SIMULINK™ MODULES THAT EMULATE DIGITAL CONTROLLERS REALIZED WITH FIXED-POINT OR FLOATING-POINT ARITHMETIC

A Thesis Presented to

The Faculty of the Russ College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Edward D. Robe

June, 1994

Acknowledgements

I would like to take this opportunity to thank the many individuals who have contributed to the development of this research project. In particular, I would like to thank my thesis advisor, Dr. Dennis Irwin, for his support, technical advice, and guidance. I would also like to especially thank Dr. Costas Vassiliadis for providing the many valuable opportunities that greatly enhanced my engineering education. Thanks is also due to Mr. Russell Glenn for his technical advice and the many discussions that we had concerning the development of this project. Most importantly, I would like to thank my parents and family for their encouragement and support.

Table of Contents

List of Figures

Chapter 1: Introduction

With the advent of more powerful and sophisticated computer systems, the simulations of digital control systems have been greatly improved by ways of speed, accessibility, complexity, and to some degree, accuracy. However, since digital control systems have a digital controller, which is typically realized with a microprocessor, and other associated hardware, there are inherent quantizing effects resulting from such an implementation. These quantization effects can introduce errors in the system due to the calculations that are involved with such a digital controller implementation. Thus, if the simulation of a digital control system does not take into account these effects, then the results of such a simulation will be inaccurate. Therefore, by emulating the fixed-point or floating-point arithmetic inherent in a digital controller and its associated hardware, these quantization effects are accounted for and a more accurate simulation can be achieved. In this thesis, new developments which have been made to simulate a digital control system more accurately with respect to these quantization effects are presented.

1.1. Organization of Thesis

This thesis contains five chapters and one appendix. Chapter 1 is the introduction which provides some background on the quantization problem and explains the motivation for this work. Chapter 2 presents the different mathematical formats for both fixed-point and floating-point arithmetic, as well as presenting the theory behind digital control

system implementation and analysis. The developed fixed-point and floating-point arithmetic emulation algorithms are presented in Chapter 3. Sections 3.3 and 3.4 present the process of integrating these emulation algorithms with that of the SIMULINK™ software. Chapter 4 presents a digital control system model which serves as an application to test the SIMULINK™ modules that emulate digital controllers realized with fixed-point or floating-point arithmetic. Results of these tests or simulations are presented throughout the various sections in Chapter 4. Finally, Chapter 5 discusses conclusions and recommendations for future work.

1.2. Motivation for Simulation Improvements

The motivation for simulating both the fixed-point and floating-point arithmetic processes is that there are certain hardware implementation issues that must be addressed in order for a digital control system to be emulated accurately. These hardware issues correspond to a digital control system's analog-to-digital (A/D) and digital-to-analog (D/A) converters, as well as its microprocessor unit (MPU), all of which must be accounted for in order to simulate a system correctly. Furthermore, the finite wordlength of each of these hardware devices must be taken into consideration since the wordlength dictates the amount of precision available in representing various numbers.

In this study, both fixed-point and floating-point number representation formats are examined with respect to their wordlengths. Furthermore, these two arithmetic

schemes are discussed with respect to their role in emulating digital controllers via the SIMULINK™ software.

SIMULINK™ is a window-based software program that allows for the simulation of dynamic systems; it is an extension to the MATLAB® software. [1, 2] It is the simulation environment that utilizes the work in this study.

SIMULINK™ is not inherently capable of emulating digital control systems that are realized with either fixed-point or floating-point arithmetic; however, it does provide the capability for incorporating specialized programming routines that do perform such operations. Therefore, C language-based software programs have been written that emulate both the fixed-point and floating-point arithmetic processes inherent in digital control systems and have been integrated into the SIMULINK™ environment.

1.3. Wordlength and Quantization

In order to simulate a digital control system accurately, the analyst must consider that the wordlength of the computer performing the simulation is typically longer than that of each of the wordlengths for the A/D, D/A, and MPU, all of which are to be used in the actual hardware implementation. As stated in [3], this introduces the task of coercing the computer performing the calculations for a given simulation to generate the same results as that of the actual A/D, D/A, and MPU.

Since most computers inherently use floating-point arithmetic, it is vital to the accuracy of the simulation that special coding be implemented in order to account for the differences in wordlength and arithmetic format. This is necessary for either emulating a fixed-point or a floating-point format.

With respect to either of these formats, the wordlength dictates the amount of precision available for which a number can be represented. In particular, three sources of errors arise due to the fixed-point calculations inherent in a digital controller and associated hardware that are realized with fixed-point arithmetic. First, the filter coefficients are modified or quantized due to the microprocessor's finite number of bits with which to represent them. Secondly, quantization of the system's inputs and outputs will result due to the finite wordlengths of the A/D and D/A converters. Thirdly, the quantization resulting from any arithmetic operations (e.g., additions or multiplications) performed by the MPU. [3] These same three sources of errors must be taken into consideration as well for the floating-point arithmetic emulation.

As described in [3], these three errors can be attributed to the finite wordlength of the system components. Therefore, when simulating a digital control system, whether it be for a fixed-point or floating-point format, the emulation code must not only account for these errors, but truncate or round-off numbers appropriately during the calculations in order to emulate the system correctly.

1.4. Past Developments on Digital Control System Simulation

There has been much study on the topic of quantization with respect to finite wordlengths of digital controllers. The three sources of errors mentioned earlier are the basic factors that have prompted the study on this topic. As stated in [3], authors, who have studied and analyzed this topic, have presented various ways in determining the degradation in response due to a digital controller that is implemented with a microprocessor and its associated hardware. [4, 5, 6, 7] To model the effects of quantization, these authors would introduce noise into the system. As stated in [8], some authors have proposed that by directly taking into account the round-off errors when designing a given digital controller, which is implemented with a finite wordlength, will prove to be an optimal solution in reducing quantization errors.

A software program has been written by Follett, [3], that emulates the fixed-point arithmetic processes of a digital controller, which can be cast into three different transfer function filter implementation schemes (e.g., direct, cascade, and parallel). His work provided much of the logistics in the development of a state-space realization scheme that implements fixed-point arithmetic emulation, and further served as a basis in the development of the floating-point arithmetic emulation.

1.5. The Goals of the Improved Simulation

The goals to be achieved in this work correspond to the implementation of both the fixed-point and floating-point arithmetic emulations within the SIMULINK™ environment. The integration of both of these arithmetic emulation processes within the SIMULINK™ environment will provide a means for simulating digital controllers, which are realized with either fixed-point or floating-point arithmetic, more accurately as well as providing a work-space that is simple and easy to use. Furthermore, with this type of graphical user-interface (GUI) environment, the wordlengths pertaining to the microprocessor and its associated hardware (e.g., A/D and D/A) can be easily adjusted to any given specifications, therefore, allowing for significant versatility for various finite wordlength emulations.

With this capability, the simulations of digital control systems are greatly enhanced with respect to the accuracy of their calculations for a given implementation scheme and its associated finite wordlengths. The accuracy that is obtained will allow the analyst to better evaluate the various designs on a digital controller within a digital control system.

Chapter 2: Theory

In order to understand the difference between fixed-point and floating-point arithmetic, it is necessary to examine their respective formats in relation to the wordlength. Knowledge of the way numbers are represented in a given wordlength and format will provide a better understanding of the concept of quantization. Furthermore, since algorithms have been developed that emulate digital controllers realized with fixed-point or floating-point arithmetic, it is necessary to understand the fundamentals of digital control system analysis and the effect of quantization on the stability of a digital controller and the closed loop system. Therefore, the z-domain, which is the transform domain in which digital control system analysis is performed, will be examined. This chapter examines both fixed-point and floating-point arithmetic formats, the definition of the z-transform, the various algorithms for implementing a digital controller, and the stability requirements of the z-domain.

2.1. Fixed-Point Arithmetic

The range of numbers that can be represented in a binary form mainly depends upon the wordlength. The wordlength corresponds to how many bits are allocated to represent a number; it consists of L+1 bits where L bits are used to represent the numerical value and one bit is used for the sign. Furthermore, in a fixed-point representation, the binary point or radix-2 point is fixed and therefore affects the range

of numbers that can be represented. The binary representation of an unsigned integer can be expressed as

$$n = \sum_{i=0}^{B-1} n^i \cdot 2^i \tag{2.1}$$

where B represents the number of bits in the wordlength and $n^i = 0$ or 1. For example, the binary number 11011 represents the decimal number 27, since

$$(1 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 16 + 8 + 2 + 1 = 27.$$

A fixed-point number is essentially separated into three parts: sign, integer, and fraction. However, depending on the location of the implied radix point, the fixed-point number may only be divided into two parts; e.g., sign and integer or sign and fraction. Nonetheless, these three parts are represented within a given wordlength (e.g., 8-bit, 16-bit, etc.) which has an implied radix point. The general notation format for a fixed-point number is

$$(-1)^s \cdot (Int + .F) \tag{2.2}$$

where s denotes the sign bit (0 = positive, 1 = negative), Int is the integer part, and .F is the fractional part. As shown in [9], an n-bit Int in binary form can be represented as

$$Int = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \cdots + b_0 \cdot 2^0 \tag{2.3}$$

and an m-bit .F can be represented as

$$.F = a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + \cdots + a_{-m} \cdot 2^{-m} \qquad (2.4)$$

where n+m is equal to the wordlength minus one (1 bit is reserved for the sign), and a_i's and b_i's are represented by either a 1 or 0.

Using Equation 2.1 it can be seen that the largest integer that can be represented by a B bit binary number is 2^B-1. This representation can be modified in order to represent fractional numbers. This is done by assuming a fixed scaling by a negative power of 2 on all numbers (as in Equation 2.4). For example, suppose 10 bits are used to represent decimal numbers where all numbers are scaled by $2^{-5} = 0.03125$. Thus, the largest integer that can be represented is $(2^{10} - 1) * (2^{-5}) = 31$, and the smallest fraction is 0.03125. As stated in [4], this is equivalent to assuming a radix point in the fifth position to the left of the least significant bit (LSB) and computing the fractional part as in Equation 2.1 by only using negative powers of 2.

Depending on the way negative numbers are represented, there are three different forms of fixed-point arithmetic. They are called sign-magnitude, 2's-complement, and 1's-complement representation. The sign-magnitude representation (which is used in the development of the fixed-point arithmetic emulation algorithms) utilizes the leading bit in a given wordlength (B) to represent the sign, 0 for positive values and 1 for negative values, and the remaining (B-1) bits are used to represent the numerical value or magnitude of a number. With the sign-magnitude form, the number 0 has two

representations, +0.0 and -0.0. In binary (assuming B=5 with two places allocated to the right of the radix point), this representation would be 000.00 and 100.00.

In 2's-complement notation, positive numbers are the same as those of the sign-magnitude representation. However, the negative of a positive number is obtained by complementing all the bits and adding 1 in the LSB, e.g.,

$$\sim(000.11) = (111.00) + (000.01)$$

$$= 111.01 \quad .$$

Equivalently, a 2's-complement representation of a negative number can be obtained by taking the absolute value and subtracting it from 2 (This is assuming that all numbers are scaled by 2^{-B} which implies that the binary point is positioned to the left of the most significant bit (MSB) and therefore makes all numbers less than or equal to 1 in magnitude.). This procedure will result in the first bit always being one. A mathematical interpretation of the 2's-complement notation can be expressed as

$$n = -n^0 + \sum_{i=1}^{B-1} n^i \cdot 2^{-i} \qquad (2.5)$$

where the sign bit is multiplied by -1. To illustrate, the 2's-complement representation of -0.375 is obtained by 2 - 0.375 = 1.625, which has a binary representation of 1101. Using Equation 2.5, the binary number 1101 represents the decimal number

$$-1 + (1 \cdot 2^{-1}) + (0 \cdot 2^{-2}) + (1 \cdot 2^{-3}) = -1 + 0.625$$

$$= -0.375$$

Furthermore, with a 2's-complement representation, zero has only one representation (0000···0), and there is one more negative number than there are positive numbers.

The 1's-complement form represents positive numbers the same way the sign-magnitude and 2's-complement representations do, but negative numbers are obtained by complementing all the bits of its positive counterpart, e.g.,

$$\sim(000.10) = 111.01$$

With respect to the three different forms for implementing fixed-point arithmetic that were described above, the location of the radix point is fixed for all numbers, and therefore it is implied and not physically stored. Consequently, due to the radix point being fixed, it is evident that a fixed-point arithmetic format has a limited or finite range for representing numbers.

2.2. Floating-Point Arithmetic

In general, the floating-point representation of a positive number consists of two fixed-point numbers, the mantissa and the exponent. [5] The floating-point number f is obtained as the product of the mantissa m with that of a given radix (typically radix-2)

raised to the power of the exponent n, e.g.,

$$f = m \cdot 2^n \tag{2.6}$$

The mantissa is usually normalized and falls within the range

$$\frac{1}{2} \leq m < 1 \tag{2.7}$$

As well, the range of the exponent is limited due to the finite number of bits allocated for its wordlength.

Negative floating-point numbers are typically represented by having the mantissa serve as a signed fixed-point number. Therefore, the sign of a floating-point number is obtained from the first bit of the mantissa. The exponent is also a signed fixed-point number where negative exponents represent numbers with magnitudes less than 0.5.

In contrast, an IEEE standard floating-point number is comprised of three fields: sign, exponent, and mantissa. As stated in [10], the only values representable in a given format are those specified via the following three integer parameters:

p = the number of significand bits (precision)

E_{max} = the maximum exponent

E_{min} = the minimum exponent

With this, a number is expressed in the form

$$(-1)^s \cdot 2^E \cdot (b_0 \,.\, b_1 b_2 \cdots b_{p-1}) \tag{2.8}$$

where s is the sign bit, E is any integer between E_{min} and E_{max}, and $b_0.b_1b_2 \cdots b_{p-1}$ is the mantissa. However, since there are a finite number of bits in the mantissa, the exponent can be adjusted so that a nonzero real number is *normalized* (e.g., 1.F where F represents a fraction) and, therefore, assigns a 1 to the b_0 bit. In other words, all the leading zeros of a nonzero real number are deleted and not included in the mantissa representation, thus reducing computer roundoff errors. By normalizing a number, it allows for maximum precision in representing that number. Furthermore, with a normalized number, there is no need to physically store the value of the b_0 bit since it is implied that it is a 1.

As shown in [10], the IEEE standard on binary floating-point arithmetic supports two basic formats: single-precision and double-precision. In both formats, numbers are comprised of the following three fields:

(1) 1-bit sign s

(2) Biased exponent e = E + bias

(3) Fraction f = $.b_1b_2 \cdots b_{p-1}$

In the single-precision format, a number is represented in a 32-bit wordlength. The biased exponent e and the fraction f have 8 and 23 bits, respectively, as shown in Figure 2.1.

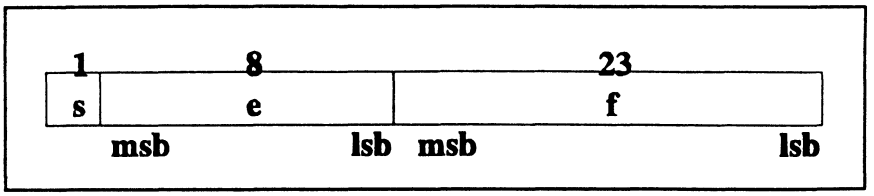

Figure 2.1: Single-Precision Format

As stated in [10], the value x of a real number can be determined by its constituent fields such that:

(1) If e = 255 and f ≠ 0, then x is NaN (Not a Number) regardless of s

(2) If e = 255 and f = 0, then $x = (-1)^s \infty$

(3) If 0 < e < 255, then $x = (-1)^s 2^{e-127}(1.f)$

(4) If e = 0 and f ≠ 0, then $x = (-1)^s 2^{-126}(0.f)$ (denormalized numbers)

(5) If e = 0 and f = 0, then $x = (-1)^s 0$ (zero)

In the double-precision format, a number is represented in a 64-bit wordlength. The biased exponent e and the fraction f have 11 and 52 bits, respectively, as shown in Figure 2.2.

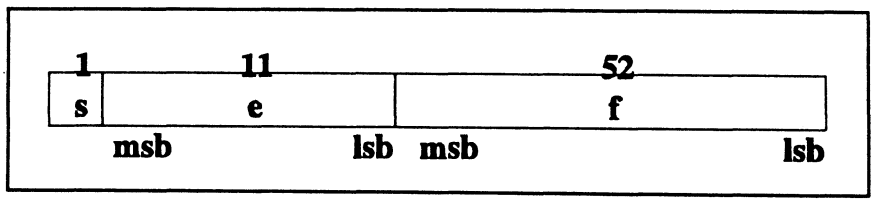

Figure 2.2: Double-Precision Format

As stated in [10], the value x of a real number can be determined by its constituent fields

such that:

(1) If e = 2047 and f ≠ 0, then x is NaN (Not a Number) regardless of s

(2) If e = 2047 and f = 0, then $x = (-1)^s \infty$

(3) If 0 < e < 2047, then $x = (-1)^s 2^{e-1023}(1.f)$

(4) If e = 0 and f ≠ 0, then $x = (-1)^s 2^{-1022}(0.f)$ (denormalized numbers)

(5) If e = 0 and f = 0, then $x = (-1)^s 0$ (zero)

2.3. z-Transforms

When studying sampled-data or discrete-time control systems, the analysis of a system involves the use of the z-transform. The z-transform is defined for any number sequence e(n), and can be used in the analysis of a system that can be described by linear, shift-invariant difference equations. The z-transform of the number sequence e(n) is defined as

$$E(z) = \sum_{n=-\infty}^{\infty} e(n) \cdot z^{-n} \qquad (2.9)$$

where z is a complex variable. The sequence e(n) is generated from a continuous-time function e(t) by sampling every T seconds, where e(n) is understood to be e(nT). [11] Therefore, this definition shows that the quantity z^{-1} is equivalent to a time delay of one sample period. The z-transform is a useful tool in analyzing the time domain implementation of a digital controller.

2.4. Algorithms for Implementing Digital Controllers

To implement a digital controller, it is necessary to develop an algorithm that can be programmed in a digital computer, e.g. a microprocessor. As described in [3], one such algorithm involves using a recurrence equation. A recurrence equation is obtained by manipulating the transfer function's output to input relationship, where the present output is a function of the present and past inputs and past outputs. In [3], this type of algorithm is outlined in great detail and further discusses the various effects that controller implementation schemes (e.g., Parallel, Direct, and Cascade) have on the stability of a system with respect to coefficient quantization.

Another algorithm which can be used to implement a digital controller involves the use of state-space equations. State-space equations for a linear time-invariant system are defined as

$$x(n + 1) = Ax(n) + Bu(n) \tag{2.10}$$

$$y(n) = Cx(n) + Du(n) \tag{2.11}$$

where A, B, C, and D are matrices that constitute a state-space realization, and u(n), x(n), and y(n) are the system's input, discrete states, and output vectors, respectively. A is an $n \times n$ matrix which contains the coefficients of the system's plant model, where n corresponds to the number of states. B is an $n \times r$ matrix which relates the system's states to its inputs, where r corresponds to the number of inputs. C is a $q \times n$ matrix

which relates the system's outputs to its states, where q corresponds to the number of outputs. Lastly, D is a $q \times r$ matrix which relates the system's outputs to its inputs.

With respect to either of these algorithms, a physically realizable linear digital controller may be written in the following form:

$$G_C(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_n z^{-n}}{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_m z^{-m}} \qquad (2.12)$$

where m and n are positive integers, a_0 and b_0 are not both zero, and m is greater than or equal to n. [12] The expression

$$G_c(z) = \frac{C(z)}{R(z)} \qquad (2.13)$$

represents the transfer function of the controller where C(z) is the output and R(z) is the input. This expression (Equation 2.13) will be used in the next section for the purpose of discussing the stability requirements in the z-domain with respect to digital controllers.

2.5. Stability in the z-Domain

To better understand the effects of quantization on a microprocessor realized digital controller that implements either fixed-point or floating-point arithmetic, an introduction to stability analysis in the z-domain will be presented. A derivation of the stability requirements in the z-domain will be conducted with respect to the linear digital controller given by Equations 2.12 and 2.13.

Using partial fraction expansion, $G_c(z)/z$ can be resolved into a sum of first order terms such that the following results:

$$\frac{G_c(z)}{z} = \frac{q_0}{z-p_0} + \frac{q_1}{z-p_1} + \cdots + \frac{q_n}{z-p_n} \quad . \tag{2.14}$$

where q_i and p_i may be positive or negative real numbers, some of which may be repeated. Multiplying both sides of Equation 2.14 by z gives

$$G_c(z) = \frac{q_0 z}{z-p_0} + \frac{q_1 z}{z-p_1} + \cdots + \frac{q_n z}{z-p_n} \quad . \tag{2.15}$$

Now taking the inverse z-transform of $G_c(z)$ yields

$$g_c(KT) = q_0(p_0^K) + q_1(p_1^K) + \cdots + q_n(p_n^K) \tag{2.16}$$

or

$$g_c(KT) = \sum_{K=0}^{\infty} q_0(p_0^K) + q_1(p_1^K) + \cdots + q_n(p_n^K) \quad . \tag{2.17}$$

It is evident that as $K \rightarrow \infty$, if any of the p_n's are greater than one in magnitude, the terms will increase without bound, indicating an unstable response. Furthermore, it can be seen that if the roots of the denominator have a magnitude greater than one, then the response of the filter will be unstable. Therefore, the region interior to the unit circle corresponds to the region of stability in the z-domain.

With respect to the above derivation, it is apparent that the pole locations of a

digital filter have a significant effect on the magnitude of its response. Thus, any quantization of a digital filter's coefficients will alter the response of the filter, and therefore possibly cause instability.

Chapter 3: Fixed-Point and Floating-Point Arithmetic Emulation Code

In the development of control systems, much analysis of the system's performance is conducted via simulation before the system is actually implemented. These simulations, however, are typically performed on a digital computer that has a longer wordlength than that of the hardware (e.g., a microprocessor, A/D, D/A) which is to be used in the actual implementation. Furthermore, another important issue that must be considered in a simulation is the arithmetic process (e.g., fixed-point or floating-point arithmetic) which is to be emulated. Therefore, this chapter discusses the task of coercing the computer performing the simulation to produce the same results as the actual hardware implementation would.

With the background information on both fixed-point and floating-point arithmetic as described in Chapter 2, it is now possible to develop the necessary algorithms which emulate digital controllers realized with either of these two arithmetic processes. Furthermore, simulation software has been chosen that allows for the implementation of these emulation algorithms.

Section 3.1 discusses the fixed-point arithmetic emulation code. Section 3.2 covers the floating-point arithmetic emulation code. Section 3.3 provides some insight on the development of the interfacing code between the emulation algorithms and the simulation software; the simulation software utilized is called SIMULINK™. Finally,

section 3.4 discusses the development of the various graphical user-interface (GUI) blocks, which were created with SIMULINK™'s S-Function block, used to emulate the hardware inherent in a digital control system.

3.1. Fixed-Point Arithmetic Emulation

Fixed-point arithmetic emulation can be accomplished on a computer which uses floating-point arithmetic by binarily shifting the resultant numbers of each calculation and then extracting only the integer portion. As stated in [3], this is equivalent to a microprocessor's calculations (which utilizes fixed-point arithmetic) since it performs all the calculations in integer arithmetic with an implied radix point. Furthermore, by adding ± 0.5 to the shifted number prior to taking the integer portion, this will round off the number and, therefore, reduce the quantization error as opposed to simple truncation. As shown in [3], this procedure can be expressed in equation form as

$$N(X) = IP[X \cdot 2^p + r \ \ sgn(X)] \tag{3.1}$$

where N(X) is the resultant integer, IP[ ] is the integer part operator, X is the number to be 'massaged', p is the number of binary places desired to the right of the radix point, r is the designation for rounding or truncation (0 for truncation and 0.5 for rounding), and sgn( ) is the sign operator.

As an example, suppose the number 8.29 is to be represented as a rounded 16-bit integer with 5 bits to the right of the radix. Equation 3.1 yields:

$$N = IP[(8.29)(2^5) + 0.5]$$

or

$$N = 265.$$

In binary, this number is represented as $(100001001)_2$. Note that when the radix point is inserted in the desired position (i.e., p = 5), the number is $(1000.01001)_2 = (8.28125)_{10}$, which is the generated decimal representation of the massaged number.

This method of representing decimal numbers is the process used for emulating fixed-point arithmetic inherent in a microprocessor. Therefore, the state-space equations previously discussed must perform its mathematical operations in an appropriate manner in order to emulate a digital controller realized with a fixed-point arithmetic microprocessor.

The mathematical operation of addition is simple and does not require any special handling except in the case of overflow. If the magnitude of the resulting sum exceeds the microprocessor's useable wordlength (B), where B+1 is the total wordlength (one bit reserved for the sign), then the magnitude is set to 2^B-1. As stated in [3], this method of handling overflows is one that produces the least amount of error.

When two fixed-point numbers are multiplied, this results in a fixed-point number that has twice the number of bits to the right of the radix point. The microprocessor, therefore, binarily shifts the number to the right corresponding to the number of places specified by p. From [3], this process is expressed in equation form as

$$M(x) = IP[x \cdot 2^{-p} + r \ \ sgn(x)] \tag{3.2}$$

where x is the floating-point number resulting from the multiplication of two numbers which have been operated on by Equation 3.1.

For example, suppose that the two numbers 8.29 and 3.76 are to be multiplied using 16-bit fixed-point arithmetic with 5 binary places to the right of the radix point. Using Equation 3.1, the results obtained are N(8.29) = 265 and N(3.76) = 120. When multiplied, these two integer numbers equate to 31,800. Therefore, this number is subsequently used in Equation 3.2 which results in M(31,800) = 994. Converting this number to its decimal equivalent produces the expected result of 31.0625.

Equations 3.1 and 3.2 provide the basis for the C language-based algorithms that were developed for emulating digital controllers that are realized with fixed-point arithmetic. These algorithms are contained within the appendix.

3.2. Floating-Point Arithmetic Emulation

Floating-point arithmetic is similar to fixed-point arithmetic in that the amount of

precision available for which a number can be represented is limited to the number of bits in a given wordlength. Furthermore, as with fixed-point arithmetic, floating-point arithmetic represents numbers in a binary integer format. However, in contrast to fixed-point, floating-point arithmetic does not have a fixed radix point (hence the name, 'floating-point') and, therefore, is able to represent a broader range of numbers. Due to this fact, the special coding that is necessary to perform the floating-point arithmetic emulation is rather extensive as compared to that of the fixed-point arithmetic emulation.

Since a simulation is conducted on a computer which in most cases uses floating-point arithmetic, there are no programming concerns with respect to format changes as is the case with the fixed-point arithmetic emulation described in section 3.1. However, the floating-point arithmetic emulation must represent numbers as defined by a given system's specifications (i.e., exponent and mantissa wordlengths). Furthermore, each of the operations specified in [10] (except for binary↔decimal conversion) "shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's format ... ." In other words, the floating-point arithmetic emulation allows for all mathematical operations to follow this standard and then appropriately quantizes the results of these operations according to a given system's wordlength specifications.

To illustrate the process of representing a floating-point number inclusive of its exponent and mantissa wordlengths, the following C code (Figure 3.1) represents a

sequence of statements and function calls that are carried out in order to quantize a number.

```
for (i = 0; i < nstates; i++) {
      for (j = 0; j < nstates; j++) {
                  .
                  .
            number = apr[i + nstates*j];
            extractexpo(number,&E);
            number2int(number,&integral,&fractional);
            convert(number,E,integral,fractional,&NUMBER);
            number = NUMBER;
            roundcheck(i,j,&number,&signalflag);
            (checks for "NaN" and "Inf" conditions)
            apr[i + nstates*j] = number;

            } /* End "j" loop */
} /* End "i" loop */
```

Figure 3.1: C code for quantizing a number

First, the 'number' is passed as an input argument to the extractexpo( ) function, which returns the corresponding biased exponent 'E'. Secondly, both the integer and fractional parts of a number are separated and placed in appropriate variable types. Thirdly, a call is made to the convert( ) function which in itself contains several other calls to functions, which perform the actual bitwise operations and calculations that generate the equivalent floating-point number representation for a given wordlength and format.

As an example, suppose the number 45.124 is to be represented in single-precision format (i.e., 8-bit exponent and 23-bit mantissa). The returned values for E, integral, and fractional, are 132, 45, and 0.124, respectively. In convert( ), the binary representation

of integral is

00000000 00000000 00000000 00101101 .

It is then binarily shifted to delete all leading zeros ('*normalization*') as shown here

10110100 00000000 00000000 00000000 .

Now, the fractional part needs to be scaled accordingly since the bitwise operators of C only function with integers (i.e., 0.124 x 2^{32} = 532575944). The scaling factor, 2^{32}, is based on a machine-dependent 4 byte unsigned long integer. If fractional parts are less than (1 / 2^{32}), then other scaling factors take place. In binary, the integer number 532575944 is expressed as

00011111 10111110 01110110 11001000 .

This scaled fractional is then binarily shifted to delete all leading zeros as shown here

11111101 11110011 10110110 01000000 .

With the use of counters and bitwise operators, the binary representation of both the integral and fractional parts together is

10110100 01111110 11111001 11011011 .

Extracting the first twenty-four bits of this number produces

10110100 01111110 11111001

which represents the mantissa. These bits are then used in a function called 'bit2deci(E, f, mantissa, &quantnumber)', which calculates the equivalent decimal representation. Therefore, the convert( ) function returns the quantized number 45.1239967346191406.

The C program which emulates floating-point arithmetic inherent in a digital controller is contained within the appendix.

3.3. SIMULINK™ and the Development of C MEX-files

SIMULINK™ is a graphical-interface software system that allows for the creation and simulation of dynamic system models; it is an extension to the MATLAB® software. In SIMULINK™, a system model is created and stored as an S-Function (for System-Function), which is the mechanism that defines and characterizes the dynamics of a system.

There are three different types of S-Functions: graphical, M-file, and MEX-file. Each of these types has its own interfacing mechanism; the graphical uses the

SIMULINK™ block diagrams, the M-file uses the MATLAB® language, and the MEX-file (which stands for Matlab Executable-file) uses either the C or Fortran language. As stated in [1], with respect to either of these types, an S-Function has the calling syntax

$$sys = model(t, x, u, flag, extra\_parameters) \tag{3.3}$$

where *model* is the model name and *flag* controls the information returned in *sys*. For example, a *flag* set to 2 returns the discrete states in the variable *sys* at the operation point defined by the time t, discrete state vector x, and input vector u. The *extra_parameters* argument allows for the inclusion of any additional data that is needed to model a system.

