545211

Creare

ALGORITHMS AND OBJECT-ORIENTED SOFTWARE FOR DISTRIBUTED PHYSICS-BASED MODELING

FISCAL YEAR REPORT
PERIOD: June 19, 2001 – September 19, 2001

CONTRACT NAS1-01085
CREARE PROJECT 8728

Marc A. Kenton
Principal Investigator

Jerry Bieszczad
Engineer

Creare Incorporated
Hanover, NH 03755

September 28, 2001
8728

1.0 INTRODUCTION

The project seeks to develop methods to more efficiently simulate aerospace vehicles. The goals are to reduce model development time, increase accuracy (e.g., by allowing the integration of multi-disciplinary models), facilitate collaboration by geographically-distributed groups of engineers, support uncertainty analysis and optimization, reduce hardware costs, and increase execution speeds. These problems are the subject of considerable contemporary research (e.g., Biedron et al. 1999; Heath and Dick, 2000).

All of these goals can be addressed by allowing complicated systems with intricate inter-connections and strong internal feedback mechanisms to be represented by semi-independent subsystems. Each subsystem is modeled in detail by separate (possibly geographically-distributed) groups of engineers and then each subsystem model is executed on a separate processor of a computer network. The development and implementation of such an approach is the subject of this project. The deliverables from the project will be mathematical techniques to allow the linking of the separate subsystem models, an object-oriented methodology for developing the models, and a software library that facilitates the process.

The potential pay-off of this technology is very large, in terms of both hardware and software development costs. Hardware costs are reduced by allowing networked processors to be efficiently used for detailed simulation of complicated aerospace systems. Software costs are reduced by a divide-and-conquer approach that directly attacks complexity, supports interdisciplinary modeling, promotes object-oriented techniques, and allows efficient re-use of component or subsystem models developed on earlier projects.

The primary initial focus of the project is on lumped parameter simulation models (not, for example, single discipline finite element models). If the overall model is to be divided into separate subsystem models, these sub-models must be mathematically integrated in a way that provides computationally-stable results in the face of strong feedbacks and complicated interactions between the subsystems. Stable integration is accomplished by automatically generating simplified versions of each subsystem model from the detailed model created by its analysts. These simplified models serve on the other processors of the network as an accurate stand-in for the associated detailed model for short periods of time. Thus, if we break an entire system into N subsystems, then each of N processors in a network could solve one detailed subsystem model and N-1 simplified models for the other subsystems. The latter provide the required feedback necessary for the stable integration of the detailed subsystem model. The simplified models are kept accurate by periodically updating them to account for changes in the nominal operating state.

The project commenced on June 19, 2001. The remainder of this report documents the work performed during FY2001.

2.0 DEFINITION OF OBJECT-ORIENTED Architecture (Task 2.1)

2.1 OVERALL DESIGN AND COMMUNICATIONS

Figure 1 shows the implementation of an aerospace vehicle simulation. In this particular example, the entire vehicle model consists of three subsystems (we expect typical cases to involve many more subsystems than used in this example). A detailed model of each of these subsystems is executed on a separate computer of a network, although this too need not necessarily be the case. For example, all subsystem models could execute on a single processor, on separate processors of a multiple-processor computer, some processors could execute more than one detailed model, etc. In any case, a number of processes are created, each of which consists of a detailed subsystem model paired with simplified versions of the other subsystem models. The latter provide the proper numerical feedbacks to enable stable integration of the detailed model's equations. Each of the detailed models periodically updates its simplified counterpart and sends a description of the new simplified model to the other processes for their use.

A key design goal is to ensure that this partitioning process can be hosted on a diverse set of computer hardware and operating systems. For this reason, a survey of available communications protocols was undertaken, and the CORBA® system for communicating between the separate processes was selected. CORBA provides all the needed functionality, is actively being developed, and is supported on a wide array of vendor hardware. One key feature of this system is that it transparently accounts for differences in how data is stored internally on different computer systems.

