Trajectory Optimization on a Parallel Processor

John T. Betts* and William P. Huffman*
Boeing Computer Services, Seattle, Washington 98124

The design of a trajectory for an aerospace vehicle involves choosing a set of variables to optimally shape the path of the vehicle. Typically the trajectory is simulated by numerically solving the differential equations describing the dynamics of the vehicle. The optimal trajectory is usually determined by using a nonlinear programming (parameter optimization) algorithm to select the variables. Problems that require choosing control functions are usually reduced to choosing a finite set of parameters. The computational expense of a trajectory optimization is dominated by two factors: the cost of simulating a trajectory and the cost of computing gradient information for the optimization algorithm. This paper presents a technique for using a parallel processor to reduce the cost of these calculations. The trajectory is broken into phases, which can be simulated in parallel, thereby reducing the cost of an individual trajectory. This multiple shooting technique has been suggested by a number of authors. The nonlinear optimization problem that results from this formulation produces a Jacobian matrix that is sparse. The Jacobian is computed using sparse finite differencing, which is also performed in parallel, thereby reducing the cost of obtaining gradient information for the optimization algorithm. This paper describes the application of sparse finite differencing to a multiple shooting formulation of the two-point boundary-value problem, in a manner suitable for implementation on a parallel processor. Computational experience with the algorithm as implemented on the BBN GP1000 (Butterfly) parallel processing computer is described.

I. Introduction

THE optimal design of a trajectory is a problem that can be solved using a wide variety of seemingly different computational techniques. Most state-of-the-art trajectory optimization programs combine a nonlinear iteration technique with a numerical trajectory simulation. Typically the programs differ in two respects: 1) the manner in which a trajectory optimization problem is formulated using a finite number of variables; and 2) the manner in which the resulting finite dimensional problem is solved. One approach is to parameterize the control function and a limited number of state end conditions. Examples of this approach include the POST program[1] and the GTS program.[2] A direct collocation technique in which both the state and control functions are parameterized has been implemented in the OTIS program.[3] Parameterization of the boundary-value problem (an indirect formulation) has also been explored.[4-7] The fundamental concept of partitioning a trajectory into a series of phases is essentially the basis for the parallel or multiple shooting method described by Keller[8] and Stoer and Bulirsch.[9]

In this paper we present a formalism that encompasses the traditional methods for solving such problems. After describing the basic elements of the process, we identify subprocesses that can be performed in parallel in order to reduce the overall computation time.

II. Nonlinear Programming Problem

The nonlinear programming problem can be stated as follows: find the N-vector x that minimizes the objective function

$$f(x) \tag{1}$$

subject to the m_e equality constraints

$$c_i(x) = 0 \tag{2}$$

for $i = 1, \ldots, m_e$ and the m_i inequality constraints

$$c_i(x) \geq 0 \tag{3}$$

for $i = (m_e + 1), \ldots m$, where the total number of constraints is $m = m_e + m_i$. The number of constraints m can be less than, equal to, or greater than the number of variables N. For clarity in presentation, it has been assumed that the first m_e constraints are equalities, and the remaining are inequalities. For a general nonlinear least squares problem the objective function is

$$f(x) = r^T r = \sum_{i=1}^{\tilde{n}} r_i^2 \tag{4}$$

where r is an $\tilde{n}$ vector of residuals.

It is assumed that the functions are continuously differentiable to second order, although in practice these attributes must be inferred. In fact, it may be necessary for the optimization algorithm to detect discontinuities and respond appropriately. Furthermore, the functions may not be computable for all x. It is convenient to visualize the function generator as a "black box," which either 1) evaluates the problem functions f and c_i, or 2) sets a flag (the function error flag) indicating that evaluation is impossible. Thus, the functions are computable over some region Ω, which in general cannot be precisely defined.

Define the basic of constraints by

$$\mathcal{Z}^* = [i \mid c_i(x^*) = 0; \qquad i \in \{1, \ldots, m\}] \tag{5}$$

The set consists of all equality constraints and all inequality constraints that are zero at the solution point x^*, and hence is denoted by the symbol $\mathcal{Z}^*$. For consistency, denote the set of constraints that are strictly positive at the solution by $\mathcal{P}$. An estimate of the basic set $\mathcal{Z}$ is referred to as a basis or active set. The Kuhn-Tucker necessary conditions for a local minimum require that

$$\nabla f(x^*) + G^T(x^*)\lambda^* = 0 \tag{6}$$

Presented as Paper 89-3613 at the AIAA Guidance, Navigation, and Control Conference, Boston, MA, Aug. 14-16, 1989; received Sept. 11, 1989; revision received Jan. 19, 1990.