When creating a MEX-file, the source code must consist of two distinct portions. The first is a 'computational routine' that contains the source code for performing the actual numerical computation that is to be implemented in the MEX-file. The second portion consists of a 'gateway routine' that interfaces the computational routine with that of the MATLAB® / SIMULINK™ software. [13] The entry point to the gateway routine must be named 'mexFunction' and have the arguments shown as

$$[a,b,c,...] = function\_name(d,e,f,...) \tag{3.4}$$

These arguments are representative of the MATLAB® arguments with which a user invokes the MEX-function. The a,b,c,... are lefthand-side (LHS) arguments and the d,e,f,... are righthand-side (RHS) arguments. More specifically, the arguments *nlhs* and *nrhs* correspond to the number of LHS and RHS arguments, respectively. 'plhs' is a pointer to an array of length nlhs, and 'prhs' is a pointer to an array of length nrhs.

These arguments and a 'mexFunction' can be found in the 'SIMULIN2.H' header file contained in the appendix. This header file performs the operation of interfacing the computational routine with that of the gateway routine. The compilation of these two routines together comprise an S-Function C MEX-file.

3.4. SIMULINK™ Modules

Having created the C programs (MEX-files) which emulate both the fixed-point and floating-point arithmetic processes, it is now possible to integrate these programs in the SIMULINK™ environment as S-Functions. Graphical representations of these C MEX-files, which emulate the A/D, D/A, and MPU processes of a digital controller, are made possible via SIMULINK™'s S-Function block (See Figure 3.2). The S-Function block has two fields: the first field allows the name of a previously created S-Function (of any type) to be entered, and the second field allows for the entry of any additional parameters that are needed for a given system.

For example, the compiled C program MPUFL.CMEX4, which emulates the microprocessor implemented digital controller realized with floating-point arithmetic, can be represented as an S-Function block by inserting the name 'mpufl' and the parameters 'A,B,C,D,X0,MPUSETUP' in the 'Subsystem function name' and 'Function parameters' fields, respectively. The 'MPUSETUP' parameter contains the system characteristics such as exponent and mantissa wordlengths. A similar process can be performed for both the

A/D and D/A converters as well. Furthermore, these S-Functio
customized graphically via a *masking* mechanism available in SIMULI
3.3).

The masking mechanism allows a SIMULINK™ graphical block to be redefined in terms of its dialog box, icon, and initialization commands. In particular, the S-Function C MEX-files that have been developed which emulate a digital controller's fixed-point or floating-point arithmetic processes, have been incorporated as S-Function blocks using this masking capability. With this capability, these masked S-Function blocks are more 'user-friendly' to the analyst who is simulating various digital controllers with respect to its wordlength and arithmetic process.

In SIMULINK™, two additional libraries have been created that contain S-Function blocks representing both fixed-point and floating-point arithmetic implemented hardware devices (e.g., A/D, D/A, and MPU); these libraries are saved as M-files. The fixed-point and floating-point arithmetic libraries exist as shown in Figures 3.4 and 3.5, respectively. With respect to the fixed-point library, the dialog box interfaces for the A/D, MPU, and D/A blocks are as shown in Figures 3.6, 3.7, and 3.8, respectively. For the floating-point library, the dialog box interface for the MPU block is shown in Figure 3.9. The dialog prompts for each of these blocks are easily understood and correspond to the required input parameters needed in order for a given digital controller to be emulated with respect to a given finite wordlength and arithmetic process.

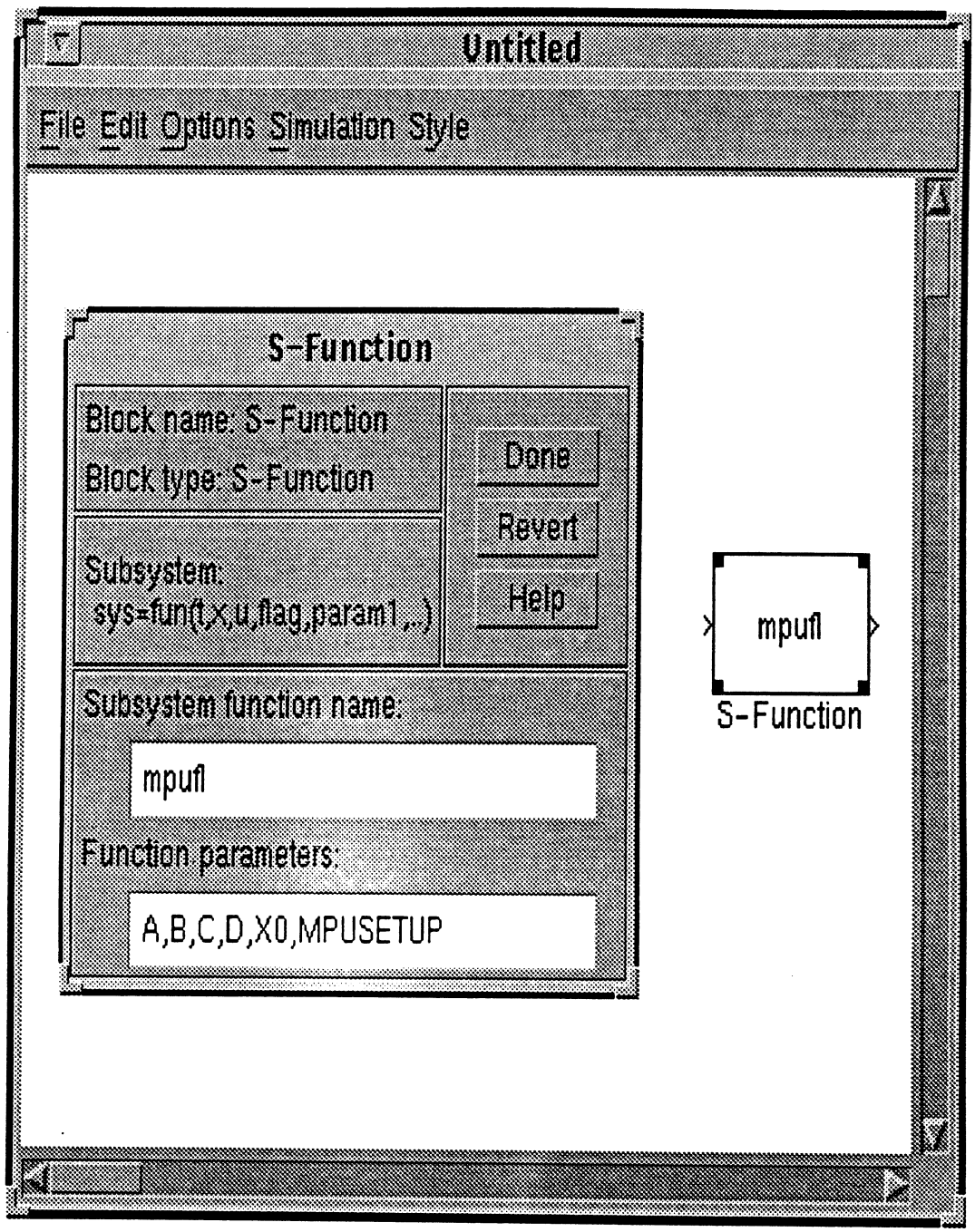

Figure 3.2: A SIMULINK™ S-Function Block

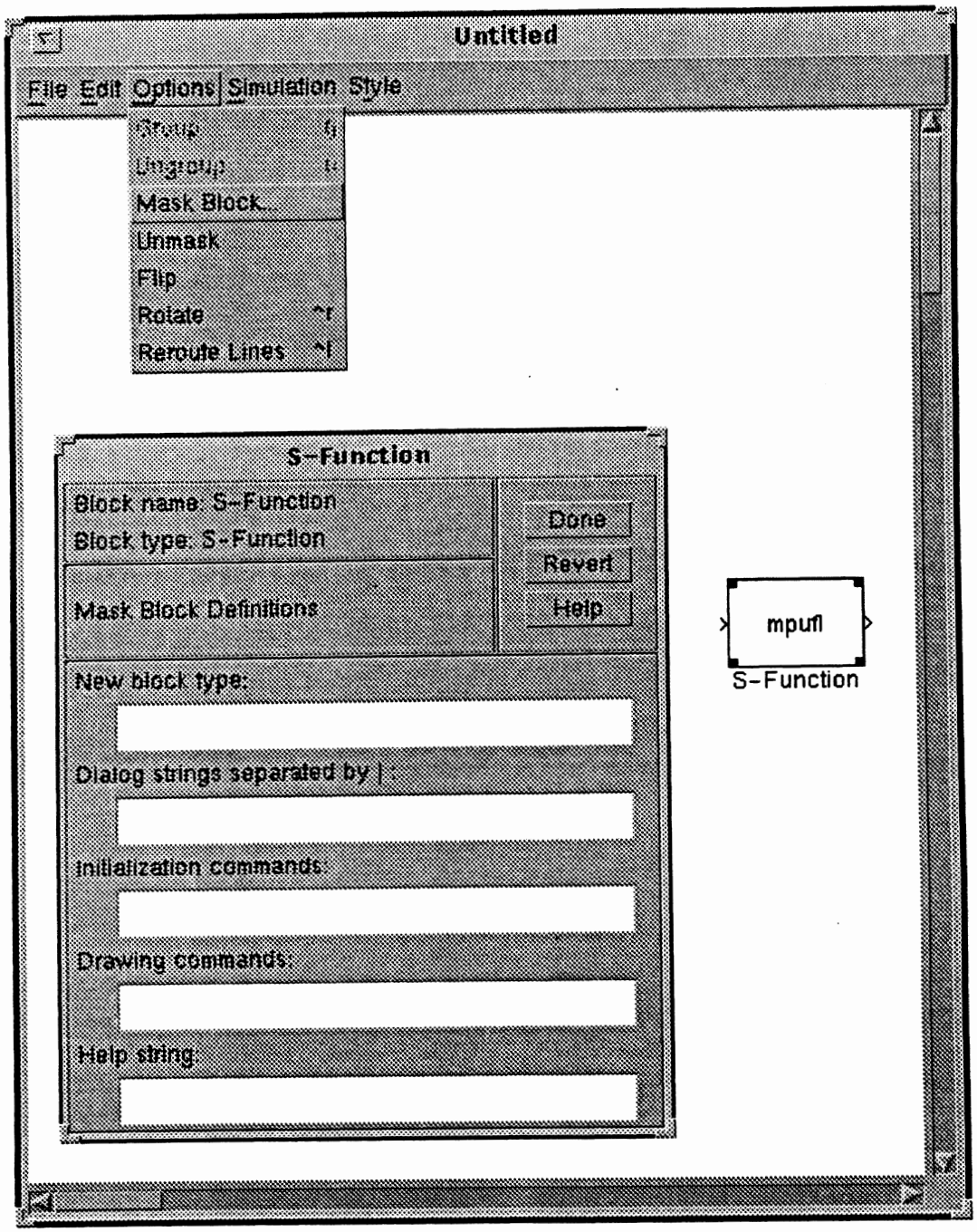

Figure 3.3: Masking an S-Function Block

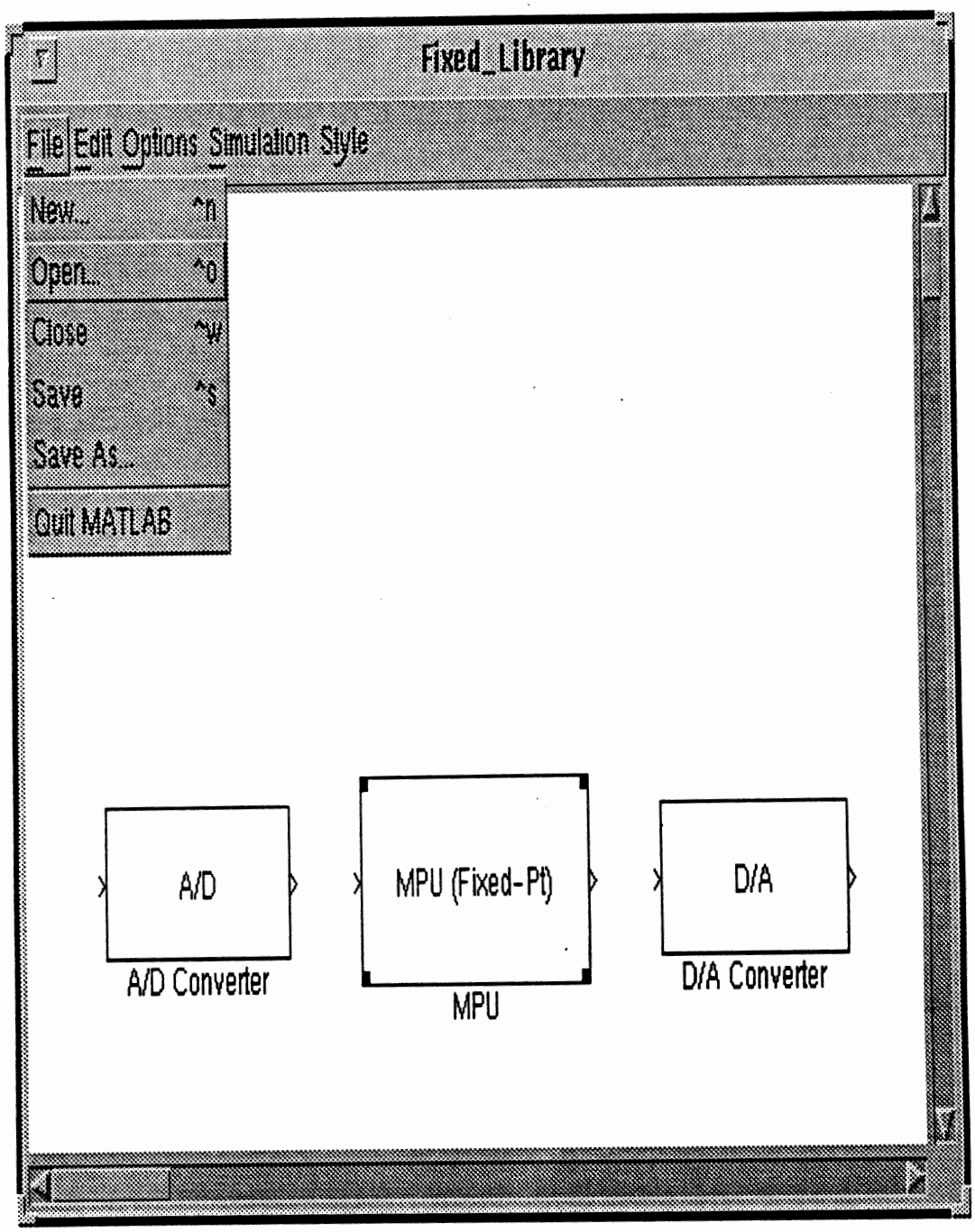

Figure 3.4: The Fixed-Point Arithmetic Library

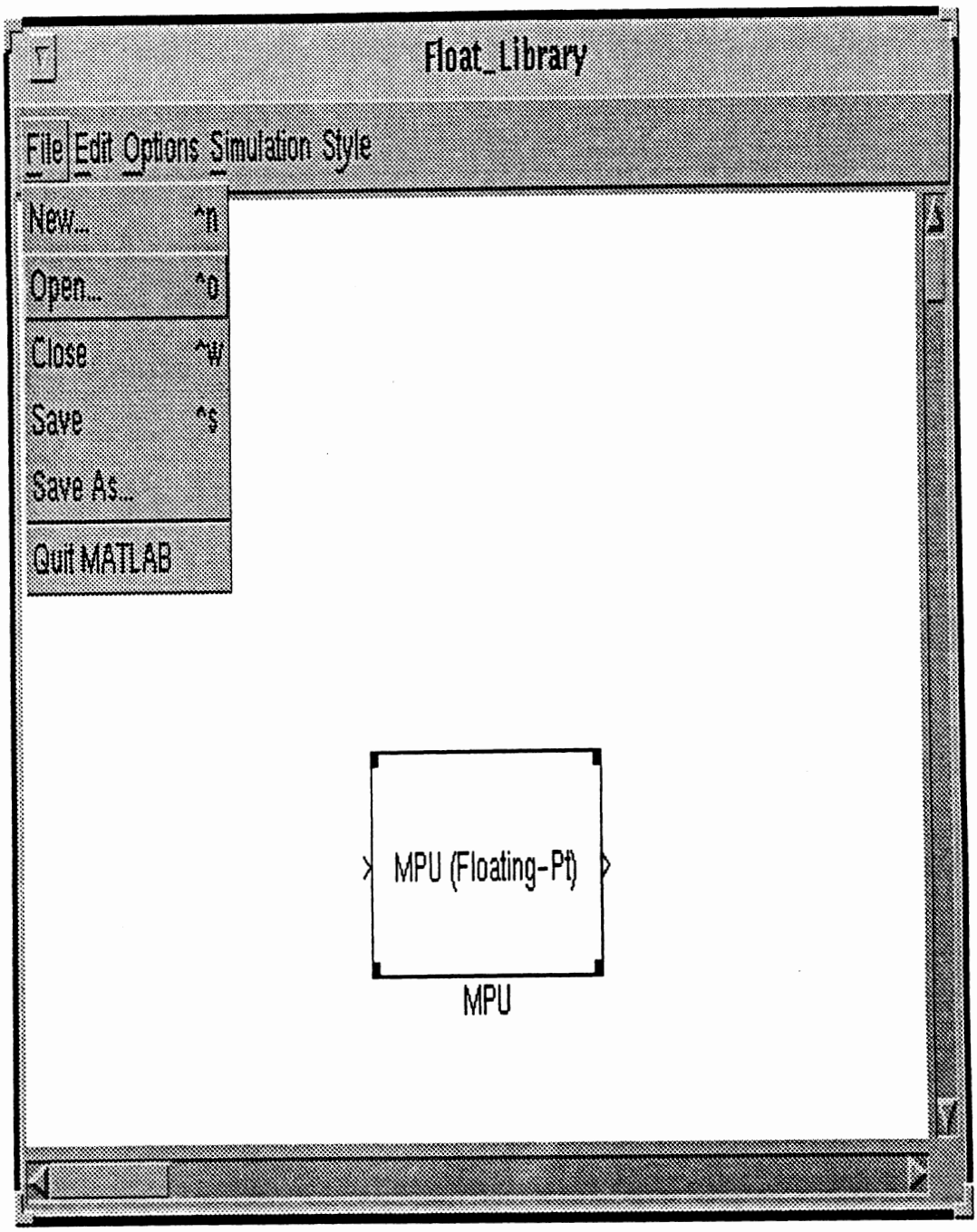

Figure 3.5: The Floating-Point Arithmetic Library

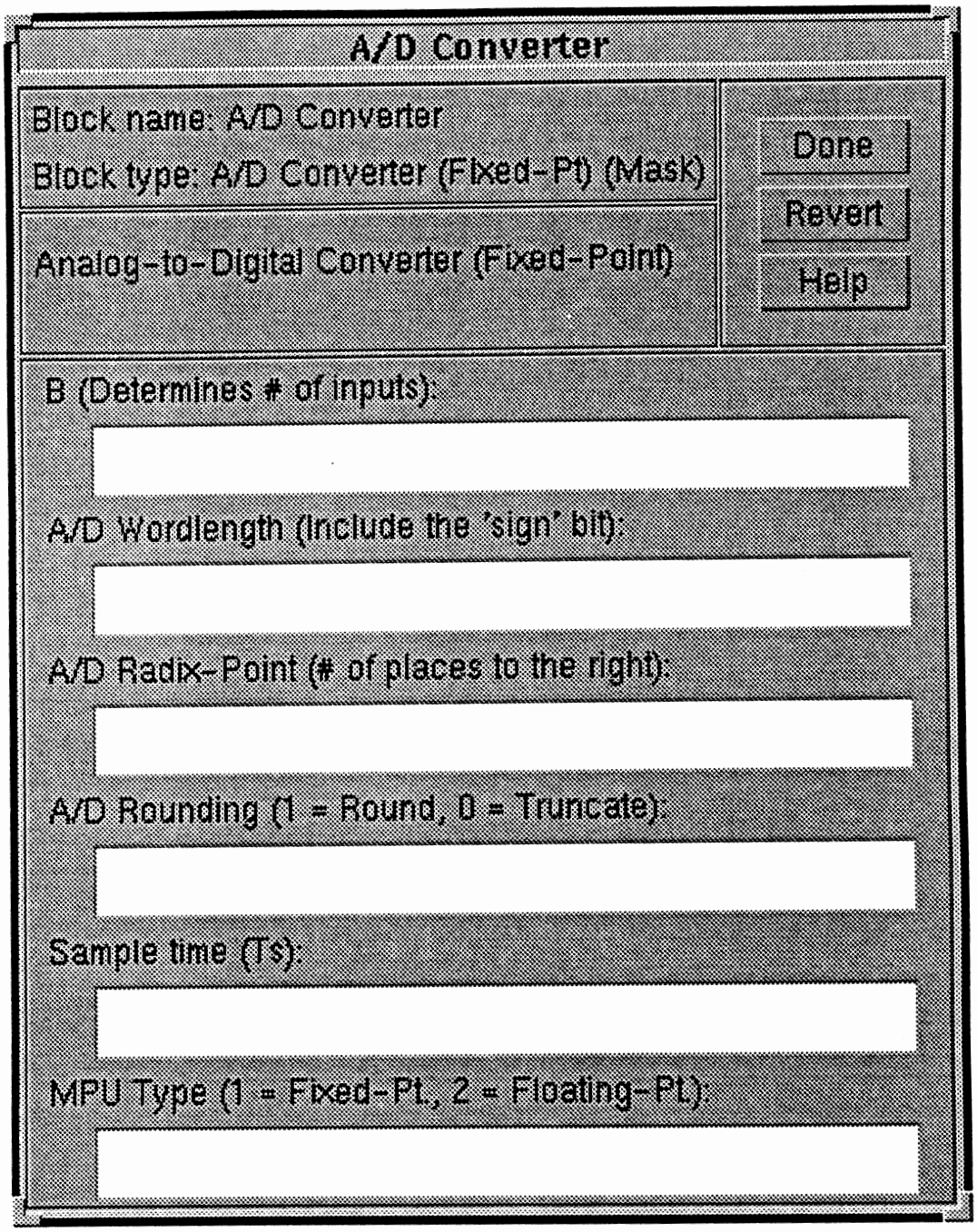

Figure 3.6: The A/D Converter Block (Fixed-Pt.)

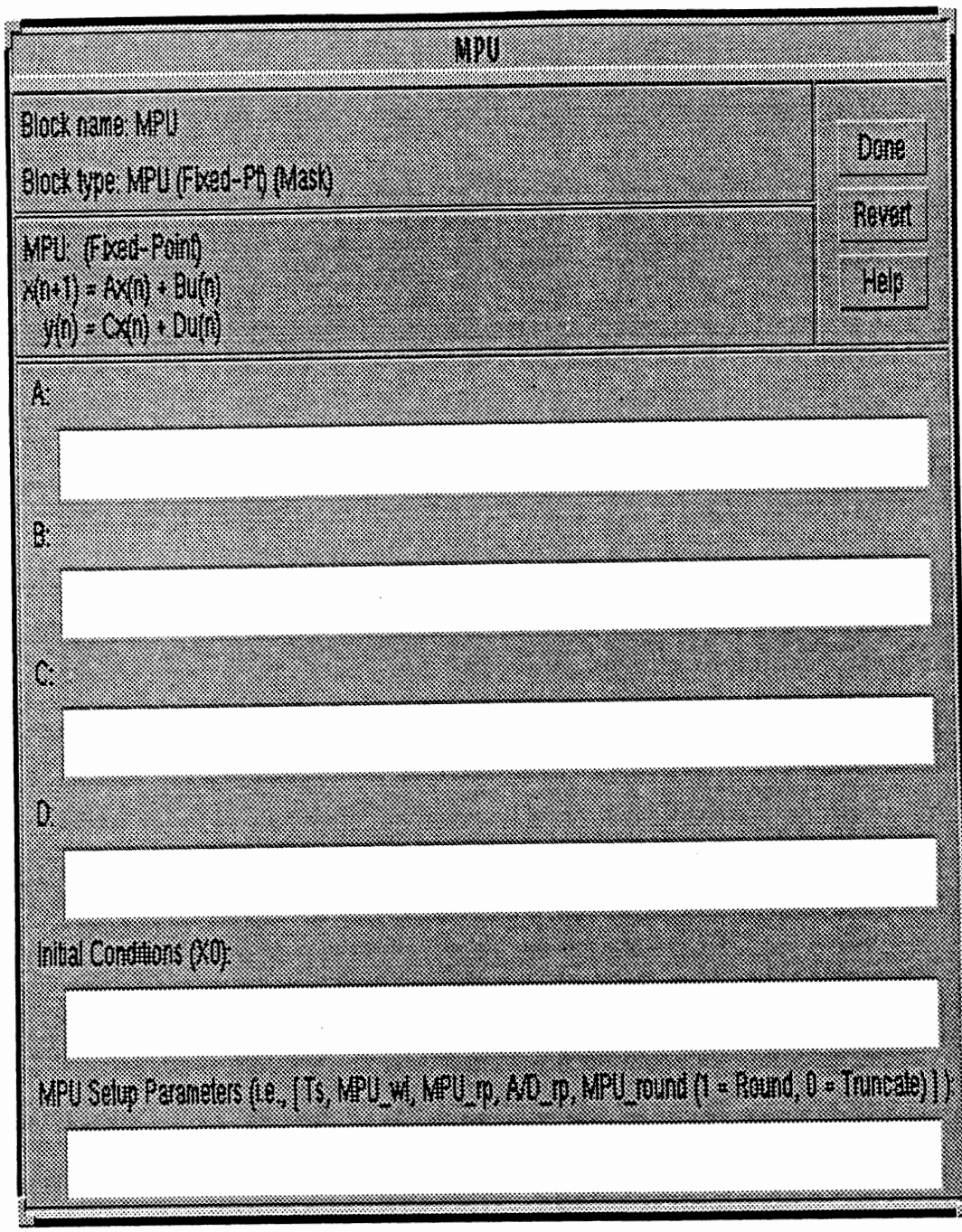

Figure 3.7: The MPU Block (Fixed-Pt.)

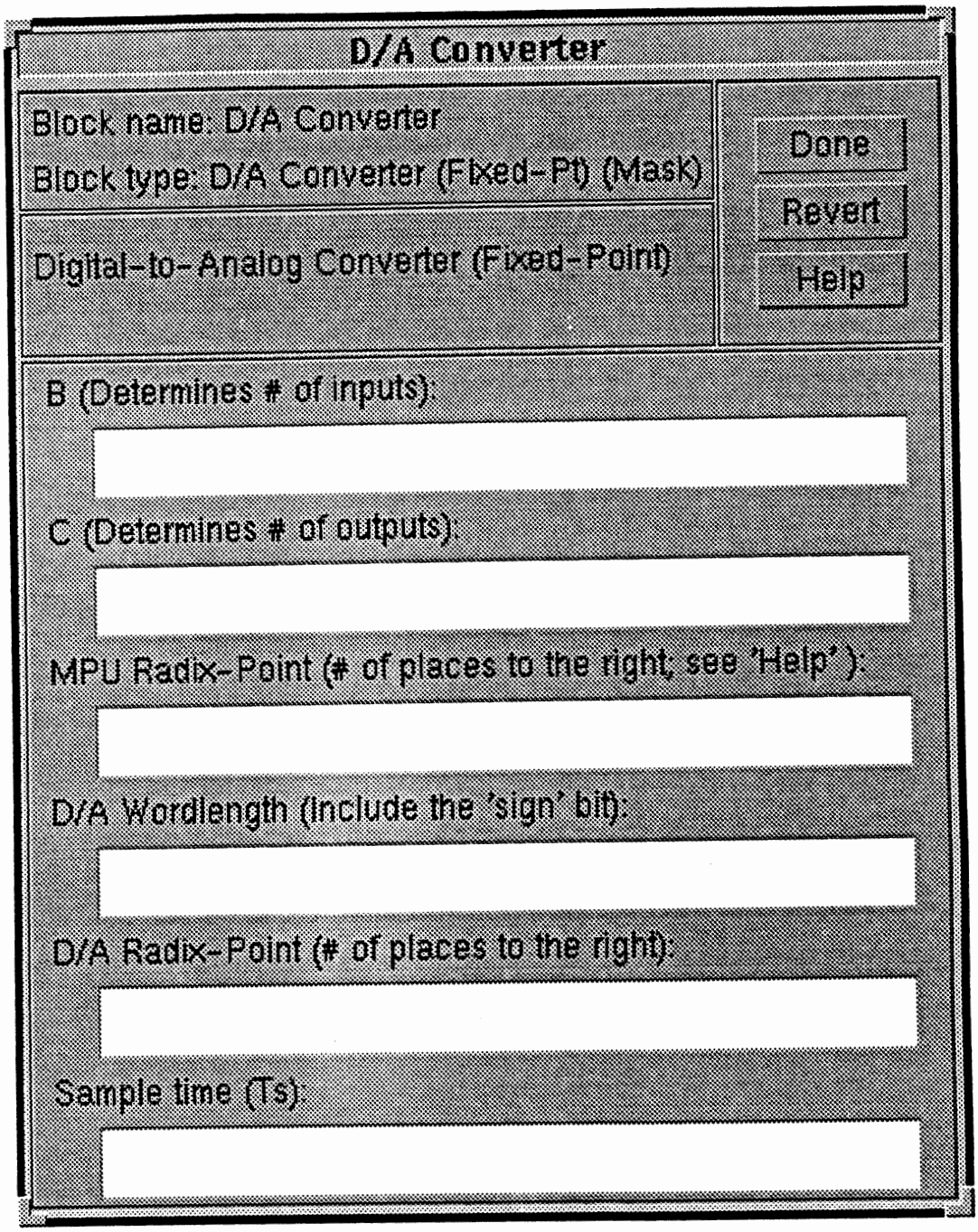

Figure 3.8: The D/A Converter Block (Fixed-Pt.)