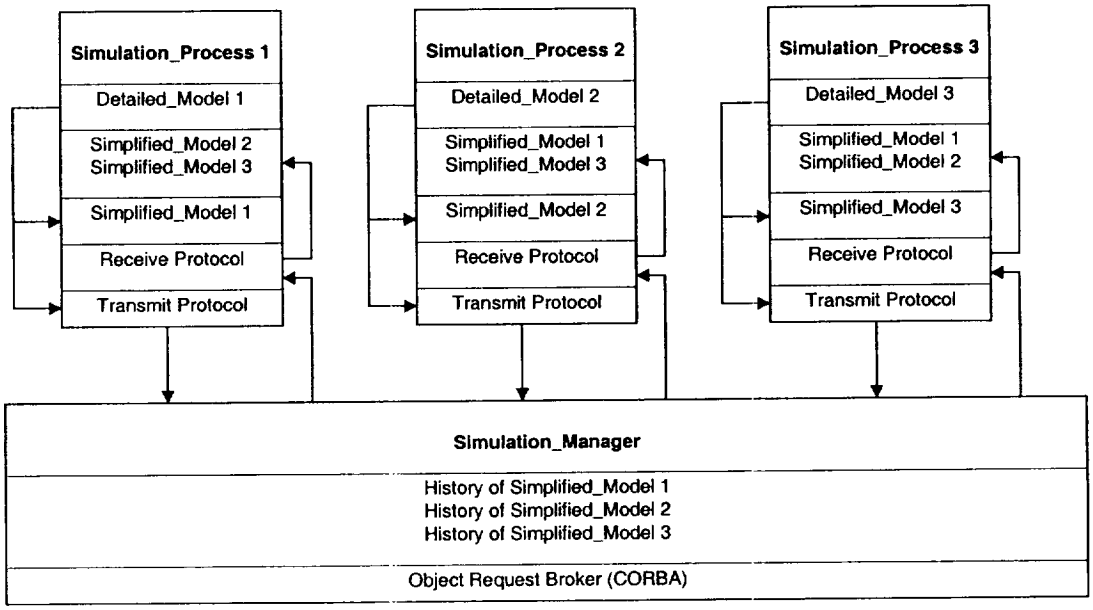

Figure 1. Architecture of an Example Simulation Utilizing 3 Processes

When a detailed model determines that its previously-generated simplified model has become obsolete and should be updated, it creates this model using the algorithms being developed in Task 2. The detailed model then sends a data structure that defines the new model via CORBA to the "Simulation Manager." The Simulation Manager then forwards the model to all the other detailed models that are executing on the remote processors of the overall system (or to detailed models executing as separate processes on the same processor, if applicable). The new model is received by a dedicated ("receiver") server that was previously spawned by each of the remote detailed models for this purpose. The receiver then pipes this data structure to the detailed model, where it replaces the now-obsolete version. While we expect that pipes are available on any candidate computer system (they are certainly available on Windows- and all Unix-based systems), other schemes for inter-process communications (e.g., shared memory) would work equally well.

A simple demonstration application was written to ensure that there were no hidden problems with this architecture. The demonstration emulated Figure 1 closely in that it involved the rapid transmission of simulated simplified models from two detailed model processes to a remote process simulating a third detailed model. This demonstration was successfully run under Windows NT, and no problems such as might be caused by collisions between simultaneously transmitted models were noted.

For transient problems, a last detail needs to be mentioned. For illustration purposes, assume that detailed model number 1 has the slowest execution time. The Simulation Manager will eventually receive information that the simplified version of model 1 has become obsolete at some simulation time t. In general, the other, faster processes will have gone past time t by the time this information is received. In this case, the Simulation Manager will send an instruction to the detailed processes that use model 1 to reinitialize at time t using a consistent set of simplified models that it supplies for this purpose. Thus, the Simulation Manager will keep a history of simplified models that it can retrieve when reinitialization is required. Since the entire problem cannot be completed until the slowest model has completed, this complication need not limit the overall execution speed. This is discussed in more detail in the next section.

2.2 Class Structure and Functionality

A key design goal of the project is to support object-oriented modeling of complex systems. Object-oriented modeling is facilitated by allowing the partitioning of a complex system into relatively independent objects. Partitioning the system in this manner supports the re-use of model objects developed on previous projects, the development of modified models through inheritance, and attacks overall model complexity by allowing the subsystem models to be developed and validated independently.

We have defined a class structure for developing models using this object-oriented approach. These modeling classes are classified into two categories. The first type, which include the classes *Detailed_Model* and *Simplified_Model*, focus primarily on localized (i.e., non-distributed) model development and simulation of a single subsystem. These localized model classes are designed to support a bottom-up modeling approach where a relatively small

number of engineers initially develop and validate a model of a particular subsystem of an aerospace vehicle. By encapsulating these subsystem models as components in a common framework, these classes facilitate subsequent large-scale simulation of the entire aerospace vehicle. The second type of modeling classes, which include the classes *Simulation_Process* and *Simulation_Manager*, implement the necessary logic and communication protocols for combining these localized subsystem models into a geographically distributed simulation of a complex process.

Before describing the details of these object-oriented modeling classes, the overall logic of a distributed multi-system simulation incorporating these classes is given. During the following description, it may be helpful to refer to the example depicted in Figure 1.

1) Before the simulation is run, an instance of *Simulation_Manager* is created on a single computer. An instance of *Simulation_Process* is created on each computer hosting a simulation of a single subsystem. The *Simulation_Process* creates an instance of *Detailed_Model* that represents the model of this subsystem. CORBA addresses are shared among the *Simulation_Manager* and the *Simulation_Processes*.

2) At simulation time zero, the *Simulation_Manager* requests that all *Simulation_Processes* initialize the state variables of their *Detailed_Models*.

3) After initialization, each *Simulation_Process* requests that its *Detailed_Model* generate the data required to instantiate a *Simplified_Model*. The *Simulation_Process* transmits this data to the *Simulation_Manager*.

4) Once the *Simulation_Manager* receives a complete set of *Simplified_Model* data structures, it transmits this set to all *Simulation_Processes*.

5) Once a *Simulation_Process* receives the complete set of *Simplified_Model* data structures, it creates the appropriate number of local instances of the *Simplified_Models*. It then begins simultaneous numerical integration of its *Detailed_Model* and the *Simplified_Models*. Note that each *Simulation_Process* is allowed to proceed at its own integration rate during the simulation.