*Applied Mathematician, Engineering Computing and Analysis Division.

where $\nabla f(x^*)$ is the N-dimensional gradient vector, $G(x^*)$ is the $m \times N$ Jacobian matrix, and λ^* is the m vector of Lagrange multipliers. Furthermore,

$$\lambda_i^* c_i(x^*) = 0 \tag{7}$$

for $i = 1,\ldots, m$ and

$$\lambda_i^* \leq 0 \tag{8}$$

for $i = (m_e + 1),\ldots, m$. These optimality conditions are necessary for a local solution.

Most state-of-the-art nonlinear programming algorithms require gradient information, i.e., the $M \times N$ Jacobian matrix

$$D \equiv \begin{bmatrix} (\nabla q_1)^T \\ (\nabla q_2)^T \\ \vdots \\ (\nabla q_M)^T \end{bmatrix} \equiv \begin{bmatrix} \frac{\partial q_1}{\partial x_1} & \frac{\partial q_1}{\partial x_2} & \cdots & \frac{\partial q_1}{\partial x_N} \\ \frac{\partial q_2}{\partial x_1} & \frac{\partial q_2}{\partial x_2} & \cdots & \frac{\partial q_2}{\partial x_N} \\ \vdots & & \ddots & \\ \frac{\partial q_M}{\partial x_1} & \frac{\partial q_M}{\partial x_2} & \cdots & \frac{\partial q_M}{\partial x_N} \end{bmatrix} \tag{9}$$

For general constrained optimization applications $M = m + 1$ and M-dimensional vector κ defines a reordering such that

$$q_{\kappa(i)} = c_i \tag{10}$$

for $i = 1,\ldots, m$ and

$$q_{\kappa(m+1)} = f \tag{11}$$

For constrained least squares minimization applications $M = m + \tilde{n}$ and the residuals are stored such that

$$q_{\kappa(i)} = r_i \tag{12}$$

for $i = 1,\ldots, \tilde{n}$.

Although any mathematical programming algorithm can be used to solve the stated problem, the numerical results presented were obtained using the NLP2 algorithm of Betts.[10]

III. Function Generator

For a trajectory optimization problem the simulation of a trajectory is performed by the function generator. A trajectory is made up of a collection of phases. Phase k is characterized by a set of ordinary differential equations

$$\dot{y}_k = h_k[y_k, t, x_k, p_k] \tag{13}$$

defined for values of the independent variable t in the region

$$t_k \leq t \leq t_{k+1} \tag{14}$$

where h_k is an n_k vector of system equations, y_k is an n_k vector of dynamic variables, and $\dot{y}_k$ is a vector of derivatives with respect to t. The quantities x_k and p_k, which are defined below in Eqs. (16) and (17), do not vary dynamically within a phase. A phase is terminated by an event that occurs when the scalar event criteria function

$$\epsilon_k[y_k(t_{k+1}), t_{k+1}] = 0 \tag{15}$$

The value t_{k+1} is referred to as the event or phase boundary. Both the dynamic variables y_k and differential equations h_k may change from one phase to another. The nonlinear programming variables x are introduced to the trajectory as inputs to a phase, and the nonlinear programming constraints c and objective function f are computed from phase outputs. For clarity, phase dependent notation is omitted. The simulation of a phase and a trajectory are explained in the following sections.

IV. Phase Simulation

The computational simulation of a phase entails three distinct subprocesses—initialization, propagation, and termination.

A. Phase Initialization

Let us introduce the partitioning of x into η blocks,

$$x = [x_1, x_2, \ldots, x_\eta] \tag{16}$$

where η is the number of phases, and x_k denotes a ν_k dimension vector of variables in block k. The initialization of a phase k requires defining the initial values of the dynamic variables $y(t_k)$ and the independent variable t_k as given by

$$[y(t_k), t_k]^T = \phi_k[x_k, p_k] \tag{17}$$

In general the $(n_k + 1)$ vector ϕ_k is a nonlinear function of the μ_k parameters p_k introduced in phase k and the trajectory optimization variables x_k. It is important to note that all optimization variables x for the nonlinear programming problem [Eqs. (1–3)] are introduced to the trajectory as inputs to a phase.