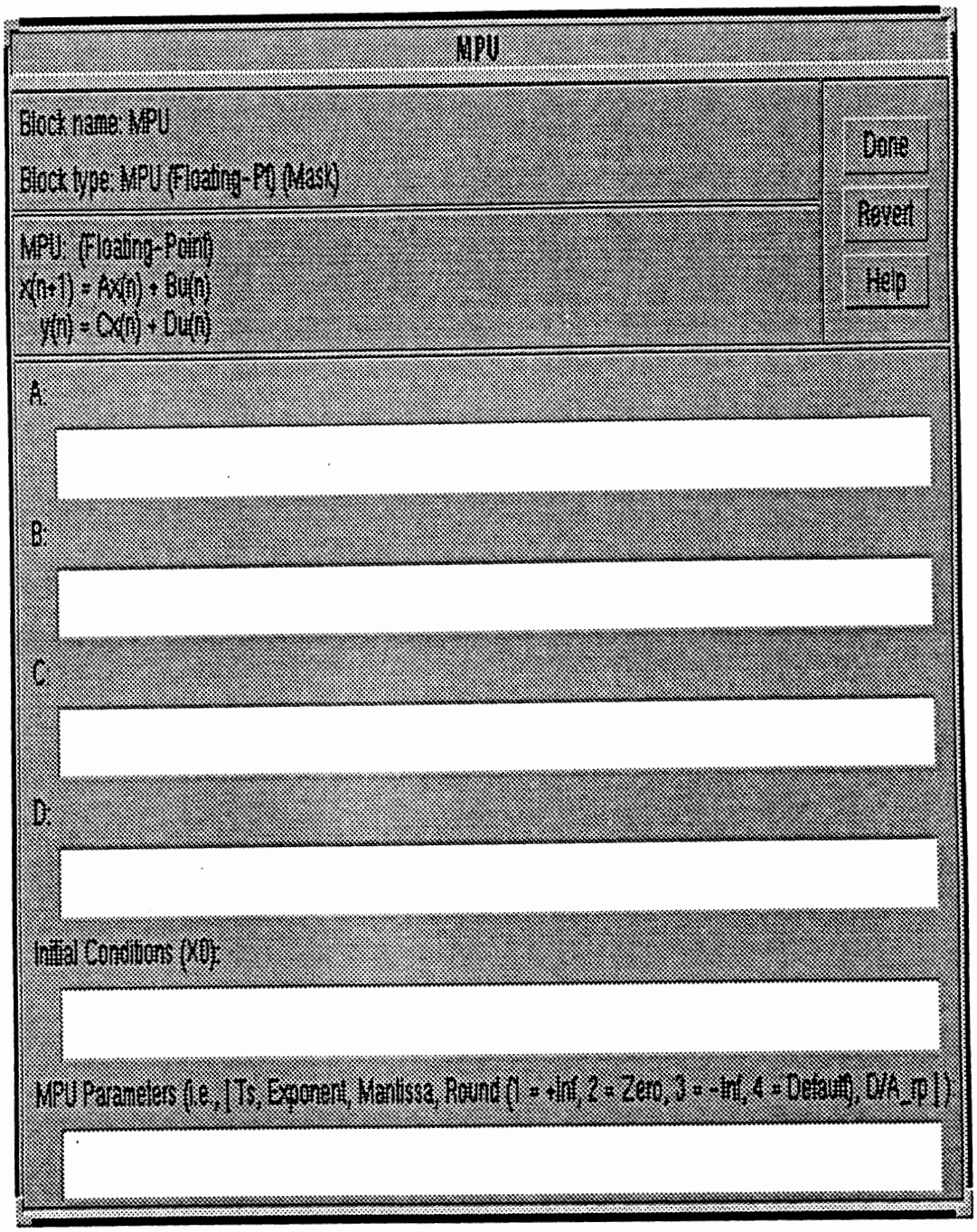

Figure 3.9: The MPU Block (Floating-Pt.)

A SIMULINK™ graphical representation of a digital controller implemented with C MEX-file S-Function blocks is shown in Figure 3.10.

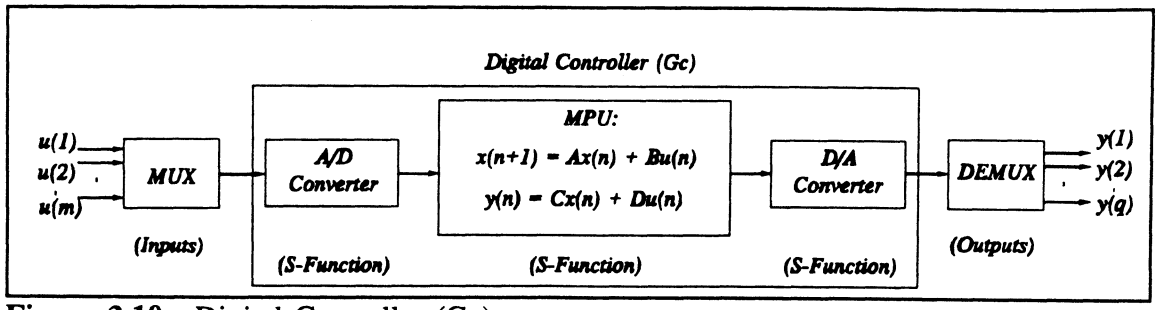

Figure 3.10: Digital Controller (Gc)

Any digital control system (as shown in Figure 3.11) realized with either fixed-point or floating-point arithmetic can, therefore, be emulated accordingly in a given system simulation via the specialized C programs that are integrated in SIMULINK™'s S-Function block.

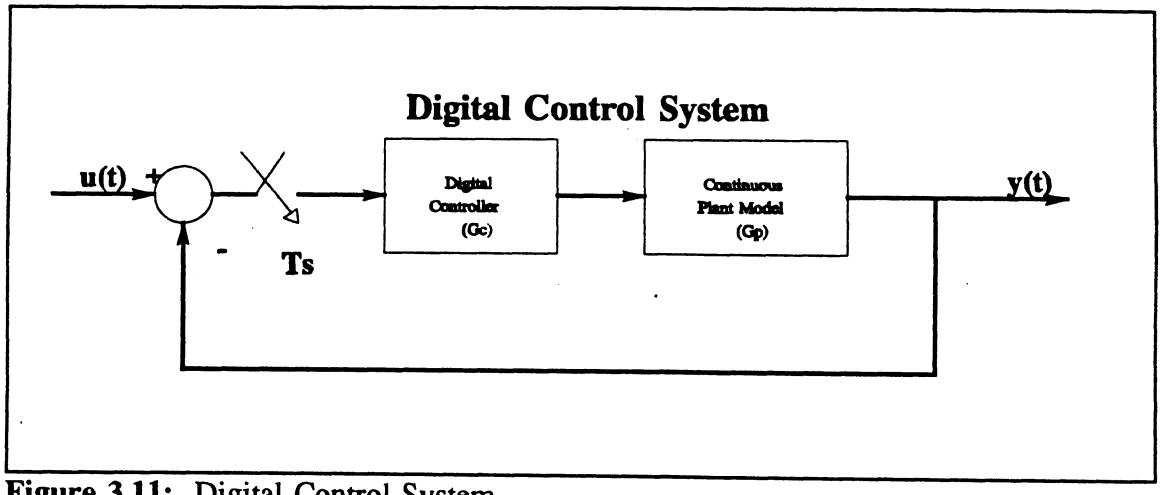

Figure 3.11: Digital Control System

Chapter 4: Applications and Code Verification

Chapter 4 presents some results of using the developed algorithms which are incorporated with the SIMULINK™ software as described in Chapter 3. The SIMULINK™ blocks which allow for the emulation of digital controllers realized with either fixed-point or floating-point arithmetic are tested with respect to a system model. Section 4.1 describes the system model used in performing the various simulations. Section 4.2 presents the results of the linear simulations (e.g., nonlinear effects such as finite wordlength, radix-point position, and arithmetic process are excluded) of the system model. Sections 4.3 and 4.4 present the results of the nonlinear simulations of the system model with respect to implementing fixed-point and floating-point arithmetic, respectively. Finally, Section 4.5 discusses the significance of performing the nonlinear simulations as opposed to the linear simulations.

4.1. System Model: The Hubble Space Telescope (HST)

The developed SIMULINK™ blocks which allow for the emulation of digital controllers realized with either fixed-point or floating-point arithmetic are tested on the Hubble Space Telescope (HST) as shown in Figure 4.1 and as described in [14]. The HST is a multi-input, multi-output (MIMO) system which has three inputs and three outputs. The HST's axes V_1, V_2, and V_3 as shown in Figure 4.1 are the roll, pitch, and yaw axes, respectively.

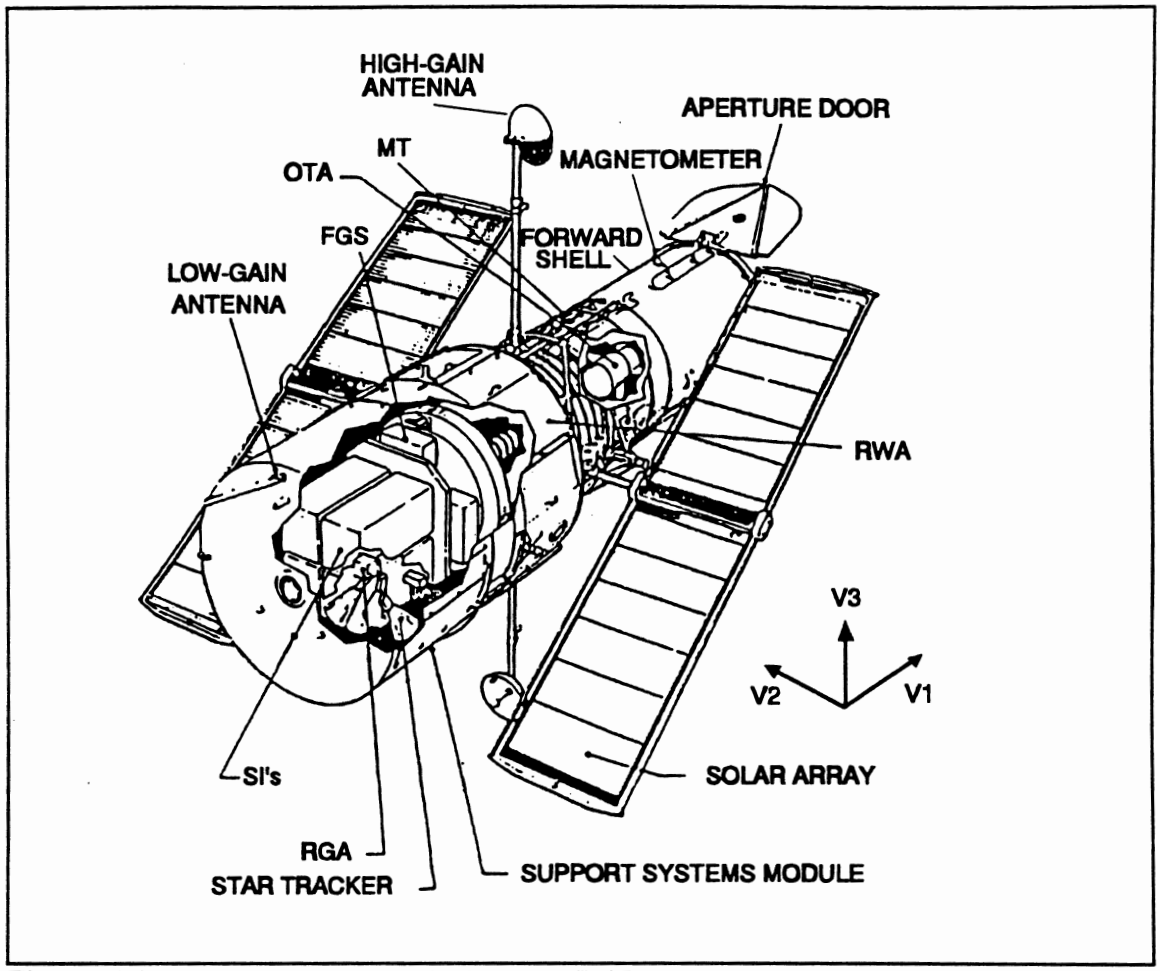

Figure 4.1: The Hubble Space Telescope (HST)

The HST has a DF-224 flight computer that implements fixed-point arithmetic which uses a 24-bit processor with 13 bits allocated to the right of the radix-point and also performs rounding (as opposed to simple truncation) during its computations. In addition, the HST has a sampling frequency of 40 Hz, or equivalently, a sampling period (which is used as a parameter in the simulations) of 0.025 seconds.

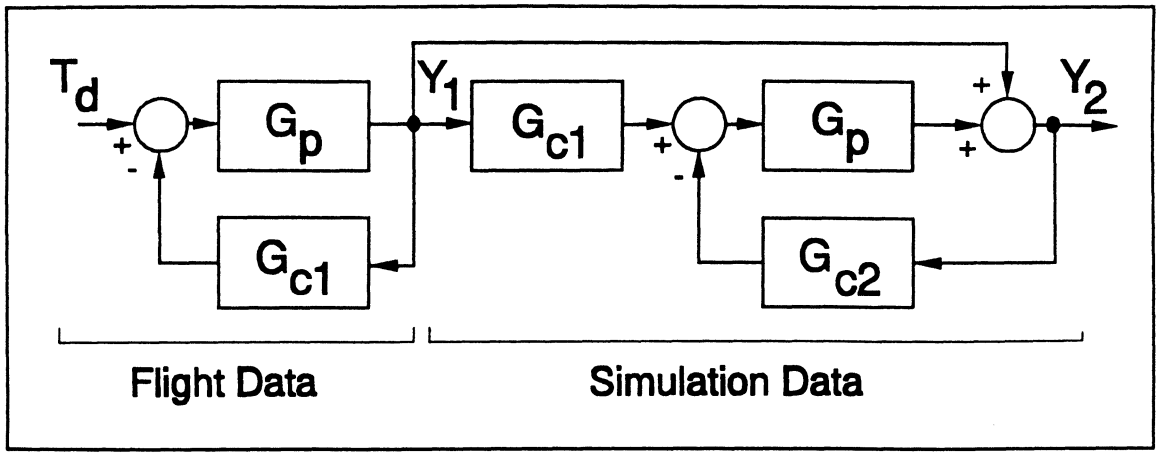

Figure 4.2: HST Simulation Block Diagram

Figure 4.2 represents the block diagram for the HST simulation. The left part of the diagram (labeled Flight Data) can be considered to be the HST, where G_{c1} is the actual in-flight controller (which is a proportional-integral-derivative (PID) controller) and Y_1 is the recorded flight data. T_d is the disturbance torque input which cannot be measured. The right part of the diagram (labeled Simulation Data) is the block diagram which represents the computer simulation under which redesigned controllers are evaluated. (For a more detailed discussion on the HST and its controller redesign efforts, refer to references [14] and [15].)

The simulations performed are based on a method whereby the actual recorded in-flight vehicle rate data, Y_1, is "played back" through the simulation, yielding the vehicle rate data, Y_2, which would result if the controller being simulated had been in the HST rather than the controller in place when the flight data was recorded. This is true under the assumption that the plant model, G_p, in the simulation exactly emulates the real HST

plant. [14] The different controllers that are used in the simulations in Sections 4.2 through 4.4 are as follows:

G_{c1} = PID controller (21^{st} order)

G_{c2} = MADCADS (30^{th} order) and SAGA-II (24^{th} order)

where the PID is the original HST controller, MADCADS is a redesigned controller generated by the Model and Data-Oriented Computer-Aided Design System software (MADCADS) at Ohio University [16], and SAGA-II (Solar Array Gain Augmentation-II) is a controller provided by NASA's Marshall Space Flight Center and Lockheed. [15] The plant model, G_p, is a mathematical model of the real HST plant and is of 126^{th} order.

4.2. Linear Simulations: Using SIMULINK™'s "Built-In" Blocks

The linear simulations that were performed on the HST involved the use of the discrete state-space blocks which are already "built-in" to the SIMULINK™ software. The user interface for this block can be seen in Figure 4.3. The HST linear simulations, which model that of the block diagram as shown in Figure 4.2, were constructed as shown in Figure 4.4. As mentioned in the previous section, the G_{c1} block corresponds to the PID controller, and the G_{c2} block corresponds to both the MADCADS and SAGA-II controllers. Two different simulations were conducted: the first involved the use of the MADCADS controller, and the second used the SAGA-II controller. The results of these two simulations can be seen in Figures 4.5 through 4.10. These linear simulations will

provide a baseline for comparison.

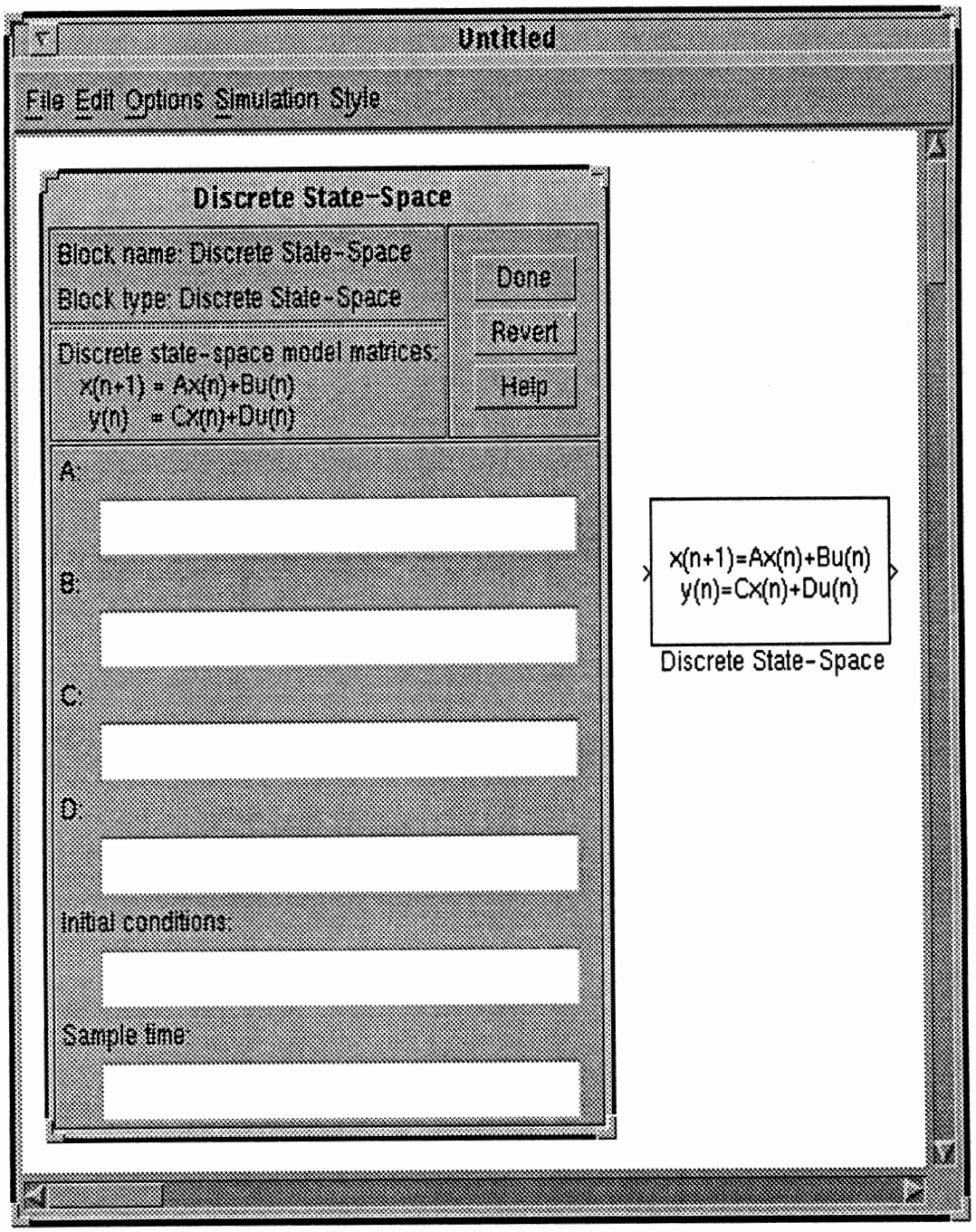

Figure 4.3: The SIMULINK™ "Built-In" Discrete State-Space Block

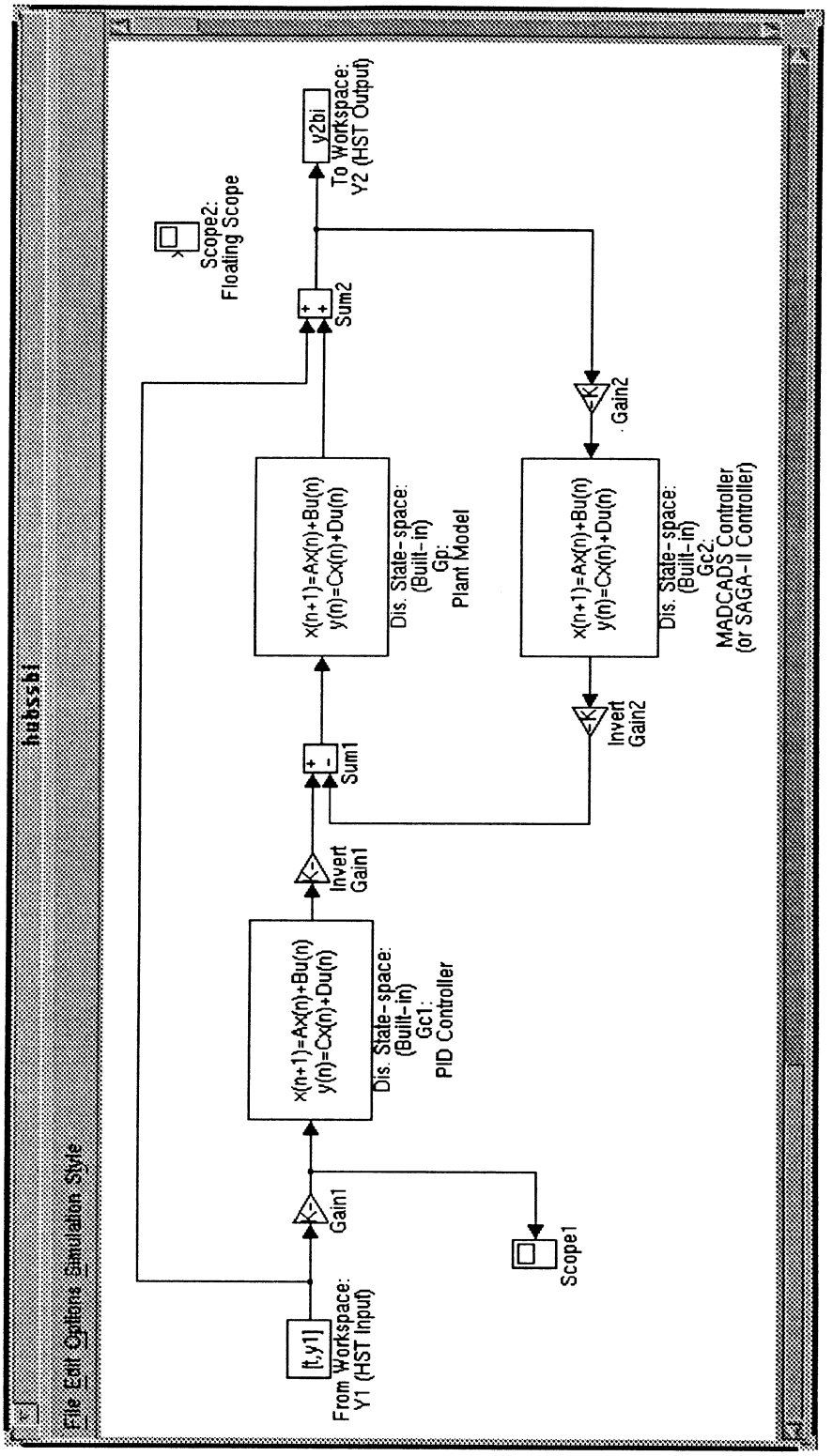

Figure 4.4: Linear Simulation Block Diagram (Using "Built-In" Blocks)