6) After every integration time step, each *Simulation_Process* verifies the validity of its local *Simplified_Model*. It does this by comparing the values of the estimated outputs from the *Simplified_Model* with the actual outputs of its *Detailed_Model*. If a specified error criterion is exceeded, the *Simulation_Process* requests an updated *Simplified_Model* data structure from its *Detailed_Model*. The *Simulation_Process* then transmits the expiration time of the old *Simplified_Model* along with the updated *Simplified_Model* data structure to the *Simulation_Manager* and continues integrating.

7) When the *Simulation_Manager* receives the first updated *Simplified_Model*, it records the expiration time of the old *Simplified_Model* as well as the updated *Simplified_Model* data structure in its history. It transmits the expiration time of the old *Simplified_Model* to all *Simulation_Processes* who in turn record it. Two possibilities can be envisioned:

a) Case-A: The *Simulation_Process* has not yet integrated to the updated expiration time. The *Simulation_Process* now knows that when it reaches this expiration time, it must request an updated *Simplified_Model* data structure from the *Simulation_Manager*. Until then, it continues integrating with the old *Simplified_Model*, but does not take a time step beyond the expiration time of any *Simplified_Model*.

b) Case-B: The *Simulation_Process* has already integrated past the expiration time. Consequently, any results since that time are invalid. The *Simulation_Process* retreats by: (i) restoring its state variables to what they were at the expiration time, (ii) asking the *Simulation_Manager* for the complete set of *Simplified_Model*s that were valid at the expiration time, and (iii) continuing integration from that time point.

8) When the *Simulation_Manager* receives subsequent updated *Simplified_Models*, it repeats the process of transmitting its expiration time to all *Simulation_Processes* (who respond as described in step 7). In addition, any *Simplified_Model*s in the *Simulation_Manager* history which were created at time points beyond the new expiration time are now invalid and are discarded.

The object-oriented structure of the four modeling classes, *Detailed_Model*, *Simplified_Model*, *Simulation_Process*, and *Simulation_Manager*, that make implementation of this logic possible are now described.

The *Detailed_Model* class is summarized in Table 1. This class encapsulates the mathematical model of a single subsystem described by a set of ordinary differential equations and algebraic equations. These differential and algebraic equations are represented by the user-defined functions *calculate_rates_of_change( )* and *calculate_outputs( )*, respectively. The engineer also specifies the number of state variables, input variables, and output variables for the model. The *Detailed_Model* class automatically allocates sufficient computer memory for storing the values of these variables. In a classic sense, an instance of *Detailed_Model* is a well-defined mathematical system that, given constant or time-dependent values of its input variables and initial values of its state variables, may be used to integrate the time-dependent behavior of its state and output variables. This integration is automatically carried out numerically by calling the *integrate( )* function of the *Detailed_Model* class. The resulting values are documented by calling the *report_values( )* function. More significantly, however, the *Detailed_Model* class extends the functionality of traditional numerical integration routines by also encapsulating the methodology of the model-order reduction algorithm. As a result, at any instance in simulation time, the *generate_simplified_model( )* function of the *Detailed_Model* class can be called to automatically create a simplified, reduced-order model. This simplified model may then be used to accurately predict values of the *Detailed_Model* outputs over short periods of simulation time. In the class structure, these reduced-order simplified models are encapsulated by instances of the *Simplified_Model* class.

Table 1. Description of *Detailed_Model* Class	
Class:	*Detailed_Model*
Inputs specified by user:	*calculate_rate_of_change( )* function for computing time-derivatives of state variables given current values of state variables, input variables, and time.
	calculate_outputs( ) function for computing values of output variables given current values of state variables, input variables, and time.
	initialize_states( ) function for initializing values of state variables.
	number_of_states, *number_of_inputs*, *number_of_outputs* integers that define the dimensions of the model.
Class methods:	*integrate( )* function numerically integrates model over a specified time interval.
	report_values( ) function reports current values of state, input, and output variables.
	generate_simplified_model( ) generates the data necessary for constructing a reduced-order model of the *Detailed_Model*.
Class variables:	*states[ ]* array representing current values of state variables. *states_ddt[ ]* array representing current values of time derivatives of state variables. *inputs[ ]* array representing current values of input variables. *outputs[ ]* array representing current values of output variables.

The *Simplified_Model* class is summarized in Table 2. This class represents a linearized reduced-order model corresponding to a particular *Detailed_Model*. The *generate_simplified_model( )* function of *Detailed_Model* generates the necessary data in *simplified_model_parameters* that is used to create any number of instances of *Simplified_Model*. The *Detailed_Model* also supplies an upper bound estimate of the duration over which the *Simplified_Model* is accurate by specifying an *expiration_time*. Given the same values of input variables as to the *Detailed_Model*, the *integrate( )* and *calculate_outputs( )* functions of the *Simplified_Model* can accurately predict the outputs of the *Detailed_Model* over short periods of simulation time. Note at any point in the simulation when the estimated output values predicted by the *Simplified_Model* diverge from the actual output values calculated by the *Detailed_Model*, the *update_expiration_time( )* function is used to signal that an updated *Simplified_Model* is required at that time point of the simulation. In this manner, an instance of the *Simplified_Model* class accurately serves as a substitute for a *Detailed_Model* over a short time interval. More significantly, a sequence of updated *Simplified_Model*s accurately serve as substitutes for calculating the outputs of a *Detailed_Model* over an entire simulation.

