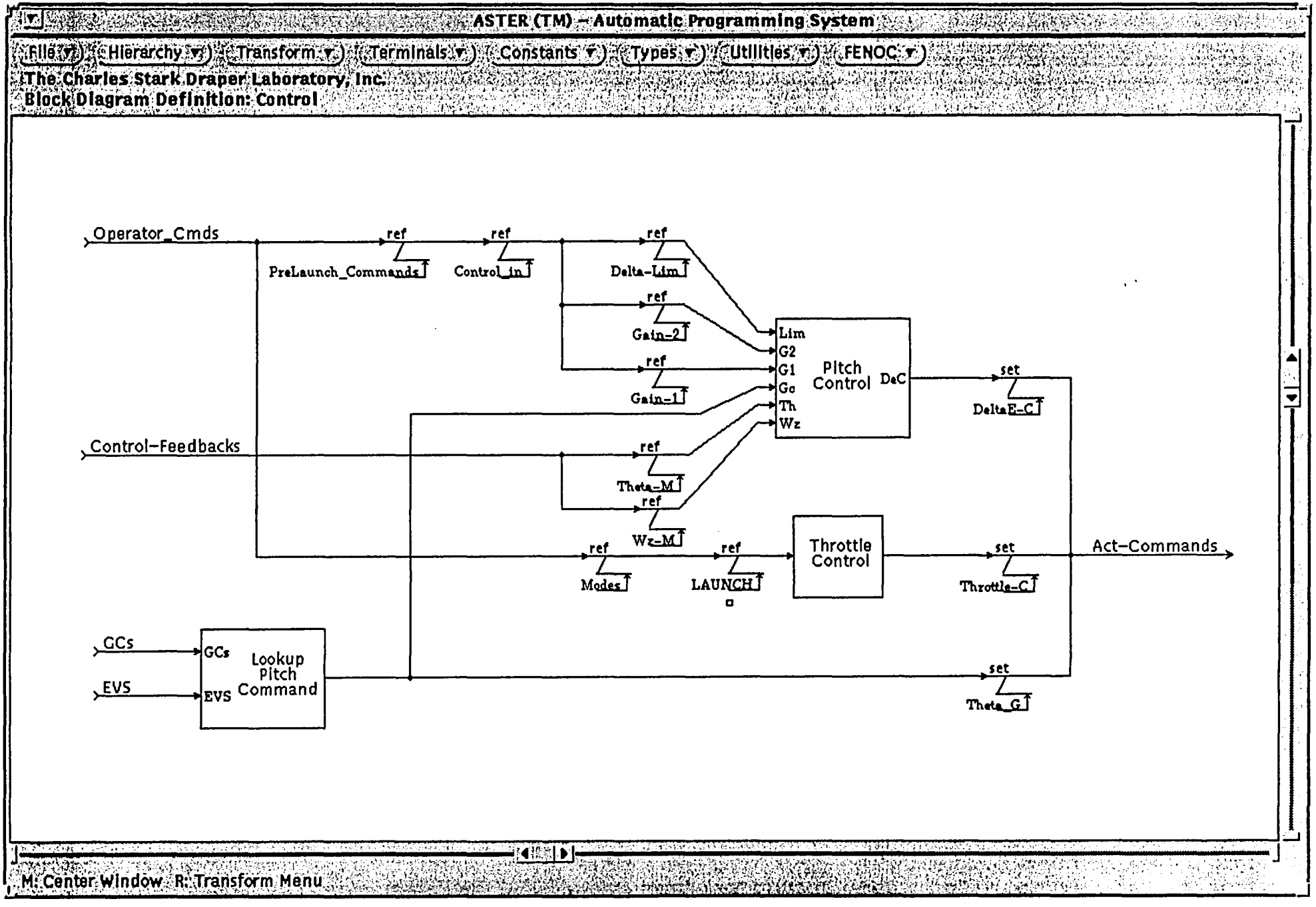


Figure 3.13 Block Diagram Definition: Control

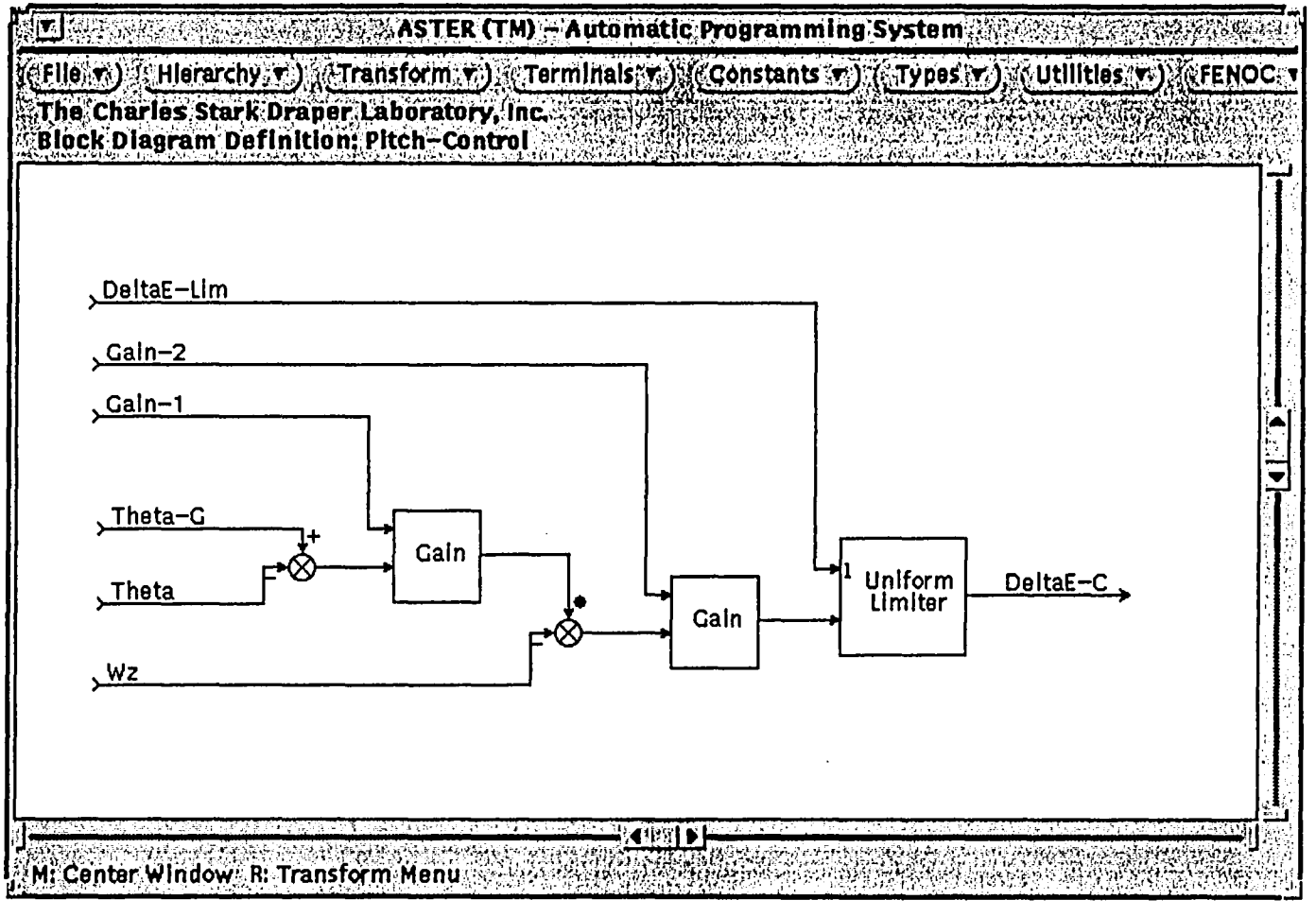


Figure 3.14 Block Diagram Definition: Pitch Control

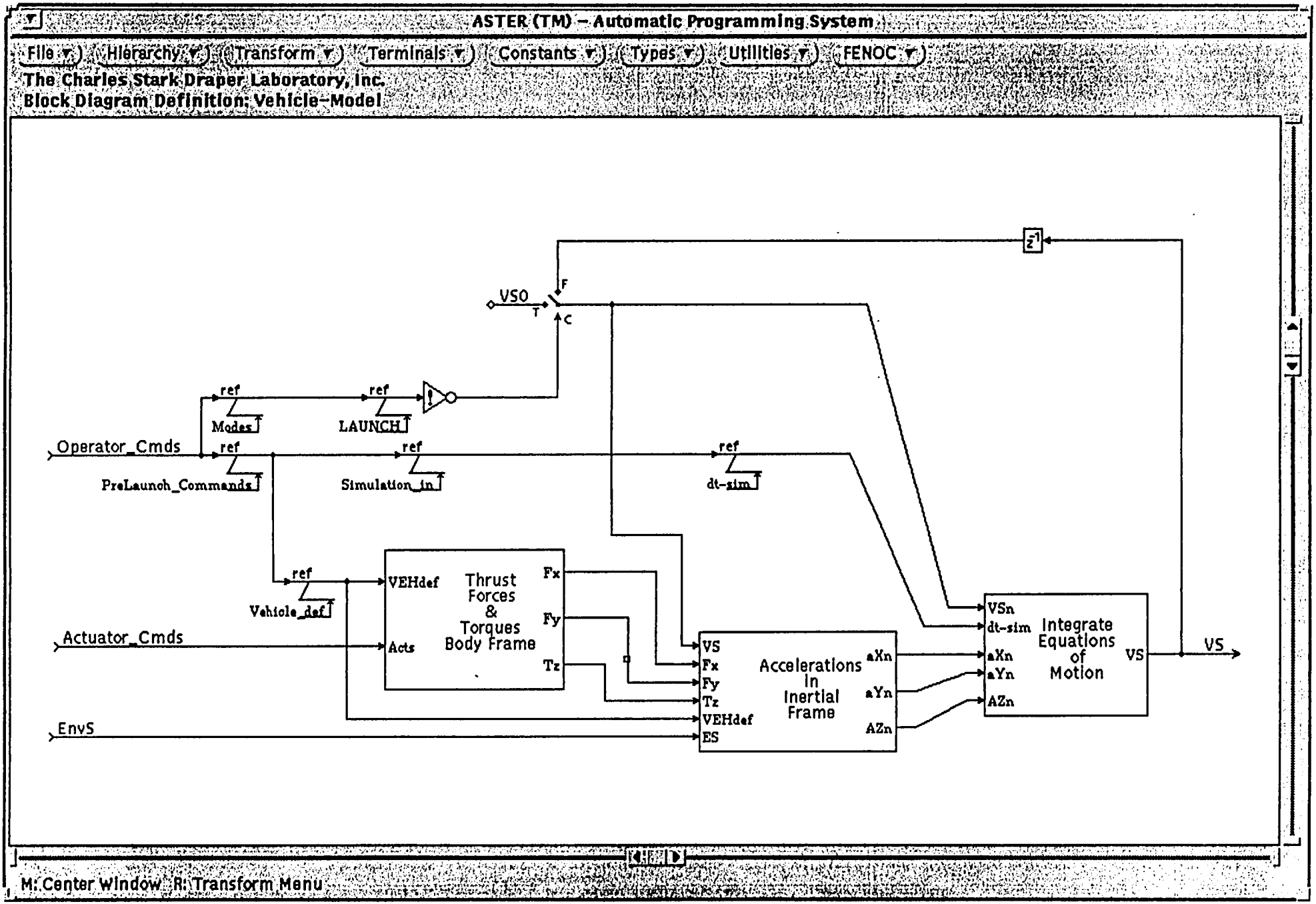
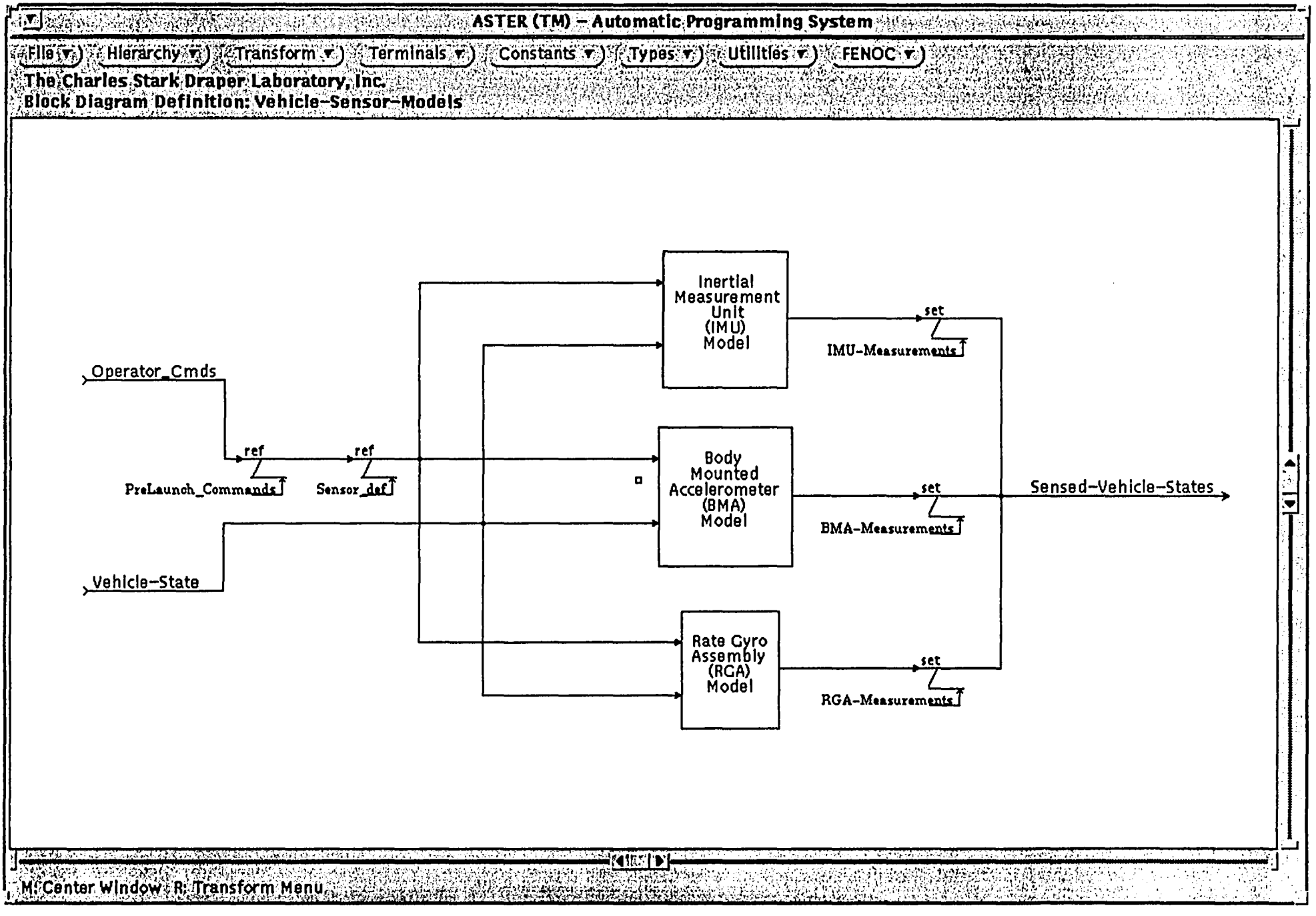


Figure 3.15 Block Diagram Definition: Vehicle Model


ASTER (TM) – Automatic Programming System
File
Hierarchy
Transform
Terminals
Constants
Types
Utilities
FENOC
The Charles Stark Draper Laboratory, Inc.
Block Diagram Definition: Vehicle-Sensor-Models
Inertial Measurement Unit (IMU) Model
Body Mounted Accelerometer (BMA) Model
Rate Gyro Assembly (RGA) Model
Operator_Cmds
ref
PreLaunch_Commands
ref
Sensor_def
Vehicle-State
set
IMU-Measurements
set
BMA-Measurements
set
RGA-Measurements
Sensed-Vehicle-States
M: Center Window R: Transform Menu

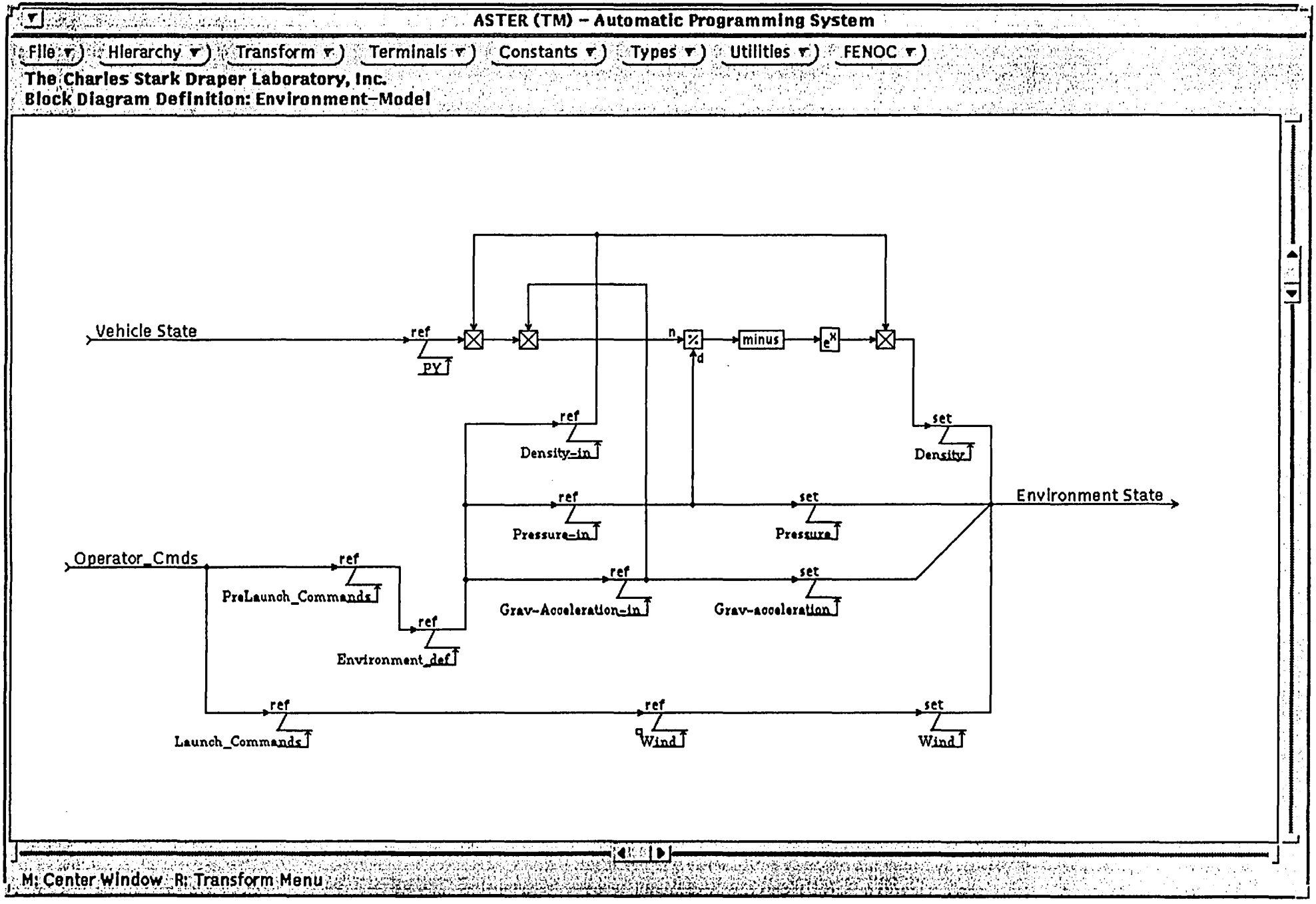


Figure 3.17 Block Diagram Definition: Environment-Model

4.0 ASTER

4.1 Technical Background

This section describes the components of ASTER's automatic programming system that are illustrated in Figure 2.1. These components are the application engineer's user interface, an automatic software designer, code generator, and document generator. The following sections describe each of these components.

User Interface

The ASTER user interface was designed with the purpose of specifying integrated systems of engineering algorithms instead of software designs. This specification emphasizes the functionality and interaction of algorithms in order to convey their meaning among engineers. Software designs by contrast do not clearly convey the meaning of algorithms and systems of algorithms. Instead, software designs dominantly reflect constraining characteristics of the execution environment such as computational resources, memory resources, input/output resources and resource connectivity.

Systems of algorithms are specified through extensive use of engineering block diagrams. We refer to the computational aspects of a diagram as "transforms" since they explicitly transform inputs into outputs with no hidden side effects. We refer to the data aspects of a diagram as "signals" that carry information from one transform to other transforms. Hierarchies of both transforms and signal types can be built either bottom-up or top-down. For bottom-up design, predefined sets of building blocks for both transforms and signal types are supplied. In the case of transforms, these are called primitive transforms and are comprised of such things as add, subtract, multiply, divide, abs, switch, etc. For signal types, these are called predefined types and include integer, float, character, string and boolean. For top-down design, engineers need only to specify the input and output characteristics of a transform before using it in an engineering block diagram. The details of the transform's block diagram and processing can be deferred until a later time.