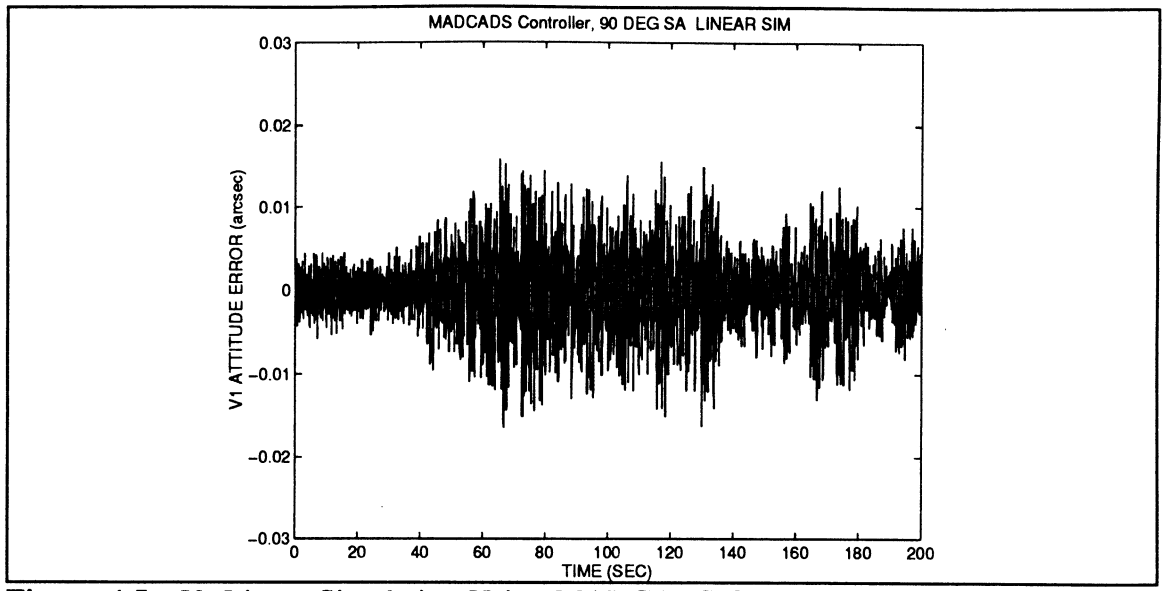

Figure 4.5: V_1 Linear Simulation Using MADCADS Controller

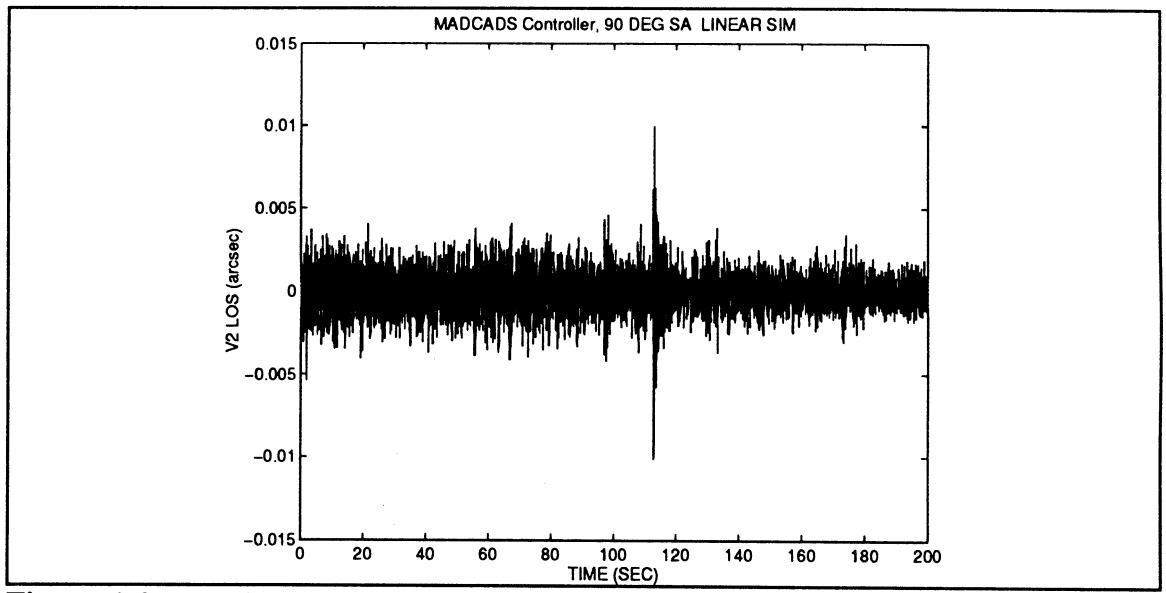

Figure 4.6: V_2 Linear Simulation Using MADCADS Controller

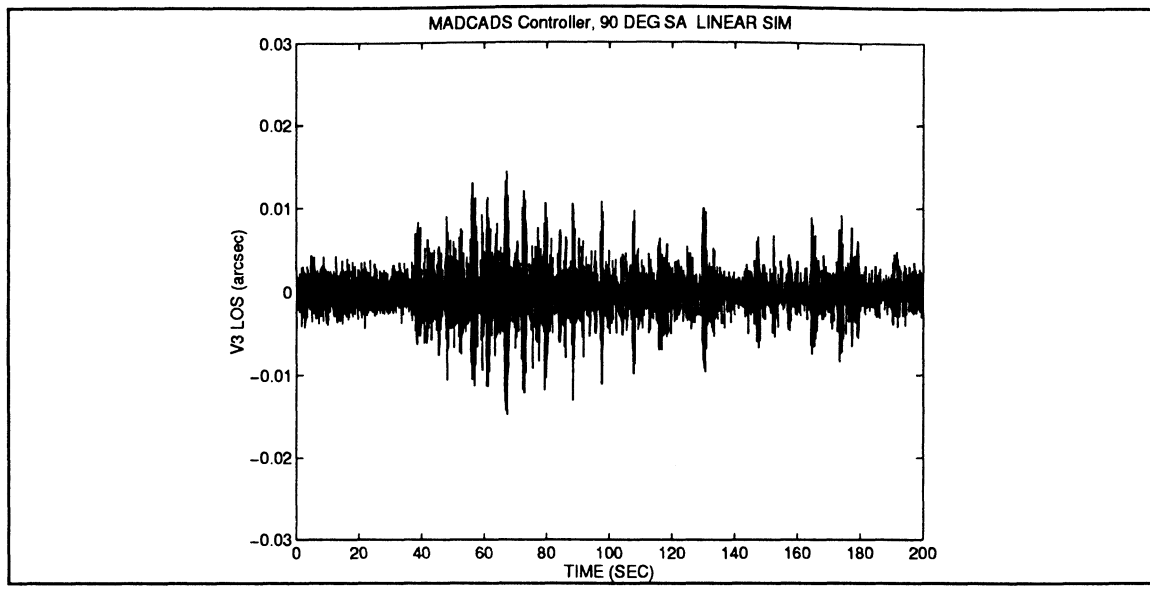

Figure 4.7: V_3 Linear Simulation Using MADCADS Controller

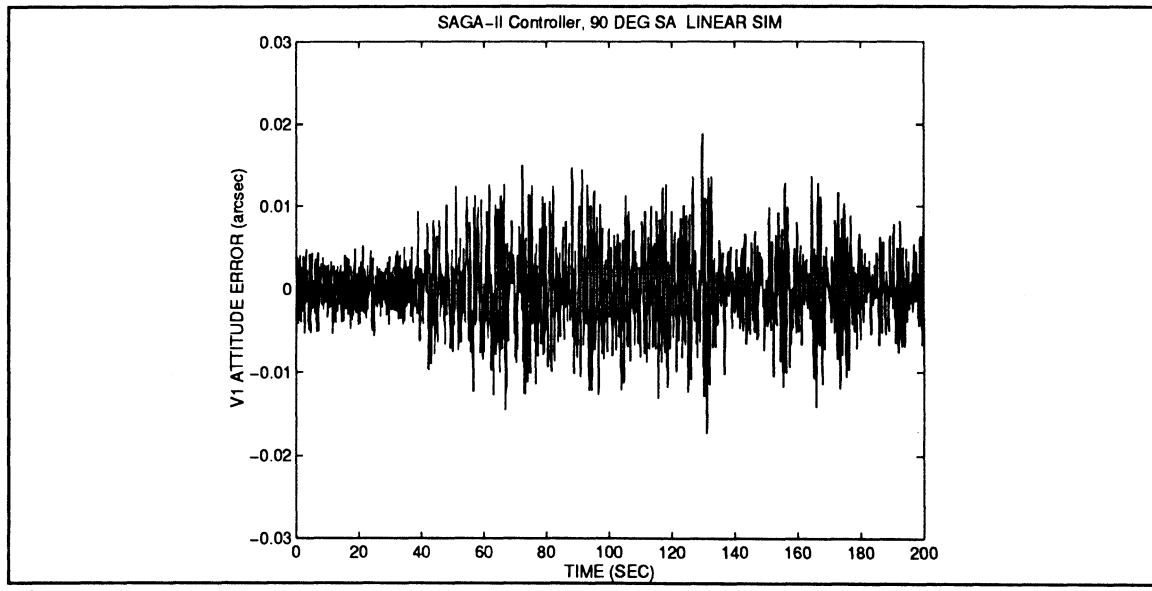

Figure 4.8: V_1 Linear Simulation Using SAGA-II Controller

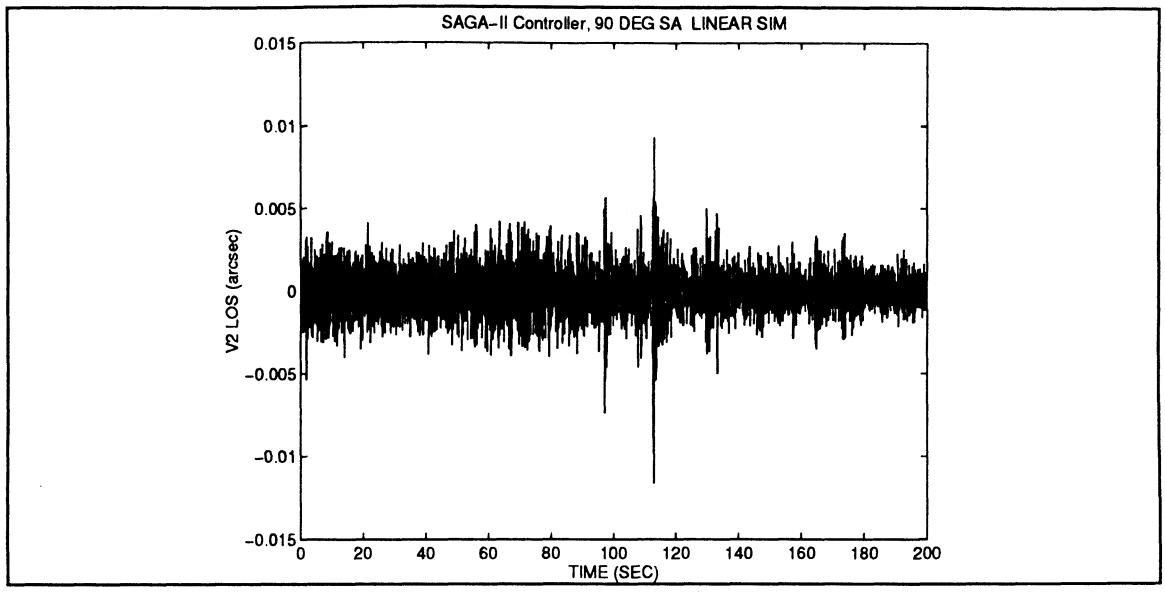

Figure 4.9: V_2 Linear Simulation Using SAGA-II Controller

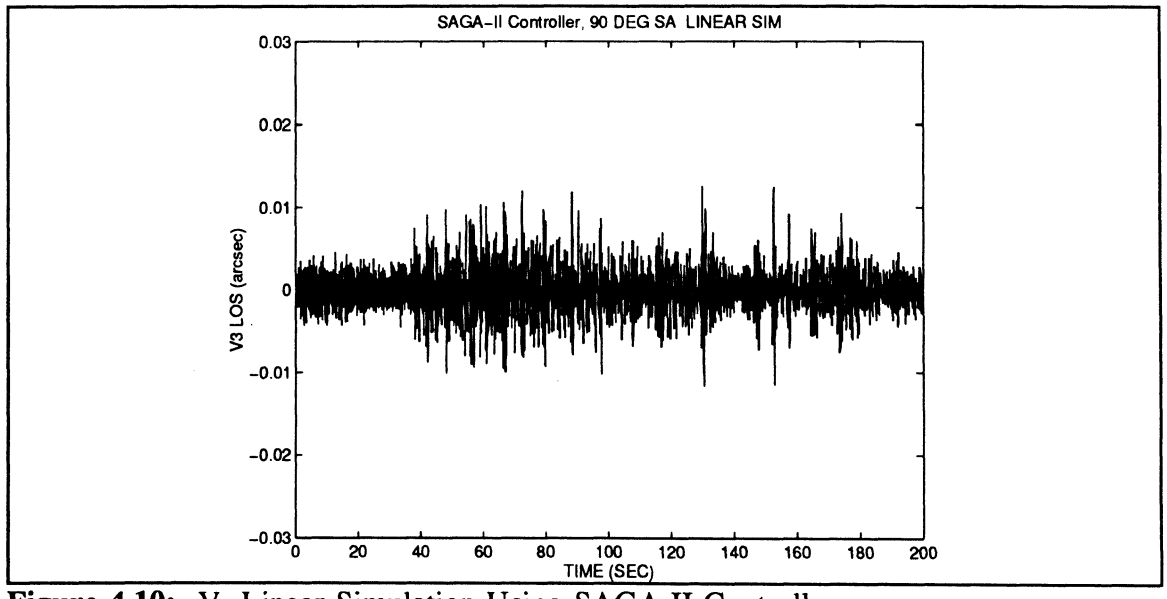

Figure 4.10: V_3 Linear Simulation Using SAGA-II Controller

4.3. Nonlinear Simulations: Fixed-Point Arithmetic Emulation

The first set of nonlinear simulations performed on the HST involved the use of the developed fixed-point arithmetic emulation SIMULINK™ blocks. The user interface for these blocks (e.g., A/D, MPU, and D/A) are as shown in Figures 3.6 through 3.8. The HST nonlinear simulations corresponding to modeling the fixed-point arithmetic processes were constructed as shown in Figure 4.11. The gain blocks that are in place before each controller are used for the purpose of scaling the input to be within the dynamic range of the controllers.

Similar to the linear simulations as discussed in Section 4.2, two sets of simulations were performed for the HST with respect to emulating its nonlinear effects such as finite wordlength, radix-point, and arithmetic process for the two different controllers, MADCADS and SAGA-II. For both simulations, the A/D, MPU, and D/A all implemented the values 24, 13, and 1 for the wordlength, radix-point position, and rounding method, respectively, as these parameters characterize the HST as outlined in Section 4.1. The results of these nonlinear simulations can be seen in Figures 4.12 through 4.17. Furthermore, as a comparison, three plots are given as shown in Figures 4.18 through 4.20, which represent the results obtained by the FORTRAN simulations conducted in the HST's controller redesign effort as described in [14]. These FORTRAN simulations, which in this particular case implemented the MADCADS controller, modeled the nonlinear effects such as finite wordlength, radix-point position, and fixed-

point arithmetic as well. Therefore, the plots as shown in Figures 4.18 through 4.20 serve as a verification to the results obtained with the developed SIMULINK™ blocks.

It is apparent by comparing the respective plots as shown in Section 4.2, which contains the linear simulation plots, with those as shown in this section, there is significant degradation of performance due to the modeling of the HST's nonlinear effects as described above.

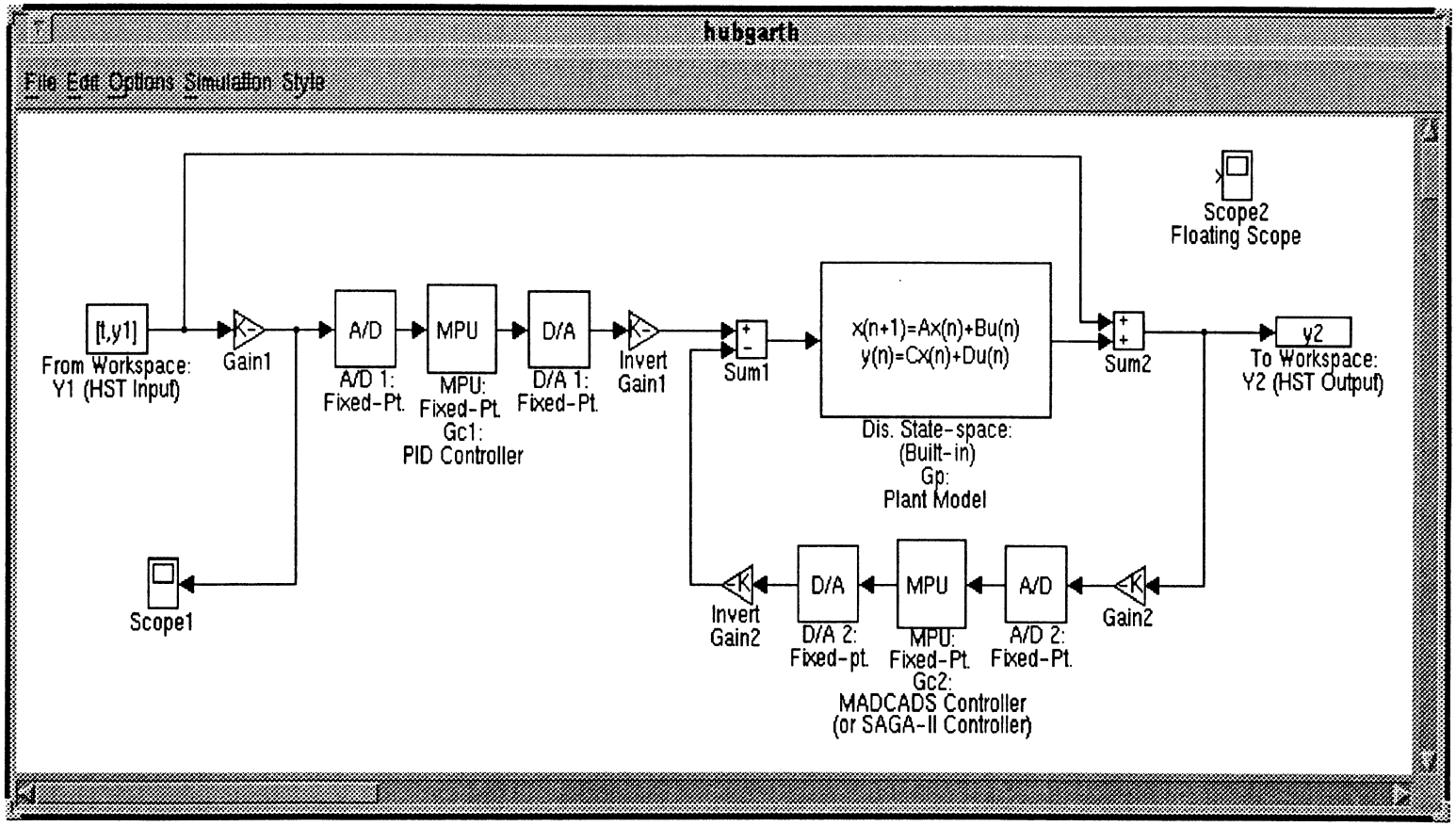

Fig 4.11: Nonlinear Simulation Block Diagram (Using Fixed-Point Arithmetic Emulation Blocks)

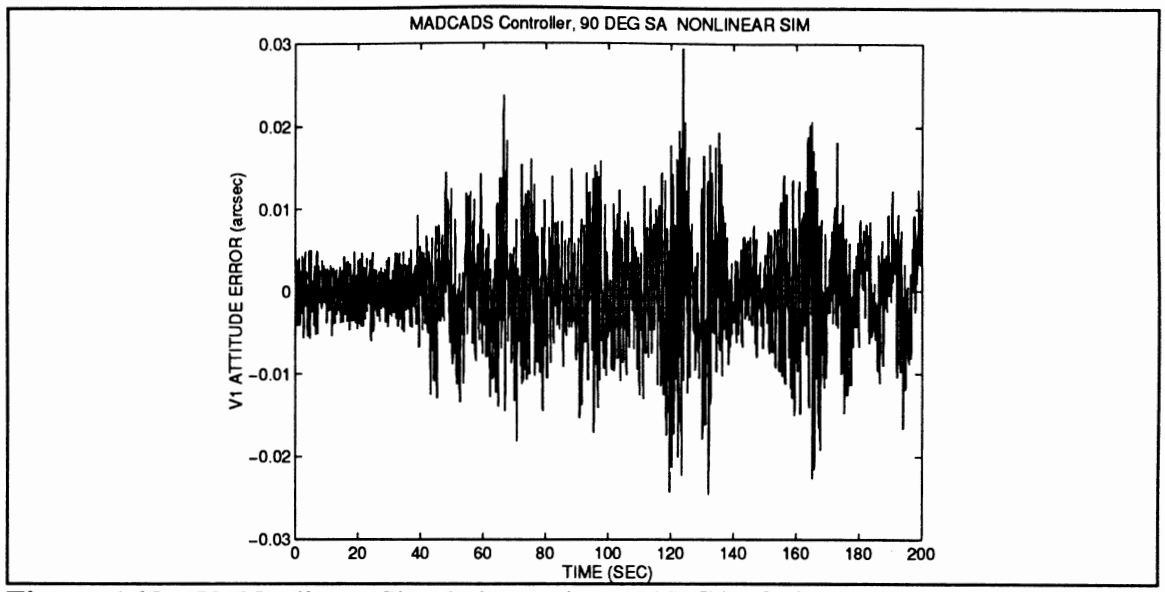

Figure 4.12: V_1 Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

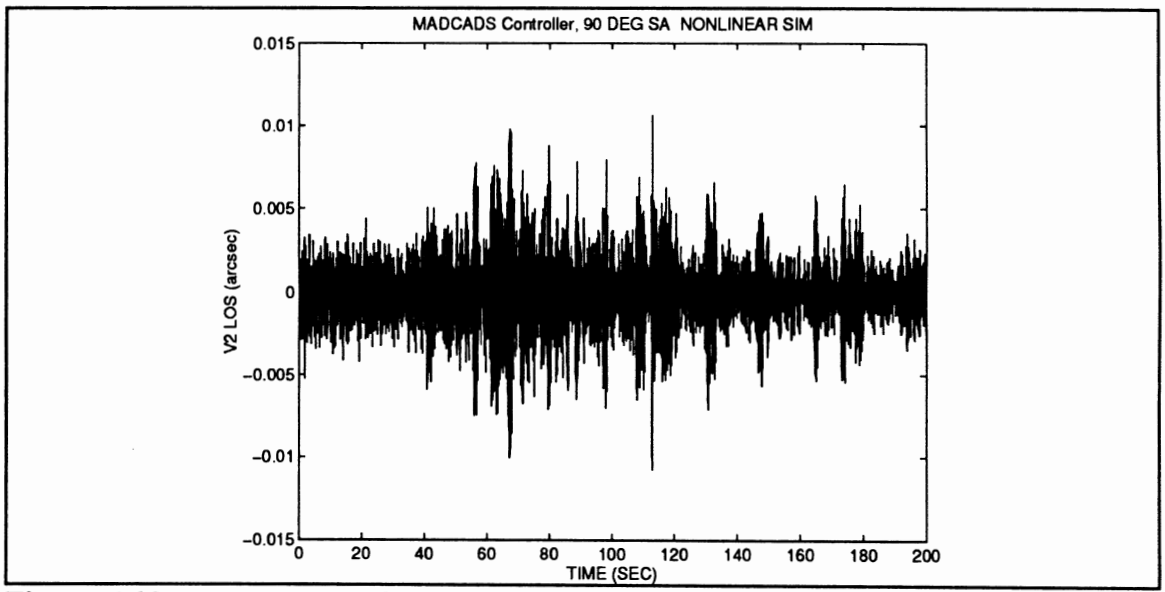

Figure 4.13: V_2 Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

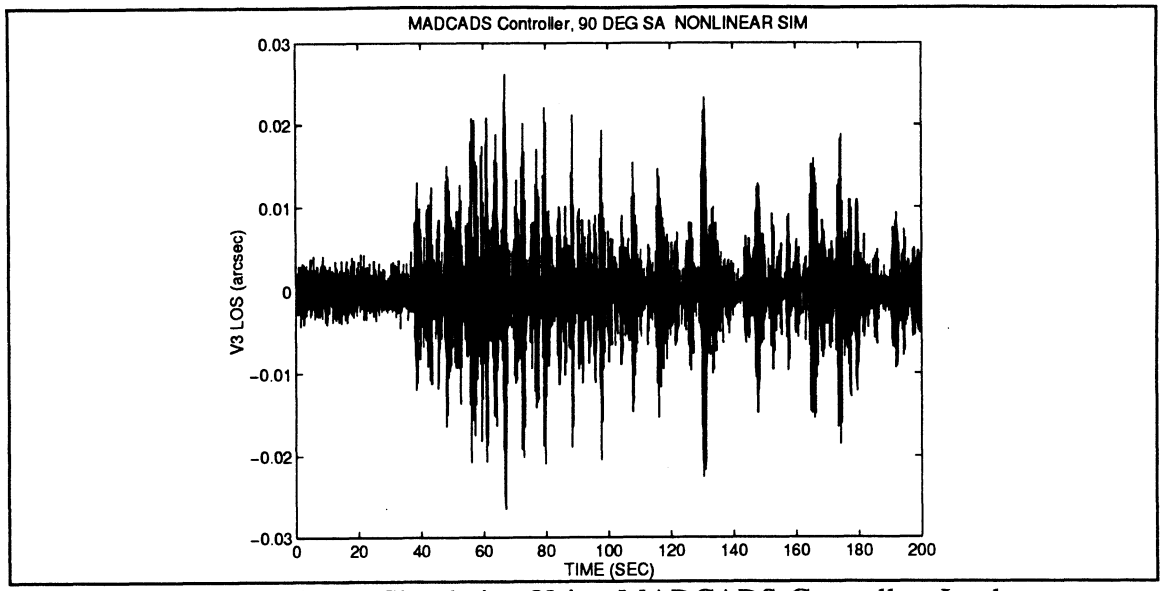

Figure 4.14: V_3 Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

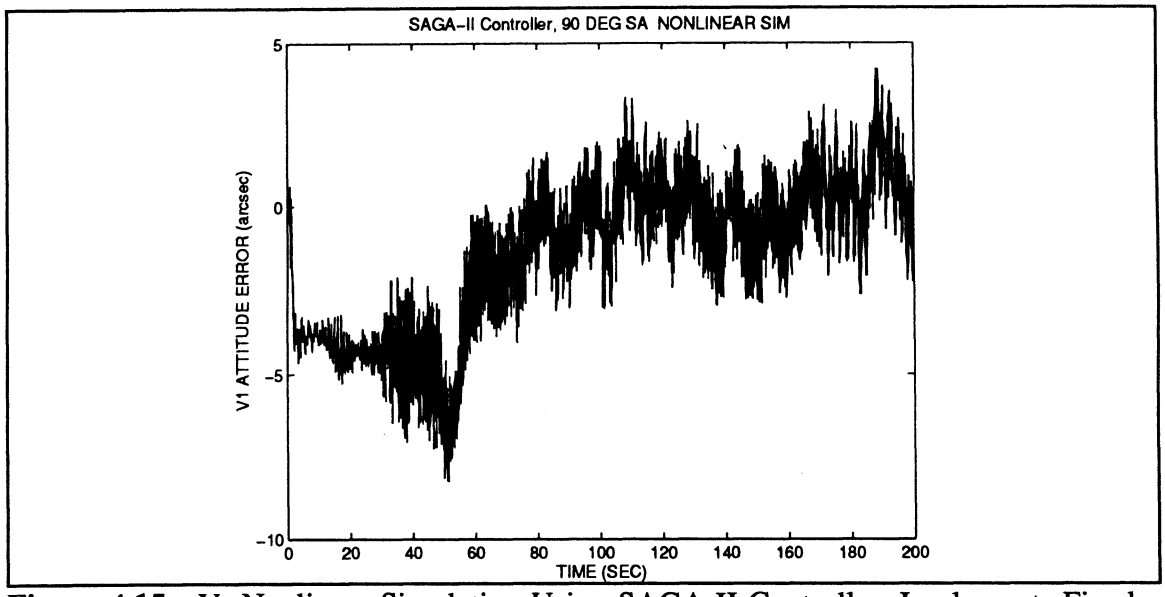

Figure 4.15: V_1 Nonlinear Simulation Using SAGA-II Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

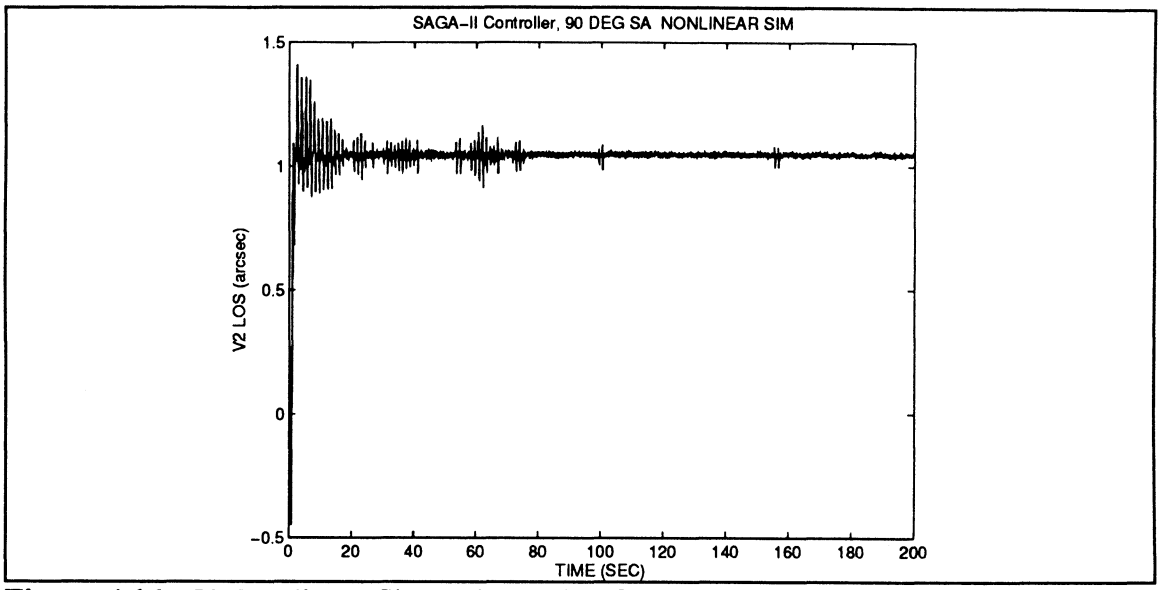

Figure 4.16: V_2 Nonlinear Simulation Using SAGA-II Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

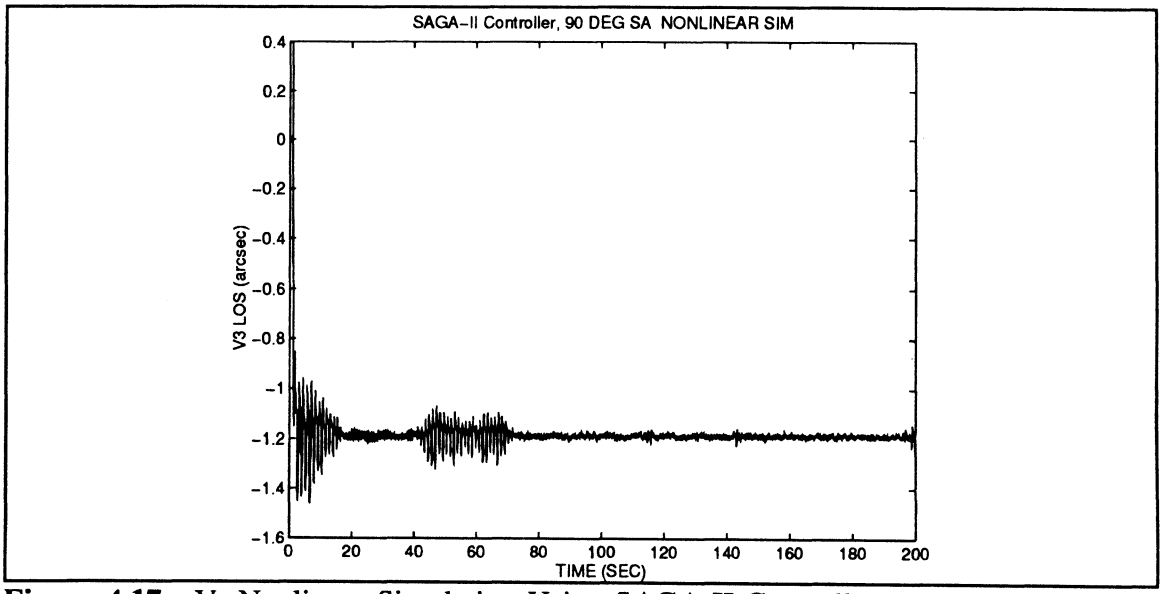

Figure 4.17: V_3 Nonlinear Simulation Using SAGA-II Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

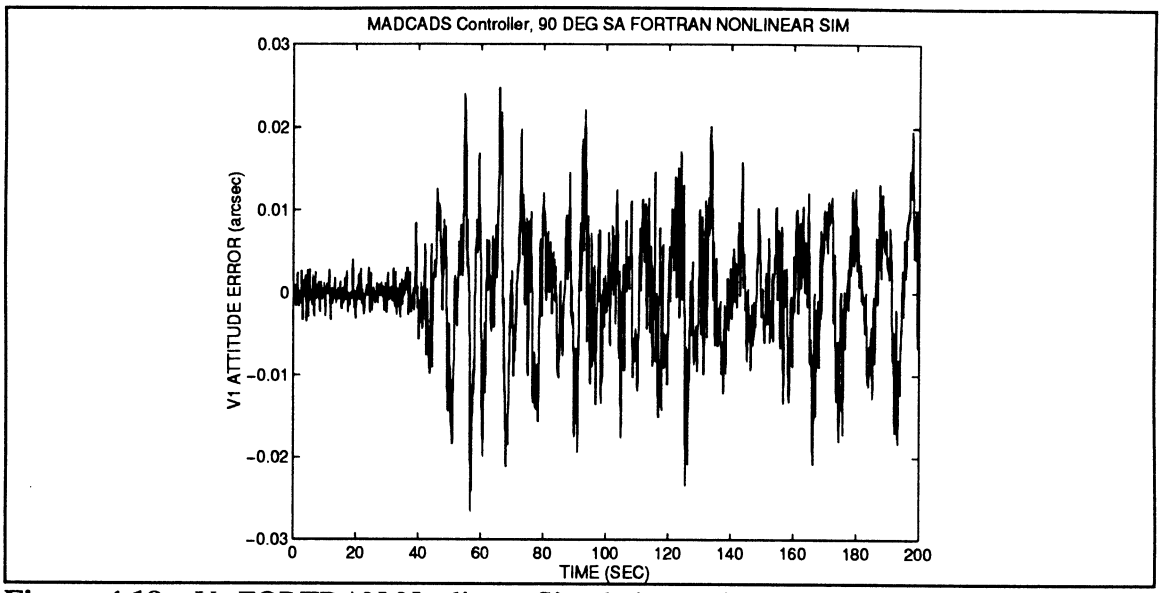

Figure 4.18: V_1 FORTRAN Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

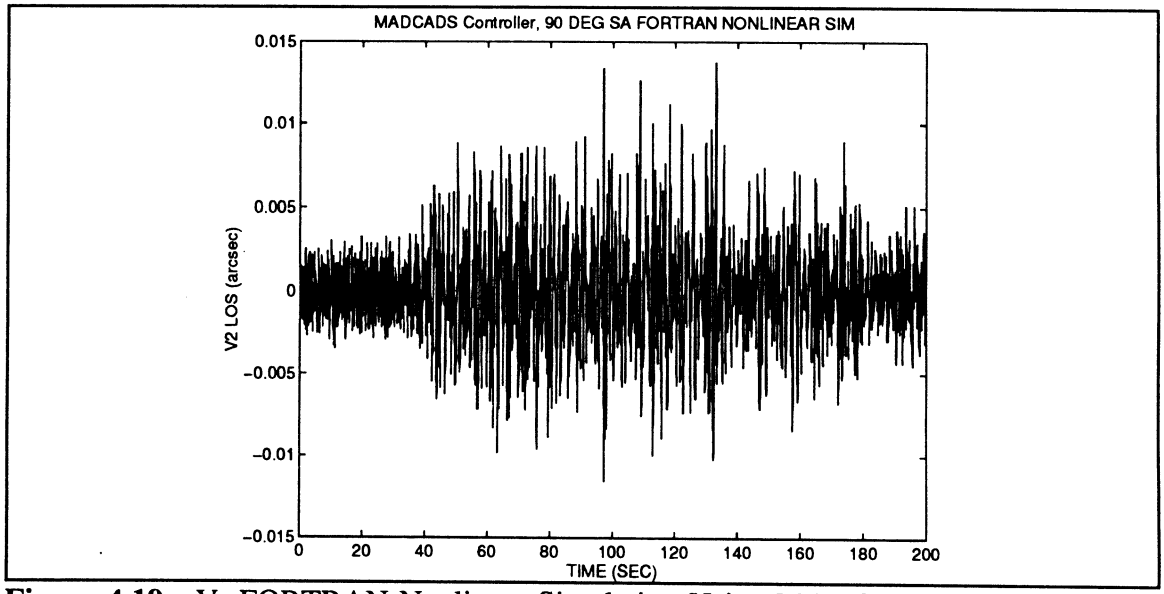

Figure 4.19: V_2 FORTRAN Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

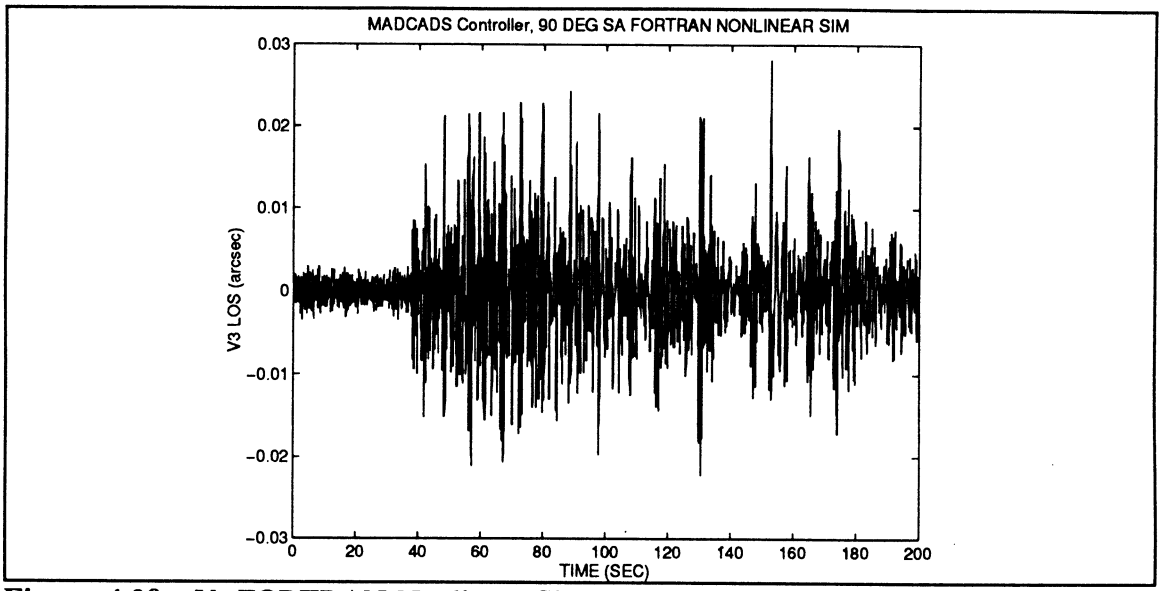

Figure 4.20: V_3 FORTRAN Nonlinear Simulation Using MADCADS Controller: Implements Fixed-Point Arithmetic Emulation with a 24 bit wordlength and 13 bits allocated to the right of the radix-point.