Table 2. Description of *Simplified_Model* Class	
Class:	*Simplified_Model*
Inputs specified by *Detailed_Model*:	*simplified_model_parameters* data structure containing information required for instantiating *Simplified_Model* class
	number_of_transformed states, *number_of_inputs*, *number_of_outputs* which are integers that define the dimensions of the model.
	initial_time which specifies the point in simulation time at which the simplified model data was generated.
	expiration_time which specifies the point in simulation time at which the simplified model becomes invalid. This time is determined by comparing predicted outputs with those of its counterpart *Detailed_Model*.
Class methods:	*integrate( )* function numerically integrates simplified model over a specified time interval.
	calculate_outputs( ) function computes values of output variables given current values of transformed state variables, input variables, and time.
	report_values( ) function reports current values of transformed state, input, and output variables.
	update_expiration_time( ) function updates the expiration time at which the *Simplified_Model* becomes invalid.
Class variables:	*transformed_states[ ]* array representing current values of the transformed state variables. *inputs[ ]* array representing current values of input variables. *outputs[ ]* array representing current values of output variables.

The *Detailed_Model* class and its *Simplified_Model* counterpart provide a common framework that encapsulates single-discipline subsystem models as modular, reusable components. These classes are designed to be minimally invasive during the modeling process, allowing engineers to develop new subsystem models or reuse legacy models with little additional overhead. They also streamline aspects of model development by providing routines for numerical integration, by automatically and dynamically creating data structures for storing variable values during simulation, and by reporting simulation results in a structured format. These classes may also be readily extended and modified through the object-oriented concept of inheritance. For example, an engineer may extend and modify existing capabilities by defining a new subclass of the *Detailed_Model* class. In this subclass, a new *integrate( )* function may be defined that replaces the default *integrate( )* function with a specialized integration routine. Likewise, a new *report_values( )* function may be defined that customizes the reporting of simulation results. Entirely new member functions and variables may also be defined. For example, a function may be defined in the subclass that determines the value of an input variable from a signal generated by the controls of a real-time training simulator. Similarly, a new

variable may be defined in the subclass that stores a bitmapped image used to depict the model graphically in a graphical user interface environment.

In our design, we have focused on keeping the *Detailed_Model* and *Simplified_Model* streamlined so that engineers developing subsystem models are not burdened with the intricacies of distributed simulation of complex systems. Consequently, in our design we have encapsulated this functionality in two separate classes, *Simulation_Process* and *Simulation_Manager*.

The *Simulation_Process* class is summarized in Table 3. This class represents an integrated multi-system model with N-component subsystems. Note, however, that as in Figure 1 only one subsystem is represented by a *Detailed_Model* and the remaining N-1 subsystems are represented by *Simplified_Models*. An instance of a *Simulation_Process* is created on each computer where the *Detailed_Model* of a particular subsystem is defined. The overall model is simulated by the *Simulation_Process* by integrating the *Detailed_Model* simultaneously with the N-1 *Simplified_Models*.

During this integration, the set of *Simplified_Models* are updated as dictated by an instance of the *Simulation_Manager*. The *Simulation_Process* communicates with the *Simulation_Manager* over the network using CORBA protocol. The *Simulation_Manager* also integrates the current simplified version of its *Detailed_Model* in order to evaluate its validity after each time step. If an updated *Simplified_Model* is required, it is generated by the *Detailed_Model* and sent by the *Simulation_Process* to the *Simulation_Manager* using the CORBA protocol.

Table 3. Description of *Simulation_Process* Class	
Class:	*Simulation_Process*
Inputs specified by user:	*Detailed_Model* which is the locally defined detailed model of a particular subsystem.
Inputs specified by *Simulation_Manager*:	*Simplified_Models* which are the current set of N-1 simplified subsystem models received from the *Simulation_Manager*.
Class methods:	*integrate( )* function simultaneously integrates the *Detailed_Model* along with the N-1 *Simplified_Models* over a specified time interval.
	update_expiration_of_simplified_model( ) function is used by *Simulation_Manager* to update the *expiration_time* of a particular *Simplified_Model*.
	transmit_updated_simple_model( ) function notifies *Simulation_Manager* that an updated *Simplified_Model* version becomes valid at the current time step.
Class variables:	*corba_address* represents the unique CORBA identification number used by the *Simulation_Manager* to communicate with the *Simulation_Process*. *current_simplified_model* represents the current simplified version of the local *Detailed_Model*, used to determine when an update is required. *input_output_map* maintains a mapping that links all input variables of all detailed and simplified model with the appropriate output variable of another detailed or simplified model.

The *Simulation_Manager* class is summarized in Table 4. The *Simulation_Manager* coordinates communication between the instances of *Simulation_Processes* running on separate computers. A single instance of *Simulation_Manager* is created for an entire distributed multi-system simulation. This "global" *Simulation_Manager* may reside on the same computer as a *Simulation_Process*, or it may reside independently on a dedicated computer. The *Simulation_Manager* serves as a central repository from which simulations are initiated and where histories of *Simplified_Models* for each *Detailed_Model* are maintained. It also serves as a communication middleman between the individual *Simulation_Processes*. In this manner, each *Simulation_Process* does not need to be aware of and communicate with all other *Simulation_Processes*, but rather communicates exclusively with the global *Simulation_Manager*.