B. Phase Propagation

The propagation of a phase entails computing the value of the dynamic variables $y(t_{k+1})$ at the phase boundary t_{k+1} defined by the event criteria (15). When the phase dynamics dictated by the differential Eqs. (13) are simple enough it may be possible to compute the terminal state $y(t_{k+1})$ algebraically. For example, Huffman[11] describes an analytic propagation technique for an orbital coast phase with a simplified gravitational force. However, in general the phase propagation must be treated as a numerical initial value problem in ordinary differential equations. The book by Gear[12] provides an overview of the techniques used to numerically propagate the set of differential equations. Unfortunately, most numerical techniques for propagation do not produce results that vary smoothly with variations in the initial conditions. Consequently, these techniques are not well suited to the numerical differentiation required by nonlinear programming algorithms.

Inconsistent behavior of the function generator may appear in a number of ways as described in Betts.[13] The two primary sources of inconsistent or nonsmooth behavior in the propagation process are 1) the adaptive quadrature method, and 2) the event detection process. Most numerical integration algorithms adjust the integration stepsize and/or the order of the approximating polynomial in an attempt to satisfy specified relative error criteria. This approach can produce an algorithm that is very efficient in terms of the number of times the right-hand sides of the differential equations are evaluated. However, integration of a neighboring trajectory using such a technique can result in a different pattern of stepsize and/or order. Stepsize or order variation can produce nonsmooth results when attempting to construct finite difference gradients. One way to achieve consistency in the integration process is to use a fixed-step, fixed-order integrator, such as the smooth method developed by Brenan.[14] An alternate approach is to use an integration process designed to be free from all branching caused by "if" tests, absolute values, "max" functions, etc. The Runge-Kutta-Taylor (RKT) method developed by Gear and Vu[15] and Vu[16] is a variable stepsize algorithm specifically designed for the smooth solution of ordinary differential equations.

Nonsmooth behavior in the computed functions can also be produced by the event detection process. Since the phase boundary is expressed as an implicit function [Eq. (15)], event detection typically requires a nonlinear iteration technique to locate the root of the event criteria function. Numerical quadrature techniques typically produce values for the dependent variables y at discrete mesh points. Let us denote the value of the variables at the mesh points t_j and $t_{j+1} = t_j + \delta t$, where δt is the integration stepsize, by y_j and y_{j+1}, respectively. After an integration step in which the event criteria function has changed sign, i.e.,

$$\epsilon_k[y(t_j),t_j]\epsilon_k[y(t_{j+1}),t_{j+1}]<0 \tag{18}$$

it is assumed that the event boundary t_{k+1} has been bracketed, and

$$t_j \le t_{k+1} \le t_{j+1} \tag{19}$$

To locate the boundary point Eq. (15) must be solved iteratively, which requires values of the dependent variables y between the mesh points. If the dependent variable is computed using another integration step the accuracy of the result is different than that at the mesh points; hence, the results are inconsistent. The preferred approach is to interpolate between the function values at the mesh points using a function compatible with the integration scheme. In fact both the methods in Refs. 14 and 16 provide compatible integration/interpolation algorithms. Smooth results can then be produced by locating the root t_{k+1} to machine precision.

C. Phase Termination

Let us introduce the partitioning of q into η blocks,

$$q = [q_1,q_2,\ldots,q_\eta] \tag{20}$$

where q_k denotes an ω_k-dimension vector of quantities computed in phase k. During the termination of a phase, functions of the dynamic variables $y(t_{k+1})$, the independent variable t_{k+1}, and the optimization variables x are computed, i.e.,

$$[q_k,o_k]^T = \psi_k[y(t_{k+1}),t_{k+1},x] \tag{21}$$

In general, the quantities q_k and o_k are computed from the nonlinear vector function ψ_k. It is important to note that all trajectory optimization constraints c and the objective function f for the nonlinear programming problem [Eqs. (1-3)] are formed from the quantities q. The phase output quantities o_k are distinguished from the quantities q_k because they do not appear explicitly as either optimization constraints or objective function. The utility of these parameters when "linking" the phases together will be described in the following section.