4.4. Nonlinear Simulations: Floating-Point Arithmetic Emulation

The second set of nonlinear simulations performed on the HST involved the use of the developed floating-point arithmetic emulation SIMULINK™ block. The user interface for this block (e.g., MPU) is as shown in Figure 3.9. The interfaces for both the A/D and D/A are the same as before as shown in Figures 3.6 and 3.8, respectively. The purpose of these simulations is to demonstrate the benefits of using floating-point calculations with respect to the HST as opposed to its inherent fixed-point arithmetic as discussed in Section 4.3. The SIMULINK™ block diagram simulation that was constructed for this purpose is shown in Figure 4.21.

Using the MADCADS controller, two different simulations were conducted. The first simulation used a floating-point configuration which assumed that the 24-bit processor implemented the values 15 and 8 for its mantissa and exponent fields, respectively. The second simulation, however, implemented the values 10 and 5 for its mantissa and exponent fields, respectively. For both simulations, the A/D and D/A converters both implemented the values 24 and 13 for the wordlength and radix-point position, respectively, as was the case in the fixed-point arithmetic simulations as described in Section 4.3. The results of these two sets of simulations can be seen in Figures 4.22 through 4.27.

A third simulation was conducted, with respect to implementing floating-point

arithmetic, using the SAGA-II controller. This setup assumed that the HST performed its calculations in double-precision mode (52 bits used for the mantissa and 11 bits used for the exponent). As can be seen in Figures 4.28 through 4.30, the results obtained in this simulation closely resemble those obtained in the linear (or un-quantized) simulation (using SAGA-II as G_{c2}) as shown in Figures 4.8 through 4.10. Recall, the nonlinear simulation involving the SAGA-II controller which implemented fixed-point arithmetic generated an unstable response as shown in Figures 4.15 through 4.17. In essence, this floating-point arithmetic (e.g., 52/11) simulation produces similar results for the case where a SUN SPARC-station (or a computer with a similar configuration) is to be emulated.

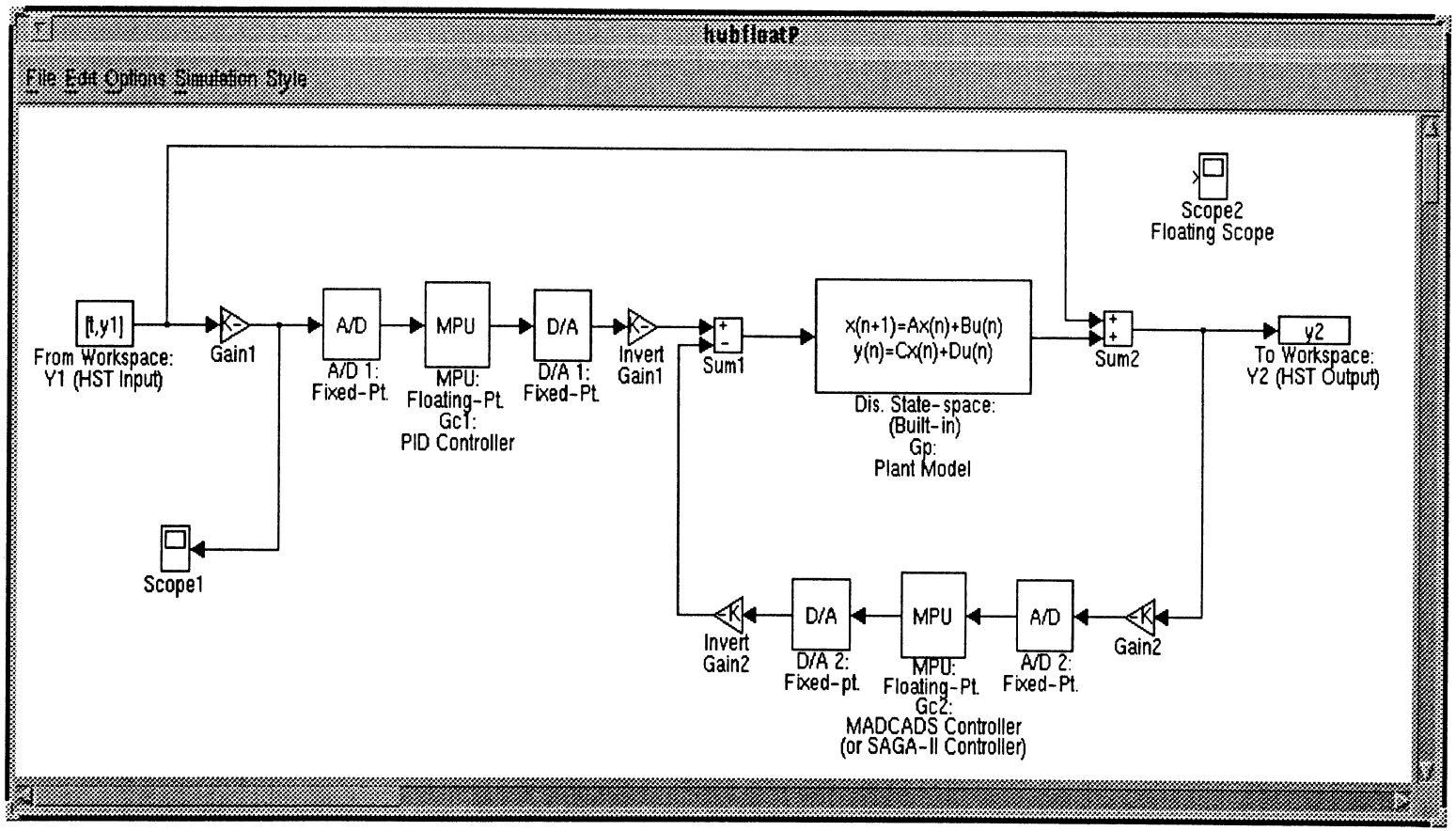

Figure 4.21: Nonlinear Simulation Block Diagram (Using the Floating-Point Arithmetic Emulation Block)

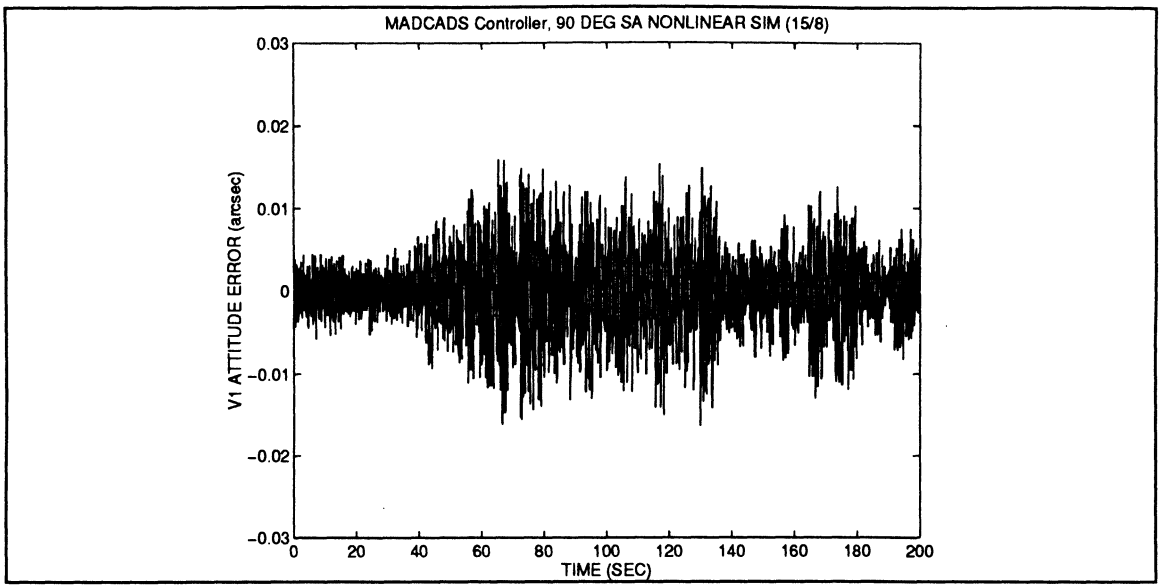

Figure 4.22: V_1 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 15 bit mantissa and an 8 bit exponent.

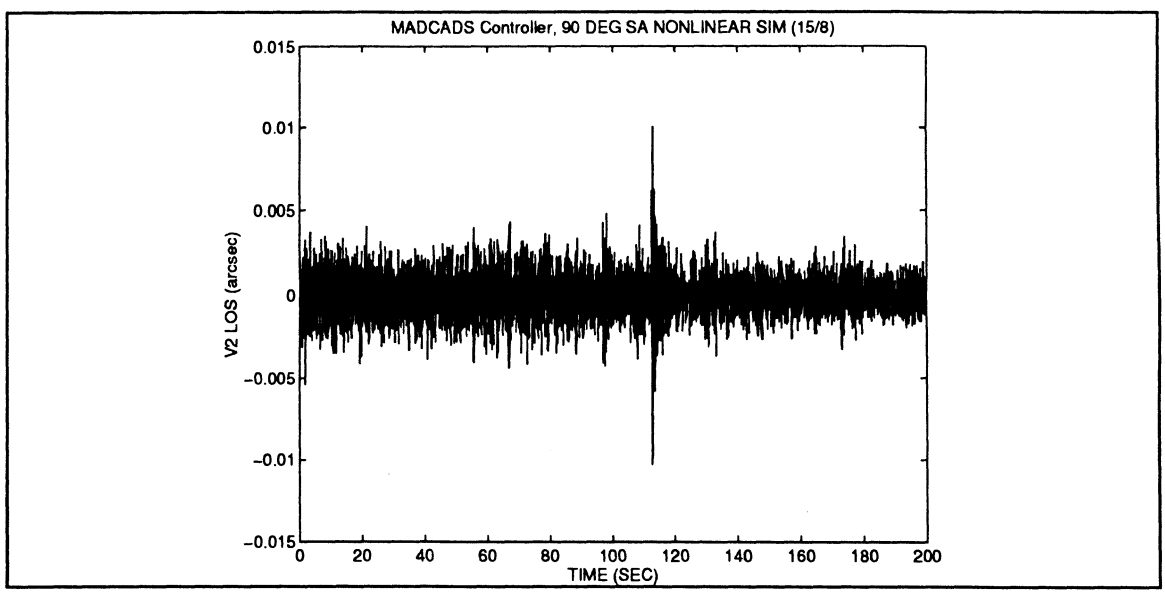

Figure 4.23: V_2 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 15 bit mantissa and an 8 bit exponent.

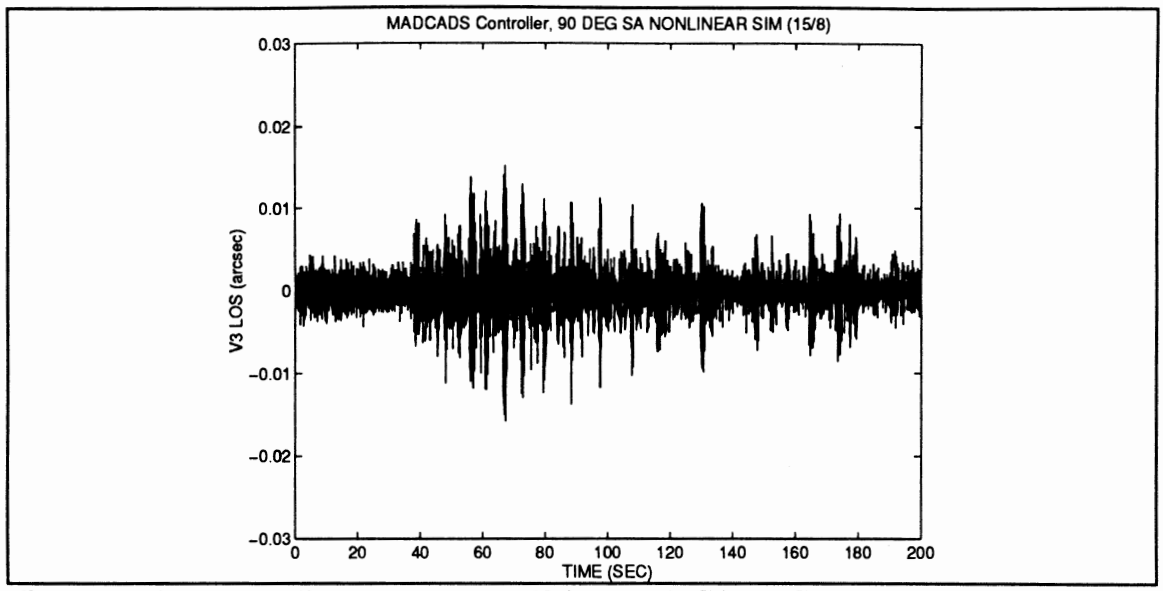

Figure 4.24: V_3 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 15 bit mantissa and an 8 bit exponent.

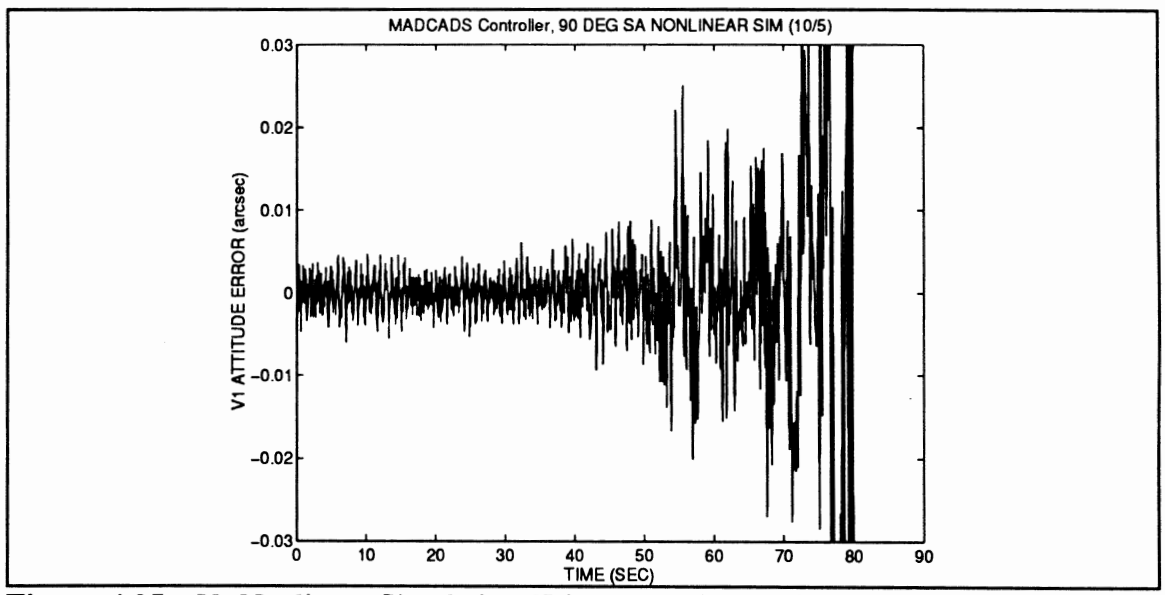

Figure 4.25: V_1 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 10 bit mantissa and a 5 bit exponent.

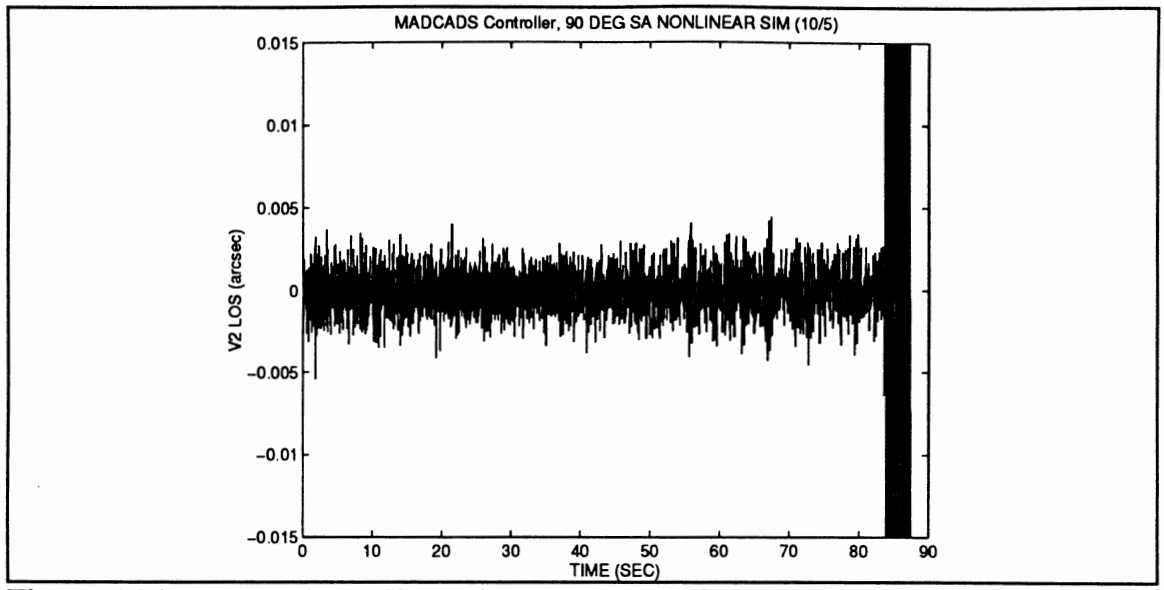

Figure 4.26: V_2 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 10 bit mantissa and a 5 bit exponent.

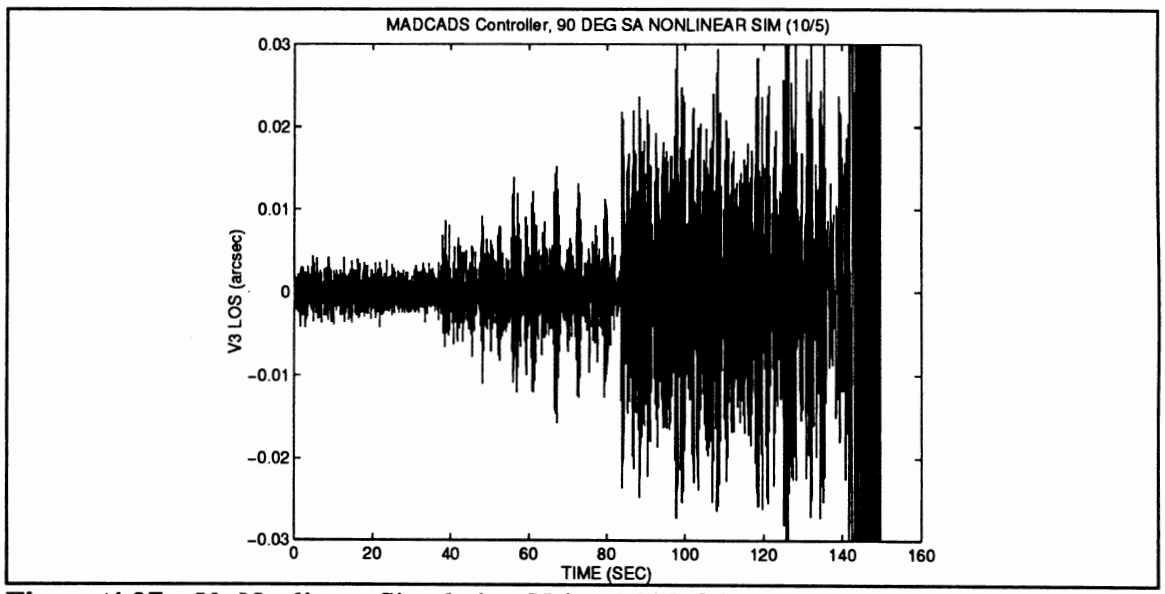

Figure 4.27: V_3 Nonlinear Simulation Using MADCADS Controller: Implements Floating-Point Arithmetic Emulation with a 10 bit mantissa and a 5 bit exponent.

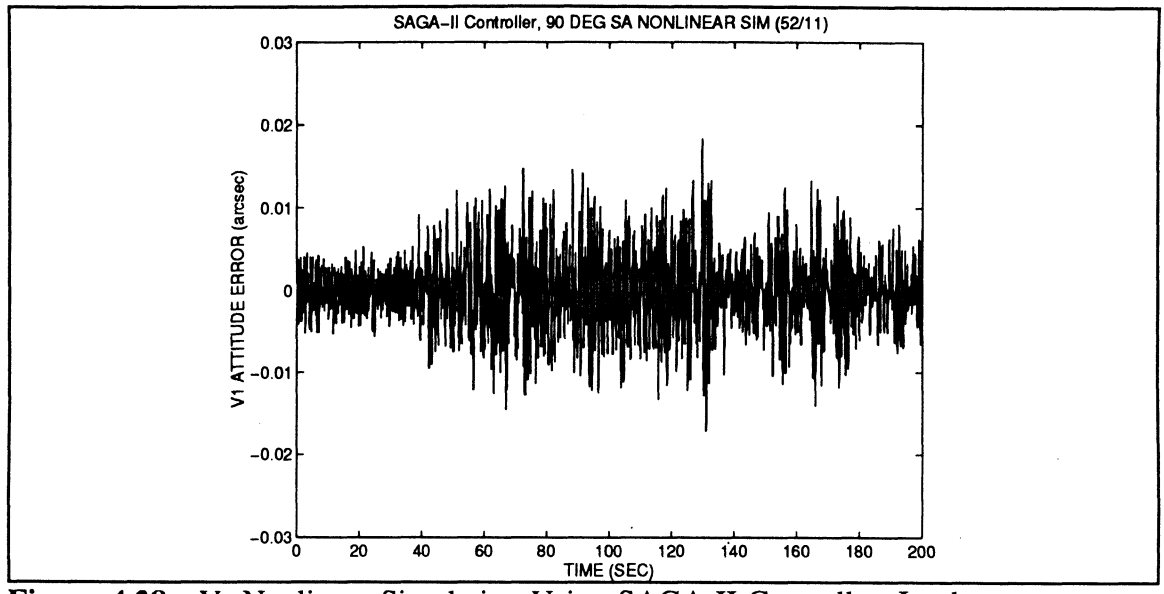

Figure 4.28: V_1 Nonlinear Simulation Using SAGA-II Controller: Implements Floating-Point Arithmetic Emulation with a 52 bit mantissa and an 11 bit exponent.

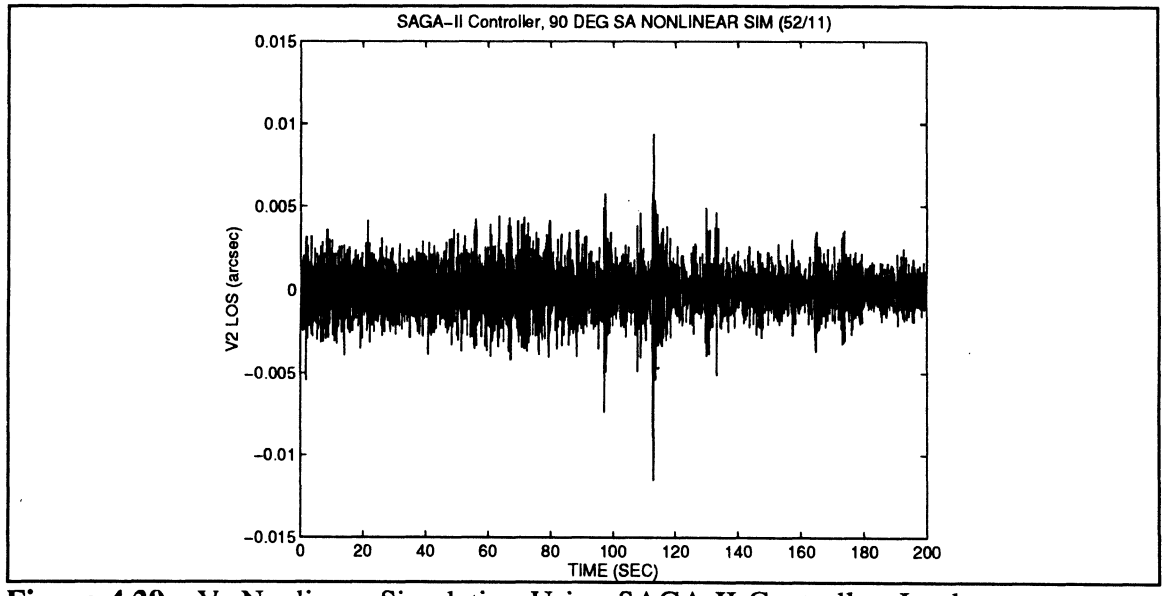

Figure 4.29: V_2 Nonlinear Simulation Using SAGA-II Controller: Implements Floating-Point Arithmetic Emulation with a 52 bit mantissa and an 11 bit exponent.

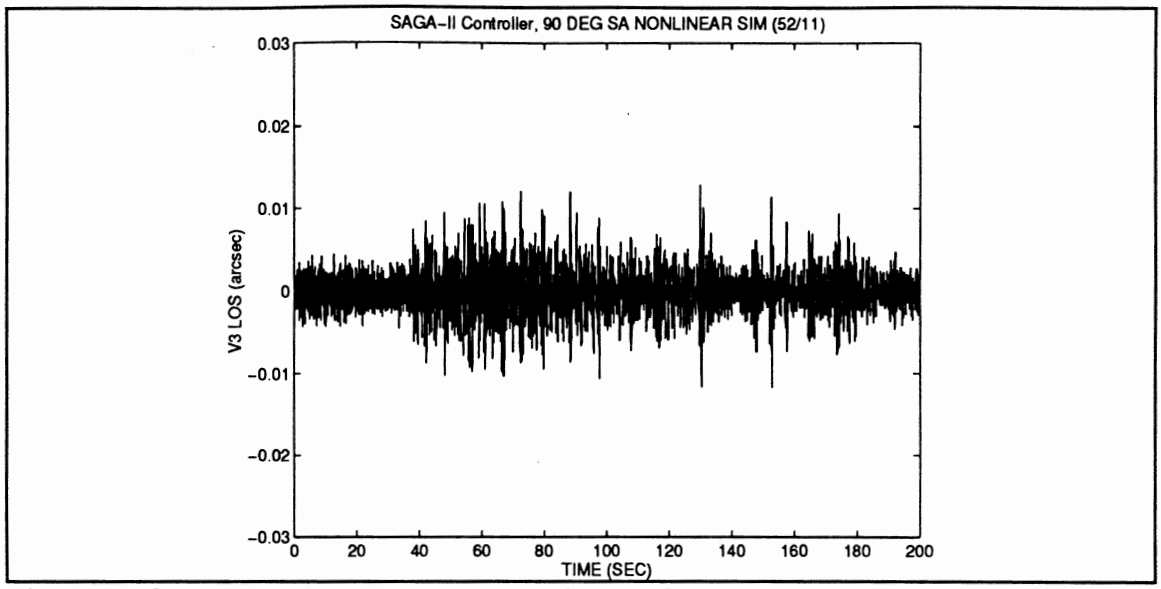

Figure 4.30: V_3 Nonlinear Simulation Using SAGA-II Controller: Implements Floating-Point Arithmetic Emulation with a 52 bit mantissa and an 11 bit exponent.