Table 4. Description of *Simulation_Manager* Class	
Class:	*Simulation_Manager*
Inputs specified by user:	*CORBA_addresses* that identify all instances of *Simulation_Process*es that compose the distributed multi-system simulation.
Class methods:	*initiate_simulation( )* function requests a *Simplified_Model* from each *Simulation_Process*, transmits these *Simplified_Model*s among all other *Simulation_Process*es, and notifies all *Simulation_Process*es to begin integration.
	register_updated_simple_model( ) function registers a data structure for constructing an updated *Simplified_Model* along with the time interval over which it is valid.
	transmit_expiration_of_simplified_model( ) function communicates to all *Simulation_Process*es that a particular *Simplified_Model* will become invalid at a specified time point.
Class variables:	*simplified_model_history* maintains a history of *Simplified_Models* for each *Simulation_Process* and the time intervals over which they are valid.

3.0 ALGORITHM DEVELOPMENT (TASK 2.2)

3.1 INTRODUCTION

To allow a model to be subdivided into semi-independent subsystems, numerical techniques are needed that will allow the individual models to be integrated. This must be done in the presence of strong feedbacks and complicated interactions while still achieving computationally-stable results. As described above, we accomplish this by automatically generating simplified versions of each subsystem model from the detailed model created by its analysts. These simplified models serve as an accurate stand-in for the associated detailed model for short periods of time. The simplified models are kept accurate by periodically updating them to account for changes in the nominal operating state.

The subsequent subsections describe an algorithm for automatically developing a simplified version of a detailed subsystem model.

3.2 MODEL DEFINITION

We assume that a particular subsystem model has been written in the following form:

$$\frac{dw_i}{dt} = e_i(w, y, \alpha) \quad \text{i=1,...n} \tag{1}$$

$$0 = g_j(w, y, \alpha) \quad \text{j=1,...m} \tag{2}$$

$$z_k = h_k(w, y, \alpha) \quad \text{k=1,...p} \tag{3}$$

In Equations (1–3), w denotes the solution of first order ordinary differential equations (ODEs), y denotes the solution of algebraic equations, t is the time, and α represents parameters. The parameters can be time-varying, but their values are assumed to be supplied to the subsystem model under consideration from outside sources (see below).

This formulation is relatively general, encompassing both transient simulations as well as purely algebraic (design-type) models (for which the number of ODEs n is zero). We do currently assume that the defining functions e and g are continuous in the variables w, y, and α; relaxation of this requirement will be investigated later in the project. Note that to simplify the subsequent development, we have assumed that the functions e, g, and h do not depend explicitly on time. Actually, this does not limit the generality, since we can formally define an additional variable to replace time:

$$\frac{dw_{n+1}}{dt} = 1 \tag{4}$$

In most cases, the number of states n+m will be relatively large, whereas the number of key outputs p and the number of parameters will be relatively small.

Let us initially consider the solution of the subsystem model Equations (1–3) in a standalone fashion, without regard to the other subsystems with which it may interact. In this parochial view, the interfaces between the model under consideration and those representing the adjoining subsystems of the vehicle are defined by what may be considered "input parameters." For example, a turbopump subsystem model requires knowledge of the inlet pressure, which we assume is supplied by a model for the fuel or oxidizer subsystem. Of course, the outputs of the latter depend on outputs of the turbopump model, so that the inlet pressure is actually determined by the interaction between the turbopump and fuel subsystem models, and perhaps others as well. However, for developing a simplified version of the turbopump model, it is convenient to temporarily ignore this complication. Thus, the quantities α include both time-dependent inputs that are known a priori as well as the outputs of other subsystem models which must be solved.

The quantities z represent output variables of this subsystem that are of particular interest. These are defined by explicit functions in the other variables. Explicitly defining key outputs in this way allows us to distinguish the important results of a model from internal details that are important only in that they affect these results. In particular, the set of the output variables z includes the interface variables between subsystem models, so that the outputs z from one model are fed into another model as input "parameters" α to that model. In the example just discussed, a key output of the turbopump model would be the calculated flow rate, since this will affect the pressure and other quantities calculated by the fuel subsystem model.

To simplify the mathematical development, it is convenient to initially combine the algebraic and ODE variables by defining a "flag" θ_i which has the value 0 for variables defined

explicitly or implicitly by the algebraic Equations (2) and has the value 1 for variables ("true states") defined by ODEs, Equations (1). Then we can replace Equations (1) and (2) with:

$$\theta_i \frac{dx_i}{dt} = f(x,\alpha) \quad \text{i=n+m} \tag{5}$$

Note that x is used to denote either an algebraically-defined variable y or an ODE-defined variable w, and f is used to represent the function f or g that defines the state.

3.3 Transforming the Model into an Equivalent System that is Amenable to Model Order Reduction

The elements of the Jacobian matrix of the subsystem model are defined by:

$$J_{ik} = \frac{\partial f_i}{\partial x_k} \quad \text{i,k =1,...n+m} \tag{6}$$

If we did wish to continue separately identifying the algebraic variables y, the structure of the matix J would be:

$$J = \begin{pmatrix} \frac{\partial e}{\partial w} & \frac{\partial e}{\partial y} \\ \frac{\partial g}{\partial w} & \frac{\partial g}{\partial y} \end{pmatrix} \tag{7}$$

For the moment, however, we continue to treat the x's and y's in the unified manner implied by Equation (5).