Automatic Software Designer

The automatic software designer takes as input the object-oriented, functional form of the specification and determines a generic, procedural form which takes into consideration characteristics of the execution environment. During the automatic design process, block diagrams are converted into procedures, functions or in-line code depending on their usage. Each component transform in an engineering block diagram becomes either a statement or a block of statements. A statement consists of an assignment to one or more variables and a call to a procedure or function. Input, output and constant terminals are converted into their respective classes of variables. The automatic designer also generates variables when it is necessary to implement state. State variables appear as a result of feedback loops in

specifications. In addition, local variables may be needed to reflect the connectivity in specifications. Connectivity is also used to determine the execution order of statements. Traversals of diagram connectivity from both outputs to inputs and inputs to outputs are performed to determine which execution sequence of statements, which may execute in parallel, and which are conditionally executed.

Automatic Code Generator

Two automatic code generators, Ada and C, exist. Each code generator takes as input a generic, procedural form of a software design and creates source code in the syntax of the corresponding target language. Both code generators produce code that is consistently organized and well commented.

The Ada code is hierarchically structured and creates one Ada function and procedure for each function and procedure in the software design.

Since C is not a hierarchical language and recognizes only function program blocks, the C generator flattens the definition hierarchy and creates functions for both functions and procedures in the software design.

Automatic Document Generator

The document generator takes as input the same object-oriented, functional form of the specification used to generate a software design and outputs a fully collated document which includes title page, table of contents, list of figures, sections and subsections and an index. Each section and subsection contain both text and graphics which reflect the specification. The entire document is composed and assembled with no manual intervention.

4.2 Algebraic Transform Engine Overview

The Algebraic Transform Engine (ATE) provides ASTER with a utility for defining transforms algebraically, using text. The ATE consists of a language parser and several interfaces to the rest of ASTER and to the user.

4.2.1 Overall Structure

The core of the ATE is designed as a meta-language that encompasses selected portions of the languages that we intend to support. This approach provides for addition of new languages at declining marginal cost, because of the extensive sharing of facilities among the several languages. Currently, only some portions of MATLAB are implemented, but it is possible to add modules for any other language, such as Ada, FORTRAN, C, or JOVIAL.

The ATE now provides services to various subsystems within the ASTER environment, including the ASTER Automatic Programming System, where the ATE is used for defining transforms algebraically. These other subsystems are viewed as *clients* of the ATE, which serves them through a well-defined interface. Figure 4.1 shows the general architecture of the ATE.

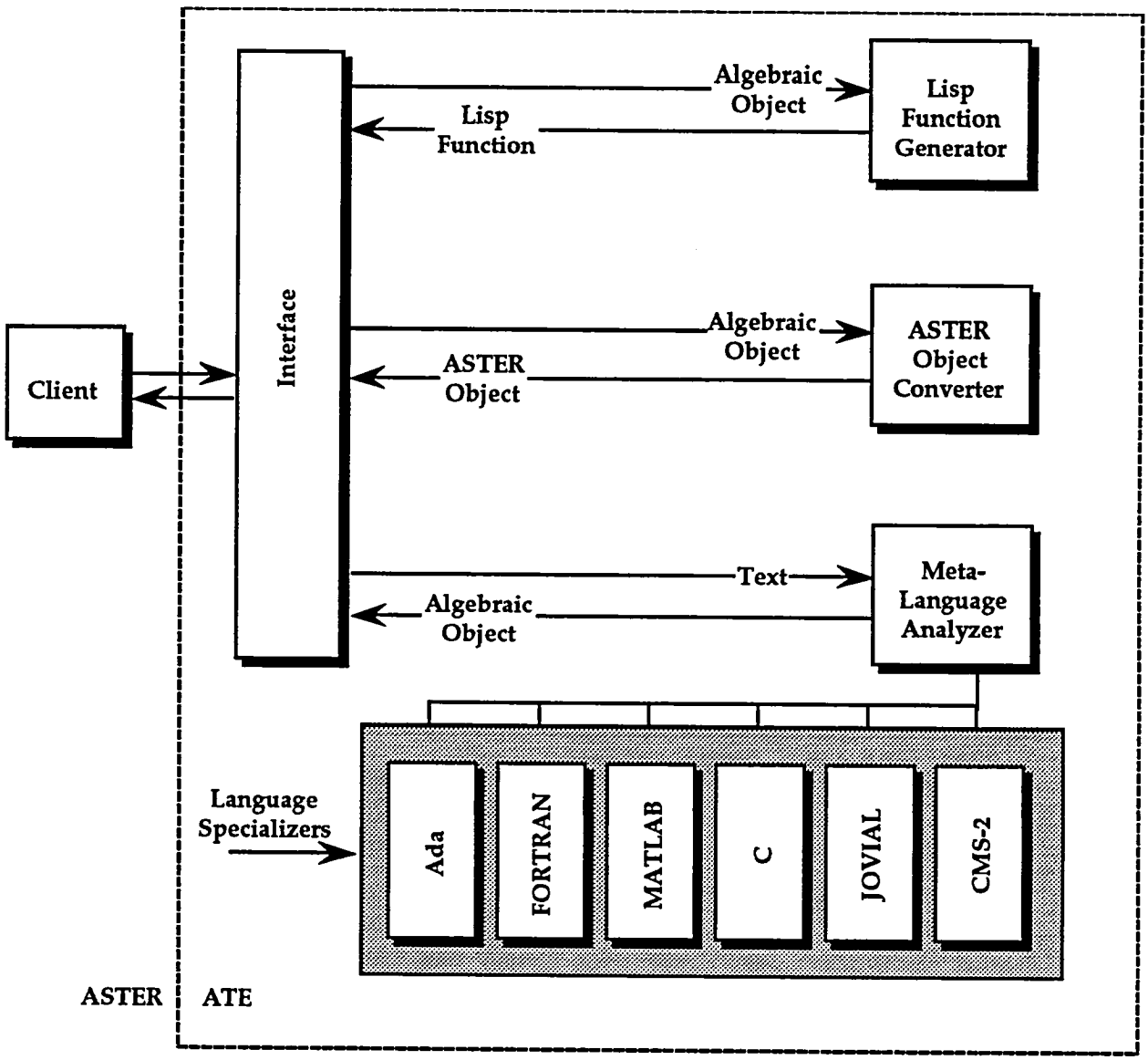


Figure 4.1 The Architecture Of The ATE

Clients are served by the ATE through the module labeled **Interface.** Although each client obtains services by invoking Lisp functions, the object-oriented design of the ATE means that successive clients need not be employing the same language. That is, a client can request FORTRAN services immediately after another client has requested MATLAB services. No overhead cost is incurred when changing client language contexts.

Two kinds of objects can be processed by the ATE:

Text The ATE can parse text with respect to any of the implemented languages. The source of the text can by the keyboard or a file. The result returned is an **Algebraic Object.**

Algebraic Objects From **Algebraic Objects,** the ATE can produce Lisp functions for use in the Automatic Testing System, or ASTER Objects for use as ASTER transform definitions.

The ATE is designed so that the specializations associated with the language used to express an algebraic transform definition are modularized. In effect, the ATE is actually a class in the sense of an object-oriented language. The various instances of that class correspond to ATEs for each language. In this analogy, the ATE instances inherit the properties of the ATE class—that is, they share an assortment of common facilities. This sharing makes it very economical to extend the set of languages that the ATE can support.

4.2.2 General Operation

The ATE performs two basic functions at this time:

- It can parse a textual specification of an algebraic transform definition.
- It can process the result of that parsing, to convert it into ASTER's internal representation of a transform definition.

The first step is called *analysis*, and the second is called *installation.* The user of ASTER enters an algebraic transform definition into ASTER by following the steps enumerated below.

1. Create a blank algebraic transform definition.
2. Insert the text that defines the transform.
3. Analyze the text. Assign types, as needed, to any variables that result from the analysis.
4. Install.

In the Analysis stage, the ATE produces its own internal representation of the text provided by the user as the algebraic transform definition. In the Installation stage, the ATE converts this internal representation to the form of the internal representation of block diagram definitions. This form is understood by the ASTER Software Designer, and it is from this block diagram form that code is generated.

4.3 ATE Implementation and Design Trade-Offs

For the purposes of this effort, we implemented the Interface module, the converter modules, and the MATLAB language specializer. Section 4.3.1 describes the design

philosophy of the MATLAB language specializer. Section 4.3.2 describes the mechanism for making type declarations in ASTER style MATLAB. Section 4.3.3 describes the ATE installation mechanism.

The MATLAB scripts developed by Martin-Marietta, which implement the FENOC algorithm, are included in Appendix A of this report. The modified form of these scripts, suitable for use in ASTER, are included in Appendix B. Detailed guidelines for performing the required modifications are included in Appendix C.

4.3.1 The MATLAB Language Specializer

Although we followed the MATLAB language specification fairly closely, there are some important differences between ASTER style MATLAB and MATLAB as provided by MathWorks. These language differences arise from differences in the working paradigms of the ASTER and MATLAB environments. MATLAB is, by nature, a programming language, while ASTER is, by nature, a workbench for capturing a functional design. This fundamental difference has important consequences for the engineer.

To carry out this work, we developed a set of guidelines for converting MATLAB code to ASTER style. To write MATLAB code in ASTER style, or to convert existing MATLAB code into ASTER style, one must define the modularity of the code to correspond with ASTER's paradigm, and one must replace certain common idiomatic constructions that conflict with the ASTER paradigm. ASTER is essentially a functional design specification language. Functions accept arguments and return values. Unlike a programming language, which permits the storage of values in cells of memory, ASTER's transforms merely operate on their inputs. Many of the restrictions below follow from this basic principle.

Examples of the more important conversion guidelines follow.

Clearly Separate the System From the Test Bench

> In most MATLAB script sets there are two basic subsystems. The first is the system one is designing. The second is a collection of routines needed to measure the behavior of that system in response to various stimuli. The conversion of the system will be dramatically simplified if one clearly separates the system itself from the test bench. For example, one may have an initialization file that performs a variety of tasks. Some of these are related to the system, some to the test bench. Split that file into two parts.

ASTER	*MATLAB*	*Comment*
No Inherent Ordering	**Inherently Ordered**	The signal flow from component to component in ASTER provides the framework in which the engineer thinks about the design. Nevertheless, there is no inherent time ordering of the various signals in a project—all signals are considered to be valid at once. On the other hand, because MATLAB is a programming language, there is a well-defined flow of control, readily deduced by examining the code.
Strongly Typed	**Untyped**	MATLAB requires no type declarations, while ASTER requires all signals to be typed.
Compiler	**Interpreter**	ASTER "programs" can run only after being compiled. MATLAB runs only in interpreted mode.
Graphical User Interface	**Textual User Interface**	ASTER programs are interconnections of transforms, in which each graphical manifestation of a transform corresponds to a subroutine call. MATLAB expresses the same concept in a textual way using a conventional programming syntax.

Table 4.1 A Comparison of ASTER and MATLAB Working Paradigms

Avoid Assignment When a Nested Function Call Will Do

Sometimes, for the sake of readability, MATLAB authors assign a value to a variable when that value is computed as a function call, even when the result is needed in only one place.

Avoid Global Variables For Input-Dependent Quantities

From time to time, one finds a need to test a condition, and to use the results of that condition later in the code. For example, one might determine whether a particular input is non-zero, and use that information in several places. In MATLAB, authors sometimes store the result of that test in a variable, and then refer to the variable later on. Typically, this variable is a global.

In ASTER, one can do the same thing, slightly differently. The signal that represents the result of the test can easily be fed as an input to the parts of the project that need it. In this role, the signal corresponds to a function argument in MATLAB. Thus, for most situations in MATLAB in which global variables pass information to the interior of subroutines, in ASTER one passes signals around as inputs.

Unify Any Split If Statements

From time to time, **if** statements are used with only a single branch active. Usually, this happens when one has already set a value, and wants to change it if certain conditions apply. For example,

```
...
Emergency = false
if slewRate > 30
  Emergency = true
end
...
```

This can also be written as

```
...
if slewRate > 30
  Emergency = true
else
  Emergency = false
end
...
```

In effect, the former is a "split" if statement. The latter form is more natural for ASTER. Rewriting code into this form often eliminates apparent re-using of variables.

Re-Using Variables Is Not Allowed

In MATLAB, one is perfectly free to assign a value to a variable several times. In certain circumstances, this cannot be done in ASTER. We first enumerate the circumstances in which one *can* reassign a value to a variable:

For loop — In a **for** loop, there is an iteration variable that is updated on each traversal through the loop. This "reassignment" is supported in both ASTER and MATLAB.

While loop — In a **while** loop, one may establish one or more counters to be incremented on each traversal through the loop. This is supported in MATLAB, but not yet supported in ASTER, unless the counter is a component of an aggregate.

Avoid all other re-use of variables. In particular, if one stores results in temporary variables, it is best to make certain that each temporary variable is used only once.

Declare Variable Types and Functions

MATLAB runs in interpretive mode. It always knows the type of every variable it deals with, because it can just look at it to find out. ASTER has no such luxury. To enable ASTER to understand what is meant, one must declare types of variables and functions. See the next section for details.

No Need For Pre-Allocating Space For Matrices

In some MATLAB programs, one may find constructions involving matrices that appear to be double assignments. Actually, they are **merges** in ASTER, and they are perfectly legal in ASTER-style MATLAB. Although they are legal, they are also unnecessary in ASTER, and one may wish to remove them.

Limitations on Iterations

Only one iteration per transform definition There can be only one iteration in any given algebraic transform definition. That means only one statement of the type "while" or "for". To work around this constraint, break up any transform definition that contain more than one iteration. If there are N iterations, make N transforms. One of them can contain manifestations of the other N-1.

Isolate all iterations Iterations must be isolated from code that is not meant to execute as the body of the iteration.

Isolate all iterations This constraint is more difficult to explain. In a transform that contains an iteration, the MATLAB code can classified as Preliminary Computation, Initialization, Iteration, and Finalization. First we explain what is meant by these classifications, and then we describe the constraints and the work around.

Preliminary Computation This code is not meant to execute as part of the body of the iteration, nor does it establish initial conditions for variables that are computed as part of the iteration.

For example, suppose we want to iterate over the elements of the k^{th} column of a matrix. To do that, we must first compute the value of k from the inputs to the transform. Then, in the body of the iteration, we access the matrix elements using the value of k we have computed.

The computation of k as described above is an example of a preliminary computation. It is not meant to execute as part of the body of the iteration, but it is used during the iteration.

Initialization This is the code that is directly involved in the iteration to the extent that it establishes initial conditions. For example, it may be used to determine the upper bound on the iteration variable of a **for** loop.

Iteration This is the code that is directly involved in the body of the iteration.

Finalization This is code that is meant to execute after the iteration. For example, if we continue with our matrix/k^{th} column illustration, suppose that we want to perform a matrix multiplication with that matrix after we have iterated over the k^{th} column. That code is in the category of Finalization.

The transform that contains the Iteration code can also contain Initialization. But it cannot contain either the Preliminary Computation or the Finalization. If it does, those portions must be removed, and inserted into one or more new algebraic transform definitions. Any such code that is not removed will be executed as part of the iteration body.

Typically, only one new transform definition is required. It would contain both the Preliminary Computation and Finalization code, and a manifestation of the transform that contains the Iteration and Initialization code. It is possible that one may prefer to move the Initialization code into the same definition that has the Preliminary Computation code. One has a certain amount of freedom in making these decisions.

4.3.2 Declarations in ASTER Style MATLAB

Two kinds of declaration statements are provided. They are the MATLAB function statement and an ASTER declare statement.

The MATLAB function statement is available as an aid to converting existing MATLAB scripts. Since we wanted to run the scripts in MATLAB as well as ASTER, we required that ASTER allow this statement. The function statement in ASTER is completely compatible with that of MATLAB, and must appear first in the body of the definition of the Algebraic Transform Definition.

One can declare symbols to be transforms or variables of various types. In some cases, such declarations are required. An example of a declaration is:

```
declare(state_vector, u, z, w)
declare(float, y)
```

This example declares u, z, and w to be of type state_vector, and y to be of type float.

Each symbol that appears in an algebraic transform definition can have a number of properties. For example, it can be pre-defined by MATLAB, or it can be defined by the user. It can be a matrix, or it can be a scalar. Declarations, if they occur, must occur before any other statements of the body of the transform definition, except that the function statement, if it appears, must occur first. In only one circumstance is a declaration required. When one has a transform and a matrix (or vector) that have the same name, one can refer to either one in the body of an algebraically defined transform, but one must declare which one is meant. Only one such declaration is permitted per definition. That is, one can*not* make a declaration that says "I mean the transform", then later on in the same definition say "I mean the matrix."

4.3.3 The ATE Installation Mechanism

The ATE Installation Mechanism consists of the submodules of the ATE that actually perform the transformation of the parsed form of the Algebraic Transform Definition. Installation is performed in two passes.

Terminal Identification In this phase, variables that have bee identified as inputs, outputs or constants are associated with ASTER artifacts called terminals. Terminals in ASTER are sources or sinks of signals.

Transform Instantiation In this phase, algebraic or logical structures in the parsed form of the textual definition are examined, and then used to drive the mechanism that instantiates component transforms such as adders, multipliers, switches, and so on. The ATE actually traverses the parsed structure from output to input, building the internal ASTER representation as it goes.

The terminal identification phase uses information provided by the user after the equations are analyzed. For example, the ATE may guess that a particular variable is an output, but the user may assert that it is a local variable, using the graphical user interface. This correction is incorporated into the ASTER internal form during the terminal identification phase.

The Transform Instantiation phase is capable of dealing with user defined transforms as well as with ASTER primitives. It also deduces the topology of **if** statements, iteration structures and the manipulation of composite data types.

4.4 Experience

4.4.1 Converting MATLAB Scripts to ASTER Style

To convert the existing MATLAB code to ASTER style, we had a choice of approaches.

A **Reimplement** Sit down in front of ASTER and re-implement the scripts using the non-ASTER style version of the MATLAB scripts as a reference.

B **Convert, then Reimplement** Convert the existing scripts to ASTER style, and then introduce the converted scripts into ASTER.

Method A (Re-implement) may seem like less work than Method B (Convert then Re-implement) but we felt that may have been an illusion. The conversion process can be complicated for even a moderate-sized project, and there is substantial opportunity for error. Method B has an important advantage for reducing errors. That is that after one has converted the code to ASTER style, one can execute it in MATLAB and compare the results to the original project. If the two systems behave differently, it is clear that something was inadvertently changed during the conversion.

Thus, Method B provided an opportunity for verifying much of the work we had to do. This provided an important advantage. Note that when one uses Method B, and differences arise in comparing the behavior of the original system to the behavior of the

system rewritten in ASTER-style MATLAB, one may not immediately conclude that the ASTER-style MATLAB code is incorrect. It may be that the original system is incorrect, or that both are incorrect. Bringing the two systems into alignment, reconciling their differences, may require changes in one or the other or both. Thus, Method B for converting to ASTER presents the opportunity to actually *improve* the reliability of the code.

4.4.2 Converting ATE Representation to ASTER Block Diagram; Representation

The architecture as implemented is deficient in one important respect. Because we elected to convert the ATE internal representation into the ASTER block diagram representation, and to generate code and software from that, certain restrictions appear at user level that would not otherwise be necessary. An alternative design in which code is generated directly from the ATE internal representation might not have resulted in these restrictions. Let us call the design we did implement, in which we first convert the ATE internal representation into block diagrams, the Block Diagram approach (BD), and the alternative design, in which we generate code directly from the ATE algebraic expressions, the Algebraic Expression (AE) approach.

Specifically, consider the restriction described above on the number of iteration constructs that are permitted in a transform definition. Clearly, both Ada and C permit multiple iterations, even nested iterations within any one definition. To allow ASTER users this capability in the BD approach, two extensions must be implemented in ASTER. ASTER's graphical block diagram language syntax must be extended to support more complicated iteration topologies, and the ASTER Software Designer must support those extensions. On the other hand, complex iterations could be supported for algebraic definitions in the AE approach if only the latter extensions to the software designer are implemented. That is, by extending the software designer to operate on ATE constructs directly, we can remove many of the restrictions that arise from difficulties in extending the graphical block diagram language.

In effect, if we had elected the ERE approach, we would have moved a step closer to implementing a MATLAB to Ada translator. Such a tool would be one of enormous value.

5.0 HOSTING OF AGN&C ON AIPS

5.1 Interprocess Communication Protocol for AGN&C

A distributed system such as that employed to launch a payload into space poses some fundamental communications problems. In such a system, hosts must communicate by sending and receiving messages over some physical medium. Whether the underlying medium uses point-to-point communication or network communication, the problem of ensuring reliable message transmissions among the hosts must be solved. The threats to which a distributed system may fall prey are of two basic types: passive and active. Both types may occur as a result of some hardware or software fault or as a result of a hostile intrusion. Passive threats result in the non-delivery of a message. Active threats result in the delivery of a modified, retransmitted or inserted message. A defense against both types of threats has been shown to be a Byzantine Resilient network such as that employed by the FTPP. However, this network does not address the issue of reliable communication with I/O devices, and it requires the full complement of interconnectivity called for by the theory of f-Byzantine Resilience.