4.5. Significance of HST Simulations: Linear vs. Nonlinear

As a result of emulating the HST's nonlinear effects with respect to finite wordlength, radix-point, and arithmetic process, it is apparent that there are significant variations in the output data, Y_2, as opposed to the output that would be obtained if a linear simulation was conducted. By emulating these nonlinear effects inherent in the HST, a given simulation will produce more accurate results and, more importantly, will allow the controller redesign effort to be carried out in a much more expedient manner via the developed SIMULINK™ blocks.

Furthermore, it was shown that the developed fixed-point arithmetic emulation algorithms were verified by comparing the results obtained with the SIMULINK™ blocks to those of the FORTRAN simulations conducted in the HST controller redesign effort as described in [14]. In addition, it must be mentioned that the FORTRAN simulation which implemented both the SAGA-II controller and the same nonlinear effects as previously mentioned, generated an unstable response as well. Therefore, this further verifies the arithmetic emulation codes that are integrated with SIMULINK™ via the S-Function Block.

The floating-point arithmetic emulation algorithm was verified by comparing its results, which were obtained from a simulation conducted in the double-precision (e.g., 52/11) mode, to that of the results obtained in the linear simulations which were

conducted on a SUN SPARC-station. Also, various simulations were conducted with respect to the HST that implemented different floating-point arithmetic formats (e.g., 15/8 and 10/5), and therefore, further tested and verified the implementation of the floating-point arithmetic emulation algorithm.

Finally, it was also shown that by implementing floating-point arithmetic as opposed to fixed-point arithmetic in a digital controller, the performance of a digital control system will increase with respect to quantization errors being significantly reduced. However, it should be understood that for any given digital control system, the implementation of a digital controller which utilizes floating-point arithmetic will naturally increase the execution time, and execution time is of major importance in implementing real time controllers. Therefore, when using the SIMULINK™ blocks in a given digital control system simulation, the analyst must have knowledge of the required performance criteria of the system in order to fully utilize the benefits of these simulation blocks.

Chapter 5: Conclusions and Recommendations

The developed fixed-point and floating-point arithmetic emulation algorithms that are integrated with the SIMULINK™ software provide a way to emulate more accurately a given digital controller with respect to its hardware architecture. The hardware architecture, of course, refers to a digital controller's analog-to-digital (A/D) and digital-to-analog (D/A) converters, as well as its microprocessing unit (MPU). With each of these devices, there are inherent characteristics such as finite wordlength, arithmetic process implementation (e.g., fixed-point or floating-point arithmetic), and rounding method, which all must be taken into consideration in order to emulate a digital controller more accurately. Furthermore, these SIMULINK™ blocks, as utilized in a given digital control system simulation, provide a means to test various designed controllers quickly and efficiently via the graphical user interface (GUI) inherent to this software.

First, the theory behind both fixed-point and floating-point arithmetic, as well as a discussion on the z-transform and the z-domain's stability requirements, were presented in Chapter 2. In Chapter 3, the fixed-point and floating-point arithmetic emulation code was discussed along with a description of the process of integrating the emulation code with that of the SIMULINK™ software. Then, in Chapter 4, testing of the developed SIMULINK™ blocks, as described above, was performed on the Hubble Space Telescope (HST) with respect to various controller implementations to verify functionality of the SIMULINK™ blocks.

Future work on the emulation programs could include the following:

1. The implementation of various fixed-point arithmetic schemes such as 2's-complement or 1's-complement arithmetic could be added, which would allow for a greater variety of digital controllers to be emulated.

2. The addition of SIMULINK™ blocks which would allow for the simulation of various digital controller realizations. For example, the inclusion of a SIMULINK™ block(s) which emulates a controller realized with a transfer function implementation. Also, the creation of blocks which would allow for various controller state-space realizations to be emulated.

3. Include an input parameter within a SIMULINK™ block(s) which would implement any necessary scaling of the input of a digital controller that is realized with fixed-point arithmetic. In other words, the input to a controller is scaled accordingly such that it falls within the dynamic range of the controller's wordlength.

References

1. The MathWorks, Inc. (1992), *SIMULINK™ User's Guide*, Natick, MA.

2. The MathWorks, Inc. (1992), *MATLAB® User's Guide*, Natick, MA.

3. Follett, R.F. (1984), "Emulation and Characterization of Digital Controllers Realized with Fixed-Point Arithmetic", Master of Science Thesis, Mississippi State University, Mississippi State, May, 1984.

4. Peled, A. and Liu, B. (1976), *Digital Signal Processing: Theory, Design, and Implementation*, John Wiley & Sons, Inc., New York, NY.

5. Rabiner, L.R. and Gold, B. (1975), *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

6. Katz, Paul (1981), *Digital Control Using Microprocessors*, Prentice-Hall International, Inc., London, England.

7. Jacquot, R.G. (1981), *Modern Digital Control Systems*, Marcel Dekker, Inc., New York, NY.

8. Liu, K., Skelton, R.E., and Grigoriadis, K. (1992), "Optimal Controllers for Finite Wordlength Implementation", *IEEE Transactions on Automatic Control*, Vol. 37, No. 9, pp. 1294-1304, September, 1992.

9. Liu, Yu-cheng (1991), *The M68000 Microprocessor Family: Fundamentals of Assembly Language Programming & Interface Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

10. "IEEE Standard for Binary Floating-Point Arithmetic" (1985), Sponsor: *Standards Committee of the IEEE Computer Society*, (ANSI/IEEE Std 754-1985), The IEEE, Inc., New York, NY.

11. Nagle, H.T. and Phillips, C.L. (1990), *Digital Control System Analysis and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

12. Kuo, B.C. (1992), *Digital Control Systems*, Saunders College Publishing, Harcourt Brace Jovanovich College Publisher, Orlando, FL.

13. The MathWorks, Inc. (1992), *MATLAB® External Interface Guide*, Natick, MA.

14. Irwin, R.D., Glenn, R.D., Frazier, W.G., Lawrence, D.A., and Follett, R.F. (1994), "Analytically and Numerically Derived H^{∞} Controller Designs for the Hubble Space Telescope", *AIAA Journal of Guidance, Control, and Dynamics*, Washington, DC.

15. Sharkey, J.P., Nurre, G.S., Beals, G.A., and Nelson, J.D. (1992), "A Chronology of the On-Orbit Pointing Control System Changes on the Hubble Space Telescope and Associated Pointing Improvements", *AIAA Guidance, Navigation, and Control Conference*, Hilton Head, SC, August, 1992.

16. Frazier, W.G. (1993), "Search-Based Methods for Computer-Aided Controller Design Improvement", Doctor of Philosophy Dissertation, Ohio University, June 1993.

Appendix

The fixed-point and floating-point arithmetic emulation algorithms have been implemented via SIMULINK™'s S-Function C MEX-file option. The "adfix.c" emulates an analog-to-digital converter. The "mpufix.c" emulates a microprocessor-based digital controller realized with fixed-point arithmetic and implements state-space equations. The "dafix.c" emulates a digital-to-analog converter. The "mpufl.c" emulates a microprocessor-based digital controller realized with floating-point arithmetic and implements state-space equations. The "simsup.h" is the header-file which allows for the inherent sampling rates in digital control systems to be incorporated. Finally, the "simulin2.h" is the header-file which serves as the gateway routine, or in other words, allows for the emulation algorithms to be integrated with the SIMULINK™ software.

Sections A.1 - A.6 contain, respectively, the source code for the "adfix.c", "mpufix.c", "dafix.c", "mpufl.c", "simsup.h", and "simulin2.h" files mentioned above. Furthermore, Section A.7 contains the UNIX command which allows for the compilation of "C" source files and thus generates the necessary C MEX-files which are to be used in the SIMULINK™ S-Function Block.

A.1. "ADFIX.C"

```
/* "C" - code written by Edward D. Robe
Date: May 4, 1994 */
/* Filename: ADFIX.C */

#define MAX 200   /* LIMIT UP TO A 200 x 1 INPUT-VECTOR (U) */
#include <math.h>
#include "matrix.h"
#include "mex.h"

/* GLOBAL VARIABLES */
        int ninputs;
        int adwl,adrp,round,mputype;
        double pbdigital;
        long int iovr;
        int uinsign[MAX];

/* Define the system sizes global in this function: */
static int NSTATES;      /* Number of continuous states */
static int NDSTATES;     /* Number of discrete states */
static int NOUTPUTS;     /* Number of outputs */
static int NINPUTS;      /* Number of inputs */
static int NSING;        /* Number of singularities */
static int NEEDINPUTS; /* Has this system got direct-feedthrough */

#define NCOEFFS 6        /* Number of extra parameters passed in*/

#define NSAMPLE 1
static double sample_times[NSAMPLE];
static double offset_times[NSAMPLE] = { 0 };
static double next_hit[NSAMPLE] = { 0 };
static double sample_hit[NSAMPLE] = { 0 };

/* Storage for output to maintain outputs */
static double lasty[200] = {1};

static Matrix *Coeffs[NCOEFFS]; /* Pointer to the matrix parameters */
/* Defines for easy access of the B,ADWL,ADRP,ADROUND,TS,MPUTYPE
parameters which are passed in */
#define B               Coeffs[0]
#define ADWL            Coeffs[1]
#define ADRP            Coeffs[2]
#define ADROUND         Coeffs[3]
#define TS              Coeffs[4]
#define MPUTYPE         Coeffs[5]

/* Function to set up global size information */
static
#ifdef __STDC__
```

```
void initialize(void)
#else
void initialize()
#endif /* __STDC__ */
{

/* Define the Sample Time (TS) */
        sample_times[0] = mxGetPr(TS)[0];
        offset_times[NSAMPLE] = 0;
        next_hit[NSAMPLE] = 0;
        sample_hit[NSAMPLE] = 0;

/* Only do size checking in this initialization for efficiency */

        adwl    = (int) (mxGetPr(ADWL)[0]);
        adrp    = (int) (mxGetPr(ADRP)[0]);
        round   = (int) (mxGetPr(ADROUND)[0]);
        mputype = (int) (mxGetPr(MPUTYPE)[0]);

/* ERROR MESSAGES */
        if (adwl > 32) {
                mexErrMsgTxt("'A/D_wordlength' cannot exceed 32 bits.");
        }
        if (adrp >= adwl) {
                mexErrMsgTxt("A 'radix-point' cannot be '>=' the wordlength.");
        }

        pbdigital = pow(2.0,(double) adrp);
        /* CALCULATE THE MAXIMUM PERMISSIBLE NUMBER FOR THE A/D */
        iovr = (long int) ((pow(2.0,(double) ((adwl - 1)))) - 1.0);

        NSTATES = 0;
        NDSTATES = 0;
        NINPUTS = mxGetN(B);
        NOUTPUTS = mxGetN(B);
        NSING = 0;

        ninputs = NINPUTS; /* GLOBAL VARIABLES */

} /* END INITIALIZE FUNCTION */

/* ***********************************************************************
** MAIN PROGRAM ************************************************************
*********************************************************************** */

/* ######################################################################
######################################################################
################### BEGIN A/D FIXED-POINT FUNCTION ###################
######################################################################
###################################################################### */
```

```
/* AD() */

void ad(u)
double *u;

{

/* LOCAL VARIABLES */
        double rnu[MAX];
        long int nnu[MAX];

        int i;

/* ANALOG-TO-DIGITAL CONVERSION OF VECTOR INPUTS */

/* INITIALIZATION OF '*SIGN[]' */

for (i = 0; i < ninputs; i++) {
        uinsign[i] = 0;
}

/* CHECK TO SEE IF THE INDIVIDUAL ELEMENTS OF A VECTOR ARE NEGATIVE, AND
IF AN ELEMENT IS, TEMPORARILY CHANGE IT TO POSITIVE */

for (i = 0; i < ninputs; i++) {
        if (u[i] < 0.0) {
                uinsign[i] = 1;
                u[i] = -u[i];
        }
}

/* CALCULATE THE INTEGER REPRESENTATION OF THE NUMBER USING THE
WORDLENGTH OF THE A/D */

for (i = 0; i < ninputs; i++) {
        rnu[i] = u[i] * pbdigital;
}

/* PERFORM ROUNDING (TRUNCATION) PROCEDURE */

if (round == 1) {
        for (i = 0; i < ninputs; i++) {
                rnu[i] = rnu[i] + 0.5;
        }
} /* END IF-CONDITION FOR ROUNDING */

/* TAKE INTEGER PART OF THE REAL NUMBER */

for (i = 0; i < ninputs; i++) {
        nnu[i] = (long int) rnu[i];
}
```

```
/* COMPARE THE INTEGER NUMBER AND THE MAXIMUM, AND USE THE
SMALLER OF THE TWO */

for (i = 0; i < ninputs; i++) {
        if (nnu[i] > iovr) {
                nnu[i] = iovr;
        }
}

if (mputype == 1) { /* IMPLEMENTING A 'FIXED-PT.' MPU */
        /* ASSIGN INTEGER VALUES BACK TO THE (U) VECTOR */

        for (i = 0; i < ninputs; i++) {
                u[i] = nnu[i];
        }
}

if (mputype == 2) { /* IMPLEMENTING A 'FLOATING-PT.' MPU */
        /* ASSIGN VALUES BACK TO THE (U) VECTOR */

        for (i = 0; i < ninputs; i++) {
                u[i] = nnu[i] / pbdigital;
        }
}

/* IF THE ELEMENT NUMBER WAS NEGATIVE, CHANGE IT BACK */

for (i = 0; i < ninputs; i++) {
        if (uinsign[i] == 1) {
                u[i] = -u[i];
        }
}

} /* END A/D FIXED-POINT FUNCTION */
/* ##########################################################################
########################################################################## */

/* END MAIN */

/* Function to return any initial conditions on the states  */
static
#ifdef __STDC__
void init_conditions(double *x0)
#else
void init_conditions(x0)
double *x0;
#endif /* __STDC__ */
{
        return; /* NO CALCULATIONS NEEDED */
}
```

```
/* Function to return derivatives */
static
#ifdef __STDC__
void derivatives(double t, double *x, double *u, double *dx)
#else
void derivatives(t,x,u,dx)
double t, *x, *u;       /* Input variables */
double *dx;             /* Output variable */
#endif /* __STDC__ */
{
        return; /* No continuous states */
}

/* Function to perform discrete state update */
static
#ifdef __STDC__
void dstates(double t, double *x, double *u, double *ds)
#else
void dstates(t,x,u,ds)
double t, *x, *u;       /* Input variables */
double *ds;                 /* Output variable */
#endif /* __STDC__ */
{
        return; /* No discrete states */
}

/* Function to return outputs */
static
#ifdef __STDC__
void outputs(double t, double *x, double *u, double *y)
#else
void outputs(t,x,u,y)
double t, *x, *u;       /* Input variables */
double *y;              /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void ad();

int i;

if (t == sample_hit[0]) {

        ad(u); /* Call to A/D function */

        for (i = 0; i < NOUTPUTS; i++) {
                y[i] = u[i];
                lasty[i] = u[i];
        }
} /* END IF-LOOP */
```

```
else {
        for (i = 0; i < NOUTPUTS; i++) {
                y[i] = lasty[i];
        }
}

} /* END OUTPUTS FUNCTION */

/* Function to return singularities */
static
#ifdef __STDC__
void singularity(double t, double *x, double *u, double *sing)
#else
void singularity(t,x,u,sing)
double t, *x, *u; /* Input variables */
double *sing;               /* Output variable */
#endif /* __STDC__ */
{
        return; /* Default is to have no singularity */
}

#include "simulin2.h" /* Include file to perform MEX glue */

/* END 'ADFIX.C' PROGRAM
##########################################################################
##########################################################################
*/
```

A.2. "MPUFIX.C"

```
/* "C" - code written by Edward D. Robe
Date: May 4, 1994 */
/* Filename: MPUFIX.C */

#define MAX 40000       /* LIMIT UP TO A 200 x 200 MATRIX */
#include <stdio.h>
#include <math.h>
#include "matrix.h"     /* needed for matrix access functions */
#include "mex.h"        /* needed mexErrMsgTxt() prototyping */

/* GLOBAL VARIABLES */
        int nstates,noutputs,ninputs;
        int mpuwl,mpurp,adrp,round;
        double pbdigital,pbshiftm,rotate;
        long int iovr;
        int asign[MAX],bsign[MAX],csign[MAX],dsign[MAX],x0sign[200];

/* Define the system sizes global in this function: */
static int NSTATES;     /* Number of continuous states */
static int NDSTATES;    /* Number of discrete states */
static int NOUTPUTS;    /* Number of outputs */
static int NINPUTS;     /* Number of inputs */
static int NSING;       /* Number of singularities */
static int NEEDINPUTS; /* Has this system got direct-feedthrough */

#define NCOEFFS 6       /* Number of extra parameters passed in*/

#define NSAMPLE 1
static double sample_times[NSAMPLE];
static double offset_times[NSAMPLE] = { 0 };
static double next_hit[NSAMPLE] = { 0 };
static double sample_hit[NSAMPLE] = { 0 };

/* General defines */
#define TRUE 1
#define FALSE 0

static Matrix *Coeffs[NCOEFFS]; /* Pointer to the matrix parameters */
/* Defines for easy access of the A,B,C,D,XO,MPSETUP matrices
which are passed in */
#define A               Coeffs[0]
#define B               Coeffs[1]
#define C               Coeffs[2]
#define D               Coeffs[3]
#define X0              Coeffs[4]
#define MPSETUP         Coeffs[5]

/* Function to set up global size information */
```

```
static
#ifdef __STDC__
void initialize(void)
#else
void initialize()
#endif /* __STDC__ */
{

/* FUNCTIONS */
void main();

        double *Mtmp;
        long Dm,Dn,Dsize;
        int i;

/* Define the Sample Time (TS) */
        sample_times[0] = mxGetPr(MPSETUP)[0];
        offset_times[NSAMPLE] = 0;
        next_hit[NSAMPLE] = 0;
        sample_hit[NSAMPLE] = 0;

/* Only do size checking in this initialization for efficiency */

        mpuwl = (int) (mxGetPr(MPSETUP)[1]);
        mpurp = (int) (mxGetPr(MPSETUP)[2]);
        adrp  = (int) (mxGetPr(MPSETUP)[3]);
        round = (int) (mxGetPr(MPSETUP)[4]);

/* ERROR MESSAGES */
        if (mpuwl > 32) {
                mexErrMsgTxt("'MPU_wordlength' cannot exceed 32 bits.");
        }
        if (mpurp >= mpuwl) {
                mexErrMsgTxt("A 'radix-point' cannot be '>=' the wordlength.");
        }

        pbdigital = pow(2.0,(double) mpurp);
        pbshiftm  = pow(2.0,(double) ((mpurp - adrp)));
        rotate = (1.0 / pbdigital);
        /* CALCULATE THE MAXIMUM PERMISSIBLE NUMBER FOR THE MPU */
        iovr = (long int) ((pow(2.0,(double) ((mpuwl - 1)))) - 1.0);

        NSTATES = 0;
        NDSTATES = mxGetM(A);
        NINPUTS = mxGetN(B);
        NOUTPUTS = mxGetM(C);
        NSING = 0;

        nstates = NDSTATES;
        noutputs = NOUTPUTS; /* GLOBAL VARIABLES */
        ninputs = NINPUTS;
```

```
        if (mxGetN(A) != NDSTATES) {
                mexErrMsgTxt("A matrix must be square");
        }
        if (mxGetM(B) != NDSTATES) {
                mexErrMsgTxt("B matrix not dimensioned correctly");
        }
        if (mxGetN(C) != NDSTATES) {
                mexErrMsgTxt("C matrix not dimensioned correctly");
        }
        if (mxGetM(D) != NOUTPUTS || mxGetN(D) != NINPUTS) {
                mexErrMsgTxt("D matrix not dimensioned correctly");
        }
        if (mxGetM(X0) != 0 && mxGetM(X0) != NDSTATES) {
                mexErrMsgTxt("X0 matrix not dimensioned correctly");
        }

/* Check for direct feedthrough (i.e., non-zero D matrix) */
/* An alternative method is to use mexCallMatlab to do the check */

        Mtmp = mxGetPr(D);
        Dm = mxGetM(D);
        Dn = mxGetN(D);
        Dsize = Dm * Dn;

        NEEDINPUTS = FALSE;
        for (i = 0; i < Dsize; i++) {
                if (Mtmp[i] != 0) {
                        NEEDINPUTS = TRUE;
                        break;
                }
        }

main(); /* Call to MAIN function for quantizing coefficients of
        the state-space realization */

} /* END INITIALIZE FUNCTION */

/* ************************************************************************
** MAIN PROGRAM ************************************************************
************************************************************************ */

/* BEGIN MAIN */
void main()
{

/* FUNCTION */
void state_space();

/* CALL TO FUNCTION TO PERFORM QUANTIZATION */

        state_space();
```

```
} /* END MAIN */

/* ##########################################################################
########################################################################
################ BEGIN STATE_SPACE FIXED-POINT FUNCTION ################
########################################################################
########################################################################## */

/* STATE_SPACE() */

void state_space()

{

/* LOCAL VARIABLES */
        double rna[MAX],rnb[MAX],rnc[MAX],rnd[MAX],rnx0[200];
        long int nna[MAX],nnb[MAX],nnc[MAX],nnd[MAX],nnx0[200];

        int i,j;
        double *apr, *bpr, *cpr, *dpr, *xopr;

        apr = (double *)mxGetPr(A);
        bpr = (double *)mxGetPr(B);
        cpr = (double *)mxGetPr(C);
        dpr = (double *)mxGetPr(D);
        xopr = (double *)mxGetPr(X0);

/* STATE_SPACE QUANT CONVERSION */

/* INITIALIZATION OF '*SIGN[]' */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                asign[i + nstates*j] = 0;
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                bsign[i + nstates*j] = 0;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                csign[i + noutputs*j] = 0;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
```

```
                dsign[i + noutputs*j] = 0;
        }
}

for (i = 0; i < nstates; i++){
        x0sign[i] = 0;
}

/* CHECK TO SEE IF THE INDIVIDUAL ELEMENTS OF A MATRIX ARE NEGATIVE, AND
IF AN ELEMENT IS, TEMPORARILY CHANGE IT TO POSITIVE */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                if (apr[i + nstates*j] < 0.0){
                        asign[i + nstates*j] = 1;
                        apr[i + nstates*j] = -apr[i + nstates*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                if (bpr[i + nstates*j] < 0.0){
                        bsign[i + nstates*j] = 1;
                        bpr[i + nstates*j] = -bpr[i + nstates*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                if (cpr[i + noutputs*j] < 0.0){
                        csign[i + noutputs*j] = 1;
                        cpr[i + noutputs*j] = -cpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                if (dpr[i + noutputs*j] < 0.0){
                        dsign[i + noutputs*j] = 1;
                        dpr[i + noutputs*j] = -dpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        if (xopr[i] < 0.0){
                x0sign[i] = 1;
                xopr[i] = -xopr[i];
```

```
        }
}

/* CALCULATE THE INTEGER REPRESENTATION OF THE NUMBER USING THE
WORDLENGTH OF THE MPU */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                rma[i + nstates*j] = apr[i + nstates*j] * pbdigital;
        }
}
for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                rmb[i + nstates*j] = bpr[i + nstates*j] * pbdigital;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                rmc[i + noutputs*j] = cpr[i + noutputs*j] * pbdigital;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                rmd[i + noutputs*j] = dpr[i + noutputs*j] * pbdigital;
        }
}

for (i = 0; i < nstates; i++){
        rmx0[i] = xopr[i] * pbdigital;
}

/* PERFORM ROUNDING (TRUNCATION) PROCEDURE */

if (round == 1){

        for (i = 0; i < nstates; i++){
                for (j = 0; j < nstates; j++){
                        rma[i + nstates*j] = rma[i + nstates*j] + 0.5;
                }
        }

        for (i = 0; i < nstates; i++){
                for (j = 0; j < ninputs; j++){
                        rmb[i + nstates*j] = rmb[i + nstates*j] + 0.5;
                }
        }

        for (i = 0; i < noutputs; i++){
                for (j = 0; j < nstates; j++){
```

```
                        rnc[i + noutputs*j] = rnc[i + noutputs*j] + 0.5;
                }
        }

        for (i = 0; i < noutputs; i++){
                for (j = 0; j < ninputs; j++){
                        rnd[i + noutputs*j] = rnd[i + noutputs*j] + 0.5;
                }
        }

        for (i = 0; i < nstates; i++){
                rnx0[i] = rnx0[i] + 0.5;
        }

} /* END IF-CONDITION FOR ROUNDING */

/* TAKE INTEGER PART OF THE REAL NUMBER */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                nna[i + nstates*j] = (long int) rna[i + nstates*j];
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                nnb[i + nstates*j] = (long int) rnb[i + nstates*j];
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                nnc[i + noutputs*j] = (long int) rnc[i + noutputs*j];
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                nnd[i + noutputs*j] = (long int) rnd[i + noutputs*j];
        }
}

for (i = 0; i < nstates; i++){
        nnx0[i] = (long int) rnx0[i];
}

/* COMPARE THE INTEGER NUMBER AND THE MAXIMUM, AND USE THE
SMALLER OF THE TWO */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
```

```
                if (nna[i + nstates*j] > iovr){
                        nna[i + nstates*j] = iovr;
                }
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                if (nnb[i + nstates*j] > iovr){
                        nnb[i + nstates*j] = iovr;
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                if (nnc[i + noutputs*j] > iovr){
                        nnc[i + noutputs*j] = iovr;
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                if (nnd[i + noutputs*j] > iovr){
                        nnd[i + noutputs*j] = iovr;
                }
        }
}

for (i = 0; i < nstates; i++){
        if (nnx0[i] > iovr){
                nnx0[i] = iovr;
        }
}

/* ASSIGN INTEGER VALUES BACK TO THE (A,B,C,D,X0) MATRICES */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                apr[i + nstates*j] = nna[i + nstates*j];
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                bpr[i + nstates*j] = nnb[i + nstates*j];
        }
}

for (i = 0; i < noutputs; i++){
```

```
        for (j = 0; j < nstates; j++){
                cpr[i + noutputs*j] = nnc[i + noutputs*j];
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                dpr[i + noutputs*j] = nnd[i + noutputs*j];
        }
}

for (i = 0; i < nstates; i++){
        xopr[i] = nnx0[i];
}

/* IF THE ELEMENT NUMBER WAS NEGATIVE, CHANGE IT BACK */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                if (asign[i + nstates*j] == 1){
                        apr[i + nstates*j] = -apr[i + nstates*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                if (bsign[i + nstates*j] == 1){
                        bpr[i + nstates*j] = -bpr[i + nstates*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                if (csign[i + noutputs*j] == 1){
                        cpr[i + noutputs*j] = -cpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                if (dsign[i + noutputs*j] == 1){
                        dpr[i + noutputs*j] = -dpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        if (x0sign[i] == 1){
```

```
                xopr[i] = -xopr[i];
        }
}

} /* END STATE_SPACE FIXED-POINT FUNCTION */
/* ###########################################################################
########################################################################### */

/* ###########################################################################
###########################################################################
################## BEGIN MPU_INPUTS FIXED-POINT FUNCTION ##################
###########################################################################
########################################################################### */

/* MPUINPUTS() */

void mpuinputs(u)
double *u;

{

/* LOCAL VARIABLES */
        int i;

/* NOW CONVERTING THE INPUTS (U) TO THE WORDLENGTH OF THE
MICROPROCESSOR (MPU) BY SHIFTING THE RADIX POINT */

for (i = 0; i < ninputs; i++) {
        u[i] = u[i] * pbshiftm;
}

} /* END MPU_INPUTS FIXED-POINT FUNCTION */
/* ###########################################################################
########################################################################### */

/* ###########################################################################
###########################################################################
################## BEGIN ROTFUNC FIXED-POINT FUNCTION ##################
###########################################################################
########################################################################### */

/* ROTFUNC() */

void rotfunc(x)
double *x;

{

/* LOCAL VARIABLES */
        int xsign;
```

```
/* THIS FUNCTION ACCEPTS THE RESULT OF AN INTEGER MULTIPLICATION
(WITH AN IMPLIED RADIX POINT) AND ADJUSTS THE NUMBER TO GET IT
BACK IN THE SAME FORM AS THE ORIGINAL NUMBERS (E.G., THE SAME
NUMBER OF PLACES TO THE RIGHT OF THE RADIX POINT). */

        xsign = 0;
        if ((*x) < 0.0) {
                xsign = 1;
                (*x) = -(*x);
        }

        (*x) = (*x) * rotate;

        if (round == 1) {
                (*x) = (long int) (((*x) + 0.5));
        }

        if ((*x) > ((double) iovr)) {
                (*x) = iovr;
        }

        if (xsign == 1) {
                (*x) = -(*x);
        }

} /* END ROTFUNC FIXED-POINT FUNCTION */
/* ########################################################################
######################################################################## */

/* ########################################################################
########################################################################
################ BEGIN CHECKLIMIT FIXED-POINT FUNCTION ################
########################################################################
######################################################################## */

/* CHECKLIMIT() */

void checklimit(xx)
double *xx;

{

/* LOCAL VARIABLES */
        int xxsign;

/* THIS FUNCTION CHECKS FOR OVERFLOW. */

        xxsign = 0;
        if ((*xx) < 0.0) {
                xxsign = 1;
                (*xx) = -(*xx);
```

```
        }

        if ((*xx) > ((double) iovr)) {
                (*xx) = iovr;
        }

        if (xxsign == 1) {
                (*xx) = -(*xx);
        }

} /* END CHECKLIMIT FIXED-POINT FUNCTION */
/* ###########################################################################
########################################################################### */

/* Function to return any initial conditions on the states  */
static
#ifdef __STDC__
void init_conditions(double *x0)
#else
void init_conditions(x0)
double *x0;
#endif /* __STDC__ */
{
        int i;
        double *pr;

        pr = (double *)mxGetPr(X0);
        for (i=mxGetM(X0); i; i--) {
                *x0++ = *pr++;
        }
}

/* Function to return derivatives */
static
#ifdef __STDC__
void derivatives(double t, double *x, double *u, double *dx)
#else
void derivatives(t,x,u,dx)
double t, *x, *u;       /* Input variables */
double *dx;             /* Output variable */
#endif /* __STDC__ */
{
        return;  /* No continuous states */
}

/* Function to perform discrete state update */
static
#ifdef __STDC__
void dstates(double t, double *x, double *u, double *ds)
#else
void dstates(t,x,u,ds)
```

```
double t, *x, *u;       /* Input variables */
double *ds;             /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void mpuinputs();
void rotfunc();
void checklimit();

/* LOCAL VARIABLES */
        int i,j;
        double accum,temp;
        double *apr, *bpr;

        apr = (double *)mxGetPr(A);
        bpr = (double *)mxGetPr(B);

        /* Matrix Multiply  x(n+1) = Ax(n) + Bu(n) */

        /* Update discrete states on a sample hit */

if (t == sample_hit[0]) {

        mpuinputs(u); /* Adjust inputs to MPU */

        for (i = 0; i < NDSTATES; i++) {
                accum = 0.0;
                temp = 0.0;
        /* Ax(n) */
                for (j = 0; j < NDSTATES; j++) {
                        temp = apr[i + NDSTATES*j] * x[j];
                        rotfunc(&temp);
                        accum = accum + temp;
                }
        /* Bu(n) */
                for (j = 0; j < NINPUTS; j++) {
                        temp = bpr[i + NDSTATES*j] * u[j];
                        rotfunc(&temp);
                        accum = accum + temp;
                }
                checklimit(&accum);
                ds[i] = accum;
        }
} /* END IF-LOOP */

} /* END DSTATES FUNCTION */

/* Function to return outputs */
static
#ifdef __STDC__
```

```
void outputs(double t, double *x, double *u, double *y)
#else
void outputs(t,x,u,y)
double t, *x, *u;       /* Input variables */
double *y;              /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void mpuinputs();
void rotfunc();
void checklimit();

/* LOCAL VARIABLES */
        int i,j;
        double accum,temp;
        double *cpr, *dpr;

        cpr = (double *)mxGetPr(C);
        dpr = (double *)mxGetPr(D);

/* Matrix Multiply  y(n) = Cx(n) + Du(n) */

        mpuinputs(u); /* Adjust inputs to MPU */

        for (i = 0; i < NOUTPUTS; i++) {
                accum = 0.0;
                temp = 0.0;
        /* Cx(n) */
                for (j = 0; j < NDSTATES; j++) {
                        temp = cpr[i + NOUTPUTS*j] * x[j];
                        rotfunc(&temp);
                        accum = accum + temp;
                }
        /* Du(n) */
                for (j = 0; j < NINPUTS; j++) {
                        temp = dpr[i + NOUTPUTS*j] * u[j];
                        rotfunc(&temp);
                        accum = accum + temp;
                }
                checklimit(&accum);
                y[i] = accum;
        }

} /* END OUTPUTS FUNCTION */

/* Function to return singularities */
static
#ifdef __STDC__
void singularity(double t, double *x, double *u, double *sing)
#else
```

```
void singularity(t,x,u,sing)
double t, *x, *u; /* Input variables */
double *sing;                /* Output variable */
#endif /* __STDC__ */
{
        return; /* Default is to have no singularity */
}

#include "simulin2.h" /* Include file to perform MEX glue */

/* END 'MPUFIX.C' PROGRAM
##########################################################################
##########################################################################
*/
```