Following Meirovitch (1980), the ith ("right") eigenvector vr_i of the Jacobian matrix is defined by:

$$J\, vr_i = \lambda_i\, vr_i \quad \text{i=1,...n} \tag{8}$$

The eigenvectors of the transpose of J are denoted by vl_j:

$$J^T vl_j = \lambda_j vl_j \quad \text{j=1,...n} \tag{9}$$

The eigenvectors of the transpose are unequal to the eigenvectors of the original matrix if the Jacobian is not symmetric. However, we are justified in using the same symbol λ for its eigenvalues, since the eigenvalues of the transpose of a matrix are the same as that of the original matrix. Taking now the transpose of both sides of Equation (9), we have:

$$vl_j^T J = \lambda_j vl_j^T \tag{10}$$

Therefore we identify the *vl* as "left" eigenvectors of the Jacobian. If we multiply Equation (10) on the right by vr_i, multiply Equation (8) on the left by vl_j^T, and subtract the results, we obtain:

$$0 = (\lambda_i - \lambda_j) vl_j^T vr_i \tag{11}$$

Thus, the left and right eigenvectors of J corresponding to distinct eigenvalues are orthogonal. In the more familiar special case in which J is symmetric, the matrix is equal to its transpose, and the right and left eigenvectors are the same. In such a case, Equation (11) reduces to the more common statement that the eigenvectors of a symmetric matrix form an orthogonal set. In the more general case of interest here, in which J is not symmetric, we say that the left and right eigenvectors form a "bi-orthogonal" set.

We can multiply either the right or left eigenvectors by an arbitrary constant without affecting the validity of Equations (8) and (10). We therefore have the freedom to normalize the right and left eigenvectors in such a way that Equation (11) implies:

$$vl_j^T vr_i = \delta_{ji} \tag{12}$$

where δ_{ji} is the Kronecker delta (the elements of the identity matrix I). Now define a matrix E whose j^{th} row contains the j^{th} left eigenvector. Similarly, the matrix EM is defined by arranging the right eigenvectors by columns. Finally, we define the matrix Λ whose i^{th} on-diagonal element is λ_i.

Given these definitions, the ji^{th} matrix element of the product of E and EM is given by the scalar product of the two corresponding eigenvectors:

$$(E\,EM)_{ji} = vl_j^T vr_i \tag{13}$$

Using Equation (12), the product of E and EM is:

$$E\,EM = I \tag{14}$$

Thus, EM is the inverse of E. If we multiply both sides of Equation (14) on the right by E and re-group, we also obtain:

$$E(EM\,E) = E \tag{15}$$

from which we conclude that:

$$EM\,E = I \tag{16}$$

In view of the definition of EM, Equation (8) can be re-written:

$$J\,EM = EM\,\Lambda \tag{17}$$

Multiplying both sides of Equation (17) on the left by E and using Equation (14) we have:

$$E\,J\,EM = \Lambda \tag{18}$$

That is, *E* and *EM* comprise a similarity transformation that diagonalizes the Jacobian.

With these preliminaries out of the way, we now proceed to develop a transformed subsystem model that will support simplified model development, i.e. "model order reduction." We assume that the model is to be used over a sufficiently short interval that linearization of the model around its initial state can be justified. The changes $\underline{\delta x}$ in the state vector $\underline{x}$ from the values about which it was linearized are given by:

$$\theta_i \frac{d\delta x_i}{dt} = J_{ik}\delta x_k + \frac{\partial f_i}{\partial \alpha_k}\delta\alpha_k + f_i^o \quad \text{i=1,...n+m} \tag{19}$$

(no sum on i)

In Equation (19) and henceforth, the Einstein summation convention is used unless stated otherwise, i.e., in this case summation is implied over the repeated index k but not over index i. Dropping the matrix and vector subscripts, pre-multiplying Equation (19) by E, and using Equation (16), we can write:

$$E\theta\frac{d\delta x}{dt} = E\,J\,(EM\;E)\delta x + E\frac{\partial f}{\partial \alpha}\delta\alpha + Ef^o \tag{20}$$

Regrouping and using Equation (18), we obtain:

$$\theta\frac{d\hat{\delta x}}{dt} = \Lambda\;\hat{\delta x} + \frac{\partial \hat{f}}{\partial \alpha}\delta\alpha + \hat{f}^o \tag{21}$$

In Equation (21), we have adopted the "^" notation to indicate quantities in a transformed coordinate system. Transformed quantities in this case are obtained by multiplying the original quantity by the matrix E. In the transformed coordinate system, the ODEs have all been "disentangled" (by diagonalizing the Jacobian), so that their rates of change no longer depend on each other. This property can be exploited for model order reduction, as shown in subsections 3.4 and 3.5.