Another approach to reliable communication among heterogeneous hosts has been selected for use in the AGN&C demonstration system. This approach is based on a communication protocol that requires each sender to unforgeably sign each message it sends and for each receiver to verify the authenticity of each message it receives. This process is referred to as the Authentication Protocol and is described in greater detail in [5].

The operating system of each host provides an application task (Ada parlance) or process (Unix parlance) with communication services based on the Authentication Protocol. Each message which a host sends is signed with a 64 bit signature which is function of the sending host's identity and the contents of the message. The receiving host authenticates the message-signature pair, thereby guaranteeing that the message is from a specific sender and the message is uncorrupted. This protocol prevents messages with erroneous contents from reaching the application. To counter the retransmission or insertion of a properly signed message, each message is assigned a sequence number which is a monotonically increasing function unique to each sender and which therefore guarantees that each message is fresh. Finally, to counter passive threats which result in the non-delivery of a message, each message is transmitted over redundant physical channels. If both messages are delivered correctly, the second one to arrive is discarded since it will at this time have a duplicate sequence number.

This reliable message delivery guaranteed by the Authentication Protocol provides a distributed application an ideal framework in which to develop algorithms. The application does not need to concern itself with heuristic approaches to deal with the myriad of "what-if" scenarios that can arise when normal execution of an application task running on one

host is dependent on timely data provided by an application task running on a remote host. The communication protocols developed for the distributed AGN&C demonstration are all based on the guarantee of reliable message delivery provided by the Authentication Protocol (AP). The execution of the distributed AGN&C application is a precisely defined sequence of communication and computation steps. The computation code is generated entirely by ASTER™ and is completely portable and host-independent. The code which provides communication, scheduling and control flow is host/operating system specific and is written by a human engineer to accommodate both the host on which the application executes as well as the requirement that the computation code be held sacrosanct, i.e. unchanged from that produced by ASTER. By requiring that the interface between these man-made and machine-made subprograms be invariant, the internal computations of the AGN&C application can be modified at the specification level, the code regenerated and then tested with relative ease since the external framework is unaffected by these modifications.

The AGN&C execution sequence is divided into three distinct phases of operation or mission modes as shown in Table 5.1: the power-on initialization mode, the pre-launch mode and the inflight mode. The four computing principals in the AGN&C application are the Vehicle Model and Environment Simulation Program (Sim), the Guidance Program (FENOC), the Control and Navigation Program (Control) and the Graphical User Interface (GUI).

During the power-on initialization mode, each of the AGN&C principals performs its own initializations and then waits for the GUI to send each the network address of the other principals. At this point the Sim executes the application code one time and waits for FENOC to request a set of sensor readings. The Sim then sends sensor readings to both FENOC and Control. FENOC reads these sensor values and executes its application code once. It then sends a trajectory table to Control which has been waiting for input from both the Sim and FENOC. Control now runs its application code one time. The application code is now initialized. All three principals then enter the pre-launch mode during which time they wait for a Launch command from the GUI. A Launch command also carries with it values for mission parameters such as the final altitude and final velocity of the launch vehicle (for use by FENOC) and ambient wind speeds (for the environment simulation). The user issues the Launch command by pressing the Launch button on the GUI. The GUI sends the Launch command to each of the AGN&C principals at the same time. After being launched, the Sim executes one time and waits for FENOC to again ask for sensor readings. The Sim sends these readings to Control and FENOC and then enters the inflight mode. FENOC computes a new trajectory table and sends it to Control which has been waiting for input from both FENOC and the Sim. FENOC now enters its inflight mode. Once Control receives its new sensor readings and new trajectory table, it computes a new commanded value of the pitch angle Theta, and it enters the inflight mode.

	Control	Model	Guidance
Init Mode	Init my AP port Wait for GUI to send AP addrs Send ack to GUI Wait for data from Guid. & Model Init my state vars Exec Control 1 time Go to Prelaunch Mode	Init my AP port Wait for GUI to send AP addrs Init my state vars Exec Model 1 time Send ack to GUI Wait for Guid to ask for data Send data to Control Send data to Guid Go to Prelaunch Mode	Init my AP port Wait for GUI to send AP addrs Send Data_Cmd to Model-Read data Init my state vars Exec FENOC 1 time Send data to control Send ack to GUI Go to Prelaunch Mode
Pre-launch Mode	Wait for Launch Cmd Send ack Wait for data from Guid. & Model Init my state vars Exec Control 1 time Go to Inflight Mode	Wait for Launch Cmd Init my state vars Exec Model 1 time Send ack to Gui Wait for Guidance to ask for data Send data to Control Send data to Guid. Go To Inflight Mode	Wait for Launch Cmd Send Data_Cmd to Model-Read data Init my state vars Exec FENOC 1 time Send data to control Send ack to GUI Go to Inflight Mode
Inflight Mode (iterative)	Send data to Model & GUI* Poll AP If abort cmd Goto Prelaunch Mode If new Traj Save data Goto Poll AP If quit cmd Terminate process If Model data Goto Exec Control If no message Goto Poll AP Exec Control	Poll AP: If abort cmd Goto Prelaunch Mode If Data_Cmd Send data to Guid Goto Poll AP If Wind Gust Cmd Init WG Vars Goto Poll AP If quit cmd Terminate process If control data Goto Exec Model If no message Goto Poll AP Exec Model Send data to Control & GUI*	Send Data_Cmd to Model Poll AP If abort cmd Goto Prelaunch Mode If quit cmd Terminate process If no message Goto Poll AP If Model data Compute new Traj Send Traj to Control & GUI *

* Data is not sent to GUI every iteration.

Table 5.1 The AGN&C Execution Sequence

During the inflight mode, Control sends an actuator command to the Sim, waits for a new set of sensor readings and then computes a new actuator command. While waiting for a new set of sensor readings, it is also willing to accept a new trajectory table from FENOC

or an abort or quit command from the GUI. The new trajectory table is applied during the computation of the next command. It repeats this sequence until it receives either an abort or a quit command from the GUI. Periodically it also sends a copy of its actuator command to the GUI to be displayed graphically for the user.

During the inflight mode, the Sim waits for an actuator command from Control, applies the command, computes a new set of sensor readings and sends them to Control. While waiting for a new actuator command, it is also willing to accept a request for sensor readings from FENOC, to which it replies by sending the latest set of sensor values, or a quit, abort or wind gust command from the GUI. It repeats this sequence until it receives an abort or a quit command from the GUI. Periodically it also sends a copy of its sensor readings to the GUI to be displayed graphically for the user. When a wind gust command is received, the Sim modifies the ambient winds vector to apply the wind gust as a step function for a period of two seconds.

During the inflight mode, FENOC requests a set of sensor readings from the Sim. After they arrive, it calculates a new trajectory and sends the new trajectory table to Control. While waiting for a new set of sensor readings, it is also willing to accept an abort or a quit command from the GUI. Periodically, it also sends a copy of its latest trajectory table to the GUI for display.

When any principals in the AGN&C application receive a quit command, they terminate their execution. When they receive an abort command, they return to their pre-launch mode.

The GUI graphically displays data which it receives from the other AGN&C principals as described above. It also relays user commands to the other principals. Nine graphical displays are maintained. They are Altitude vs Time, Downrange vs Time, Altitude vs Downrange, Horizontal Velocity vs Time, Vertical Velocity vs Time, Commanded and Actual Values of the Pitch Angle vs Time, the difference between the Actual and Commanded Pitch Angle, and the trajectory table (Pitch Angle vs Altitude) computed by FENOC at two minute intervals. The GUI also sends commands to the other AGN&C principals as directed by the user. These commands are Launch, Abort, Trigger Wind Gust, or Quit.

Figure 5.1 shows the communication interfaces among the four principals in the AGN&C demonstration. Each principal has one external port on which it can receive messages. Each principal receives several different kinds of messages; messages are differentiated by a "message type" field embedded in each message.

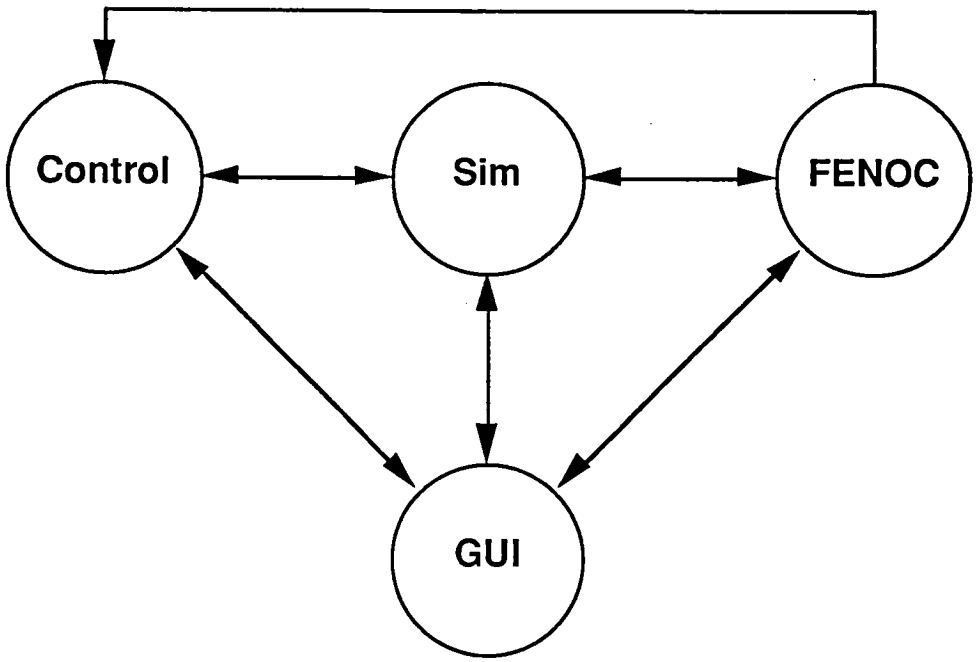


Figure 5.1 Communication Interfaces of the Distributed AGN&C Application

Table 5.2 contains information about each type of message: the sender; the receiver(s), the number of bytes of data, and the frequency of the transmission.

TYPE	SIZE	SENDER	RECEIVER(s)	FREQUENCY
Launch Cmd	64	GUI	All Others	aperiodic
Quit Cmd	8	GUI	All Others	aperiodic
Abort Cmd	8	GUI	All Others	aperiodic
Wind Gust Cmd	24	GUI	Sim	aperiodic
Init Cmd	24	GUI	All Others	at startup only
Init Ack	8	All Others	GUI	at startup only
Display Vehicle State	56	Sim	GUI	0.5
Display Control Data	32	Control	GUI	0.5
Display FENOC Data	68	FENOC	GUI	0.00833
Sensor Readings	40	Sim	Control	50
Sensor Readings	40	Sim	FENOC	aperiodic
Actuator Cmds	32	Control	Sim	50
Guidance Cmds	60	FENOC	Control	aperiodic
Request Sensor Data	8	FENOC	Sim	aperiodic

Table 5.2 Distributed AGN&C Message Specifications

5.2 Task Scheduling

All four principals are expected to run in real time. The Control and Sim, which are iterative tasks, have the most stringent timing requirements. The control laws are based on a frame execution time of 20 ms, i.e. the Control task must run at an iteration frequency of 50 Hz. The 20 ms frame includes time for communication with the Sim and with FENOC. A simplified execution time line for the Control and the Sim tasks is shown in Figure 5.2. The arrows represent inter-processor communication since the Sim and Control tasks are running on different computers. During the initialization mode, the Sim computes an initial set of sensor readings to send to Control. Once Control reads these values, it begins its periodic, 50 Hz operation. In each frame, Control reads a new set of sensor values, computes a new actuator command, and sends that value to the Sim. The Sim must then read the actuator command, compute the new state of the vehicle and send new sensor readings back to Control in time for the start of the next frame.

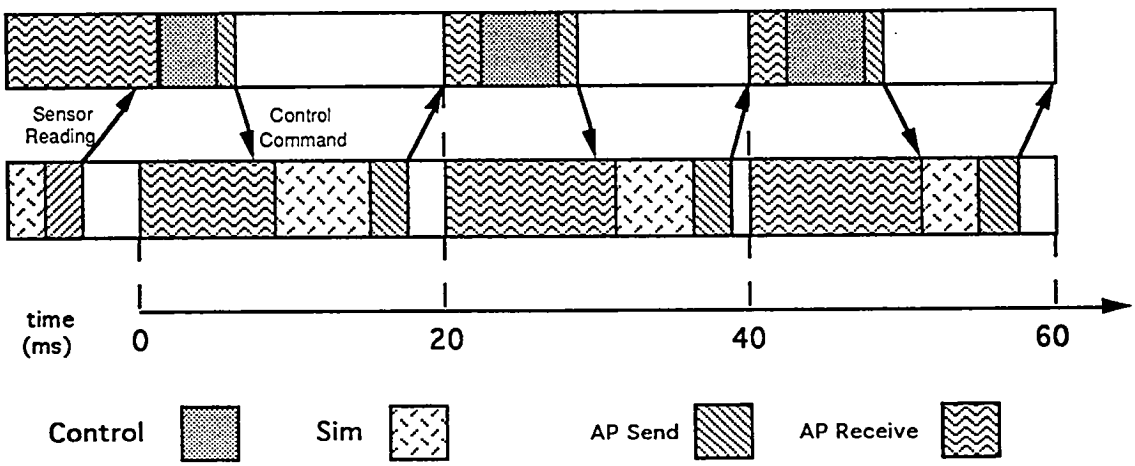


Figure 5.2 Sim and Control Execution Time Line

Although the Control and Sim are periodic tasks, FENOC is not. Its aperiodic behavior derives from the fact that the amount of time required for the calculation of a new trajectory is non-deterministic, although it may be bounded, as well as from the fact that a new trajectory is not required on a periodic basis. For example, changes in the weather may create conditions which make a new trajectory desirable at a certain rate. For the AGN&C demonstration, a new trajectory is computed approximately every 60 seconds. FENOC sends this new trajectory table to Control. Thus during certain frames, Control will receive two messages, one from the Sim and one from FENOC. This creates the possibility of introducing jitter into the Control system and the effects of these aperiodic messages on the real-time behavior of the AGN&C system must be measured and evaluated.

Both the Control and FENOC principals execute as Ada™ tasks on FTPP computers whose operating system is based on a modified version of the XD-Ada™ runtime system.

The Ada runtime system has been modified to allow a timer-driven rate-group scheduler to schedule rate group tasks. This scheduler works in conjunction with special message passing services, including the AP, to meet the requirements for synchronization and interactive consistency of a Byzantine Resilient computer. Any task which performs inline calls to AP must run at the same frequency as the AP task. The Control task is periodic and hence is easily scheduled as a high priority, rate-group task. However, FENOC is not intrinsically periodic. To maintain the required synchronicity among the redundant copies of this task, FENOC has been designed to run as two cooperating tasks. The main task is run at a frequency intended to be slow enough to guarantee that the new trajectory calculation will complete in the allocated amount of time. The second task handles all the I/O operations for the first task. This task is scheduled to run in the same rate group as the AP task. The period of the main task is some exact multiple N of the period of its I/O task. Whenever the main task needs to communicate with one of the other AGN&C principals, it prepares buffers to hold its outgoing and incoming messages. When the I/O task has completed N iterations, the required AP operations are carried out on behalf of the main task. When the main task resumes execution, it continues as if it had conducted the AP calls in line. The scheduling paradigm of the FENOC computation task and its cooperating I/O task are shown in Figure 5.3.

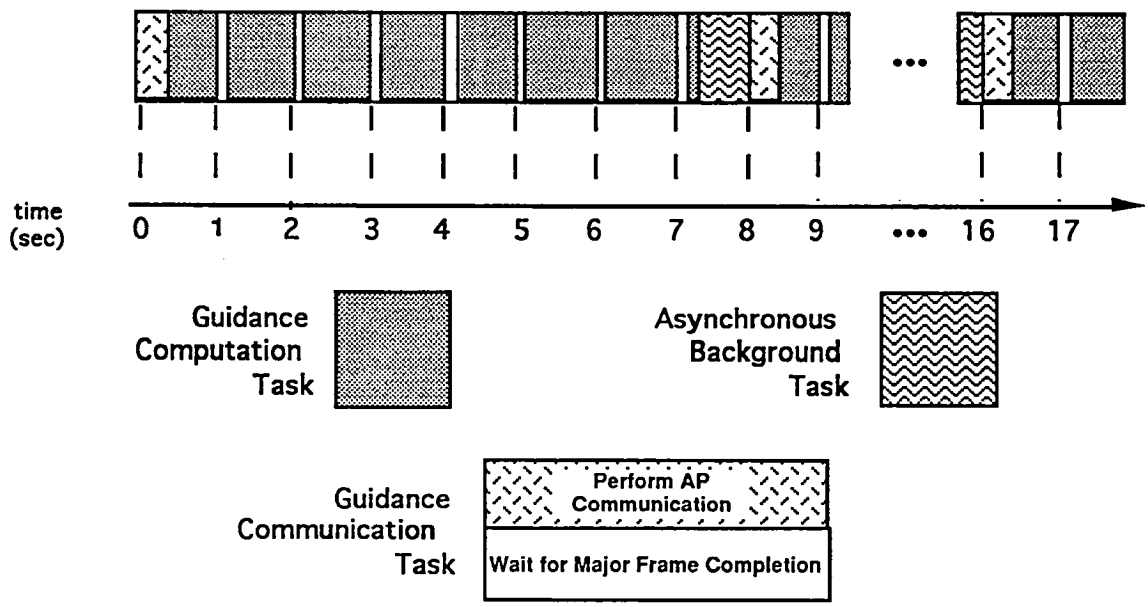


Figure 5.3 Scheduling Paradigm of the FENOC Computation and Communication Tasks

The Sim executes as a POSIX thread on an MVME-147 single board computer running a Lynx™ Real-Time Operating System. This task is scheduled to run at a frequency of 100 Hz so that it will be able to respond in time to any incoming Control messages and immediately return a new set of sensor readings to the Control task.

The GUI runs as an XView client. XView clients are event driven processes. When an event occurs, a particular subroutine is called by the XView notifier. However, AP communications do not provide any signals, i.e. detectable events, to indicate I/O activity. This poses no problem in sending messages, since the GUI only sends messages in response to user actions which do cause events to occur. However, this does not solve the problem posed by "silent" incoming messages. To allow the GUI to poll for any new messages, a periodic timer is set up to run at a frequency of 5 Hz. The Sim and Control send their data to the GUI at a frequency of 0.5 Hz and FENOC sends its data every two minutes. When the timer period expires, an event is triggered which causes a subroutine to be called. It polls for all AP messages received by the GUI during the intervening interval and display any new data which has arrived.

5.3 Demonstration Hardware Configuration

The hardware configuration for the AGN&C demonstration system is shown in Figure 5.4.