A.3. "DAFIX.C"

```
/* "C" - code written by Edward D. Robe
Date: May 4, 1994 */
/* Filename: DAFIX.C */

#define MAX 200         /* LIMIT UP TO A 200 x 1 OUTPUT-VECTOR (Y) */
#include <math.h>
#include "matrix.h"
#include "mex.h"

/* GLOBAL VARIABLES */
        int noutputs;
        int mpurp,dawl,darp;
        long int iovr;
        double pbshiftd, pbanalog;
        int youtsign[MAX];

/* Define the system sizes global in this function: */
static int NSTATES;      /* Number of continuous states */
static int NDSTATES;     /* Number of discrete states */
static int NOUTPUTS;     /* Number of outputs */
static int NINPUTS;      /* Number of inputs */
static int NSING;        /* Number of singularities */
static int NEEDINPUTS;   /* Has this system got direct-feedthrough */

#define NCOEFFS 6        /* Number of extra parameters passed in*/

#define NSAMPLE 1
static double sample_times[NSAMPLE];
static double offset_times[NSAMPLE] = { 0 };
static double next_hit[NSAMPLE] = { 0 };
static double sample_hit[NSAMPLE] = { 0 };

/* Storage for output to maintain outputs */
static double lasty[200] = {1};

static Matrix *Coeffs[NCOEFFS];  /* Pointer to the matrix parameters */
/* Defines for easy access of the B,C,MPURP,DAWL,DARP,TS parameters
which are passed in */
#define B       Coeffs[0]
#define C       Coeffs[1]
#define MPURP   Coeffs[2]
#define DAWL    Coeffs[3]
#define DARP    Coeffs[4]
#define TS      Coeffs[5]

/* Function to set up global size information */
static
#ifdef __STDC__
```

```
void initialize(void)
#else
void initialize()
#endif /* __STDC__ */
{

/* Define the Sample Time (TS) */
        sample_times[0] = mxGetPr(TS)[0];
        offset_times[NSAMPLE] = 0;
        next_hit[NSAMPLE] = 0;
        sample_hit[NSAMPLE] = 0;

/* Only do size checking in this initialization for efficiency */

        mpurp = (int) (mxGetPr(MPURP)[0]);
        dawl  = (int) (mxGetPr(DAWL)[0]);
        darp  = (int) (mxGetPr(DARP)[0]);

/* ERROR MESSAGES */
        if (dawl > 32) {
                mexErrMsgTxt("'D/A_wordlength' cannot exceed 32 bits.");
        }
        if (darp >= dawl) {
                mexErrMsgTxt("A 'radix-point' cannot be '>=' the wordlength.");
        }

        pbshiftd = pow(2.0,(double) ((mpurp - darp)));
        pbanalog = pow(2.0,(double) darp);
        /* CALCULATE THE MAXIMUM PERMISSIBLE NUMBER FOR THE A/D */
        iovr = (long int) ((pow(2.0,(double) ((dawl - 1)))) - 1.0);

        NSTATES = 0;
        NDSTATES = 0;
        NINPUTS = mxGetN(B);
        NOUTPUTS = mxGetM(C);
        NSING = 0;

        noutputs = NOUTPUTS; /* GLOBAL VARIABLES */

} /* END INITIALIZE FUNCTION */

/* ***************************************************************************
** MAIN PROGRAM ************************************************************
*************************************************************************** */

/* ###########################################################################
###########################################################################
################### BEGIN D/A FIXED-POINT FUNCTION ########################
###########################################################################
########################################################################### */
```

```
/* DA() */

void da(u)
double *u;

{

/* LOCAL VARIABLES */
        int i;

/* DIGITAL-TO-ANALOG CONVERSION OF VECTOR OUTPUTS */

/* INITIALIZATION OF '*SIGN[]' */

for (i = 0; i < noutputs; i++) {
        youtsign[i] = 0;
}

/* CHECK TO SEE IF THE INDIVIDUAL ELEMENTS OF A VECTOR ARE NEGATIVE, AND
IF AN ELEMENT IS, TEMPORARILY CHANGE IT TO POSITIVE */

for (i = 0; i < noutputs; i++) {
        if (u[i] < 0.0) {
                youtsign[i] = 1;
                u[i] = -u[i];
        }
}

/* NOW CONVERT THE OUTPUTS (Y) TO THE WORDLENGTH OF THE D/A
BY SHIFTING THE RADIX POINT */

for (i = 0; i < noutputs; i++) {
        u[i] = u[i] / pbshiftd;
}

/* NOW MAKE THE D/A CONVERSION */

for (i = 0; i < noutputs; i++) {
        u[i] = u[i] / pbanalog;
}

/* COMPARE THE NUMBER AND THE MAXIMUM, AND USE THE
SMALLER OF THE TWO */

for (i = 0; i < noutputs; i++) {
        if (u[i] > (double) iovr) {
                u[i] = iovr;
        }
}

/* IF THE ELEMENT NUMBER WAS NEGATIVE, CHANGE IT BACK */
```

```
for (i = 0; i < noutputs; i++) {
        if (youtsign[i] == 1) {
                u[i] = -u[i];
        }
}

} /* END D/A FIXED-POINT FUNCTION */
/* ########################################################################
######################################################################## */

/* END MAIN */

/* Function to return any initial conditions on the states  */
static
#ifdef __STDC__
void init_conditions(double *x0)
#else
void init_conditions(x0)
double *x0;
#endif /* __STDC__ */
{
        return; /* NO CALCULATIONS NEEDED */
}

/* Function to return derivatives */
static
#ifdef __STDC__
void derivatives(double t, double *x, double *u, double *dx)
#else
void derivatives(t,x,u,dx)
double t, *x, *u;       /* Input variables */
double *dx;             /* Output variable */
#endif /* __STDC__ */
{
        return; /* No continuous states */
}

/* Function to perform discrete state update */
static
#ifdef __STDC__
void dstates(double t, double *x, double *u, double *ds)
#else
void dstates(t,x,u,ds)
double t, *x, *u;       /* Input variables */
double *ds;                 /* Output variable */
#endif /* __STDC__ */
{
        return; /* No discrete states */
}

/* Function to return outputs */
```

```
static
#ifdef __STDC__
void outputs(double t, double *x, double *u, double *y)
#else
void outputs(t,x,u,y)
double t, *x, *u;       /* Input variables */
double *y;              /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void da();

int i;

if (t == sample_hit[0]) {

        da(u); /* Call to D/A function */

        for (i = 0; i < NOUTPUTS; i++) {
                y[i] = u[i];
                lasty[i] = u[i];
        }
} /* END IF-LOOP */
else {
        for (i = 0; i < NOUTPUTS; i++) {
                y[i] = lasty[i];
        }
}

} /* END OUTPUTS FUNCTION */

/* Function to return singularities */
static
#ifdef __STDC__
void singularity(double t, double *x, double *u, double *sing)
#else
void singularity(t,x,u,sing)
double t, *x, *u; /* Input variables */
double *sing;               /* Output variable */
#endif /* __STDC__ */
{
        return; /* Default is to have no singularity */
}

#include "simulin2.h" /* Include file to perform MEX glue */

/* END 'DAFIX.C' PROGRAM
##########################################################################
##########################################################################
*/
```