3.4 SOLUTION OF SUBSYSTEM MODEL

By utilizing an integrating factor:

$$s = e^{-\lambda_i t} \tag{22}$$

one can readily obtain the exact solution of the ODEs represented by the linearized and transformed system model Equation (21) (Kaplan, 1952). At this point, it is now useful to resume distinguishing the ODEs from algebraic equations, and we restore the *w-y/e-g* notation for that purpose.

After integration, we obtain for the transformed ODEs:

$$\delta \hat{w}_i(t) = e^{\lambda_i t}\{\int_o^t e^{-\lambda_i \tau} \frac{\partial \hat{e}_i}{\partial \alpha_k} \delta\alpha_k(t) d\tau + \int_o^t e^{-\lambda_i \tau} \hat{e}_i^o \, d\tau\} \tag{23}$$

(no sum on i)

The algebraic equations represented by the terms in Equation (21) for which $\theta_j = 0$ need not be integrated, of course. Representing their counterparts in the transformed coordinate system by $\hat{y}$, their solution is:

$$\delta \hat{y}_j = \frac{\frac{\partial \hat{g}_j}{\partial \alpha_k} \delta\alpha_k + \hat{g}_j^o}{-\lambda_j} \tag{24}$$

For either the algebraic or differential equations, the states in the original coordinate system can be calculated if desired by reversing the effects of the coordinate transformation E using its inverse, EM :

$$\delta x_k = EM_{ki} \, \delta \hat{x}_i \tag{25}$$

3.5 MODEL ORDER REDUCTION

Our ultimate goal is to obtain a simplified model that can calculate the key outputs z to a specified accuracy over a short time interval. The time interval is limited to a period over which the linearized version of the nonlinear subsystem model is considered sufficiently accurate. To develop the simplified model, we linearize the equations for the output variables:

$$z_k(t) = \frac{\partial h_k}{\partial x_i} \delta x_i(t) + \frac{\partial h_k}{\partial \alpha_m} \delta\alpha_m(t) + z_k^o \tag{26}$$

Equation (26) can also be expressed in terms of the transformed (^) variables. To this end, we first use Equation 25 to define a transformed version of $\frac{\partial h}{\partial x}$ that utilizes transformed variables $\delta \hat{x}$ rather than the original states δx:

$$\frac{\partial \hat{h}_k}{\partial x_i} = \frac{\partial h_k}{\partial x_m} EM_{mi} \tag{27}$$

Substituting Equations (23), (24), and (27) into Equation (26), we have:

$$z_k(t) = \frac{\partial \hat{h}_k}{\partial w_i} e^{\lambda_i t} \left(\int_o^t e^{-\lambda_i \tau} \frac{\partial \hat{e}_i}{\partial \alpha_m} \delta\alpha_m(\tau) d\tau + \int_o^t e^{-\lambda_i \tau} \hat{e}_i^o \, d\tau \right) + \frac{\partial \hat{h}_k}{\partial y_j} \frac{\frac{\partial \hat{g}_j}{\partial \alpha_m} \delta\alpha_m + \hat{g}_j^o}{-\lambda_j} + \frac{\partial h_k}{\partial \alpha_m} \delta\alpha_m + z_k^o \tag{28}$$

Equation (28) formally represents the linearized values of the output variables in terms of the input parameters, various constants, and time. The first set of terms on the right-hand-side of the equation represent the contributions of the transformed ODEs, and the second set of terms represent the effects of the transformed algebraic variables. The second to last term represents the "direct" effect of changing parameters on z. The only approximations involved in writing Equation (28) are those associated with linearizing the functions e, g, and h appearing in Equations (1–3).

As it stands, Equation (28) is not especially useful, since it includes numerous terms resulting from the large number of equations n+m in the subsystem model. To simplify this expression, first note that the contributions to the outputs from the algebraic equations can be simply combined with the last two terms. The terms z_k^o in Equation (28) become:

$$z_k^o \rightarrow z_k^o + \frac{\partial \hat{h}_k}{\partial y_j} \frac{\hat{g}_j^o}{-\lambda_j} \tag{29}$$

Similarly, we add to the direct effect term $\frac{\partial h_k}{\partial \alpha_m}$ a quantity that reflects the effect of a change in parameter α_m acting through the variables $\delta \hat{y}$:

$$\frac{\partial h_k}{\partial \alpha_m} \rightarrow \frac{\partial h_k}{\partial \alpha_m} + \frac{\partial \hat{h}_k}{\partial y_j} \frac{\frac{\partial \hat{g}_j}{\partial \alpha_m}}{-\lambda_j} \tag{30}$$

This eliminates all the algebraic terms from Equation (28), affording a considerable simplification in the approximate subsystem model if the number of algebraic equations m is large.