V. Trajectory Simulation

A. Mission Description – Linking Phases

The basic building block used to construct a trajectory is the phase as described in the preceding section. The complete definition of the trajectory simulation requires linking together a number of phases to model the vehicle mission. To define a mission timeline it will be useful to introduce the concept of phase linkage conditions. For example, suppose we are describing a three-burn orbit transfer such as that in Ref. 17. This trajectory could be modeled with six distinct phases—phase 1 being the first coast, phase 2 the first burn, phase 3 the second coast, etc. Since the first burn follows the first coast phase, it is necessary to postulate the conditions that link phases 1 and 2. In general, the linkage conditions between phase i and phase j are of the form

$$L_{ij}[x_j,p_j,o_i] = 0 \tag{22}$$

Notice that the input to phase j, namely x_j and p_j, is linked to the output of phase i, namely o_i. The notation L_{ij} is used to describe the nonlinear vector function linking phases i and j.

For most typical mission definitions it is natural to assume the phases occur sequentially, i.e., phase i is followed in time by phase $i+1$. This assumption is consistent with the fact that the independent variable describing the dynamics is denoted by t, i.e., time. In fact, there is no need to impose this restriction on the model. First, the independent variable need not be time. Second, there is no restriction on the order in which phases can be linked to describe a mission profile. In particular, the linkage conditions permit description of multipath and multivehicle trajectory simulations. In essence then, the phases can be linked in any meaningful manner required to model the mission profile.

In many cases the linkage conditions may be of the form

$$[x_j,p_j]^T = L_{ij}[o_i] \tag{23}$$

that is, the inputs to phase j are given as explicit functions of the outputs from phase i. Let us refer to these as explicit linkage conditions, in contrast to the implicit linkage conditions in Eq. (22). When explicit linkage conditions are available, initialization of phase j is accomplished directly from Eq. (17) since

$$[y(t_j),t_j]^T = \phi_j[L_{ij}(o_i)] \tag{24}$$

On the other hand implicit conditions require an iterative solution. Specifically the linkage conditions between phase i and j must be treated as constraints on phase i, and, therefore, must be included in the set of computed quantities q_i. Furthermore, the set of optimization variables introduced at the beginning of phase j must be sufficient to initialize the dynamic variables.

The most common type of linkage condition insures continuity in the dynamic variables. At an event it may be necessary that

$$[y(t^-),t^-]^T = [y(t^+),t^+]^T \tag{25}$$

where t^- denotes the end of phase i and t^+ denotes the beginning of phase j. If the linkage is explicit the phase termination Eq. (21) for phase i should set

$$o_i = [y(t^-),t^-]^T \tag{26}$$

The phases are actually linked in the trajectory propagation process when

$$p_j = o_i \tag{27}$$

The initialization for phase j in Eq. (17) then becomes

$$[y(t^+),t^+]^T = p_j \tag{28}$$

On the other hand if the linkage is implicit, the original set of constraints can be augmented to include $c_i = q_i = 0$ where the phase termination Eq. (21) for phase i should augment any other computed values of q_i with the set

$$q_i = [y(t^-),t^-]^T - x_j \tag{29}$$

The initialization for phase j in Eq. (17) is then achieved by setting

$$[y(t^+),t^+]^T = x_j \tag{30}$$

where the original set of variables is augmented to include the initial values for the dynamic variables. It should be observed that if a dynamic variable is constant it is not necessary to add

a variable or constraint. While the most common use of a linkage condition is to force continuity, an equally important use is to introduce a naturally occurring discontinuity, such as vehicle staging. This can be achieved by introducing the value of the discontinuity as either a parameter or variable in either the explicit or implicit linkage condition.

B. Simulation Considerations

On a serial or sequential computer the simulation of a phase occurs one at a time. That is, we may simulate phase 1, then phase 2, followed by phase 3, and so on until all phases have been simulated. The trajectory is constructed using either explicit or implicit phase linkage conditions. On a parallel processor we can exploit the independence of the phase simulations by calculating individual phases on independent arithmetic processors. In so doing the total computation time for a trajectory can be reduced to the computation time of the most costly phase, provided enough processors are available. The overall mission description is still recovered by linking the phases together with implicit linkage conditions.