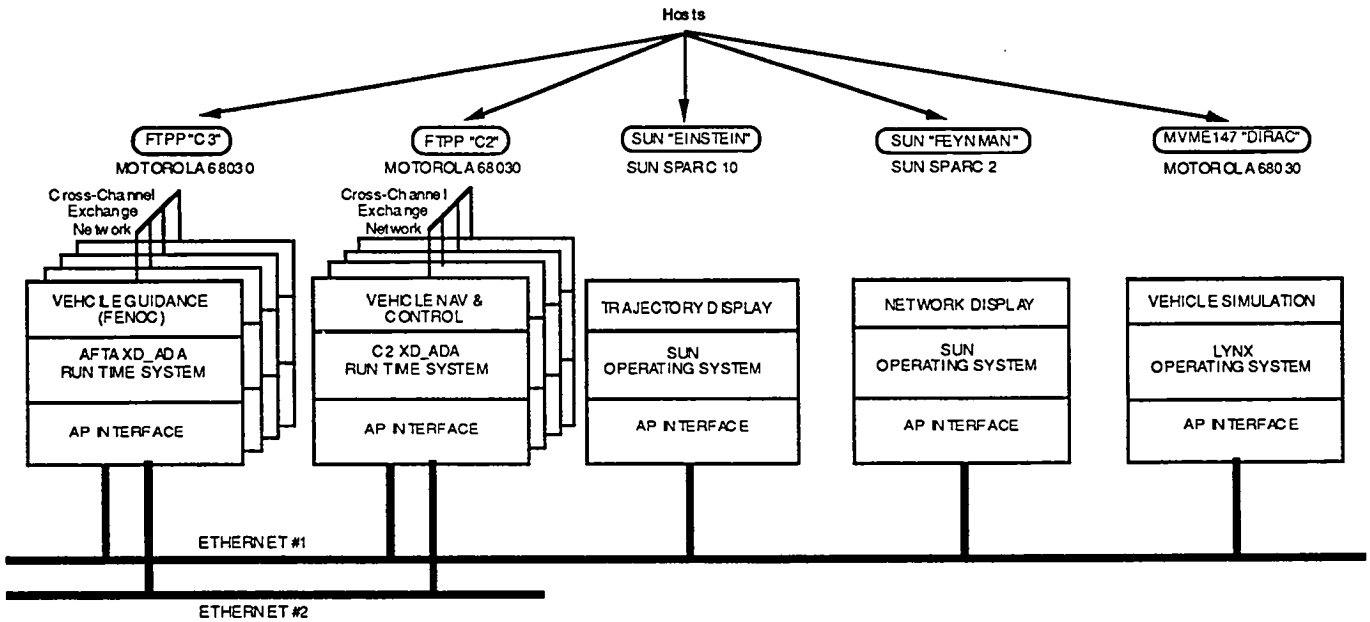


Figure 5.4 Hardware Configuration For The AGN&C Demonstration System

Several key components of this system are designed to provide Byzantine resilience to faults. These are the quad-redundant FTPPs called cluster C2 and C3, and the dual AP communication link between them. Of the four AGN&C demonstration computing principals, the two corresponding to the flight software run on fault tolerant platforms: the Guidance Program (FENOC) runs on C3 and the Control and Navigation Program (Control) runs on C2. The Vehicle Model and Environment Simulation Program (Sim) runs on an MVME-147 board, and the Graphical User Interface (GUI) runs on a Sun workstation. In a real distributed AGN&C application, the Sim would be replaced by a set of redundant sensors and actuators which would use an Authentication Protocol to

communicate with Control and FENOC. The GUI would evolve into a mission command and control center which would have reliability requirements dictating the use of both a fault tolerant computer and communications system.

5.4 Graphical User Interface

The Graphical User Interface (GUI) program is an XView application which provides a user with the means to control the operation of the AGN&C demonstration system as well as a means to graphically display data computed by the AGN&C principals. The GUI program requires three arguments to begin execution. These are the names of the hosts of the three other AGN&C principals, i.e. the Vehicle Model and Environment Simulation Program (Sim), the Guidance Program (FENOC), and the Control and Navigation Program (Control). The GUI executes a handshake protocol with each of these principals to ensure that correctly functioning, bi-directional communications among all four are established. It then creates the XView objects for the control panel and data displays to be used in the AGN&C demonstration. Figure 5.5 shows the GUI during a representative AGN&C mission.

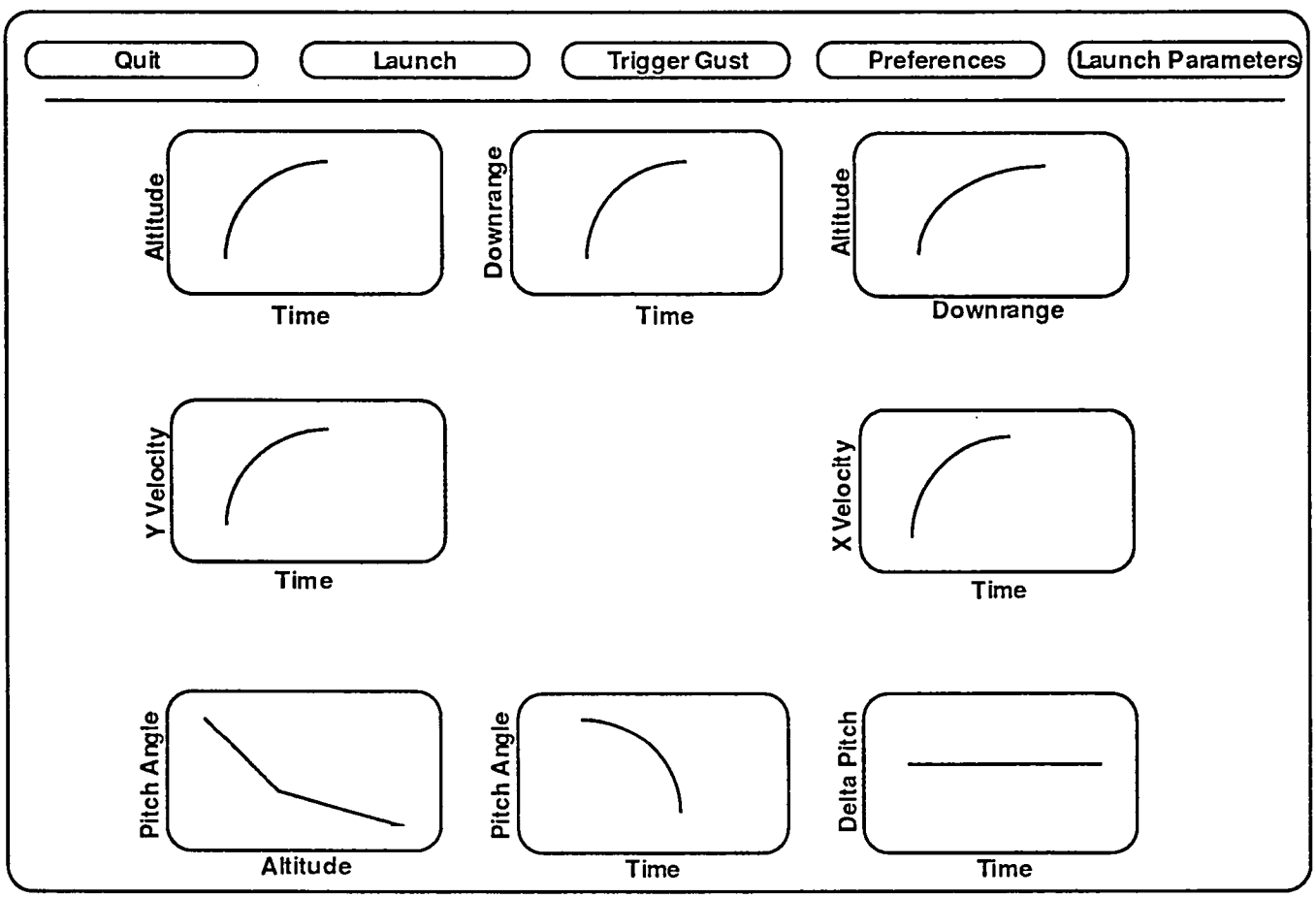


Figure 5.5 The AGN&C Graphical User Interface

Control of the operation of the demonstration system is accomplished by manipulating a set of buttons and menus provided in a control panel displayed across the top of the main GUI window. The data plots use a graphical display package developed at Draper to present selected real-time information during a mission. The function of these buttons and the data plotted in the graphical display windows are described in detail below.

There are six buttons in the GUI control panel. They are "Quit", "Launch/Abort", "Trigger Gust", "Select Input Source", "Preferences", and "Set Input Parameters". These buttons are controlled by using the *select* button on the mouse. When "Quit" is selected, the GUI sends a quit command to the other AGN&C principals and then terminates its own execution gracefully. When these programs receive this command they terminate their execution gracefully. Quitting the program is not a reversible operation. Once the Quit command has been selected, the programs must be restarted to resume execution.

The Launch/Abort button toggles between its two commands. By selecting this button when it displays the Launch command, the user starts a mission and "launches" the vehicle. The GUI processes a launch command by sending the selected set of launch parameters to the other AGN&C principals and by plotting the data that they send back. After a launch command is issued, the Launch/Abort button label toggles to display the Abort command. By selecting the Abort button during a Launch, the system can be returned to its prelaunch state. Data plotted during the previous Launch is not discarded until a new Launch command is issued and hence can be saved at this time. In this way data from several different "runs", i.e. with different initial Launch parameters, can be collected without the need to restart the system.

The Trigger Gust button causes the GUI to send a command to the Sim to simulate a constant wind gust for the next two seconds. The FENOC algorithm is intended to allow a new trajectory to be plotted in real time in response to changing conditions. The trajectory calculated during and after the wind gust command is a measure of the effectiveness of the FENOC algorithm.

The Select Input Source button presents a menu which allows the user to choose to send a set of default launch parameters or a set of user defined launch parameters. Menu choices are provided to allow the user to make this decision one time in advance, or to make it on a launch-by-launch basis.

The Preferences button allows a user to customize two aspects of the testing environment. Wind gust commands can be sent periodically or on user command only. The confirmation window associated with the Launch/Abort button can be disabled to allow a Launch or Abort command to be acted on immediately.

The Set Launch Parameters button allows a user to display the values of the default launch parameters, to display the values of the current user defined launch parameters, and to change the values of the user defined launch parameters. Figure 5.6 shows two of these panels. The one entitled "Current Input Values" shows the values of the launch parameters as currently set by the user. The one entitled "Change Input Values" shows the panel used to change the values of the launch parameters. The input field is selected by positioning the mouse over the desired field and clicking the select key. The triangular caret appears in the active field. The values can be changed either by manipulating the arrows with the mouse or by directly typing input from the keyboard. The GUI does not accept values outside the indicated ranges. A single field can be altered by clicking the select button over the "Enter" button on the same line as the active field. Alternatively, all current entries can be set simultaneously by clicking on the "Enter All" button at the bottom of the panel. The most recent settings are immediately displayed in the "Current Input Values" panel to allow the user to confirm that the indicated values have been noted by the GUI.

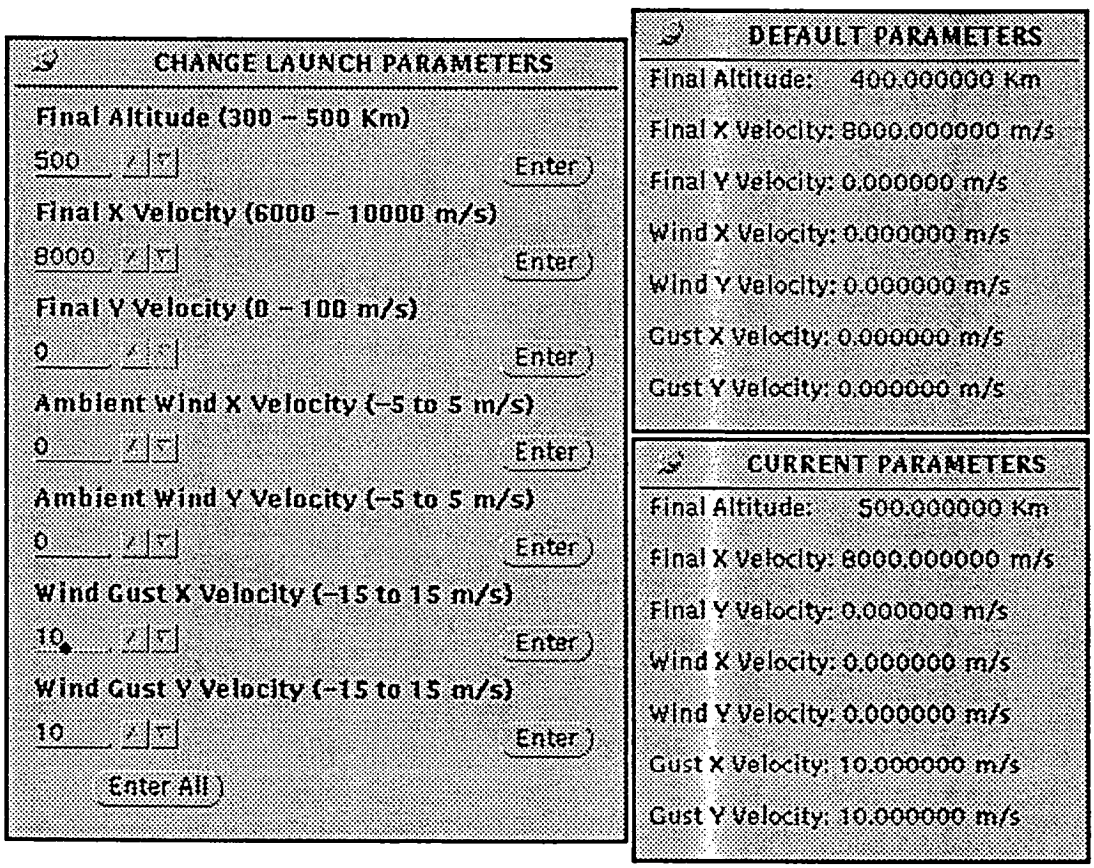


Figure 5.6 Launch Parameters

Eight graphical displays are maintained. They are Altitude vs Time, Downrange vs Time, Altitude vs Downrange, Horizontal Velocity vs Time, Vertical Velocity vs Time, Commanded and Actual Values of the Pitch Angle vs Time, the difference between the Actual and Commanded Pitch Angle, and the trajectory table (Pitch Angle vs Altitude) computed by FENOC.

Data for the first five plots, representing the current state of the vehicle, is sent directly from the Sim, where it is computed, to the GUI. Data for the next two plots is sent to the GUI from the Control program. However, only the value of the commanded pitch angle is computed by the Control. The value of the actual pitch angle is sent by the Sim program to the control and represents the pitch angle that resulted after the commanded angle was applied. The difference between these values is displayed in the seventh plot. Finally, the trajectory table computed by FENOC is plotted.

The GUI runs on a Sun workstation and simply cannot update its displays fast enough to allow it to plot data from every iteration of the control task which with the Sim run at 50 Hz. If they both send data to the GUI after every iteration, the GUI would receive data at 100 Hz and would soon have a huge backlog of points to plot. Instead, the Control and Sim send data every 100 iterations and hence data packets arrive at the GUI at approximately 1 Hz. Guidance sends its data to the GUI every two minutes. Under the present AP implementation, messages may only be received by an in-line query of the AP interface. Since XView is a signal driven system, polling this interface in a "busy wait" mode is not allowed. Instead, a timer is created to signal the system at a rate of 5 Hz. When the timer signals an event, the GUI queries the AP interface for any incoming messages. If data from one of the AGN&C principals has arrived, that data is displayed in the appropriate data plot. Since messages are only expected to arrive at a rate of 1 Hz, this rate of polling the interface should be adequate to display data in a timely fashion without being burdensome to the rest of the system.

Once an abort command is issued, the data from the previous launch is preserved until another Launch command is issued. At this point, the data in any or all windows can be captured using the Openwindows Xview utility *snapshot*. This creates a raster file which can be printed on printers accepting raster format or converted to a postscript format for printers accepting postscript. Figures 5.7 to 5.13 are postscript versions of snapshots taken during a trial run of the AGN&C demonstration system.

5.5 Integration and Testing

The AGN&C demonstration system is a very complex system so an orderly test and integration sequence is very important to the success of the project. The complexities arise in many areas. First of all, many pieces of the system are under development at the same time. If one part of the system depends on another, a temporary work around must be

found so that development may proceed on all fronts. For example, ASTER which generates the application code is under development in this task. Furthermore, the system is distributed and a great deal of interprocessor communication is required. However, the Authentication Protocol communication system is also under development. Finally, both the hardware and operating system of one of the fault tolerant computers which is intended to be a part of the final system is also under development.

Another factor contributing to the complexity of the AGN&C system is the distributed and heterogeneous nature of the system. Several different computer platforms are involved. These are the fault tolerant processors C2 and C3, at least one Sun workstation and an MVME-147 board. Each of these computers has a unique operating system. The C2 operating system is based on a frozen version of the XD-Ada runtime system for the MVME-147 board. The C3 operating system is a second and completely different variation on the same XD-Ada base. The Sun workstations run their own proprietary version of UNIX®. The operating system for the MVME-147 board is another proprietary version of UNIX called LynxOS. This operating system was developed by Lynx Real-Time Systems as a real-time POSIX compliant operating system.

The requirements of the AGN&C application also add to the complexity of the system. For example, the control task and the vehicle simulation are expected to run in real time at 50 Hz. However, the guidance algorithm is not periodic in nature and while it is expected to run in real time, its execution time could be on the order of several seconds. These mixed performance requirements introduce another set of problems to be solved. Both the control