To simplify the terms associate with the ODEs, we assume that the model is to be used over an interval [0,T], at the end of which the model is to be updated to account for nonlinear effects. We further assume that the maximum duration T that the model could be used can be specified. For the limited purpose of assessing the relative importance of the remaining terms in Equation (28), we assert that it is sufficient to consider the special case where the parameters α are held constant over the interval [0,T]. In this case, the ODE terms in Equation (28) can be integrated and the result substituted into Equation (26) to yield their contribution to the output variables:

$$z_k(T) = \frac{\partial \hat{h}_k}{\partial w_i} \frac{(e^{\lambda_i T} - 1)}{\lambda_i} \left(\frac{\partial \hat{e}_i}{\partial \alpha_m} \alpha_m + \hat{e}_i^o \right) + \ldots \tag{31}$$

This expression allows the order of the model to be systematically reduced as will now be discussed.

a. Elimination of Numerically Stiff ODE States

Consider the product of each eigenvalue λ_i and a characteristic minimum time of interest, e.g., the time step used for integration. If this product is sufficiently negative, the associated state goes through a very fast transient and reaches a new steady-state quickly. Such states are often termed "stiff," and in this case the underlying ODE can be algebraically eliminated by assuming that the state variable instantly achieves its steady-state value in response to changes in input parameters. Therefore, to eliminate a stiff state we solve Equation (21) by assuming that the transient happens over such a short interval that the states can always be assumed to reflect their quasi-steady-state values. These steady-state solutions are obtained by setting the rate-of-change to zero, yielding:

$$\delta \hat{w}_i^{ss} = \frac{\frac{\partial \hat{e}_i}{\partial \alpha_m} \delta \alpha_m + \hat{e}_i^o}{-\lambda_i} \tag{32}$$

Not surprisingly, this is the same form obtained for the model equations that were originally given in algebraic form, Equation (24). As was done for the algebraic equations, the contribution of this state to the output variable z_k^o can be included by adding to each of the terms z_k^o in Equation (26) the following term:

$$z_k^o \rightarrow z_k^o + \frac{\partial \hat{h}_k}{\partial w_i} \frac{\hat{e}_i^o}{-\lambda_i} \tag{33}$$

Similarly, we add to the $\frac{\partial h_k}{\partial \alpha_m}$ term a quantity that reflects the effect of a change in the parameter acting through the variable $\delta \hat{w}_i$:

$$\frac{\partial h_k}{\partial \alpha_m} \rightarrow \frac{\partial h_k}{\partial \alpha_m} + \frac{\partial \hat{h}_k}{\partial w_i} \frac{\frac{\partial \hat{e}_i}{\partial \alpha_m}}{-\lambda_i} \tag{34}$$

This process eliminates all the stiff states, subsuming their contributions within the direct effects of parameter changes on the output variables.

b. Elimination of Unobservable or Barely Observable ODE States

We can also eliminate states that have a transient behavior, but do not contribute substantially to the output variable. In control theory, such states are often called "unobservable," since they don't affect the outputs appreciably. From Equation (31), for any set of constant inputs $\delta\alpha_m$ the contribution of state i to the change in a given output z_k over T is given by:

$$\frac{\partial \hat{h}_k}{\partial w_i} \frac{(e^{\lambda_i T} - 1)}{\lambda_i} \left(\frac{\partial \hat{e}_i}{\partial \alpha_m} \delta\alpha_m + \hat{e}_i^o \right) \tag{35}$$

(no sum on i)

In general, we expect only a relatively small number of the transformed states to contribute significantly to each output variable. When we evaluate each of the states, several situations may be found that permit simplification:

1) If a state $\hat{w}_i$ contributes very little to one of the z_k, all the associated terms can be eliminated completely.

2) If only the last term of Equation (35) is significant, then the state can be integrated to time t<T and the result added to the "base" term z_k^o in Equation (26). The base term now becomes an explicit function of time:

$$z_k^o \rightarrow z_k^o + \frac{\partial \hat{h}_k}{\partial w_i} \frac{(e^{\lambda_i t} - 1)}{\lambda_i} \hat{e}_i^o \tag{36}$$

If convenient for the software implementation, this term can alternatively remain represented by an integral, albeit a trivial one.

3) If, as expected, some but not all of the parameter-dependent terms of the state contribute significantly to the total change in z_k^o, then only the significant terms need to be included in Equation (28). Note that the magnitude of the changes in parameter $\delta\alpha_m(t)$ need not be known ahead of time since we base the decision on whether to include a term on its relative contribution to the sum over states i for a given m.

Using this process, the form of Equation (28) can be simplified until it contains the minimum number of terms necessary to calculate the key outputs of a subsystem model over a limited time interval.

3.6 CONCLUSIONS

A method to develop simplified subsystem models defined by first-order ODEs and algebraic equations has been outlined. The simplified models are created by linearizing the full system equations around the current operating point, diagonalizing the linearized versions of the model equations, and then including only those terms necessary to calculate specifically identified outputs to a defined accuracy. Assuming the time interval over which the model is used is sufficiently short to justify linearization, the only additional assumption is that the defining equations do not possess discontinuities. Means for relaxing this last assumption will be evaluated as the project proceeds.

4.0 REFERENCES

Biedron, R. T., Mehrotra, P., Nelson, M. L., Preston, F. S., Rehder, J. J., Rogers, J. L., Rudy, D. H., Sobieski, J., and Storaasli, O.O., "Compute as Fast as the Engineers Can Think!," Langley Research Center, NASA/TM-1999-209715, September 1999.

Heath, M.T. and Dick, W.A., "Virtual Prototyping of Solid Propellant Rockets, *Computing in Science and Engineering*, March/April 2000.

Kaplan, W., *Advanced Calculus*, Addison-Wesley, 1952.

Meirovitch, L., *Computational Methods in Structural Dynamics*, Sijthoff and Noordhoff, 1980, p 70–72.