C. Multiphase Functions

Suppose the statement of the trajectory optimization problem requires computing an objective or constraint function of the form

$$b[y(t_\alpha), y(t_\beta)] \tag{31}$$

where $y(t_\alpha)$ and $y(t_\beta)$ are computed outputs from phases α and β, with $\alpha<\beta$. When the phases are executed sequentially with phase α occuring first, a function of this form can be accumulated by a "cascading" approach. Specifically we can set

$$o_k = [p_k, y(t_k)] \tag{32}$$

during the termination processing for phase k, and set

$$p_k = o_{k-1} \tag{33}$$

when linking the phases together. The accumulation process must be initiated during phase α and continued until phase β. The function is then formed at the end of the accumulation in phase β, using the expression

$$q_\beta = b[o_\beta, y(t_\beta)] \tag{34}$$

However, if the phases are simulated in parallel on different processors the multiphase function cannot be computed until the results from all phases have been completed. This is undesirable for two reasons. First, it imposes a "hold" condition on the parallel computations. Second, it changes the sparsity pattern of the derivative matrix D defined in Eq. (9), which will be discussed in the next section. To avoid this, let us augment the set of computed functions in phase α to include the quantities

$$\hat{q}_\alpha = y(t_\alpha) - \hat{x}_\beta \tag{35}$$

with the variables in phase β augmented to include the new variables $\hat{x}_\beta$. The function [Eq. (31)] can then be formed as an additional computed output from phase β as

$$q_\beta = b[\hat{x}_\beta, y(t_\beta)] \tag{36}$$

Effectively the original function can be replaced by an alternate form of Eq. (36), which can be computed at the end of phase β, provided the original set of constraints is augmented to include the synchronization constraints

$$\hat{c}_\alpha = \hat{q}_\alpha = 0 \tag{37}$$

where $\hat{q}_\alpha$ is given by Eq. (35).

Multiphase functions of the form of Eq. (31) describe a single relationship between the outputs from two or more phases. When a single variable is an input to two or more phases, the converse situation exists. Since all variables are available when the simulation of a phase begins, multiphase variables do not impact computational sequencing on a parallel processor. However, a multiphase variable does affect the sparsity pattern of D. For this reason it may be desirable to introduce a separate variable in each phase, and then introduce a constraint that forces the individual variables to be equal. In fact, it may be necessary to augment the natural set of problem variables in order to create the partitioning required by Eqs. (13) and (17).

VI. Sparse Finite Difference Jacobian

A. Sparsity Pattern

When a trajectory is treated using the phase structure, the natural partitioning of the variables and computed functions produces a corresponding partitioning of the Jacobian matrix. Specifically, Eq. (9) can be written as

$$D = \begin{bmatrix} D_{11} & D_{12} & \cdots & D_{1\eta} \\ D_{21} & D_{22} & \cdots & D_{2\eta} \\ \vdots & & \ddots & \\ D_{\eta 1} & D_{\eta 2} & \cdots & D_{\eta\eta} \end{bmatrix} \tag{38}$$

where the ij block of the matrix consists of the derivatives of the quantities computed in phase i with respect to the variables introduced in phase j, that is

$$D_{ij} \equiv \frac{\partial q_i}{\partial x_j} \tag{39}$$

The number of columns in the ij block is equal to ν_j, the number of variables introduced in phase j, and the number of rows in the block is equal to ω_i, the number of quantities computed in phase i.

When the phases are linked explicitly, there are no multiphase variables or constraints, and the phases are performed sequentially, the Jacobian becomes somewhat simpler. In this case Eq. (38) can be written as

$$D = \begin{bmatrix} D_{11} & 0 & \cdots & 0 \\ D_{21} & D_{22} & \cdots & 0 \\ \vdots & & \ddots & \\ D_{\eta 1} & D_{\eta 2} & \cdots & D_{\eta\eta} \end{bmatrix} \tag{40}$$

where $D_{ij}=0$ for $i<j$. This is because quantities "early" in the trajectory are not affected by variables introduced "later." The GTS trajectory program[2] exploits this property using the so-called partial trajectory mechanism. In this approach only those phases that occur after a variable is introduced to the trajectory are executed when evaluating a derivative.