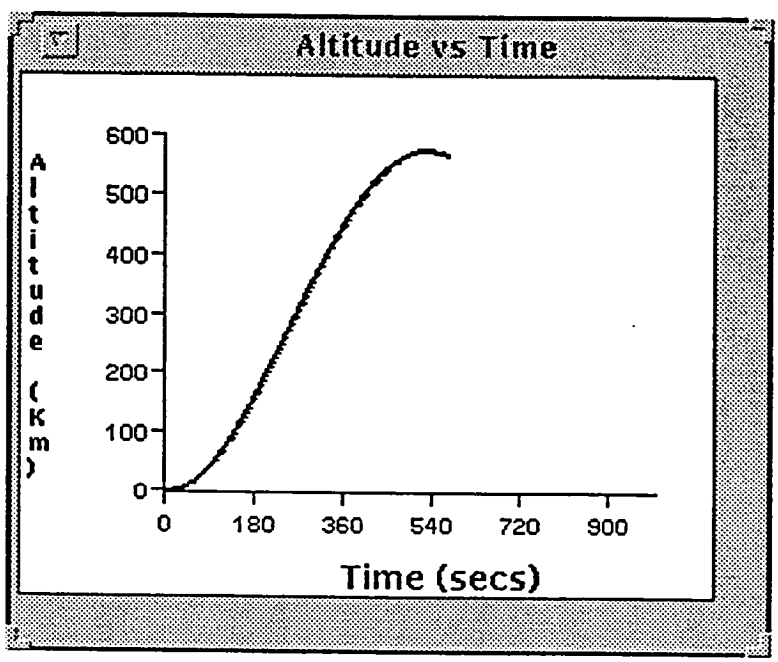


Figure 5.7 Vehicle Altitude vs Time

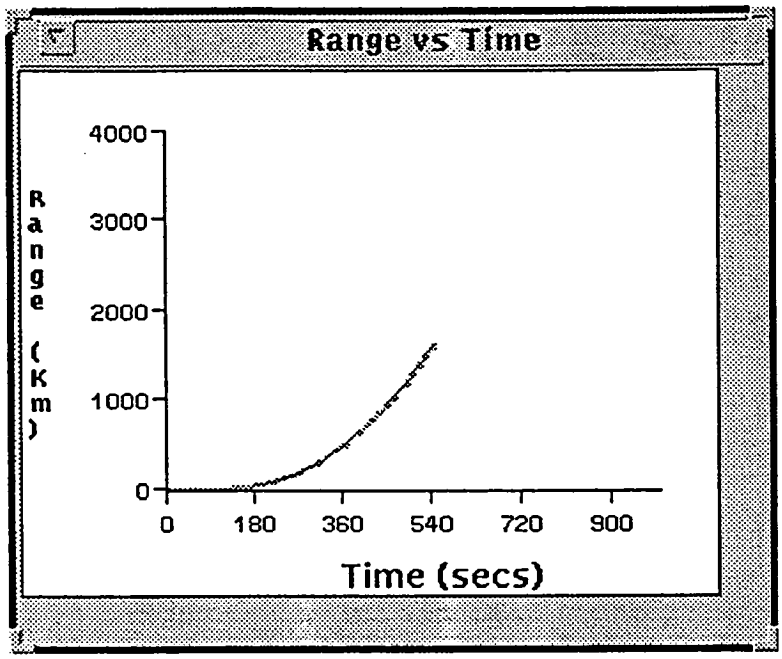


Figure 5.8 Vehicle Range vs Time

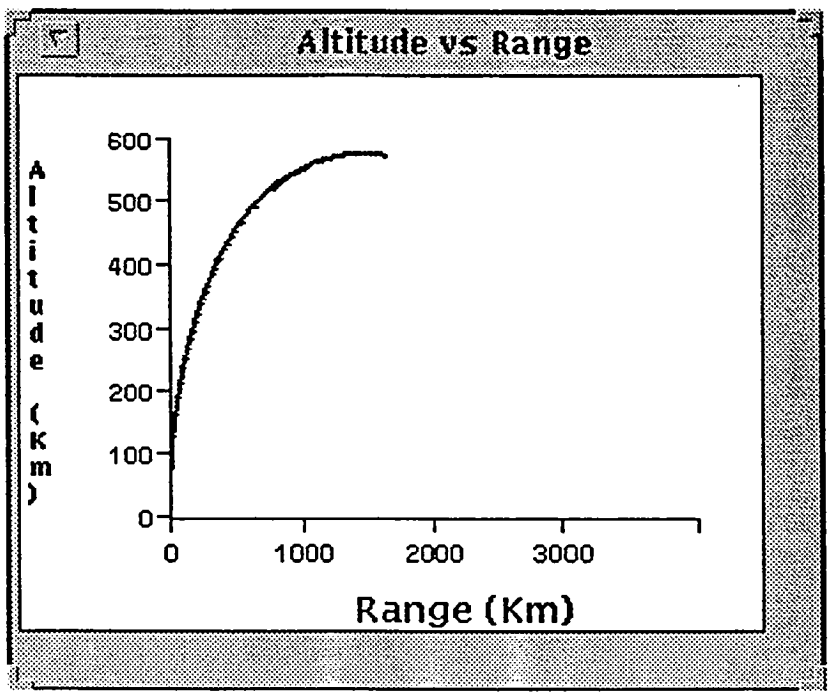


Figure 5.9 Vehicle Altitude vs Vehicle Range

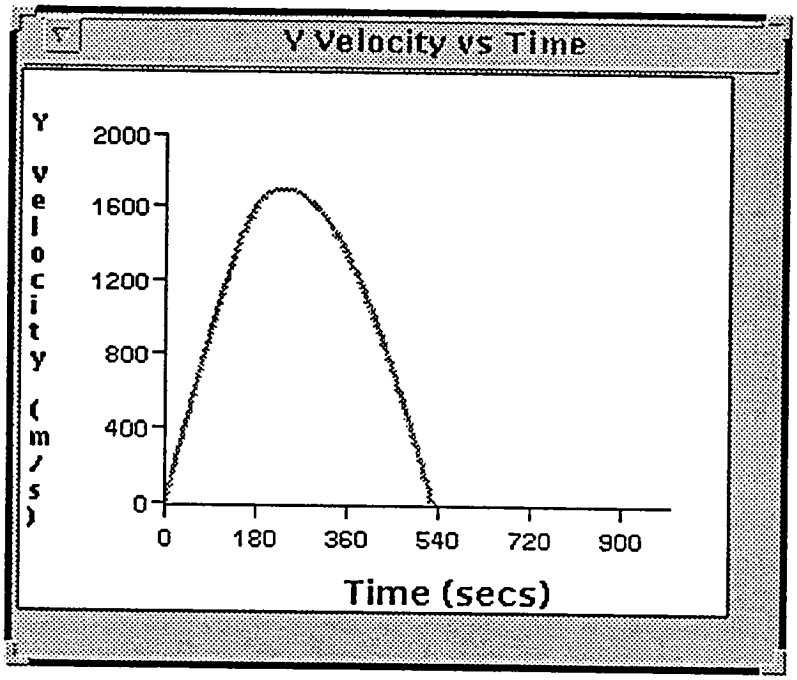


Figure 5.10 Vehicle Y Velocity vs Time

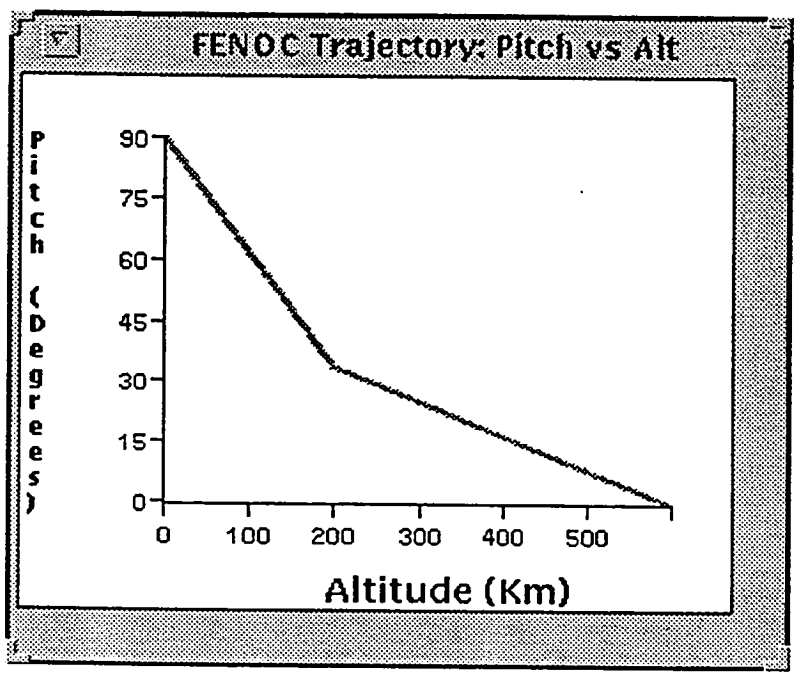


Figure 5.11 FENOC Trajectory

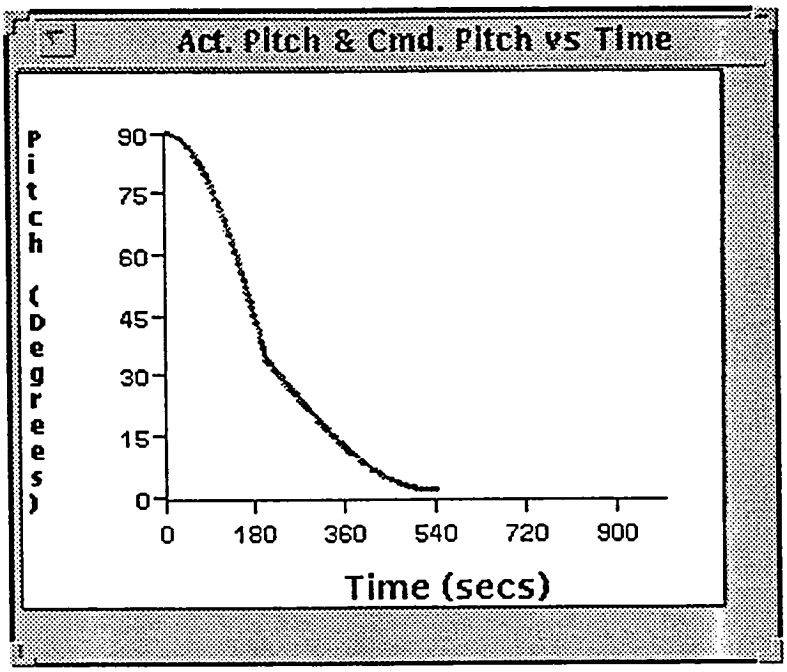


Figure 5.12 Actual Pitch Angle and Commanded Pitch Angle vs Time

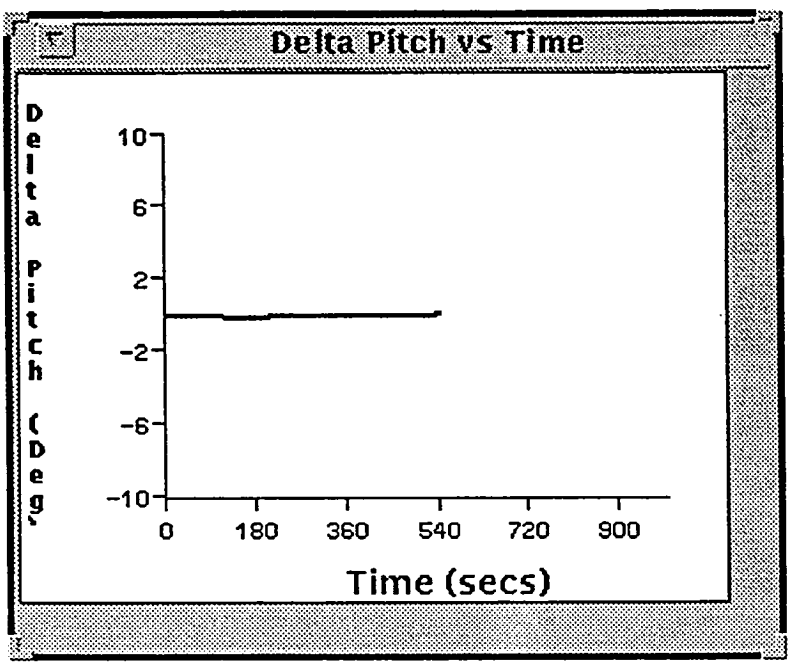


Figure 5.13 Delta Pitch Angle vs Time

and guidance tasks have high reliability requirements and hence must execute on fault tolerant computers. However, the vehicle simulation and the graphical user interface are only part of the demonstration and hence may run on simplex machines. Thus, issues arising from mixed levels of redundancy must be dealt with. Furthermore, there is a requirement that the code for the guidance be written in Ada. Since the control task is running on a platform which only fully supports Ada and assembly language, the control must also be written in Ada. Other parts of the system are written in C. Extra care must be taken to ensure that data passed between programs written in different languages is interpreted correctly by both sides. Finally, there is a requirement that the source code files which are generated by ASTER not be edited in any way prior to compilation. Since ASTER at the present time does not support the logic needed for sequential program control flow or interfaces for scheduling and I/O on various operating systems, these had to be provided by external interface programs.

As the above discussion shows, the development of the AGN&C demonstration system is a major undertaking. Many pieces have to come together in a timely way for the project to be a complete success. If the system is to ultimately possess the attributes of high reliability in both computation and communication and real-time performance and testability, consideration for these attributes must be part of the design from the start. Furthermore, in such a complex system, it is highly desirable to build portable software components which can be reused on many platforms thereby reducing the problems of configuration management which can seriously slow down development.

The application software for the AGN&C demonstration system is designed and implemented to provide these key features: portability, reliability, real-time performance, and testability.

Since ASTER would not be able to translate the Matlab scripts for the real FENOC algorithm until late in the project, and even the real control and simulation applications would not be ready until the middle of the project, a greatly simplified set of applications was generated on ASTER for use in developing the communication and scheduling interface programs. During this phase of development, new features were added to ASTER to allow it to generate a stand alone main or "driver" module. This was a fully distributed system with the control, model, and graphical user interface programs written in C running on Sun workstations and guidance written in Ada running on C2. An Ethernet communications package using a UDP/IP protocol was developed for the C2 system which provided fault-tolerant communications. Finally, a rudimentary graphical user interface program was also developed which provided a user with a graphical display of altitude, downrange, and pitch angle vs time.

During the next phase of development, the actual control and vehicle simulation programs were brought on line and the graphical user interface was further developed and enhanced. The distributed paradigm of the earlier system was not followed. Instead a single module comprising control, simulation and guidance was developed and this single program communicated with the graphical user interface. The next phase required the separation of these integrated components. Issues related to initialization and sequencing came up which probably could have been more readily addressed earlier in the design had a modularized system been integrated for initial testing. Nevertheless, these issues were resolved during several intense test and debug sessions. One feature which would greatly enhance the testability of ASTER code would be the ability to identify variables whose values need to be tracked and then to generate code to either display these values on the console in real-time or to save them in a file for later review. The well defined start-up and execution sequence assisted in the test effort by providing message traceability. This feature would become even more important during the last phase of the test and integration effort.

During the next-to-last phase of development all communications were conducted by means of Ethernet sockets since the AP communication system was still under development. Prior to integration with the AP communications package, the AGN&C demonstration system was fully functional with control running in real-time on C2, the simulation running on LynxOS in real-time and guidance and the GUI running on Sun workstations.

The final phase involved integration with AP. Initially, the AP integration effort required code that was previously portable at least among all UNIX based systems to become node specific, i.e. the code had to linked for a specific Sun workstation or LynxOS. To some extent this problem was removed by outfitting the application code with some additional calls that eliminated this node specificity. Code for C2 and C3 is required to be host specific since these are customized Ada operating systems. An Ada interface for the guidance application was developed by reusing as much of the Ada control interface as possible, not so much as a means of saving time but rather as a means of using tested, reliable code.

Table 5.3 shows the incremental addition of complexity to the system under test. The system aspects such as computational nodes, application code and communication modules that were changed from one milestone to the next are shown in bold in the table. The incremental approach to the test and integration of a huge distributed heterogeneous system such as the present one proved to be of utmost value.

	Milestone 0	Milestone 1	Milestone 2	Milestone 3	Milestone 4	Milestone 5
Control & DISPLAY	GUI with No Inputs	GUI with User Inputs & Expanded Plots	GUI with User Inputs & Expanded Plots	GUI with User Inputs & Expanded Plots	GUI with User Inputs & Expanded Plots	GUi with User Inputs & Expanded Plots
APPLICA-TIONS	Distributed Application Mock Model Mock Control Static Guidance Some Hand Code	Integrated Application Model Control Static Guidance Hand Code & ASTER code Decoupled	Distributed Application Model Control Static Guidance Hand Code & ASTER code Decoupled	Distributed Application Model Control Static Guidance Hand Code & ASTER code Decoupled	Distributed Application Model Control Static Guidance Hand Code & ASTER code Decoupled	Distributed Application Model Control Dynamic Guidance Hand Code & ASTER code Decoupled
COMMUNI-CATION	Simplex Socket-Based Communication	Simplex Socket-Based Communication	Simplex Socket-Based Communication	Redundant Socket-Based Communication	Simplex AP Communica-tions	Redundant AP Communica-tions
COMPUTING PLATFORMS	Sun Workstations FTPP-C2	Sun Workstations	Sun Workstations Lynx Real-Time (POSIX) Workstation	Sun Workstations Lynx Real-Time (POSIX) Workstation FTPP-C2	Sun Workstations Lynx Real-Time (POSIX) Workstation FTPP-C2 FTPPC3/ AFTA	Sun Workstations Lynx Real-Time (POSIX) Workstation FTPP-C2 FTPP-C3/AFTA
FAULT TOLERANCE	Real-Time Fault Tolerance for Mock Control No Transient Recovery	No Fault Tolerance	No Fault Tolerance	Real-Time Fault Tolerance for Control Transient Recovery in Prelaunch Mode for Control	Real-Time Fault Tolerance for Control Real-Time Fault Tolerance for Static Guidance	Real-Time Fault Tolerance for Control Real-Time Fault Tolerance for Dynamic Guidance Fault Tolerant Inter-Processor Communica-tions

Table 5.3 AGN&C Development Approach

6.0 SUMMARY AND CONCLUSIONS

6.1 Summary

The goal of this project was to demonstrate a unified application of a diverse but inter-related set of technologies to mission- and/or safety-critical systems that are of interest to NASA and DOD. An advanced guidance algorithm for a space launch vehicle was chosen to provide a focus for the demonstration. The demonstration met most of its objectives.

A set of MATLAB scripts that implemented a Finite Element Numerical Optimal Control (FENOC) algorithm was provided to Draper by Martin Marietta. The scripts were modified in accordance with certain guidelines and input to the ASTER tool. Capabilities of the ASTER tool itself were enhanced to accept MATLAB scripts. ASTER produced Ada code and documentation for the FENOC algorithm.

Concurrently, a launch vehicle dynamics model was created and input to ASTER using its traditional block diagram input capability to produce C code. Vehicle control and navigation algorithms were also created and processed similarly by ASTER to produce both C as well as Ada code.

For the purpose of continuing with the task of integrating application code, vehicle model, etc. with target computers and inter-computer communications, it was decided to produce a static guidance algorithm as a place holder for FENOC since the MATLAB interface for ASTER was still under development.

The system integration was performed in small steps that gradually added complexity to the hardware and software. As a first step, the C version of all of the application code (static guidance, navigation, control, and vehicle model) was hosted on a single Sun workstation. This workstation also acted as the operator's display and control console and executed the Graphical User Interface (GUI) program. Subsequently, the applications were distributed on different Sun workstations interconnected by Ethernet and socket communications. Finally, the flight software, i.e., guidance, control and navigation, were moved to two Fault Tolerant Parallel Processors (FTPPs): clusters C2 and C3. The socket-based communication between computers was replaced by the Authentication Protocols (AP)-based communication.

The final integrated system was shown to be able to tolerate faults in various components of the FTPPs such as processors and communications links while still correctly executing the flight software in a timely fashion.

The next section discusses some avenues for further research and development in this area.

6.2 Future work

FENOC Algorithm

A second generation of the FENOC algorithm has been devised by Martin Marietta. This new algorithm would be used as the guidance specification in future work. The second generation FENOC algorithm includes a number of new characteristics. These characteristics include the solution to a multistage vehicle guidance problem, inclusion of FENOC specification into KMS (a multimedia document) [8], and the use of FORTRAN for a portion of the specification.

The second generation FENOC algorithm supports multiple stage launch vehicles whereas the original algorithm can only solve single stage vehicle problems. In future work, the MATLAB scripts for the new algorithm would be brought into ASTER. The vehicle model, control system, and workstation displays would be modified to include the characteristics of a multiple stage vehicle. Staging sensors would also be modeled.

Martin Marietta's specification of the new FENOC algorithm resides in a distributed hypermedia system for workgroups. This system is KMS. The MATLAB scripts for FENOC are only part of the hypermedia document. The document also contains textual descriptions, block diagrams, flowcharts, memos and other correspondence. In future work, the MATLAB scripts would to be either referenced by or copied into an ASTER specification.

The second generation of FENOC includes not only MATLAB scripts to describe guidance behavior but also draws upon FORTRAN code for part of the specification. In order for ASTER to automatically digest the FORTRAN portion of the specification, ASTER's algebraic transform engine would be modified to recognize the FORTRAN syntax.

Parallel and Distributed Processing

The AIPS environment is inherently a distributed processing system. One configuration of AIPS has been demonstrated by the work summarized in this report. Alternate configurations can readily be evaluated in future work. ASTER is one tool that will allow rapid evaluation of alternate configurations.

ASTER is designed so that its automatic software designer can be modified to accept functional specifications and produce parallel and distributed software designs. As part of a future effort, parallel and distributed code for the AGN&C system would be generated. The FENOC algorithm was selected because it is inherently computationally intense and is parallelizable. In the work described in this report, only sequential Ada code was automatically generated for FENOC. Also in the work described in this report, both Ada and C were automatically generated for the remaining portions of the AGN&C system. All

of the components were manually distributed on AIPS processors. In the future effort ASTER would automatically distribute code on AIPS processors.

A thesis topic is currently investigating the benefits of allowing users to modulate the design characteristics of ASTER's automatic software designer. The automatic software designer will continue to accept functional designs from the application user interface, but will also accept constraints and design strategies that map functional specifications into tasks and communication interfaces.

7.0 REFERENCES

[1] Lala, J. H., et al, "Advanced Information Processing System for Advanced Launch System: Avionics Architecture Synthesis," The Charles Stark Draper Laboratory, Inc., NASA Contractor Report-187554, September 1991.

[2] Harper, R. E., Alger, L. S., and Lala, J. H., "Advanced Information Processing System: Design and Validation Knowledgebase," The Charles Stark Draper Laboratory, Inc., NASA Contractor Report-187544, September 1991.

[3] Harper, R.E., et al, "Advanced Information Processing System: The Army Fault Tolerant Architecture Conceptual Study - Vol. I Army Fault Tolerant Architecture Overview," The Charles Stark Draper Laboratory, Inc., NASA Contractor Report-189632, Volume I, July 1992.

[4] Harper, R.E., et al, "Advanced Information Processing System: The Army Fault Tolerant Architecture Conceptual Study - Vol. II Army Fault Tolerant Architecture Design and Analysis," The Charles Stark Draper Laboratory, Inc., NASA Contractor Report-189632, Volume II, July 1992.

[5] Harper, R.E., et al, "Advanced Information Processing System: Authentication Protocols for Network Communication", NASA Contractor Report 194923, June 1994.

[6] -, ASTER™ User Guide, The Charles Stark Draper Laboratory, Inc., 1992.

[7] McDowell, Mark E.: Computer Aided Software Engineers at The Charles Stark Draper Laboratory Inc., April 1988.

[8] -, Knowledge Management System User Guide, Knowledge Systems, 1992.

[9] Hodges, Dewey H. and Robert R. Bless, "Weak Hamiltonian Finite Element Method for Optimal Control Problems", Journal Guidance - June 1989.