A.4. "MPUFL.C"

```
/* "C" - code written by Edward D. Robe
Date: May 4, 1994 */
/* Filename: MPUFL.C */

#define MAX 40000              /* LIMIT UP TO A 200 x 200 MATRIX */
#define MAXSTRING 54           /* LIMIT TO 54 BITS IN MANTISSA WORDLENGTH */
#include <math.h>
#include "matrix.h"            /* needed for matrix access functions */
#include "mex.h"               /* needed mexErrMsgTxt() prototyping */

/* STRUCTURE TYPES: ASSIGN INTEGER WORDS TO '8' BYTES */
struct Integer8 {
        unsigned long int upper,lower;
};

typedef struct Integer8 Integer8;

/* GLOBAL VARIABLES */
        int nstates,noutputs,ninputs,expox,mantx,roundx,matrixstar;
        int bound,biaslimit,darp;
        int mw,mx,mz;
        unsigned long int my;
        double infinityy,nearzero,pbda;
        double shiftpoint1,shiftpoint2,shiftpoint3,shiftpoint4;
        double shiftpoint5,shiftpoint6;
        double invshiftpt1,invshiftpt2,invshiftpt3,invshiftpt4;
        double invshiftpt5;
        double LFN,MNFN;
        int asign[MAX],bsign[MAX],csign[MAX],dsign[MAX],x0sign[200];
        int uinsign[200],quantsign[1];
        int signalflag;

/* Define the system sizes global in this function: */
static int NSTATES;      /* Number of continuous states */
static int NDSTATES;     /* Number of discrete states */
static int NOUTPUTS;     /* Number of outputs */
static int NINPUTS;      /* Number of inputs */
static int NSING;        /* Number of singularities */
static int NEEDINPUTS; /* Has this system got direct-feedthrough */

#define NCOEFFS 6      /* Number of extra parameters passed in*/

#define NSAMPLE 1
static double sample_times[NSAMPLE];
static double offset_times[NSAMPLE] = { 0 };
static double next_hit[NSAMPLE] = { 0 };
static double sample_hit[NSAMPLE] = { 0 };
```

```
/* General defines */
#define TRUE 1
#define FALSE 0

static Matrix *Coeffs[NCOEFFS]; /* Pointer to the matrix parameters */
/* Defines for easy access of the A,B,C,D,XO,MPSETUP matrices
which are passed in */
#define A               Coeffs[0]
#define B               Coeffs[1]
#define C               Coeffs[2]
#define D               Coeffs[3]
#define X0              Coeffs[4]
#define MPSETUP         Coeffs[5]

/* Function to set up global size information */
static
#ifdef __STDC__
void initialize(void)
#else
void initialize()
#endif /* __STDC__ */
{

/* FUNCTIONS */
void main();

        double *Mtmp;
        long Dm,Dn,Dsize;

        int i;
        unsigned long int temp = 1;
        double temp1,temp2,max;

/* Define the Sample Time (TS) */
        sample_times[0] =  mxGetPr(MPSETUP)[0];
        offset_times[NSAMPLE] = 0;
        next_hit[NSAMPLE] = 0;
        sample_hit[NSAMPLE] = 0;

/* Only do size checking in this initialization for efficiency */

        expox  = (int) (mxGetPr(MPSETUP)[1]);
        mantx  = (int) (mxGetPr(MPSETUP)[2]);
        roundx = (int) (mxGetPr(MPSETUP)[3]);
        darp   = (int) (mxGetPr(MPSETUP)[4]);

/* ERROR MESSAGES */
        if ((expox + mantx + 1) > 64) {
                mexErrMsgTxt("'MPU_wordlength' cannot exceed 64 bits.");
        }
```

```
        NSTATES = 0;
        NDSTATES = mxGetM(A);
        NINPUTS = mxGetN(B);
        NOUTPUTS = mxGetM(C);
        NSING = 0;

        nstates = NDSTATES;
        noutputs = NOUTPUTS;  /* GLOBAL VARIABLES */
        ninputs = NINPUTS;

        if (mxGetN(A) != NDSTATES) {
                mexErrMsgTxt("A matrix must be square");
        }
        if (mxGetM(B) != NDSTATES) {
                mexErrMsgTxt("B matrix not dimensioned correctly");
        }
        if (mxGetN(C) != NDSTATES) {
                mexErrMsgTxt("C matrix not dimensioned correctly");
        }
        if (mxGetM(D) != NOUTPUTS || mxGetN(D) != NINPUTS) {
                mexErrMsgTxt("D matrix not dimensioned correctly");
        }
        if (mxGetM(X0) != 0 && mxGetM(X0) != NDSTATES) {
                mexErrMsgTxt("X0 matrix not dimensioned correctly");
        }

/* Check for direct feedthrough (i.e., non-zero D matrix) */
/* An alternative method is to use mexCallMatlab to do the check */

        Mtmp = mxGetPr(D);
        Dm = mxGetM(D);
        Dn = mxGetN(D);
        Dsize = Dm * Dn;

        NEEDINPUTS = FALSE;
        for (i = 0; i < Dsize; i++) {
                if (Mtmp[i] != 0) {
                        NEEDINPUTS = TRUE;
                        break;
                }
        }

/* DETERMINE THE 'INTEGER' WORDLENGTH OF THE MACHINE */
mx = sizeof(temp);          /* Returns # of bytes for an 'unsigned long int' */
mz = mx * 8;                /* Determines # of bits based on 8 bits per 1 byte */
mw = mz - 1;
my = (unsigned long int) (pow(2.0,(double) mw));

/* DETERMINE 'EXPONENT' BOUNDS AND LIMITS */
temp2 = pow(2.0,(double) expox);
bound = (int) (temp2 - 1.0);
```

```
temp2 = temp2 / 2.0;
biaslimit = (int) (temp2 - 1.0);
infinityy = pow(2.0,(double) (biaslimit + 1));
nearzero = pow(2.0,(double) (-(biaslimit - 1)));

/* CALCULATE FOR THE FORMAT'S LARGEST FINITE NUMBER */
max = 0.0;
temp1 = 0.0;
LFN = 0.0;
for (i = (biaslimit - mantx); i <= biaslimit; i++){
        temp1 = pow(2.0,(double) i);
        max = max + temp1;
}
LFN = max;
MNFN = LFN; /* without (-) sign */

/* USED IN THE 'CALCFRACTION' AND 'NUMBER2INT' FUNCTIONS: */
/* USED FOR SCALING DUE TO MACHINE LIMITATIONS FOR REPRESENTING INTEGERS,
AND FOR TRANSFORMING FRACTIONAL NUMBERS INTO INTEGER NUMBERS WHICH THEN
CAN BE MANIPULATED BY THE 'BITWISE OPERATORS' */
shiftpoint1 = pow(2.0,(double) mz);
shiftpoint2 = pow(2.0,(double) ((2*mz)));
shiftpoint3 = pow(2.0,(double) ((4*mz)));          /* Covers to Single Prec. */
shiftpoint4 = pow(2.0,(double) ((8*mz)));
shiftpoint5 = pow(2.0,(double) ((16*mz)));
shiftpoint6 = pow(2.0,(double) ((32*mz))); /* Covers to Double Prec. */
                                                   /* and Single Extended */
invshiftpt1 = (1.0 / shiftpoint1);
invshiftpt2 = (1.0 / shiftpoint2);
invshiftpt3 = (1.0 / shiftpoint3);
invshiftpt4 = (1.0 / shiftpoint4);
invshiftpt5 = (1.0 / shiftpoint5);

pbda = pow(2.0,(double) darp);

main(); /* Call to MAIN function for quantizing coefficients of
        the state-space realization */

} /* END INITIALIZE FUNCTION */

/* ************************************************************************
** MAIN PROGRAM ***********************************************************
************************************************************************** */

/* BEGIN MAIN */
void main()
{

/* FUNCTION */
void state_space();
```

```
/* CALL TO FUNCTION TO PERFORM STATE-SPACE QUANTIZATION */

        state_space();

} /* END MAIN */


/* ########################################################################
########################################################################
################ BEGIN STATE_SPACE FLOATING-POINT FUNCTION ###############
########################################################################
######################################################################## */

/* STATE_SPACE() */

void state_space()

{

/* FUNCTIONS */
void extractexpo();
double number;
int E;

void number2int();
Integer8 integral;
double fractional;

void convert();
double NUMBER;

void roundcheck();

/* LOCAL VARIABLES */
        int i,j;
        double *apr, *bpr, *cpr, *dpr, *xopr;

        apr = (double *)mxGetPr(A);
        bpr = (double *)mxGetPr(B);
        cpr = (double *)mxGetPr(C);
        dpr = (double *)mxGetPr(D);
        xopr = (double *)mxGetPr(X0);

/* QUANT CONVERSION FOR THE STATE_SPACE: */

/* INITIALIZATION OF '*SIGN[]' */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                asign[i + nstates*j] = 0;
        }
```

```
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                bsign[i + nstates*j] = 0;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                csign[i + noutputs*j] = 0;
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                dsign[i + noutputs*j] = 0;
        }
}

for (i = 0; i < nstates; i++){
        x0sign[i] = 0;
}

/* CHECK TO SEE IF THE INDIVIDUAL ELEMENTS OF A MATRIX ARE NEGATIVE, AND
IF AN ELEMENT IS, TEMPORARILY CHANGE IT TO POSITIVE */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                if (apr[i + nstates*j] < 0.0){
                        asign[i + nstates*j] = 1;
                        apr[i + nstates*j] = -apr[i + nstates*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                if (bpr[i + nstates*j] < 0.0){
                        bsign[i + nstates*j] = 1;
                        bpr[i + nstates*j] = -bpr[i + nstates*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                if (cpr[i + noutputs*j] < 0.0){
                        csign[i + noutputs*j] = 1;
                        cpr[i + noutputs*j] = -cpr[i + noutputs*j];
                }
```

```
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                if (dpr[i + noutputs*j] < 0.0){
                        dsign[i + noutputs*j] = 1;
                        dpr[i + noutputs*j] = -dpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        if (xopr[i] < 0.0){
                x0sign[i] = 1;
                xopr[i] = -xopr[i];
        }
}

/* CALCULATE THE FLOATING-POINT REPRESENTATION OF THE NUMBER
USING BOTH THE EXPONENT AND MANTISSA WORDLENGTHS OF THE MPU */

matrixstar = 0;

/* CONVERT 'A' MATRIX: */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                matrixstar = 1;
                signalflag = 0;
                number = apr[i + nstates*j];
                extractexpo(number,&E);
                number2int(number,&integral,&fractional);
                convert(number,E,integral,fractional,&NUMBER);
                number = NUMBER;
                roundcheck(i,j,&number,&signalflag);
                if (signalflag == 1) {
                  printf("\nThe A[%d][%d] element is: %s \n",i,j,"NaN"); }
                if (signalflag == 2) {
                  printf("\nThe A[%d][%d] element is: %s \n",i,j,"Inf"); }
                apr[i + nstates*j] = number;
        } /* THE OUTER 'j' INDICE LOOP */
} /* THE OUTER 'i' INDICE LOOP */

/* CONVERT 'B' MATRIX: */

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                matrixstar = 2;
                signalflag = 0;
                number = bpr[i + nstates*j];
```

```
                extractexpo(number,&E);
                number2int(number,&integral,&fractional);
                convert(number,E,integral,fractional,&NUMBER);
                number = NUMBER;
                roundcheck(i,j,&number,&signalflag);
                if (signalflag == 1) {
                  printf("\nThe B[%d][%d] element is: %s \n",i,j,"NaN"); }
                if (signalflag == 2) {
                  printf("\nThe B[%d][%d] element is: %s \n",i,j,"Inf"); }
                bpr[i + nstates*j] = number;
        } /* THE OUTER 'j' INDICE LOOP */
} /* THE OUTER 'i' INDICE LOOP */

/* CONVERT 'C' MATRIX: */

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                matrixstar = 3;
                signalflag = 0;
                number = cpr[i + noutputs*j];
                extractexpo(number,&E);
                number2int(number,&integral,&fractional);
                convert(number,E,integral,fractional,&NUMBER);
                number = NUMBER;
                roundcheck(i,j,&number,&signalflag);
                if (signalflag == 1) {
                  printf("\nThe C[%d][%d] element is: %s \n",i,j,"NaN"); }
                if (signalflag == 2) {
                  printf("\nThe C[%d][%d] element is: %s \n",i,j,"Inf"); }
                cpr[i + noutputs*j] = number;
        } /* THE OUTER 'j' INDICE LOOP */
} /* THE OUTER 'i' INDICE LOOP */

/* CONVERT 'D' MATRIX: */

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                matrixstar = 4;
                signalflag = 0;
                number = dpr[i + noutputs*j];
                extractexpo(number,&E);
                number2int(number,&integral,&fractional);
                convert(number,E,integral,fractional,&NUMBER);
                number = NUMBER;
                roundcheck(i,j,&number,&signalflag);
                if (signalflag == 1) {
                  printf("\nThe D[%d][%d] element is: %s \n",i,j,"NaN"); }
                if (signalflag == 2) {
                  printf("\nThe D[%d][%d] element is: %s \n",i,j,"Inf"); }
                dpr[i + noutputs*j] = number;
        } /* THE OUTER 'j' INDICE LOOP */
```

```
} /* THE OUTER 'i' INDICE LOOP */

/* CONVERT 'X0' MATRIX: */

for (i = 0; i < nstates; i++){
        j = 0;
        matrixstar = 5;
        signalflag = 0;
        number = xopr[i];
        extractexpo(number,&E);
        number2int(number,&integral,&fractional);
        convert(number,E,integral,fractional,&NUMBER);
        number = NUMBER;
        roundcheck(i,j,&number,&signalflag);
        if (signalflag == 1) {
                printf("\nThe X0[%d][%d] element is: %s \n",i,j,"NaN"); }
        if (signalflag == 2) {
                printf("\nThe X0[%d][%d] element is: %s \n",i,j,"Inf"); }
        xopr[i] = number;
} /* THE OUTER 'i' INDICE LOOP */

/* IF THE ELEMENT WAS NEGATIVE, CHANGE IT BACK */

for (i = 0; i < nstates; i++){
        for (j = 0; j < nstates; j++){
                if (asign[i + nstates*j] == 1){
                        apr[i + nstates*j] = -apr[i + nstates*j];
                }
        }
}

for (i = 0; i < nstates; i++){
        for (j = 0; j < ninputs; j++){
                if (bsign[i + nstates*j] == 1){
                        bpr[i + nstates*j] = -bpr[i + nstates*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < nstates; j++){
                if (csign[i + noutputs*j] == 1){
                        cpr[i + noutputs*j] = -cpr[i + noutputs*j];
                }
        }
}

for (i = 0; i < noutputs; i++){
        for (j = 0; j < ninputs; j++){
                if (dsign[i + noutputs*j] == 1){
                        dpr[i + noutputs*j] = -dpr[i + noutputs*j];
```

```
                }
        }
}

for (i = 0; i < nstates; i++){
        if (x0sign[i] == 1){
                xopr[i] = -xopr[i];
        }
}

} /* END STATE_SPACE FLOATING-POINT FUNCTION */
/* ###################################################################
################################################################### */

/* ###################################################################
################################################################### BEGIN MPU_INPUTS FLOATING-POINT FUNCTION ###############
###################################################################
################################################################### */

/* MPUINPUTS() */

void mpuinputs(u)
double *u;

{

/* FUNCTIONS */
void extractexpo();
double number;
int E;

void number2int();
Integer8 integral;
double fractional;

void convert();
double NUMBER;

void roundcheck();

/* LOCAL VARIABLES */
        int i,j;

/* NOW CONVERTING THE INPUTS (U) TO THE WORDLENGTH OF THE
MICROPROCESSOR (MPU) BY ADJUSTING THE FORMAT */

/* INITIALIZATION OF '*SIGN[]' */

for (i = 0; i < ninputs; i++){
        uinsign[i] = 0;
```

```
}

/* CHECK TO SEE IF THE INDIVIDUAL ELEMENTS OF A VECTOR ARE NEGATIVE, AND
IF AN ELEMENT IS, TEMPORARILY CHANGE IT TO POSITIVE */

for (i = 0; i < ninputs; i++){
        if (u[i] < 0.0){
                uinsign[i] = 1;
                u[i] = -u[i];
        }
}

/* CALCULATE THE FLOATING-POINT REPRESENTATION OF THE NUMBER USING BOTH THE
EXPONENT AND MANTISSA WORDLENGTHS OF THE MPU */

/* CONVERT 'U' VECTOR: */

for (i = 0; i < ninputs; i++){
        j = 0;
        matrixstar = 6;
        signalflag = 0;
        number = u[i];
        extractexpo(number,&E);
        number2int(number,&integral,&fractional);
        convert(number,E,integral,fractional,&NUMBER);
        number = NUMBER;
        roundcheck(i,j,&number,&signalflag);
        if (signalflag == 1) {
                printf("\nThe U[%d][%d] element is: %s \n",i,j,"NaN"); }
        if (signalflag == 2) {
                printf("\nThe U[%d][%d] element is: %s \n",i,j,"Inf"); }
        u[i] = number;
} /* THE OUTER 'i' INDICE LOOP */

/* IF THE ELEMENT WAS NEGATIVE, CHANGE IT BACK */

for (i = 0; i < ninputs; i++){
        if (uinsign[i] == 1){
                u[i] = -u[i];
        }
}

} /* END MPU_INPUTS FLOATING-POINT FUNCTION */
/* ######################################################################
###################################################################### */

/* ######################################################################
######################################################################
################### BEGIN QUANT FLOATING-POINT FUNCTION ################
######################################################################
###################################################################### */
```

```
/* QUANT() */

void quant(type,i,temp)
int type; /* Determines either 'dstates' or 'outputs' function call */
int i;
double *temp;

{

/* FUNCTIONS */
void extractexpo();
double number;
int E;

void number2int();
Integer8 integral;
double fractional;

void convert();
double NUMBER;

void roundcheck();

/* LOCAL VARIABLES */
        int j;

/* NOW QUANTIZING THE NUMBER (*TEMP) TO THE WORDLENGTH OF THE
MICROPROCESSOR (MPU) */

/* INITIALIZATION OF '*SIGN[]' */
        quantsign[0] = 0;

/* CHECK TO SEE IF THE NUMBER IS NEGATIVE, AND IF IT IS, TEMPORARILY
CHANGE IT TO POSITIVE */

if ((*temp) < 0.0){
        quantsign[0] = 1;
        (*temp) = -(*temp);
}

/* CALCULATE THE FLOATING-POINT REPRESENTATION OF THE NUMBER USING BOTH THE
EXPONENT AND MANTISSA WORDLENGTHS OF THE MPU */
j = 0;
/* QUANTIZE THE NUMBER: */
matrixstar = 7;
signalflag = 0;
number = (*temp);
extractexpo(number,&E);
number2int(number,&integral,&fractional);
convert(number,E,integral,fractional,&NUMBER);
number = NUMBER;
```

```
roundcheck(i,j,&number,&signalflag);
if (type == 1) {
        if (signalflag == 1) {
        printf("\nThe ds[%d][%d] element is: %s \n",i,j,"NaN"); }
        if (signalflag == 2) {
        printf("\nThe ds[%d][%d] element is: %s \n",i,j,"Inf"); }
}
if (type == 2) {
        if (signalflag == 1) {
        printf("\nThe y[%d][%d] element is: %s \n",i,j,"NaN"); }
        if (signalflag == 2) {
        printf("\nThe y[%d][%d] element is: %s \n",i,j,"Inf"); }
}

(*temp) = number;

/* IF THE NUMBER WAS NEGATIVE, CHANGE IT BACK */

if (quantsign[0] == 1) {
        (*temp) = -(*temp);
}

} /* END QUANT FLOATING-POINT FUNCTION */
/* ##########################################################################
########################################################################## */

/* **************************************************************************
**********************BEGIN 'EXTRACTEXPO' FUNCTION*************************
************************************************************************** */

void extractexpo(number,E)

/* INPUT ARGUMENTS */
double number;

/* OUTPUT ARGUMENTS */
int *E;

{

/* LOCAL VARIABLES */
        static double lower,upper;
        static int i,xx,yy,flag1;

/* EXTRACT EXPONENT (E) */
*E = 0;
flag1 = 0;

if (number < nearzero) {
        *E = 0;
        return;
```

```
}
if (number >= infinityy) {
        *E = bound;
        return;
}

if (number >= 1.0) {
        for (i = biaslimit; i <= bound; i++){ /* BEGIN FOR-LOOP */
                xx = i - biaslimit;
                lower = pow(2.0,(double) xx);
                yy = (i + 1) - biaslimit;
                upper = pow(2.0,(double) yy);

                if ((number >= lower) && (number < upper)){
                        *E = i;
                        flag1 = 1;
                }
                if (flag1 == 1){
                        break;
                }
        } /* END FOR-LOOP */
} /* END IF-LOOP */

if ((number >= nearzero) && (number < 1.0)) {
        for (i = (biaslimit - 1); i >= 0; i--){ /* BEGIN FOR-LOOP */
                xx = i - biaslimit;
                lower = pow(2.0,(double) xx);
                yy = (i + 1) - biaslimit;
                upper = pow(2.0,(double) yy);

                if ((number >= lower) && (number < upper)){
                        *E = i;
                        flag1 = 1;
                }
                if (flag1 == 1){
                        break;
                }
        } /* END FOR-LOOP */
} /* END IF-LOOP */

} /* END 'EXTRACTEXPO' FUNCTION */
/* ########################################################################
######################################################################## */

/* ************************************************************************
**********************BEGIN 'NUMBER2INT' FUNCTION**************************
************************************************************************ */

void number2int(number,integral,fractional)

/* INPUT ARGUMENTS */
```

```
double number;

/* OUTPUT ARGUMENTS */
Integer8 *integral;
double *fractional;

{

/* FUNCTION */
void calcshiftpt();
double shiftpoint;

/* LOCAL VARIABLES */
        static double temp,templower,integraltemp;

/* THIS FUNCTION ASSIGNS THE INPUTED 'DOUBLE' NUMBER TO AN 8-BYTE
'INTEGER' NUMBER; THIS 'INTEGER' NUMBER IS STORED IN TWO SEPARATE
'UNSIGNED LONG INT' VARIABLES CALLED 'UPPER' AND 'LOWER'. */

shiftpoint = 1.0;
calcshiftpt(number,&shiftpoint);
if (number < shiftpoint6) {
        temp = (number / shiftpoint);
        (*integral).upper = (unsigned long int) temp;
        templower = temp - ((double) (*integral).upper);
        if (number < shiftpoint1) {
                integraltemp = 0.0;
                (*integral).lower = 0; }
        else {
                integraltemp = (templower * shiftpoint);
                temp = (templower * shiftpoint1);
                (*integral).lower = (unsigned long int) (temp);
        }
        temp = ((double) (*integral).upper) * shiftpoint;
        temp = (temp + integraltemp);
        *fractional = (double) (number - temp); }
else {
        (*integral).upper = (unsigned long int) (shiftpoint1 - 1.0);
        (*integral).lower = (*integral).upper;
        *fractional = 0.0;
}

} /* END 'NUMBER2INT' FUNCTION */
/* ##########################################################################
########################################################################## */

/* **************************************************************************
**********************BEGIN 'CALCSHIFTPT' FUNCTION************************
************************************************************************** */

void calcshiftpt(number,shiftpoint)
```

```
/* INPUT ARGUMENTS */
double number;

/* OUTPUT ARGUMENTS */
double *shiftpoint;

{

/* DETERMINE APPROPRIATE 'SHIFTPOINT' FOR THE 'NUMBER2INT' FUNCTION */
/* THESE ARE 'SCALING' FACTORS FOR MACHINE DEPENDENT INTEGER WORDLENGTH */

if ((number >= shiftpoint5) && (number < shiftpoint6)) {
        *shiftpoint = shiftpoint5; }
else if ((number >= shiftpoint4) && (number < shiftpoint5)) {
        *shiftpoint = shiftpoint4; }
else if ((number >= shiftpoint3) && (number < shiftpoint4)) {
        *shiftpoint = shiftpoint3; }
else if ((number >= shiftpoint2) && (number < shiftpoint3)) {
        *shiftpoint = shiftpoint2; }
else if ((number >= shiftpoint1) && (number < shiftpoint2)) {
        *shiftpoint = shiftpoint1; }
else if ((number >= 0.0) && (number < shiftpoint1)) {
        *shiftpoint = 1.0; } /* NO SHIFT (SCALING) NECESSARY */
else {
        *shiftpoint = shiftpoint6; /* THE 'NUMBER' IS TOO LARGE. */
}

} /* END 'CALCSHIFTPT' FUNCTION */
/* ##########################################################################
########################################################################## */

/* ************************************************************************
**********************BEGIN 'CONVERT' FUNCTION**************************
************************************************************************ */

void convert(number,E,integral,fractional,NUMBER)

/* INPUT ARGUMENTS */
double number;
int E;
Integer8 integral;
double fractional;

/* OUTPUT ARGUMENTS */
double *NUMBER;

{

/* FUNCTIONS */
void calcfraction();
double fractionalx;
```

```
Integer8 mantfrac;
int counta;

void extractbit();
Integer8 mant;
char mantissa[MAXSTRING];
int f;

void bit2deci();
double quantnumber;

/* LOCAL VARIABLES */
        Integer8 integralx,mantintegral,integralxx;
        static unsigned long int mask1,mask2,tempint,bit1;
        Integer8 shiftmantfrac;
        static int i,countupper,countlower;
        static int bitstakenupper,bitstakenlower,bitstaken;

/* RE-ASSIGN THE INTEGER AND FRACTION PARTS TO NEW VARIABLES */
integralx.upper = integral.upper;
integralx.lower = integral.lower;
fractionalx = fractional;

mask1 = my; /* HAS A '1' IN THE HIGH ORDER BIT */
mask2 = 1; /* USED FOR SHIFTING OPERATIONS FROM '*.LOWER' TO '*.UPPER' */
countupper = 0;
countlower = 0;

/* CONVERT DECIMAL-INTEGER-PART AND DECIMAL-FRACTIONAL-PART SEPARATELY,
TO BINARY EQUIVALENT */

if (integralx.upper == 0){
        calcfraction(fractionalx,&mantfrac,&counta);
        mant.upper = mantfrac.upper;
        mant.lower = mantfrac.lower;
        /* EXTRACT BIT-BY-BIT FROM MANTISSA AND ASSIGN TO A 'STRING' */
        extractbit(mant,mantissa,&f); }
else {
        while (integralx.upper < my){
                integralx.upper <<= 1;
                countupper = countupper + 1;
                tempint = mask1 & integralx.lower;
                tempint >>= (mz - 1);
                tempint = tempint & mask2;
                integralx.upper = integralx.upper | tempint;
                integralx.lower <<= 1;
        }
        mantintegral.upper = integralx.upper;
        mantintegral.lower = integralx.lower;
        integralxx.lower = integralx.lower;
```

```
for (i = 0; i < mz; i++) {
        bit1 = integralxx.lower & mask2;
        if (bit1 == 0) {
                countlower = countlower + 1;
                integralxx.lower >>= 1; }
        else {
                break;
        }
} /* END FOR-LOOP */

if (number >= ((shiftpoint2 - 1.0))) {
        bitstaken = (2*mz); }
else if (number >= ((shiftpoint1 - 1.0))) {
        bitstakenupper = mz;
        bitstakenlower = (mz - countlower);
        bitstaken = (bitstakenupper + bitstakenlower); }
else {
        bitstaken = (mz - countupper);
} /* END IF */

if (bitstaken >= (mantx + 1)) {
        mantintegral.upper = integralx.upper;
        mantintegral.lower = integralx.lower;
        mantfrac.upper = 0;
        mantfrac.lower = 0;
        counta = 0; }
else {
        calcfraction(fractionalx,&mantfrac,&counta);
} /* END IF */

/* SHIFT FRACTIONAL-PART TO RIGHT, WILL APPEND TO INTEGRAL-PART */
for (i = 0; i < (bitstaken + counta); i++) {
        tempint = mask2 & mantfrac.upper;
        tempint <<= (mz - 1);
        mantfrac.lower >>= 1;
        mantfrac.lower = mantfrac.lower | tempint;
        mantfrac.upper >>= 1;
} /* END FOR-LOOP */

shiftmantfrac.upper = mantfrac.upper;
shiftmantfrac.lower = mantfrac.lower;

if (number >= ((shiftpoint1 - 1.0))) {
        mant.upper = mantintegral.upper;
        mant.lower = mantintegral.lower | shiftmantfrac.upper; }
else {
        mant.upper = mantintegral.upper | shiftmantfrac.upper;
        mant.lower = shiftmantfrac.lower;
}
/* EXTRACT BIT-BY-BIT FROM MANTISSA AND ASSIGN TO A STRING */
extractbit(mant,mantissa,&f);
```

```
} /* END 'ELSE' CONDITION */

bit2deci(E,f,mantissa,&quantnumber);

*NUMBER = quantnumber;

} /* END 'CONVERT' FUNCTION */
/* ##########################################################################
########################################################################## */

/* ************************************************************************
**********************BEGIN 'CALCFRACTION' FUNCTION************************
************************************************************************ */

void calcfraction(fractionalx,mantfrac,counta)

/* INPUT ARGUMENTS */
double fractionalx;

/* OUTPUT ARGUMENTS */
Integer8 *mantfrac;
int *counta;

{

/* FUNCTION */
void frac2int();

/* LOCAL VARIABLES */
        static unsigned long int mask1,mask2,tempint;

mask1 = my;     /* HAS A '1' IN THE HIGH ORDER BIT */
mask2 = 1;      /* USED FOR SHIFTING OPERATIONS FROM '*.LOWER' TO '*.UPPER' */

/* MAKE 'FRACTION' INTO A MACHINE DEPENDENT BIT-INTEGER REPRESENTATION */
(*mantfrac).upper = 0;
(*mantfrac).lower = 0;
*counta = 0;

if (fractionalx < invshiftpt5) {
        frac2int(fractionalx,shiftpoint6,mantfrac); }
else if (fractionalx < invshiftpt4) {
        frac2int(fractionalx,shiftpoint5,mantfrac); }
else if (fractionalx < invshiftpt3) {
        frac2int(fractionalx,shiftpoint4,mantfrac); }
else if (fractionalx < invshiftpt2) {
        frac2int(fractionalx,shiftpoint3,mantfrac); }
else if (fractionalx < invshiftpt1) {
        frac2int(fractionalx,shiftpoint2,mantfrac); }
else {
        frac2int(fractionalx,shiftpoint1,mantfrac);
```

```
}

if ((*mantfrac).upper != 0) {
        while ((*mantfrac).upper < my) {
                (*mantfrac).upper <<= 1;
                *counta = *counta + 1;
                tempint = mask1 & (*mantfrac).lower;
                tempint >>= (mz - 1);
                tempint = tempint & mask2;
                (*mantfrac).upper = (*mantfrac).upper | tempint;
                (*mantfrac).lower <<= 1;
        } /* END 'WHILE' LOOP */ } /* END 'IF' LOOP */
else {
        (*mantfrac).upper = 0;
        (*mantfrac).lower = 0;
        *counta = 0;
}

} /* END 'CALCFRACTION' FUNCTION */
/* ########################################################################
######################################################################## */

/* ************************************************************************
**********************BEGIN 'FRAC2INT' FUNCTION**************************
************************************************************************ */

void frac2int(fractionalx,shiftpointx,mantfrac)

/* INPUT ARGUMENTS */
double fractionalx,shiftpointx;

/* OUTPUT ARGUMENTS */
Integer8 *mantfrac;

{

/* LOCAL VARIABLES */
        static double temp,templower,shiftpoint;

/* THIS FUNCTION SCALES THE 'FRACTIONAL'-PART OF A NUMBER TO AN 8-BYTE
'INTEGER' NUMBER; THIS 'INTEGER' NUMBER IS STORED IN TWO SEPARATE
'UNSIGNED LONG INT' VARIABLES CALLED 'UPPER' AND 'LOWER'. */

shiftpoint = shiftpointx;
if (fractionalx >= invshiftpt5) {
        temp = (fractionalx * shiftpoint);
        (*mantfrac).upper = (unsigned long int) temp;
        templower = temp - ((double) (*mantfrac).upper);
        (*mantfrac).lower = (unsigned long int) (templower * shiftpoint); }
else {
        (*mantfrac).upper = 0;
```

```
        (*mantfrac).lower = 0;
}

} /* END 'FRAC2INT' FUNCTION */
/* ##########################################################################
########################################################################## */

/* ************************************************************************
**********************BEGIN 'EXTRACTBIT' FUNCTION**************************
************************************************************************ */

void extractbit(mant,mantissa,f)

/* INPUT ARGUMENTS */
Integer8 mant;

/* OUTPUT ARGUMENTS */
char mantissa[MAXSTRING];
int *f;

{

/* LOCAL VARIABLES */
        static int i;
        static unsigned long int mask,mask2,tempint;

/* EXTRACT BITS FROM BIT-INTEGER AND ASSIGN TO A CHARACTER STRING */
mask = my;      /* MASK HAS A '1' IN THE HIGH ORDER BIT */
mask2 = 1;      /* USED FOR SHIFTING OPERATIONS FROM '*.LOWER' TO '*.UPPER' */
*f = 0;

for (i = 0; i <= mantx; i++){
   tempint = mant.upper & mask;
   tempint >>= (mz - 1);
   if (tempint){
        mantissa[i] = '1'; }
   else {
        mantissa[i] = '0';
   }
   *f = *f + (mantissa[i] - '0');
   mant.upper <<= 1;
   tempint = mask & mant.lower;
   tempint >>= (mz-1);
   tempint = tempint & mask2;
   mant.upper = mant.upper | tempint;
   mant.lower <<= 1;
}

} /* END 'EXTRACTBIT' FUNCTION */
/* ##########################################################################
########################################################################## */
```

```
/* ************************************************************************
**********************BEGIN 'BIT2DECI' FUNCTION**************************
************************************************************************ */

void bit2deci(E,f,mantissa,quantnumber)

/* INPUT ARGUMENTS */
int E,f;
char mantissa[MAXSTRING];

/* OUTPUT ARGUMENTS */
double *quantnumber;

{

/* LOCAL VARIABLES */
        static int i;
        static double pp,power,factor,temp;
        static int shift;

/* QUANTIZE NUMBER INTO FLOATING-POINT FORMAT AS SPECIFIED BY THE
USER */
*quantnumber = 0.0;
temp = 0.0;
factor = 0.0;
shift = E - biaslimit;

if ((E == 0) && (f != 0)) { /* FOR 'VERY_SMALL' NUMBERS */
        for (i = 0; i < mantx; i++) {
                pp = (double) (shift - i);
                power = pow(2.0,pp);
                factor = ((double) ((mantissa[i] - '0'))) * power;
                temp = temp + factor;
        } /* END 'FOR' LOOP */ } /* END 'IF' LOOP */
else {
        for (i = 0; i <= mantx; i++) {
                pp = (double) (shift - i);
                power = pow(2.0,pp);
                factor = ((double) ((mantissa[i] - '0'))) * power;
                temp = temp + factor;
        }
} /* END 'ELSE' CONDITION */

*quantnumber = temp;

} /* END 'BIT2DECI' FUNCTION */
/* ########################################################################
######################################################################## */

/* ************************************************************************
**********************BEGIN 'ROUNDCHECK' FUNCTION************************
```

```
********************************************************************** */

void roundcheck(i,j,number,signalflag)

/* INPUT ARGUMENTS */
int i,j;

/* OUTPUT ARGUMENTS */
double *number;
int *signalflag;

{

/* LOCAL VARIABLES */
        int ii = i;
        int jj = j;
        int starsign[1];

/* CHECK FOR THE ROUNDING METHOD CHOSEN, THEN PERFORM APPROPRIATE
OPERATIONS */

if (matrixstar == 1){
        starsign[0] = asign[ii + nstates*jj]; }
else if (matrixstar == 2){
        starsign[0] = bsign[ii + nstates*jj]; }
else if (matrixstar == 3){
        starsign[0] = csign[ii + noutputs*jj]; }
else if (matrixstar == 4){
        starsign[0] = dsign[ii + noutputs*jj]; }
else if (matrixstar == 5){
        starsign[0] = x0sign[ii + nstates*jj]; }
else if (matrixstar == 6){
        starsign[0] = uinsign[ii + ninputs*jj]; }
else if (matrixstar == 7){
        starsign[0] = quantsign[0]; }

if (roundx == 1){ /* ROUND TO +Inf */
        if (((*number) > LFN) && (starsign[0] == 1)){
                *number = MNFN; }
        else {
                *number = *number;
        }
}
else if (roundx == 2){ /* ROUND TO Zero */
        if ((*number) > LFN){
                *number = LFN; }
        else {
                *number = *number;
        }
}
else if (roundx == 3){ /* ROUND TO -Inf */
```

```
        if (((*number) > LFN) && (starsign[0] == 0)){
                *number = LFN; }
        else {
                *number = *number;
        }
}
else {
        *number = *number; /* No 'other' rounding method
chosen, therefore, return the default mode (e.g., R2Nearest),
which returns infinitely precise value as determined by
the given destination format. */
}

if ((*number) > infinityy) {
        *signalflag = 1; } /* 'NaN' */
if ((*number) == infinityy) {
        *signalflag = 2; } /* 'Inf' */

} /* END 'ROUNDCHECK' FUNCTION */
/* ########################################################################
######################################################################## */

/* Function to return any initial conditions on the states  */
static
#ifdef __STDC__
void init_conditions(double *x0)
#else
void init_conditions(x0)
double *x0;
#endif /* __STDC__ */
{
        int i;
        double *pr;

        pr = (double *)mxGetPr(X0);
        for (i=mxGetM(X0); i; i--) {
                *x0++ = *pr++;
        }
}

/* Function to return derivatives */
static
#ifdef __STDC__
void derivatives(double t, double *x, double *u, double *dx)
#else
void derivatives(t,x,u,dx)
double t, *x, *u;       /* Input variables */
double *dx;             /* Output variable */
#endif /* __STDC__ */
{
        return;    /* No continuous states */
```

```
}

/* Function to perform discrete state update */
static
#ifdef __STDC__
void dstates(double t, double *x, double *u, double *ds)
#else
void dstates(t,x,u,ds)
double t, *x, *u;    /* Input variables */
double *ds;          /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void mpuinputs();
void quant();

/* LOCAL VARIABLES */
        int i,j;
        double accum;
        double *apr, *bpr;

        apr = (double *)mxGetPr(A);
        bpr = (double *)mxGetPr(B);

        /* Matrix Multiply  x(n+1) = Ax(n) + Bu(n) */

        /* Update discrete states on a sample hit */

if (t == sample_hit[0]) {

        mpuinputs(u); /* Adjusts inputs to MPU */

        for (i = 0; i < NDSTATES; i++) {
                accum = 0.0;
        /* Ax(n) */
                for (j = 0; j < NDSTATES; j++) {
                        accum = accum + apr[i + NDSTATES*j] * x[j];
                }
        /* Bu(n) */
                for (j = 0; j < NINPUTS; j++) {
                        accum = accum + bpr[i + NDSTATES*j] * u[j];
                }
                quant(1,i,&accum); /* QUANTIZE NUMBER */
                ds[i] = accum;
        }
} /* END IF-LOOP */

} /* END DSTATES FUNCTION */

/* Function to return outputs */
```

```
static
#ifdef __STDC__
void outputs(double t, double *x, double *u, double *y)
#else
void outputs(t,x,u,y)
double t, *x, *u;       /* Input variables */
double *y;              /* Output variable */
#endif /* __STDC__ */
{

/* FUNCTIONS */
void mpuinputs();
void quant();

/* LOCAL VARIABLES */
        int i,j;
        double accum;
        double *cpr, *dpr;

        cpr = (double *)mxGetPr(C);
        dpr = (double *)mxGetPr(D);

/* Matrix Multiply  y(n) = Cx(n) + Du(n) */

        mpuinputs(u); /* Adjusts inputs to MPU */

        for (i = 0; i < NOUTPUTS; i++) {
                accum = 0.0;
        /* Cx(n) */
                for (j = 0; j < NDSTATES; j++) {
                        accum = accum + cpr[i + NOUTPUTS*j] * x[j];
                }
        /* Du(n) */
                for (j = 0; j < NINPUTS; j++) {
                        accum = accum + dpr[i + NOUTPUTS*j] * u[j];
                }
                quant(2,i,&accum); /* QUANTIZE NUMBER */
                accum = accum * pbda;
                y[i] = accum;
        }

} /* END OUTPUTS FUNCTION */

/* Function to return singularities */
static
#ifdef __STDC__
void singularity(double t, double *x, double *u, double *sing)
#else
void singularity(t,x,u,sing)
double t, *x, *u; /* Input variables */
double *sing;             /* Output variable */
```

```
#endif /* __STDC__ */
{
        return; /* Default is to have no singularity */
}

#include "simulin2.h" /* Include file to perform MEX glue */

/* END 'MPUFL.C' PROGRAM
##########################################################################
##########################################################################
*/
```

A.5. "SIMSUP.H"

```
/* Filename: SIMSUP.H */
/* Date: May 4, 1994 */
/* Header file for discrete-time hits. */
/* This code was provided by The MathWorks, Inc. */

#include <math.h>    /* needed for floor() prototyping */

#ifdef NSAMPLE

static
#ifdef __STDC__
double next_hit_ftn(double ts, double offset, double t)
#else
double next_hit_ftn(ts, offset, t)
double ts, offset, t;
#endif /* __STDC__ */
{
        double nosamples, tnext;

        nosamples = (t-offset)/ts;
        tnext =  offset  + (1 + floor(nosamples + 1e-13*(1 + nosamples))) * ts;
        return(tnext);
}

/*--------------------------------------------------------------------*/
/* code_tnext_ftn    - works out the next sample hits for             */
/*                     sampled data blocks.                           */
/*--------------------------------------------------------------------*/
static
#ifdef __STDC__
void code_tnext_ftn(double t,double *OffsetTimes,double *SampleTimes,double *SampleHit,double
*NextHit,double *tnext)
#else
code_tnext_ftn(t, OffsetTimes, SampleTimes, SampleHit, NextHit, tnext)
double t, *OffsetTimes, *SampleTimes; /* Input variables */
double *SampleHit, *NextHit, *tnext;  /* Output variables */
#endif /* __STDC__ */
{

        int i;
        static double Neareps = 1e-13;
        double nearest_next_hit, b_next;

/*  Loop through sample times working out when the next sample hit is for
all of the sample times store this in the vector 'next_hit' and return
the nearest one to the present time, t. */

for (i=0; i<NSAMPLE; i++) {
```

```
/* On the first iteration the sample times are set to -1e30 */
        if (SampleHit[i]  == -1e30) {
/* First update */
                SampleHit[i] = next_hit_ftn(SampleTimes[i], OffsetTimes[i], t - SampleTimes[i]);
                b_next = next_hit_ftn(SampleTimes[i], OffsetTimes[i], t);

        }
        else {

        /* Next scheduled sample hit is stored in next_hit */
        b_next = NextHit[i];
                SampleHit[i] = b_next;

        /* If the next hit has been overtaken then calculate another one */
                if (b_next <= t)
                b_next = next_hit_ftn(SampleTimes[i], OffsetTimes[i], t);
        }

        /* Calculate nearest hit to the present time */

        /* On the first iteration set this to be the first one */
        if (i == 0)
                nearest_next_hit = b_next;

        else if (fabs(nearest_next_hit - b_next) < Neareps) {

        /* Attempt to correct sample clash problem for clashing sample hits
        i.e. set near sample hits to the same time */
                        b_next = nearest_next_hit;

        }
        else {
                if (b_next < nearest_next_hit)
                        nearest_next_hit = b_next;
        }

NextHit[i] = b_next;

}

tnext[0] = nearest_next_hit;

}

#endif

/* END 'SIMSUP.H' HEADER-FILE */
```

A.6. "SIMULIN2.H"

```
/* Filename: SIMULIN2.H */
/* Date: May 4, 1994 */
/* Header file for interfacing C source code (e.g., Computational Routine) with that of MATLAB /
SIMULINK code (e.g., Gateway Routine). */

/*  SIMULIN2.H - Include file for making MEX-file systems and blocks
 *              with variables passed down from the workspace.
 *
 *  This file should be included at the end of a MEX-file system.
 *  It performs an interface to the MEX-file mechanism which allows
 *  blocks and systems to be entered only as their corresponding
 *  mathematical functions.
 *
 *  This template performs all checks for sizes of parameters.
 *  With this include file you can pass down parameters from the
 *  workspace to the system. (See for example stspace.c ).
 *
 *
 *      Syntax  of MEX-file S-function:
 *
 *     [sys, x0] = filename(t,x,u,flag)
 *
 *  Andrew Grace  Oct 11, 1990
 *  Copyright (c) 1990-93 The Mathworks, Inc.
 *  All Rights Reserved
 */

/* Discrete-time Code */

#include "simsup.h" /* Header file for discrete-time hits. */

/* ************** */

/* Input arguments */
#define T_IN            prhs[0]
#define X_IN            prhs[1]
#define U_IN            prhs[2]
#define FLAG_IN         prhs[3]

#ifdef NO_COEFF
/* NCOEFFS now replaces the number of external parameters in simulin2.h */
#define NCOEFFS NO_COEFF
#endif

/* Output Arguments */

#define SYS_OUT         plhs[0]
#define X0_OUT          plhs[1]
```

```
#ifdef __STDC__
void mexFunction(int nlhs, Matrix *plhs[], int nrhs, Matrix *prhs[])
#else
mexFunction(nlhs, plhs, nrhs, prhs)
int nlhs, nrhs; Matrix *plhs[], *prhs[];
#endif /* __STDC__ */
{
        int flag, i;
        double *pr;

        /* Check validity of arguments */

        if (nlhs > 2 || nrhs != 4 + NCOEFFS) {
                mexErrMsgTxt("System MEX file called with wrong number of arguments. ");
        }
        if (mxGetM(FLAG_IN) != 1 || mxGetN(FLAG_IN) != 1) {
                mexErrMsgTxt("FLAG must be a scalar in System MEX file.");
        }
        flag = (int) mxGetPr(FLAG_IN)[0];

/* Load coefficients into global matrix structure */
        for (i=0; i<NCOEFFS; i++) {
                Coeffs[i] = prhs[4+i];
        }

        /* Special case FLAG=0 return sizes and initial conditions */
        if (flag==0) {
                initialize();
                SYS_OUT = mxCreateFull(6, 1, REAL);
                pr = (double *)mxGetPr(SYS_OUT);
                pr[0] = NSTATES;
                pr[1] = NDSTATES;
                pr[2] = NOUTPUTS;
                pr[3] = NINPUTS;
                pr[4] = NSING;
                pr[5] = NEEDINPUTS;
                if (nlhs >1) {
                        X0_OUT = mxCreateFull(NSTATES + NDSTATES, 1, REAL);
                        init_conditions(mxGetPr(X0_OUT));
                }
                return;
        }
        if (nlhs >= 0) {
                if (nlhs > 1) {
                    mexErrMsgTxt("System MEX file called with wrong number of output arguments.");
                }
                /* Error checking - ommitted for speed
                   - but may cuase segmentation faults if
                   called with the wrong sizes of arguments */
                if (mxGetM(X_IN)*mxGetN(X_IN) != NSTATES + NDSTATES) {
                        mexErrMsgTxt("State vector X wrong size in System MEX file.");
```

```
        }
        if (mxGetM(U_IN)*mxGetN(U_IN) != NINPUTS) {
            mexErrMsgTxt("Input vector U wrong size in System MEX file.");
        }
        if (mxGetM(T_IN) != 1 || mxGetN(T_IN) != 1) {
            mexErrMsgTxt("T must be a scalar in System MEX file.");
        }
    }
    switch (flag) {
    case 1:
    case -1:
        if (nlhs >= 0)
            SYS_OUT = mxCreateFull(NSTATES, 1, REAL);
        derivatives(*mxGetPr(T_IN), mxGetPr(X_IN), mxGetPr(U_IN), mxGetPr(SYS_OUT));
        break;
    case 2:
    case -2:
        if (nlhs >= 0)
            SYS_OUT = mxCreateFull(NDSTATES, 1, REAL);
        dstates(*mxGetPr(T_IN), mxGetPr(X_IN), mxGetPr(U_IN), mxGetPr(SYS_OUT));
        break;
    case 3:
        if (nlhs >= 0)
            SYS_OUT = mxCreateFull(NOUTPUTS, 1, REAL);
        outputs(*mxGetPr(T_IN), mxGetPr(X_IN), mxGetPr(U_IN), mxGetPr(SYS_OUT));
        break;

    case 4:
        if (nlhs >= 0) {
            SYS_OUT = mxCreateFull(1, 1, REAL);
            *mxGetPr(SYS_OUT) = 1e30; /* Defaults to very large number */
        }
        *mxGetPr(SYS_OUT) = 1e30; /* Defaults to very large number */
        code_tnext_ftn(*mxGetPr(T_IN), offset_times, sample_times, sample_hit, next_hit,
                mxGetPr(SYS_OUT));
        break;

    case 5:
        if (nlhs >= 0)
            SYS_OUT = mxCreateFull(NSING, 1, REAL);
        singularity(*mxGetPr(T_IN), mxGetPr(X_IN), mxGetPr(U_IN), mxGetPr(SYS_OUT));
        break;

    default:
        mexErrMsgTxt("Not a valid flag number for MEX System.");
        break;
    }
}


/* END 'SIMULIN2.H' HEADER-FILE */
```

A.7. Compilation Command for C MEX-files

The following UNIX command allows for the compilation of a given "C" source file (e.g., "mpufl.c"), and therefore creates the necessary "*.CMEX4 (or equivalent)" executable file. This executable file's filename (e.g., "mpufl") is to be used in a given S-Function Block.

Example:

The UNIX command is:

```
/usr/local/matlab40/bin/cmex  -I/usr/local/matlab40/extern/include -lm mpufl.c
```

Note: '-lm' links the math library.