APPENDIX A

JACOBIAN.M:
AN EXCERPT OF ORIGINAL MARTIN MARIETTA MATLAB SCRIPT

```
function [jacobian,nfunc] = JACOBIAN(Z,param);
%-------------------------------------------------------------------------
% This function evaluates the Jacobian of the FENOC system of equations.
% The FENOC equations are implemented in the function FEM.m.
%
% This Jacobian is used by the root finding routine (such as FSOLVE) that
% is employed to find the zero of the FEM.m function.
%
% Note that Tf_free flag controls whether the derivatives of the transversality
% equation and the derivatives with respect to Tf are evaluated, permitting
% evaluation for both fixed Tf and free Tf.
%
% The FENOC equations, FEM.m, must be run prior to this script so that Uf_global
% has the approproiate value.
%
% REQUIRED FUNCTIONS:
%     SYSTEM_INIT initializes problem-specific constants
%
%     FF     n col system dynamics
%     Fx     n x n partial wrt x of system dynamics
%     Fu     n x m partial wrt u of system dynamics
%     Ft     n col partial wrt t of system dynamics
%     Fxx    n x n 2n partial d/dx(Fx'*LAMBDA)
%     Fxu    n x m 2n partial d/du(Fx'*LAMBDA)
%     Fuu    m x m 2n partial d/du(Fu'*LAMBDA)
%
%     L      scalar       cost function
%     Lx     n row partial wrt x of cost function
%     Lu     m row partial wrt u of cost function
%     Lt     scalar       partial wrt t of cost function
%     Lxx    n x n 2nd partial d/dx(Lx')
%     Lxu    n x m 2nd partial d/du(Lx')
%     Luu    m x m 2nd partial d/du(Lu')
%
%     PHIx   n col partial wrt x of terminal cost function
%                  (PHI incl. adjoined terminal constraints)
%     PHIt   scalar       partial wrt t of terminal cost function
%     PHIxx  n x n 2nd partial d/dx(PHIx)
%     PHItx  n row 2nd partial d/dt(PHIx)
%     PHItt  scalar       2nd partial d/dt(PHIt)
%     PHInux       n x q 2nd partial d/dNU(PHIx)
%     PHInut       q row 2nd partial d/dNU(PHIt)
%
%     PSI    q col terminal constraint function
%     PSIx   q x n partial wrt x of PSI
%     PSIt   q col partial wrt t of PSI
%
% INPUTS:
%     Z                  assembled state vector (see notes)
%                        2*n*(N+1)+m*N+q (+1 if Tf_free) column
%     param              not used
%
% OUTPUTS:
%     jacobian           Jacobian (see notes)
```

```
%     nfunc              number of function evaluations (for fsolve stats)
%
% GLOBALS:
%     N_global     scalar      number of elements
%     Uf_global    m col passing the terminal control from FEM
%     DEBUG        scalar      controls output of diagnostics
%
% INTERNAL:
%     N            scalar      number of elements
%     n            scalar      number of states
%     m            scalar      number of controls
%     q            scalar      number of terminal constraints
%     dt           scalar      finite element time step
%     t            scalar      time at element i midpoint (!NB!)
%     t_array            N row vector of element midpoint times
%     Tf           scalar      final time
%     Tf_free            scalar      final time free flag (=1 free, =0 fixed)
%     i            scalar      current element number
%     Xi           n col state vector at current element i
%     X_prev             n col state vector at previous element i-1
%     Xo           n col state vector at initial time
%     Xf           n col state vector at final time Tf
%     LAMBDAo            n col costate at initial time
%     LAMBDAf            n col costate at final time
%     LAMBDAi            n col costate vector at current element i
%     LAMBDA_prev n col costate vector at previous element i-1
%     Ui           m col control vector at current element i
%     Uf           m col control vector at final time
%     NU           q col vector of terminal constraint multipliers
%
%     xx_i               function xx evaluated at i
%     xx_prev                  (i-1) value of variable xx
%
%     row                range of indeces for rows of jacobian
%     xx_column          range of indeces for columns of jacobian for
%                        the variable xx
%
% HISTORY:
% 5 April 91
%     Created; M.Corvin; References:
%     - Development of Finite Element Equations, pp25, FENOC notebook
%     - Definition of Assembled State Vector, p36, FENOC notebook
%     - Definition of Assembled Function Vector, p37, FENOC notebook
%     - Definition of Jacobian, p59, FENOC notebook
%     - Terms in Jacobian, pp61, FENOC notebook
% 9 April 91
%     Debugged and running.
% 11 April 91
%     Output validated for exmaple case against Jacobian estimate developed
%     by the fsolve routine.
% 14 April 91
%     Misc. cleaning-up.
%
%-------------------------------------------------------------------------
```

```
% >>>> INITIALIZATON <<<<

N     = N_global;

n     = SYSTEM_INIT(1);
m     = SYSTEM_INIT(2);
q     = SYSTEM_INIT(3);
xo    = SYSTEM_INIT(4);

% TIME

Tf_free     = SYSTEM_INIT(5);
if Tf_free,
   Tf = Z( 2*n*(N+1) + N*m + q + 1 ); % get current value from Z
else
   Tf = SYSTEM_INIT(6);
end;

dt = Tf/N;
half_dt = 0.5*dt;
t_array = half_dt:dt:Tf; % N.B. initial time is assumed to be zero


% INITIALIZE OUTPUT

if Tf_free,
  jacobian = zeros(2*n*(N+1) + m*N + q + 1);
else
  jacobian = zeros(2*n*(N+1) + m*N + q);
end;

nfunc = 1; % meaningless value to satisfy fsolve requirements for output

% CALCULATE LAMBDAf

LAMBDAo     = Z(1:n);
LAMBDAf = LAMBDAo;
for i = 1:N,
      LAMBDAi     = Z( 2*i*n + (i-1)*m + 1      : (2*i+1)*n + (i-1)*m );
      LAMBDAf = LAMBDAf + 2*(LAMBDAi - LAMBDAf);
end;

% LOAD Xf and NU

Xf    = Z( (2*N+1)*n + N*m + 1  :  2*n*(N+1) + N*m     );
NU    = Z( 2*n*(N+1) + N*m + 1  :  2*n*(N+1) + N*m + q );

% Uf

Uf = Uf_global;

% EVALUATE REQUIRED FUNCTIONS

% ----------------------------------------
```

```
% NONZERO TERMS WITHIN JACOBIAN BY COLUMNS:
% ------------------------------------------

% >>>> EQUATION Y1 IN COLUMN 1 <<<<

% d_Y1/d_LAMBDAo

jacobian(1:n,1:n) = -eye(n);


% >>>> EQUATIONS Y1, Y2, Y3, Y4, Y5, Y6 IN COLUMNS FOR ELEMENTS 1 TO N <<<<
% >>>>                        and Tf COLUMN if Tf_free                 <<<<

Tf_column = 2*n*(N+1) + N*m + q + 1;

for i=1:N,

%-------------------------------------------------------------------------

% COLUMN INDEX RANGES

X_column =   (2*i-1)*n + (i-1)*m + 1 : 2*i*n + (i-1)*m;
LAMBDA_column =   2*i*n + (i-1)*m + 1     : (2*i+1)*n + (i-1)*m;
U_column =   (2*i+1)*n + (i-1)*m + 1 : (2*i+1)*n + i*m;

% ELEMENT TIME

t = t_array(i);

% LOAD ELEMENT DATA, Xi, LAMBDAi, Ui FROM ASSEMBLED INPUT VECTOR

Xi     = Z( (2*i-1)*n + (i-1)*m + 1  : 2*i*n + (i-1)*m     );
LAMBDAi      = Z( 2*i*n + (i-1)*m + 1      : (2*i+1)*n + (i-1)*m );
Ui     = Z( (2*i+1)*n + (i-1)*m + 1  : (2*i+1)*n + i*m     );

% EVALUATE FUNCTIONS FOR ELEMENT i

F_i    = FF(Xi,Ui,t);
Fx_i   = Fx(Xi,Ui,t);
Fu_i   = Fu(Xi,Ui,t);
Fxx_i  = Fxx(Xi,LAMBDAi,Ui,t);
Fxu_i  = Fxu(Xi,LAMBDAi,Ui,t);
Fux_i  = Fxu_i';
Fuu_i  = Fuu(Xi,LAMBDAi,Ui,t);

Lx_i   = Lx(Xi,Ui,t);
Lu_i   = Lu(Xi,Ui,t);
Lxx_i  = Lxx(Xi,Ui,t);
Lxu_i  = Lxu(Xi,Ui,t);
Lux_i  = Lxu_i';
Luu_i  = Luu(Xi,Ui,t);

if i==1,
```

```
      % ---- FIRST THREE LINES, EQUATIONS Y1, Y2, Y3 ----

      % Y1 LINE

      % d_Y1/d_X1
      jacobian(1:n,X_column) = half_dt*(Lxx_i + Fxx_i);

      % d_Y1/d_LAMBDA1
      jacobian(1:n,LAMBDA_column) = half_dt*Fx_i' + eye(n);

      % d_Y1/d_U1
      jacobian(1:n,U_column) = half_dt*(Lxu_i + Fxu_i);

      if Tf_free,
            % d_Y1/d_Tf
            jacobian(1:n,Tf_column) = (0.5/N)*(Lx_i' + Fx_i'*LAMBDAi);
      end;


      % Y2 LINE

      % d_Y2/d_X1
      jacobian(n+1:2*n,X_column) = half_dt*Fx_i - eye(n);

      % d_Y2/d_U1
      jacobian(n+1:2*n,U_column) = half_dt*Fu_i;

      if Tf_free,
            % d_Y2/d_Tf
            jacobian(n+1:2*n,Tf_column) = (0.5/N)*F_i;
      end;

      % Y3 LINE

      % d_Y3/d_X1
      jacobian(2*n+1:2*n+m,X_column) = Lux_i + Fux_i;

      % d_Y3/d_LAMBDA1
      jacobian(2*n+1:2*n+m,LAMBDA_column) = Fu_i';

      % d_Y3/d_U1
      jacobian(2*n+1:2*n+m,U_column) = Luu_i + Fuu_i;

else
      % >>>> EQUATIONS Y4, Y5, Y6, FOR j=i <<<<

      j = i;

      % Y4 LINE
      row = 2*(j-1)*n+(j-1)*m+1 : (2*j-1)*n+(j-1)*m;

      % d_Y4i/d_Xi
      jacobian(row,X_column) = half_dt*(Lxx_i + Fxx_i);
```

```
        % d_Y4i/d_LAMBDAi
        jacobian(row,LAMBDA_column) = half_dt*Fx_i' + eye(n);

        % d_Y4i/d_Ui
        jacobian(row,U_column) = half_dt*(Lxu_i + Fxu_i);

        if Tf_free,
              % d_Y4j/d_Tf
              jacobian(row,Tf_column) = (0.5/N)*(Lx_i'+Lx_prev'...
                                    +Fx_i'*LAMBDAi+Fx_prev'*LAMBDA_prev);
        end;

        % Y5 LINE
        row = (2*j-1)*n+(j-1)*m+1 : 2*j*n+(j-1)*m;

        % d_Y5i/d_Xi
        jacobian(row,X_column) = half_dt*Fx_i - eye(n);

        % d_Y5i/d_Ui
        jacobian(row,U_column) = half_dt*Fu_i;

        if Tf_free,
              % d_Y5j/d_Tf
              jacobian(row,Tf_column) = (0.5/N)*(F_i + F_prev);
        end;

        % Y6 LINE
        row = 2*j*n+(j-1)*m+1 : 2*j*n+j*m;

        % d_Y6i/d_Xi
        jacobian(row,X_column) = Lux_i + Fux_i;

        % d_Y6i/d_LAMBDAi
        jacobian(row,LAMBDA_column) = Fu_i';

        % d_Y6i/d_Ui
        jacobian(row,U_column) = Luu_i + Fuu_i;

end;

% >>>> EQUATIONS Y4, Y5, Y6, FOR j=i+1 <<<<

j = i + 1; % these are the n-1 parts of the equations

% Y4 LINE
row = 2*(j-1)*n+(j-1)*m+1 : (2*j-1)*n+(j-1)*m;

% d_Y4i+1/d_Xi
jacobian(row,X_column) = half_dt*(Lxx_i + Fxx_i);

% d_Y4i+1/d_LAMBDAi
jacobian(row,LAMBDA_column) = half_dt*Fx_i' - eye(n);

% d_Y4i+1/d_Ui
```

```
jacobian(row,U_column) = half_dt*(Lxu_i + Fxu_i);

% Y5 LINE
row = (2*j-1)*n+(j-1)*m+1 : 2*j*n+(j-1)*m;

% d_Y5i+1/d_Xi
jacobian(row,X_column) = half_dt*Fx_i + eye(n);

% d_Y5i+1/d_Ui
jacobian(row,U_column) = half_dt*Fu_i;

% Y6 LINE all zero

% STORE CURRENT VALUES FOR USE IN NEXT ELEMENT in Tf COLUMN EVALUATIONS

if Tf_free,
      LAMBDA_prev = LAMBDAi;
      F_prev      = F_i;
      Fx_prev     = Fx_i;
      Lx_prev     = Lx_i;
end;

%-------------------------------------------------------------------------

end; % of for-loop

% >>>> EQUATIONS Y7, Y8, Y9 <<<<

X_column =   (2*N-1)*n + (N-1)*m + 1 : 2*N*n + (N-1)*m;
LAMBDA_column =   2*N*n + (N-1)*m + 1      : (2*N+1)*n + (N-1)*m;
U_column =   (2*N+1)*n + (N-1)*m + 1 : (2*N+1)*n + N*m;
Xf_column =  (2*N+1)*n + N*m +1      : 2*n*(N+1) + N*m;
NU_column =        2*n*(N+1) + N*m + 1      : 2*n*(N+1) + N*m + q;

% EVALUATE REQUIRED FUNCTIONS

PHIxx_Tf  = PHIxx(Xf,Uf,NU,Tf);
PHIxux_Tf = PHIxux(Xf,Uf,NU,Tf);

PSIx_Tf   = PSIx(Xf,Uf,Tf);

% EQUATION Y7
row = N*(2*n+m)+1:(2*N+1)*n+N*m;

% d_Y7/d_XN
jacobian(row,X_column) = half_dt*(Lxx_i + Fxx_i);

% d_Y7/d_LAMBDAN
jacobian(row,LAMBDA_column) = half_dt*Fx_i' - eye(n);

% d_Y7/d_UN
jacobian(row,U_column) = half_dt*(Lxu_i + Fxu_i);

% d_Y7/d_Xf
```

```
jacobian(row,Xf_column) = PHIxx_Tf;

% d_Y7/d_NU
jacobian(row,NU_column) = PHInux_Tf;


% EQUATION Y8
row = (2*N+1)*n+N*m+1:2*n*(N+1)+N*m;

% d_Y8/d_XN
jacobian(row,X_column) = half_dt*Fx_i + eye(n);

% d_Y8/d_UN
jacobian(row,U_column) = half_dt*Fu_i;

% d_Y8/d_Xf
jacobian(row,Xf_column) = -eye(n);

% EQUATION Y9
row = 2*n*(N+1)+N*m+1:2*n*(N+1)+N*m+q;

% d_Y9/d_Xf
jacobian(row,Xf_column) = PSIx_Tf;

% >>>> TF TERMS <<<<

if Tf_free,

      % EVALUATE REQUIRED FUNCTIONS AT Tf

      F_Tf  = FF(Xf,Uf,Tf);
      Fx_Tf = Fx(Xf,Uf,Tf);
      Ft_Tf = Ft(Xf,Uf,Tf);

      Lx_Tf = Lx(Xf,Uf,Tf);
      Lt_Tf = Lt(Xf,Uf,Tf);

      PHIx_Tf     = PHIx(Xf,Uf,NU,Tf);
      PHIt_Tf     = PHIt(Xf,Uf,NU,Tf);
      PHItx_Tf = PHItx(Xf,Uf,NU,Tf);
      PHInut_Tf = PHInut(Xf,Uf,NU,Tf);
      PHItt_Tf = PHItt(Xf,Uf,NU,Tf);

      PSIt_Tf     = PSIt(Xf,Uf,Tf);

      % EQUATION Y10
      row = 2*n*(N+1)+N*m+q+1;

      % d_Y10/d_Xf
      jacobian(row,Xf_column) = Lx_Tf + PHIx_Tf'*Fx_Tf + F_Tf'*PHIxx_Tf + PHItx_Tf

      % d_Y10/d_NU
      jacobian(row,NU_column) = F_Tf'*PHInux + PHInut_Tf;
```

```
        % d_Y10/d_Tf
        jacobian(row,Tf_column) = Lt_Tf+PHItx_Tf'*F_Tf+PHIx_Tf'*Ft_Tf+PHItt_Tf;

        % ---- Tf COLUMN ----

        % d_Y7/d_Tf
        row = N*(2*n+m)+1:(2*N+1)*n+N*m;
        jacobian(row,Tf_column) = (0.5/N)*(Lx_i'+Fx_i'*LAMBDAi) + PHItx_Tf;

        % d_Y8/d_Tf
        row = (2*N+1)*n+N*m+1:2*n*(N+1)+N*m;
        jacobian(row,Tf_column) = (0.5/N)*F_i;

        % d_Y9/d_Tf
        row = 2*n*(N+1)+N*m+1:2*n*(N+1)+N*m+q;
        jacobian(row,Tf_column) = PSIt_Tf;

end;

%-------------------------------------------------------------------------------
```

APPENDIX B
(PART I)

JACOBIAN.M:
AN EXCERPT OF REVISED MATLAB SCRIPT FOR ASTER

```
function [jacobian] = JACOBIAN(Z,Zuf,Xo,To,Tf_free,dyn_param,term_cons);
%
%-------------------------------------------------------------------------
% This function evaluates the Jacobian of the FENOC system of equations.
% The FENOC equations are implemented in the function FEM.m.
%
% Note that Tf_free flag controls whether the derivatives of the transversality
% equation and the derivatives with respect to Tf are evaluated, permitting
% evaluation for both fixed Tf and free Tf.
%
% REQUIRED FUNCTIONS:

% JACOBIAN_ITER iterative part of the Jacobian generation
%     FF          n col      system dynamics
%     Fx     n x n      partial wrt x of system dynamics
%     Ft         n col  partial wrt t of system dynamics
%
%     Lx          n row       partial wrt x of cost function
%     Lt          scalar     partial wrt t of cost function
%
%     PHIx         n col     partial wrt x of terminal cost function
%                           (note: PHI includes adjoined terminal constraints)
%     PHIt         scalar    partial wrt t of terminal cost function
%     PHIxx  n x n       2nd partial d/dx(PHIx)
%     PHItx  n row       2nd partial d/dt(PHIx)
%     PHItt scalar       2nd partial d/dt(PHIt)
%     PHInux       n x q 2nd partial d/dNU(PHIx)
%     PHInut       q row 2nd partial d/dNU(PHIt)
%
%     PSIx          q x n     partial wrt x of PSI
%     PSIt          q col     partial wrt t of PSI
%
% INPUTS:
%     Z                  assembled state vector (see notes)
%                         size: n*(2N+2) + m*N + q (+1 if Tf_free) column
% Zuf terminal control vector
%     Xo  n col vector of initial conditions
% To  scalar initial time
% Tf_free scalar flag for Tf
% dyn_param  system dynamics parameters
% term_cons  terminal constraints
%
% OUTPUTS:
%     jacobian          Jacobian (see notes)
%
% GLOBAL:
%     N           scalar      number of elements
%     n           scalar      number of states
%     m           scalar      number of controls
%     q           scalar      number of terminal constraints

% INTERNAL:
%     Tf          scalar      final time
%     dt          scalar      finite element time step
```

```
% half_dt scalar 0.5*dt

%     Xf              n col   state vector at final time Tf
%     LAMBDAN      n col costate vector at element N
% LAMBDAf    n col costate vector at final time
%     Uf           m col control vector at final time
%     NU           q col vector of terminal constraint multipliers
%
%     xx_N                  function xx evaluated at N
%     xx_Tf                function xx evaluated at Tf
%
%     row_sB,E                 range of indices for rows of jacobian (s=n7, n8 or q
%     xx_colB,E          range of indices for columns of jacobian for
%                              the variable xx
%
% HISTORY:
% 5 April 91
%     Created; M.Corvin; References:
%     - Development of Finite Element Equations, pp25, FENOC notebook
%     - Definition of Assembled State Vector, p36, FENOC notebook
%     - Definition of Assembled Function Vector, p37, FENOC notebook
%     - Definition of Jacobian, p59, FENOC notebook
%     - Terms in Jacobian, pp61, FENOC notebook
% 9 April 91
%     Debugged and running.
% 11 April 91
%     Output validated for example case against Jacobian estimate developed
%     by the fsolve routine.
% 14 April 91
%     Misc. cleaning-up.

% 3 May 93 A. Schor
%          Removed SYSTEM_INIT, N_global
%          Added Tf_input
%          Some clean-up
% 5 May 93 A. Schor
%          Removed Uf_global, DEBUG, nfunc
%          Added To, Zuf
% 8 May 93 A. Schor
%          Xo, To and Tf_free passed through argument list
%          Streamlined Tf setting logic and removed Tf_input
%          Made Jacobian size independent of Tf_free, clean-ups
% 13 May 93 A. Schor
%          dyn_param and term_cons passed through argument list
%          changed calls to FF, Fx, Fu, Ft, Fxx, Fxu, Fuu, PSIx and PSIt
%          N,n,m,q assumed global
% 12 June 93 A. Schor
%          Added declare statements
%          row changed to row_n, row_m and row_q for unique type declaration
%          Correction PHImux_Tf (instead of PHImux !) in d_Y10/d_NU
% 25 August 93 A. Schor
%          Modified to only include the non-iterative portions of the previous
%          JACOBIAN
% 26 August 93 A. Schor
```

```
%          Obtains the last element's function values from JACOBIAN_ITER
% 22 September 93 A.Schor
%          Added declarations for To and Tf_free
% 23 September 93 A. Schor
%          Change range addressing
%-------------------------------------------------------------------------
% Declarations:

declare(solution_vector,  Z);
declare(state_vector,  Xo, Xf);
declare(state_vector,  LAMBDAN, LAMBDAf);
declare(state_vector,  F_N, F_Tf, Ft_Tf, PHIx_Tf);
declare(state_vector_row,  Lx_N, Lx_Tf, PHItx_Tf);
declare(control_vector,  Zuf, Uf);
declare(cons_vector,  term_cons, NU, PSIt_Tf);
declare(cons_vector_row,  PHInut_Tf);

declare(Jacobian_matrix, jacobian);
declare(matrix_nxn,  Fx_N, Fxx_N, Lxx_N, Fx_Tf, PHIxx_Tf);
declare(matrix_nxm,  Fu_N, Fxu_N, Lxu_N);
declare(matrix_nxq,  PHInux_Tf);
declare(matrix_qxn,  PSIx_Tf);

declare(array_dynamics,  dyn_param);
declare(Float,  To);
declare(Integer  ,Tf_free);

%-------------------------------------------------------------------------

Tf    = Z( 2*n*(N+1) + N*m + q + 1 ); % get current value from Z
dt = (Tf-To)/N;
half_dt = 0.5*dt;

% ----------------------------------------
% NONZERO TERMS WITHIN JACOBIAN BY COLUMNS:
% ----------------------------------------

% >>>> EQUATIONS Y1, Y2, Y3, Y4, Y5, Y6 IN COLUMNS FOR ELEMENTS 1 TO N <<<<
% >>>>                    and Tf COLUMN if Tf_free                    <<<<

Tf_column = 2*n*(N+1) + N*m + q + 1;

[jacobian, F_N,Fx_N,Fu_N,Fxx_N,Fxu_N,Lx_N,Lxx_N,Lxu_N,LAMBDAN] = ...
    JACOBIAN_ITER(Z,dt,half_dt,Tf_column,Tf_free,dyn_param);

% >>>> EQUATIONS Y7, Y8, Y9 <<<<

% COLUMN INDEX RANGES

X_colB =       (2*N-1)*n + (N-1)*m + 1;
X_colE =        2*N   *n + (N-1)*m;
LAMBDA_colB =       2*N   *n + (N-1)*m + 1;
LAMBDA_colE =      (2*N+1)*n + (N-1)*m;
U_colB =          (2*N+1)*n + (N-1)*m + 1;
```

```
U_colE =          (2*N+1)*n +  N   *m;
Xf_colB =        (2*N+1)*n +  N   *m + 1;
Xf_colE =        (2*N+2)*n +  N   *m;
NU_colB =       (2*N+2)*n +  N   *m + 1;
NU_colE =       (2*N+2)*n +  N   *m + q;

% CALCULATE LAMBDAf

LAMBDAf = LAMBDAF_CALC(Z);

% LOAD Xf, NU, Uf

Xf    = Z( Xf_colB : Xf_colE );
NU    = Z( NU_colB : NU_colE );
Uf = Zuf;

% EVALUATE REQUIRED FUNCTIONS at the end node

PHIxx_Tf  = PHIxx(Xf,Uf,NU,Tf);
PHInux_Tf = PHInux(Xf,Uf,NU,Tf);
PSIx_Tf   = PSIx(Xf,Uf,Tf,term_cons);

% EQUATION Y7
row_n7B =  2*N   *n + N*m + 1;
row_n7E = (2*N+1)*n + N*m;

jacobian(row_n7B:row_n7E, X_colB:X_colE) = ...
                        half_dt*(Lxx_N + Fxx_N);            % d_Y7/d_XN

jacobian(row_n7B:row_n7E, LAMBDA_colB:LAMBDA_colE) = ...
                                    half_dt*Fx_N' - eye(n); % d_Y7/d_LAMBDAN

jacobian(row_n7B:row_n7E, U_colB:U_colE) = ...
                        half_dt*(Lxu_N + Fxu_N);            % d_Y7/d_UN

jacobian(row_n7B:row_n7E, Xf_colB:Xf_colE) = PHIxx_Tf;      % d_Y7/d_Xf

jacobian(row_n7B:row_n7E, NU_colB:NU_colE) = PHInux_Tf;     % d_Y7/d_NU

% EQUATION Y8
row_n8B = (2*N+1)*n + N*m + 1;
row_n8E = (2*N+2)*n + N*m;

jacobian(row_n8B:row_n8E, X_colB:X_colE) = ...
                           half_dt*Fx_N + eye(n);           % d_Y8/d_XN

jacobian(row_n8B:row_n8E, U_colB:U_colE) = half_dt*Fu_N;    % d_Y8/d_UN

jacobian(row_n8B:row_n8E, Xf_colB:Xf_colE) = -eye(n);       % d_Y8/d_Xf

% EQUATION Y9
row_qB =  (2*N+2)*n + N*m + 1;
row_qE =  (2*N+2)*n + N*m + q;
```

```
jacobian(row_qB:row_qE, Xf_colB:Xf_colE) = PSIx_Tf;        % d_Y9/d_Xf

% >>>> TF TERMS <<<<

if Tf_free,

      % EVALUATE REQUIRED FUNCTIONS AT Tf

      F_Tf   = FF(Xf,Uf,Tf,dyn_param);
      Fx_Tf  = Fx(Xf,Uf,Tf,dyn_param);
      Ft_Tf  = Ft(Xf,Uf,Tf,dyn_param);

      Lx_Tf  = Lx(Xf,Uf,Tf);
      Lt_Tf  = Lt(Xf,Uf,Tf);

      PHIx_Tf       = PHIx(Xf,Uf,NU,Tf);
      PHIt_Tf       = PHIt(Xf,Uf,NU,Tf);
      PHItx_Tf  = PHItx(Xf,Uf,NU,Tf);
      PHInut_Tf = PHInut(Xf,Uf,NU,Tf);
      PHItt_Tf  = PHItt(Xf,Uf,NU,Tf);

      PSIt_Tf       = PSIt(Xf,Uf,Tf,term_cons);

      % ---- Tf COLUMN (for EQUATIONS Y7, y8, y9) ----

      jacobian(row_n7B:row_n7E, Tf_column) = ...
  (0.5/N)*(Lx_N'+Fx_N'*LAMBDAN) + PHItx_Tf;                % d_Y7/d_Tf

      jacobian(row_n8B:row_n8E, Tf_column) = (0.5/N)*F_N;        % d_Y8/d_Tf

      jacobian(row_qB:row_qE, Tf_column) = PSIt_Tf;              % d_Y9/d_Tf

      % EQUATION Y10
      row_f = Tf_column;

      jacobian(row_f, Xf_colB:Xf_colE) = ...
    Lx_Tf + PHIx_Tf'*Fx_Tf + F_Tf'*PHIxx_Tf + PHItx_Tf';       % d_Y10/d_Xf

      jacobian(row_f, NU_colB:NU_colE) = ...
            F_Tf'*PHInux_Tf + PHInut_Tf;                     % d_Y10/d_NU

      jacobian(row_f, Tf_column) = ...
    Lt_Tf + PHIx_Tf'*Ft_Tf + PHItx_Tf'*F_Tf + PHItt_Tf;        % d_Y10/d_Tf

else

      % EQUATION Y10 (modified to preserve Tf)
      row_f = Tf_column;
 jacobian(row_f,Tf_column) = 1;

end;

%-------------------------------------------------------------------------
```

APPENDIX B
(PART II)

JACOBIAN_ITER.M:
AN EXCERPT OF REVISED MATLAB SCRIPT FOR ASTER

```
function [jacobian, F_i,Fx_i,Fu_i,Fxx_i,Fxu_i,Lx_i,Lxx_i,Lxu_i,LAMBDAi] = ...
    JACOBIAN_ITER(Z,dt,half_dt,Tf_column,Tf_free,dyn_param);
%
%-------------------------------------------------------------------------
% This function evaluates the element-related portion of
% the Jacobian of the FENOC system of equations.
%
% Note that Tf_free flag controls whether the derivatives of the transversality
% equation and the derivatives with respect to Tf are evaluated, permitting
% evaluation for both fixed Tf and free Tf.
%
% REQUIRED FUNCTIONS:

%     FF          n col      system dynamics
%     Fx     n x n      partial wrt x of system dynamics
%     Fu     n x m      partial wrt u of system dynamics
%     Ft         n col  partial wrt t of system dynamics
%     Fxx   n x n 2nd partial d/dx(Fx'*LAMBDA)
%     Fxu   n x m 2nd partial d/du(Fx'*LAMBDA)
%     Fuu   m x m 2nd partial d/du(Fu'*LAMBDA)
%
%     L            scalar    cost function
%     Lx           n row      partial wrt x of cost function
%     Lu           m row      partial wrt u of cost function
%     Lt           scalar     partial wrt t of cost function
%     Lxx   n x n 2nd partial d/dx(Lx')
%     Lxu   n x m 2nd partial d/du(Lx')
%     Luu   m x m 2nd partial d/du(Lu')
%
%
% INPUTS:
%     Z                 assembled state vector (see notes)
%                        size: n*(2N+2) + m*N + q (+1 if Tf_free) column
%     dt           scalar      finite element time step
%     half_dt           scalar      half time step
% Tf_column index for the Tf's column of Jacobian
% Tf_free scalar flag for Tf
% dyn_param  system dynamics parameters
%
% OUTPUTS:
%     jacobian          Jacobian matrix
% F_i,Fx_i,Fu_i,Fxx_i,Fxu_i,Lx_i,Lxx_i,Lxu_i (see required functions)
%     LAMBDAi              n col    costate vector at current element i

% GLOBAL:
%     N           scalar      number of elements
%     n           scalar      number of states
%     m           scalar      number of controls
%     q           scalar      number of terminal constraints

% INTERNAL:
%     t            scalar     time at element i midpoint (!NB!)
%     i               scalar current element number
%     Xi             n col   state vector at current element i
```

```
%     X_prev            n col state vector at previous element i-1
%     Xf              n col   state vector at final time Tf
%     LAMBDAo             n col    costate at initial time
%     LAMBDAf             n col    costate at final time
%     LAMBDA_prev n col costate vector at previous element i-1
%     Ui          m col control vector at current element i
%     Uf          m col control vector at final time
%     NU          q col vector of terminal constraint multipliers
%
%     xx_i                function xx evaluated at i
%     xx_prev               (i-1) value of variable xx
%     xx_Tf               function xx evaluated at Tf
%
%     row_sB,E              range of indeces for rows of jacobian
(s=n4,n4p,n5,n5p, m)
%     xx_colB,E         range of indices for columns of jacobian for the variable :
%
%
% HISTORY:
% 5 April 91
%     Created; M.Corvin; References:
%     - Development of Finite Element Equations, pp25, FENOC notebook
%     - Definition of Assembled State Vector, p36, FENOC notebook
%     - Definition of Assembled Function Vector, p37, FENOC notebook
%     - Definition of Jacobian, p59, FENOC notebook
%     - Terms in Jacobian, pp61, FENOC notebook
% 9 April 91
%     Debugged and running.
% 11 April 91
%     Output validated for example case against Jacobian estimate developed
%     by the fsolve routine.
% 14 April 91
%     Misc. cleaning-up.

% 3 May 93 A. Schor
%          Removed SYSTEM_INIT, N_global
%          Added Tf_input
%          Some clean-up
% 5 May 93 A. Schor
%          Removed Uf_global, DEBUG, nfunc
%          Added To, Zuf
% 8 May 93 A. Schor
%          Xo, To and Tf_free passed through argument list
%          Streamlined Tf setting logic and removed Tf_input
%          Made Jacobian size independent of Tf_free, clean-ups
% 13 May 93 A. Schor
%          dyn_param and term_cons passed through argument list
%          changed calls to FF, Fx, Fu, Ft, Fxx, Fxu, Fuu, PSIx and PSIt
%          N,n,m,q assumed global
% 12 June 93 A. Schor
%          Added declare statements
%          row changed to row_n, row_m and row_q for unique type declaration
%          Correction PHInux_Tf (instead of PHInux !) in d_Y10/d_NU
% 25 August A. Schor
```

```
%          Created from previous Jacobian to incorporate only the "for" loop
% 26 August 93 A. Schor
%          Outputs the last element's function values for use by JACOBIAN
% 22 September 93 A. Schor
%          Added declarations for dt, half_dt, Tf_column, Tf_free
% 23 September 93 A. Schor
%          Change range addressing
%-------------------------------------------------------------------------
% Declarations:

declare(solution_vector,  Z);
declare(state_vector,  Xi);
declare(state_vector,  LAMBDAi, LAMBDA_prev);
declare(state_vector,  F_i, F_prev);
declare(state_vector_row,  Lx_i, Lx_prev);
declare(control_vector,  Ui);
declare(control_vector_row,  Lu_i);

declare(Jacobian_matrix, jacobian);
declare(matrix_nxn,  Fx_i, Fxx_i, Lxx_i, Fx_prev);
declare(matrix_nxm,  Fu_i, Fxu_i, Lxu_i);
declare(matrix_mxn,  Fux_i, Lux_i);
declare(matrix_mxm,  Fuu_i, Luu_i);

declare(array_dynamics,  dyn_param);

declare(Float,  dt, half_dt);
declare(Integer,  Tf_column, Tf_free);

%-------------------------------------------------------------------------

% INITIALIZE time

t = - half_dt;

% INITIALIZE OUTPUT MATRIX (independent of Tf_free)

jacobian = zeros(2*n*(N+1) + m*N + q + 1);

% ---------------------------------------
% NONZERO TERMS WITHIN JACOBIAN BY COLUMNS:
% ---------------------------------------

% >>>> EQUATIONS Y1, Y2, Y3, Y4, Y5, Y6 IN COLUMNS FOR ELEMENTS 1 TO N <<<<
% >>>>                    and Tf COLUMN if Tf_free                    <<<<

for i=1:N,

%-------------------------------------------------------------------------

% COLUMN INDEX RANGES

X_colB =      (2*i-1)*n + (i-1)*m + 1;
X_colE =       2*i   *n + (i-1)*m;
```

```
LAMBDA_colB =      2*i   *n + (i-1)*m + 1;
LAMBDA_colE =     (2*i+1)*n + (i-1)*m;
U_colB =          (2*i+1)*n + (i-1)*m + 1;
U_colE =          (2*i+1)*n +  i   *m;

% ELEMENT TIME
  told = t;
  t = told + dt;

% LOAD ELEMENT DATA, Xi, LAMBDAi, Ui FROM ASSEMBLED INPUT VECTOR

Xi     =      Z( X_colB : X_colE );
LAMBDAi      = Z( LAMBDA_colB : LAMBDA_colE );
Ui     =      Z( U_colB : U_colE );

% EVALUATE FUNCTIONS FOR ELEMENT i

F_i     = FF(Xi,Ui,t,dyn_param);
Fx_i    = Fx(Xi,Ui,t,dyn_param);
Fu_i    = Fu(Xi,Ui,t,dyn_param);
Fxx_i = Fxx(Xi,LAMBDAi,Ui,t,dyn_param);
Fxu_i = Fxu(Xi,LAMBDAi,Ui,t,dyn_param);
Fux_i = Fxu_i';
Fuu_i = Fuu(Xi,LAMBDAi,Ui,t,dyn_param);

Lx_i    = Lx(Xi,Ui,t);
Lu_i    = Lu(Xi,Ui,t);
Lxx_i = Lxx(Xi,Ui,t);
Lxu_i = Lxu(Xi,Ui,t);
Lux_i = Lxu_i';
Luu_i = Luu(Xi,Ui,t);

if i==1,

      % ---- FIRST THREE LINES, EQUATIONS Y1, Y2, Y3 ----

      % Y1 LINE

      jacobian(1:n, 1:n) = -eye(n);                                  % d_Y1/d_LAMBD

      jacobian(1:n, X_colB:X_colE) = half_dt*(Lxx_i + Fxx_i);        % d_Y1/d_X1

      jacobian(1:n, LAMBDA_colB:LAMBDA_colE) = ...
                        half_dt*Fx_i' + eye(n);                  % d_Y1/d_LAMBDA1

      jacobian(1:n, U_colB:U_colE) = half_dt*(Lxu_i + Fxu_i);          % d_Y1/d_U

      if Tf_free,
            jacobian(1:n, Tf_column) = (0.5/N)*(Lx_i' + Fx_i'*LAMBDAi);  % d_Y1/d_'
 end;


      % Y2 LINE
```

```
      jacobian((n+1):(2*n), X_colB:X_colE) = half_dt*Fx_i - eye(n); % d_Y2/d_X1

      jacobian((n+1):(2*n), U_colB:U_colE) = half_dt*Fu_i;          % d_Y2/d_U1

      if Tf_free,
           jacobian((n+1):(2*n), Tf_column) = (0.5/N)*F_i;              % d_Y2/d
      end;


      % Y3 LINE

      jacobian((2*n+1):(2*n+m), X_colB:X_colE) = Lux_i + Fux_i;       % d_Y3/d_X1

      jacobian((2*n+1):(2*n+m), LAMBDA_colB:LAMBDA_colE) = Fu_i';   % d_Y3/d_LAMB

      jacobian((2*n+1):(2*n+m), U_colB:U_colE) = Luu_i + Fuu_i;       % d_Y3/d_U1

else
      % >>>> EQUATIONS Y4, Y5, Y6, FOR j=i <<<<

      j = i;

      % Y4 LINE
      row_n4B = (2*j-2)*n+(j-1)*m+1;
 row_n4E = (2*j-1)*n+(j-1)*m;

      jacobian(row_n4B:row_n4E, X_colB:X_colE) = ...
                        half_dt*(Lxx_i + Fxx_i);    % d_Y4i/d_Xi

      jacobian(row_n4B:row_n4E, LAMBDA_colB:LAMBDA_colE) = ...
                         half_dt*Fx_i' + eye(n);       % d_Y4i/d_LAMBDAi

      jacobian(row_n4B:row_n4E, U_colB:U_colE) = ...
                         half_dt*(Lxu_i + Fxu_i);  % d_Y4i/d_Ui

      if Tf_free,
           jacobian(row_n4B:row_n4E, Tf_column) = (0.5/N)*(Lx_i' + Lx_prev' ...
                        + Fx_i'*LAMBDAi + Fx_prev'*LAMBDA_prev);  % d_Y4j/d_Tf
      end;

      % Y5 LINE
 row_n5B = (2*j-1)*n+(j-1)*m+1;
 row_n5E =  2*j   *n+(j-1)*m;

      jacobian(row_n5B:row_n5E, X_colB:X_colE) = half_dt*Fx_i - eye(n);  %
d_Y5i/d_Xi

      jacobian(row_n5B:row_n5E, U_colB:U_colE) = half_dt*Fu_i;              %
d_Y5i/d_Ui

      if Tf_free,
           jacobian(row_n5B:row_n5E, Tf_column) = (0.5/N)*(F_i + F_prev);
% d_Y5j/d_Tf
      end;
```

```
      % Y6 LINE
      row_mB = 2*j    *n+(j-1)*m+1;
 row_mE = 2*j    *n+ j    *m;

      jacobian(row_mB:row_mE, X_colB:X_colE) = Lux_i + Fux_i;     % d_Y6i/d_Xi

      jacobian(row_mB:row_mE, LAMBDA_colB:LAMBDA_colE) = Fu_i';  % d_Y6i/d_LAMBDAi

      jacobian(row_mB:row_mE, U_colB:U_colE) = Luu_i + Fuu_i;     % d_Y6i/d_Ui

end;

% >>>> EQUATIONS Y4, Y5, Y6, FOR j=i+1 <<<<

j = i + 1; % these are the n-1 parts of the equations

% Y4 LINE
row_n4pB = (2*j-2)*n+(j-1)*m+1;
row_n4pE = (2*j-1)*n+(j-1)*m;

jacobian(row_n4pB:row_n4pE, X_colB:X_colE) = ...
                                half_dt*(Lxx_i + Fxx_i);       % d_Y4i+1/d_Xi

jacobian(row_n4pB:row_n4pE, LAMBDA_colB:LAMBDA_colE) = ...
                                    half_dt*Fx_i' - eye(n);  % d_Y4i+1/d_LAMBDAi

jacobian(row_n4pB:row_n4pE, U_colB:U_colE) = ...
                           half_dt*(Lxu_i + Fxu_i);            % d_Y4i+1/d_Ui

% Y5 LINE
row_n5pB = (2*j-1)*n+(j-1)*m+1;
row_n5pE =  2*j    *n+(j-1)*m;

jacobian(row_n5pB:row_n5pE, X_colB:X_colE) = ...
                                half_dt*Fx_i + eye(n);         % d_Y5i+1/d_Xi

jacobian(row_n5pB:row_n5pE, U_colB:U_colE) = half_dt*Fu_i;    % d_Y5i+1/d_Ui

% Y6 LINE all zero

% STORE CURRENT VALUES FOR USE IN NEXT ELEMENT in Tf COLUMN EVALUATIONS

if Tf_free,
      LAMBDA_prev = LAMBDAi;
      F_prev           = F_i;
      Fx_prev          = Fx_i;
      Lx_prev          = Lx_i;
end;

%-------------------------------------------------------------------------

end; % of for-loop
```

APPENDIX C

ASTER STYLE MATLAB GUIDELINES

1.0 A Comparison of the MATLAB and ASTER Working Paradigms

MATLAB and ASTER operate under two distinct working paradigms. MATLAB is, by nature, a programming language, while ASTER is, by nature, a workbench for capturing a functional design. This fundamental difference has important consequences for the engineer.

ASTER	*MATLAB*	*Comment*
No Inherent Ordering	**Inherently Ordered**	The signal flow from component to component in ASTER provides the framework in which the engineer thinks about the design. Nevertheless, there is no inherent time ordering of the various signals in a project—all signals are considered to be valid at once. On the other hand, because MATLAB is a programming language, there is a well-defined flow of control, readily deduced by examining the code.
Strongly Typed	**Untyped**	MATLAB requires no type declarations, while ASTER requires all signals to be typed.
Compiler	**Interpreter**	ASTER "programs" can run only after being compiled. MATLAB runs only in interpreted mode.
Graphical User Interface	**Textual User Interface**	ASTER programs are interconnections of transforms, in which each graphical manifestation of a transform corresponds to a subroutine call. MATLAB expresses the same concept in a textual way using a conventional programming syntax.

Table 1. A Comparison of ASTER and MATLAB Working Paradigms

2.0 A Strategy for Converting MATLAB Script Sets to ASTER

Successful conversion from MATLAB to ASTER requires an understanding of the differences between the two working paradigms. If you haven't yet written your MATLAB code, and you know in advance that you will be converting to ASTER, you can simplify the conversion process by writing MATLAB code in ASTER style. In this section we'll describe this style, and discuss strategies for conversion of MATLAB codes.

If you're trying to convert an existing MATLAB code to ASTER, and that code wasn't written in ASTER style, you have a choice of two methods.

A	**Re-implement**	You can sit down in front of ASTER and re-implement your scripts using the non-ASTER style version of the MATLAB scripts as a reference.
B	**Convert, then re-implement**	You can convert your existing scripts to ASTER style, and then re-implement the converted scripts in ASTER.

Method A (Re-implement) may seem like less work than Method B (Convert then Re-implement) but that may be an illusion. The conversion process can be complicated for even a moderate-sized project, and there is substantial opportunity for error. Method B has an important advantage for reducing errors. That is that after you have converted the code to ASTER style, you can execute it in MATLAB and compare the results to the original project. If the two systems behave differently, you know that you have inadvertently changed something during the conversion.

Thus, Method B provides an opportunity for verifying much of the work you must do. This can be an important advantage. In the remainder of this note, we'll assume that all projects are converted using Method B.

Note that when you use Method B, and differences arise when you compare the behavior of the original system to the behavior of the system rewritten in ASTER-style MATLAB, you may not immediately conclude that the ASTER-style MATLAB code has bugs. It may be that the original system is incorrect, or that both are incorrect. Bringing the two systems into alignment, reconciling their differences, may require changes in one or the other or both. Thus, Method B for converting to ASTER presents the opportunity to actually *improve* the reliability of the code.

3.0 What is ASTER Style MATLAB?

To write MATLAB code in ASTER style, or to convert existing MATLAB code into ASTER style, you must define the modularity of the code to correspond with ASTER's paradigm, and you must replace certain common idiomatic constructions that conflict with the ASTER paradigm. In this section, we give guidelines for these tasks.

Keep in mind that ASTER is essentially a functional design specification language. Functions accept arguments and return values. Unlike a programming language, which allows you to store values in cells of memory, ASTER transforms merely operate on their inputs. Many of the restrictions below follow from this basic principle.

Restrictions on Iterations

Iterations cannot be nested. If you must nest, create an inferior transform definition that contains the nested iteration. Iterations cannot appear inside IF statements. Use the same approach to get around this restriction.

Only one iteration per transform definition is permitted. Any code not directly related to initialization of variables to be assigned values in the body of the iteration, or the body of the iteration itself, must be removed to another transform definition from which the iteration's transform definition is manifested.

The command **break** is not supported.

Pay Attention To The Length of Variable and Function Names

MATLAB restricts names to 19 characters or fewer. ASTER has no such restriction. If you want your code to run in MATLAB, be certain that your names are unique in their first 19 characters. MATLAB allows longer names, but it ignores characters after the nineteenth.

Clearly Separate the System From the Test Bench

In most MATLAB script sets you can find two basic subsystems. The first is the system you are designing. The second is a collection of routines needed to measure the behavior of that system in response to various stimuli.

You will be converting to ASTER only the first part, the system you're designing. You won't be converting the test bench. The conversion of the system will be dramatically simplified if you clearly separate the system itself from the test bench. For example, you may have an initialization file that performs a variety of tasks. Some of these are related to the system, some to the test bench. Split that file into two parts.

You may notice an increase in the number of modules in the ASTER style MATLAB version of your system after you have performed this division. But don't worry too much about it—you now have the advantage that the test bench modules are clearly separated, and require very little attention or conversion work. There may be some incidental changes needed to support changes in your system modules, but other than that, you can leave them alone, since you won't be converting them to ASTER.

Avoid Assignment When A Nested Function Call Will Do

Sometimes, for the sake of readability, MATLAB authors assign a value to a variable when that value is computed as a function call, even when the result is needed in only one place. For example, consider the following fragment:

```
...
DrivingSignal = sin(omega*t)
Response = Convolve(ImpulseResponse, DrivingSignal)
...
```

This is perfectly acceptable in both ASTER and MATLAB. In ASTER, DrivingSignal is viewed as a local variable, and you might, for example, label its signal segment. You need not change this when you do the conversion. But let's suppose that you don't really care that DrivingSignal carries a name. You're perfectly happy to implement this as:

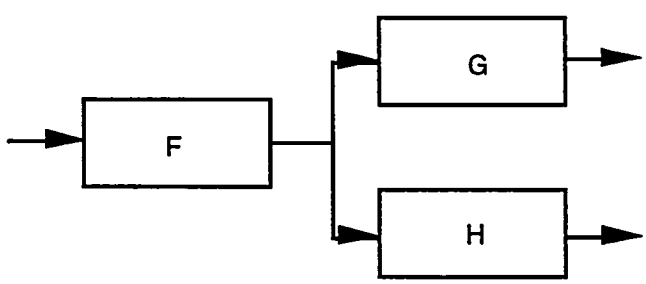
```
...
Response = Convolve(ImpulseResponse, sin(omega*t))
...
```

If this is acceptable to you, then you should make the change in the MATLAB code.

Avoid Global Variables For Input-Dependent Quantities

From time to time, you find a need to test a condition, and to use the results of that condition later in your code. For example, you might determine whether a particular input is non-zero, and use that information in several places. In MATLAB, authors sometimes store the result of that test in a variable, and then refer to the variable later on. Typically, this variable is a global.

In ASTER, you can do the same thing, but you can do it a little differently. The signal that represents the result of the test can easily be fed as an input to the parts of the project that need it. In this role, the signal corresponds to a function argument in MATLAB. Thus, for most situations in MATLAB in which you use global variables to pass information to the interior of subroutines, in ASTER you will find yourself passing signals around as inputs. This corresponds to using arguments of functions in MATLAB.



Unify Any Split if Statements

From time to time, **if** statements are used with only a single branch active. Usually, this happens when you have already set a value, and you want to change it if certain conditions apply. For example,

```
...
Emergency = false
if slewRate > 30
   Emergency = true
end
...
```

This can also be written as

```
...
if slewRate > 30
   Emergency = true
else
   Emergency = false
end
...
```

In effect, the former is a "split" if statement. The latter form is more natural for ASTER. Rewriting code into this form often eliminates apparent re-using of variables, described below.

Unbalanced if Statements

Unbalanced if statements are allowed. By this we mean that the else and elseif clauses are optional. However, unless each clause of the if assigns values to the same set of variables, ASTER assumes that any missing assignments are implicit delayed reassignments.

Specifically, consider the example

```
if x<= 0
  y=1
  z=2
else
  y=2
endif
```

This is exactly equivalent to

```
if x<= 0
  y=1
  z=2
else
  y=2
  z=delay(z)
endif
```

Re-Using Variables Is Not Allowed

In MATLAB, you're perfectly free to assign a value to a variable several times. In certain circumstances, this cannot be done in ASTER. Let's first enumerate the circumstance in which you *can* reassign a value to a variable:

For loop — In a **for** loop, there is an iteration variable that is update on each traversal through the loop. This "reassignment" is supported in both ASTER and MATLAB.

While loop — In a **while** loop, you may establish one or more counters to be incremented on each traversal through the loop. This is supported in MATLAB, but not yet supported in ASTER. We are now designing this facility. For the time being, consider it permissible.

Avoid all other re-use of variables. In particular, if you store results in temporary variables, make certain that you use each temporary variable only once.

Use M-Files

MATLAB supports subroutines through a facility called an m-file. An m-file is a file that holds a single function definition. The ASTER analog of an m-file is a transform definition.

Recasting your scripts as a collection of m-files makes conversion to ASTER much easier. When writing in a textual programming language, there is a tendency to avoid defining a function unless it is called at least twice. ASTER is different. You'll find it much easier to think about your design if you make transform definitions, the analog of function definitions, even when they are used only once. So be generous with them. If you find a section of your project that has a well-defined role, redefine it as an m-file even if you intend to call it only once. Later on, when you try to convert it to ASTER, function definitions will become transform definitions, and you will find it much easier to convert your project to ASTER.

Localize State Variables And Convert Them To Arguments

Whenever you use a delay, you must consider the initial state associated with the delay. In code generated from an ASTER project, the initial state is treated as an argument of the function that represents the transform that contains state. This is true whether you specify the initialization explicitly or implicitly.

To minimize the difficulty of converting your scripts to ASTER, we recommend that you specify state variables and initialization explicitly, especially if you are a new user. In terms of ASTER block diagrams, the possible representation is shown below:

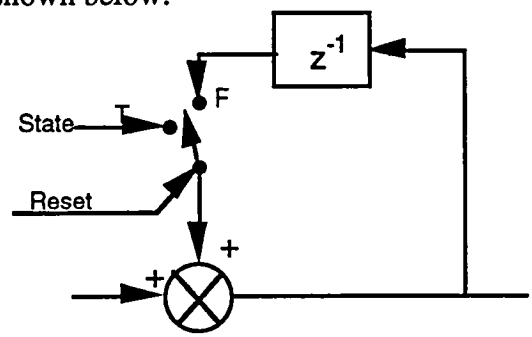


Figure A.Explicit Initialization

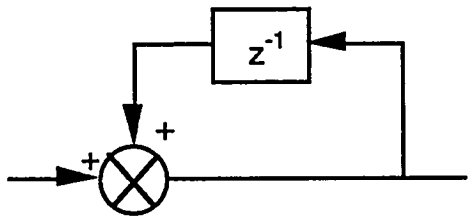


Figure B. Implicit Initialization

There is a direct MATLAB analog to using Explicit Initialization, as shown in Figure A. Although the scheme show in Figure B also has an analog in MATLAB, it is less clear. We recommend that you use Explicit Initialization at first, until you are comfortable with the way ASTER treats delays.

This means that the MATLAB version of each transform that contains a delay should be a function that accepts arguments for the reset signal and for the initial value of the state variable.

Define A Project Hierarchy For Your M-Files

In ASTER, all definitions reside in a hierarchy that has scope. That is, transforms can call other transforms, but only those that are defined within the scope of the caller. Below is brief definition of transform scope in ASTER.

Every transform in a project has one and only one superior. The set of superiors of a transform is its superior, plus the superiors of its superior. Transforms can have any number of inferiors, including 0. A given transform can use any of its direct inferiors, or any direct inferior of any of its superiors. In this note, we call this set of transforms the *scope* of the transform.

If you've defined an m-file that is relatively generic, you will want its ASTER analog to be within the scope of everything that needs it. You accomplish that by placing it near the top of the hierarchy. The more specific the m-file, the lower you can place it in the hierarchy.

We recommend that you create a directory structure for MATLABPATH that closely parallels the hierarchy you will establish in ASTER. This practice will help

you get used to the project geometry you intend to use for ASTER. It has no impact on the functioning of the MATLAB model, once you establish the correct MATLABPATH.

Declare Variable Types and Functions

MATLAB runs in interpretive mode. It always knows the type of every variable it deals with, because it can just look at it to find out. ASTER has no such luxury. To enable ASTER to understand what you mean, you have to declare types of variables and functions. See the next section for details.

No Need for Pre-Allocating Space For Matrices

In some MATLAB programs, you may find constructions involving matrices that appear to be double assignments. Actually, they are **merges** in ASTER, and they are perfectly legal in ASTER-style MATLAB. Although they are legal, they are also unnecessary in ASTER, and you may wish to remove them.

Here is an example.

The MATLAB manual recommends that you pre-allocate space for matrices to improve performance. For example,

```
y = zeros(1,100);
for i= 1:100
    y(i) = det(x^i)
end
```

The pre-allocation in the example above is accomplished by setting the value of y to a matrix of zeros. This eliminates the need to dynamically grow the matrix y, one step at a time, as you iterate over i.

In ASTER, on the other hand, there is no benefit to pre-allocation. In fact, the target language for your generated code probably requires an explicit type declaration, unless, of course, the target is MATLAB. So, assuming your target is not MATLAB, you will have included elsewhere in your project a type declaration for y, say, in the above example. If y appears for the first time in the algebraic transform definition, you can use instead the type declaration mechanism described in the next section.

There are several examples of this construction in the FENOC scripts, as noted below. These cases were found in the directory CoreCode. It is reasonable to expect to find other examples elsewhere in the scripts.

File name	*Variables Initialized*
CONTROL.m	U_hat
FEM.m	G
JACOBIAN.m	jacobian
REFINE.m	X_hat_new, LAMBDA_hat_new, and U_hat_new; X_bar, LAMBDA_bar, and U_bar; and Z.
UNPACK_NODES.m	X_hat, and LAMBDA_hat; X_bar, LAMBDA_bar, and U_bar;

Limitations on Iterations

Only one iteration per transform definition

You can have only one iteration in any given algebraic transform definition. That means only one statement of the type "while" or "for". To work around this constraint, break up any transform definition that contain more than one iteration. If you have N iterations, make N transforms. One of them can contain manifestations of the other N-1.

Isolate all iterations

Iterations must be isolated from code that isn't meant to execute as the body of the iteration.

This constraint is more difficult to explain. In a transform that contains an iteration, the MATLAB code can classified as Preliminary Computation, Initialization, Iteration, and Finalization. First we explain what is meant by these classifications, and then we describe the constraints and the work around.

Preliminary Computation

This code is not meant to execute as part of the body of the iteration, nor does it establish initial conditions for variables that are computed as part of the iteration.

For example, suppose you want to iterate over the elements of the kth column of a matrix. To do that, you first compute the value of *k* from the inputs to the transform. Then, in the body of the iteration, you access the matrix elements using the value of *k* you have computed.

The computation of *k* as described above is an example of a preliminary computation. It is not meant to execute as part of the body of the iteration, but it is used during the iteration.

Initialization This is the code that is directly involved in the iteration in connection with termination conditions or initial values.

Iteration This is the code that is directly involved in the body of the iteration.

Finalization This is code that is meant to execute after the iteration. For example, if we continue with our matrix/kth column illustration, suppose that we want to perform a matrix multiplication with that matrix after we have iterated over the kth column. That code would be in the category of Finalization.

The transform that contains the Iteration code can also contain Initialization. But it cannot contain either the Preliminary Computation or the Finalization. If it does, those portions must be removed, and inserted into one or more new algebraic transform definitions. Any such code that you do not remove will be executed as part of the iteration body.

Typically, you have to make only one new transform definition. It would contain both the Preliminary Computation and Finalization code, and a manifestation of the transform that contains the Iteration and Initialization code. But it is possible that you may prefer to move the Initialization code into the same definition that has the Preliminary Computation code. You have a certain amount of freedom in making these decisions.

4.0 Declarations in ASTER

Two kinds of declaration statements are provided. They are the function statement and an ASTER declare statement.

4.1 The Function Statement

This statement is available as an aid to converting existing scripts. You may also wish to run your scripts in, so ASTER allows this statement to facilitate use of your scripts in. The function statement in ASTER is completely compatible with that of, and must appear first in the body of the definition of the Algebraic Transform Definition.

4.2 The ASTER Declare Statement

You can declare symbols to be transforms or variables of various types. In some cases, you are required to make such declarations. In the following section, we describe the

circumstances under which you are required to make declarations, and how to make them. This section is just an introduction.

An example of a declaration is:

```
declare(state_vector, u, z, w)
declare(float, y)
```

This example declares u, z, and w to be of type state_vector, and y to be of type float.

Each symbol that appears in an algebraic transform definition can have a number of properties. For example, it can be pre-defined by, or it can be defined by the user. It can be a matrix, or it can be a scalar. Below are descriptions of the properties of symbols:

Origin	The origin of a symbol is either or the user. If defined by, you cannot make any declarations for that symbol.
Part of speech	The *part of speech* of a symbol is either noun or verb. A noun is a variable name. A verb is a transform name.
Category	Nouns of user origin can be any of five categories—input, output, parameter, constant, or local variable.
Type	Nouns of user origin can also have a type, such as float, integer, matrix, and so on. The full space of types includes user-defined types.

You can make declarations for the part of speech, the category, and the type for any symbol of user origin. You can make declarations of all three properties, or any combination of the three, but you can make only one declaration per property per symbol per algebraic transform definition. Declarations have lexical scope. That is, a declaration made in the body of a given algebraic transform definition has no effect outside the context of that definition.

Declarations, if they occur, must occur before any other statements of the body of the transform definition, except that the function statement, if it appears, must occur first. See Section 6 for a complete list of symbols of origin that are recognized in ASTER.

4.3 When Are You Required to Make Declarations?

In only one circumstance is a declaration required. When you have a transform and a matrix (or vector) that have the same name, you can refer to either one in the body of an algebraically defined transform, but you must declare which one you mean. Only one such declaration is permitted per definition. That is, you can*not* make a declaration that says "I mean the transform", then later on in the same definition say "I mean the matrix."

Any other declarations are optional. That is, you can declare a symbol to refer to a matrix, or another transform, or anything else you like. ASTER uses such information if provided.

4.4 What Can You Declare?

Below is a summary of property names and their meanings.

Property Name	*Meaning*
inputs, input	Declares the named symbol or symbols to be an input. Consequently, the symbol is implicitly a noun.
outputs, output	Declares the named symbol or symbols to be an output. Consequently, the symbol is implicitly a noun.
parameters, parameter	Declares the named symbol or symbols to be a parameter. Consequently, the symbol is implicitly a noun.
constants, constant	Declares the named symbol or symbols to be a constant. Consequently, the symbol is implicitly a noun.
local_variables, local_variable	Declares the named symbol or symbols to be a local variable. Consequently, the symbol is implicitly a noun.
any ASTER type	Declares the named symbol or symbols to be the named ASTER type. For built-in ASTER types, in most cases, you can use the name of the ASTER type as it appears in the type menu. For user-defined types, you may have to define a textual representation of the type for.
transform, transforms	Declares the named symbol to be a transform (a verb).

4.5 Other Restrictions

The variable NIL (all uppercase) is forbidden. allows you to assign a range value to a variable. For example, x=1:3; is a legal assignment statement. ASTER does not support this construct. Range accessors can be used only within a reference to a particular composite quantity.

4.6 MATLAB-Defined Symbols

abs	cos	if	rem
acos	cosh	inv	round
acosh	det	log	sin
asin	diag	log10	sinh
asinh	exp	logm	sqrt
atan	expm	max	sqrtm
atan2	eye	min	tan
atanh	fix	ones	tanh
ceil	floor	pi	trace
	for	rand	zeros

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1994	Contractor Report

4. TITLE AND SUBTITLE

Advanced Information Processing System: Hosting of Advanced Guidance, Navigation and Control Algorithms on AIPS using ASTER

5. FUNDING NUMBERS

C NAS1-18565

WU 506-59-61-03

6. AUTHOR(S)

Richard Brenner, Jaynarayan H. Lala, Gail A. Nagle, Andrei Schor, and John Turkovich

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

NASA CR-194960

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: Charles W. Meissner, Jr.
Final Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unclassified-Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This program demonstrated the integration of a number of technologies that can increase the availability and reliability of launch vehicles while lowering costs. Availability is increased with an advanced guidance algorithm that adapts trajectories in real-time. Reliability is increased with fault-tolerant computers and communication protocols. Costs are reduced by automatically generating code and documentation. This program was realized through the cooperative efforts of academia, industry, and government. The NASA-LaRC coordinated the effort, while Draper performed the integration. Georgia Institute of Technology supplied a weak Hamiltonian finite element method for optimal control problems. Martin Marietta used MATLAB to apply this method to a launch vehicle (FENOC). Draper supplied the fault-tolerant computing and software automation technology. The fault-tolerant technology includes sequential and parallel fault-tolerant processors (FTP & FTPP) and authentication protocols (AP) for communication. Fault-tolerant technology was incrementally incorporated. Development culminated with a heterogeneous network of workstations and fault-tolerant computers using AP. Draper's software automation system, ASTER, was used to specify a static guidance system based on FENOC, navigation, flight control (GN&C), models, and the interface to a user interface for mission control. ASTER generated Ada code for GN&C and C code for models. An Algebraic Transform Engine (ATE) was developed to automatically translate MATLAB scripts into ASTER.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Launch vehicle avionics, fault-tolerant systems, fault-tolerant computers, authentication protocols, finite element guidance law, automatic software generation	130
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		