When the phases occur sequentially and are linked implicitly, and there are no multiphase variables or constraints, the Jacobian becomes simpler. In this case Eq. (38) can be written as

$$D = \begin{bmatrix} D_{11} & D_{12} & \cdots & 0 \\ 0 & D_{22} & \cdots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & D_{\eta\eta} \end{bmatrix} \tag{41}$$

where $D_{ij}=0$ for $(i+1)<j$ and for $i>j$. The blocks along the

diagonal are nonzero because the functions computed in phase i are affected by the variables introduced in phase i. The blocks above the diagonal are nonzero because of the linkage conditions between phases. All other blocks are zero because of the independence of the phases.

When the linkage conditions have the special form given by Eq. (29) the Jacobian becomes still simpler, i.e.,

$$D_{i,i+1}=A_{i,i+1} \tag{42}$$

for $i=1,\ldots,(\eta-1)$ where the blocks $A_{i,i+1}$ are constants. In this case it is convenient to write

$$D=C+A \tag{43}$$

where the block diagonal matrix of general derivatives

$$C=\begin{bmatrix} C_{11} & 0 & \ldots & 0 \\ 0 & C_{22} & \ldots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \ldots & C_{\eta\eta} \end{bmatrix} \tag{44}$$

is segregated from the constant matrix

$$A=\begin{bmatrix} 0 & A_{12} & \ldots & 0 \\ \vdots & & \ddots & A_{\eta-1,\eta} \\ 0 & 0 & \ldots & 0 \end{bmatrix} \tag{45}$$

The sparsity pattern is altered slightly when a multiphase variable or constraint is introduced or when the phases do not occur sequentially and are linked accordingly. Specifically, if the constraint [Eq. (31)] is introduced or the trajectory definition requires linkage between phase α and β according to Eq. (22), then Eq. (41) must include a nonzero block $D_{\alpha\beta}$. When the linkage conditions are given by the form of Eq. (29), and/or the multiphase constraint is treated as described in Eqs. (35) and (36) the constant matrix becomes

$$A=\begin{bmatrix} 0 & A_{12} & 0 & & & \ldots & & 0 \\ & \ddots & \ddots & & & & & \\ & & 0 & A_{\alpha,\alpha+1} & 0 & A_{\alpha\beta} & \ldots & 0 \\ \vdots & & & \ddots & \ddots & 0 & & \\ & & & & 0 & A_{\beta-1,\beta} & & \\ & & & & & \ddots & \ddots & \\ & & & & & & 0 & A_{\eta-1,\eta} \\ 0 & & & \ldots & & & & 0 \end{bmatrix} \tag{46}$$

Notice that the sparsity is altered by the introduction of a nonzero constant matrix $A_{\alpha\beta}$ above the diagonal.

B. Sparse Differences

An efficient method for computing the Jacobian of a sparse matrix was suggested by Curtis et al.[18] Essentially, the approach requires partitioning the column indices of the Jacobian D into subsets Γ^k. Each subset of the columns of D has the property that there is at most one nonzero element in each row. Then define the perturbation direction vector by

$$\Delta=\sum_{j\in\Gamma^k}\delta_j e_j \tag{47}$$

where δ_j is the perturbation size for variable j, and e_j is a unit vector in direction j. Then for each nonzero row i of a column $j\in\Gamma^k$, the forward difference approximation to the Jacobian is

$$D_{ij}\approx\frac{q_i(x+\Delta)-q_i(x)}{\delta_j} \tag{48}$$

and the central difference approximation is

$$D_{ij}\approx\frac{q_i(x+\Delta)-q_i(x-\Delta)}{2\delta_j} \tag{49}$$

Denote the total number of index sets Γ^k needed to span the columns of D by γ. Then, a forward difference estimate for the Jacobian requires γ perturbations and a central difference estimate requires evaluations at 2γ perturbed points. For a sparse matrix the number of index sets can be significantly less than the number of variables, i.e., $\gamma\ll N$. Consequently, the cost of evaluating a Jacobian using sparse differences can be substantially less than standard finite difference methods.

Sparse differencing could be applied directly to produce a Jacobian with the sparsity structure of Eq. (41). However, additional simplification is possible by exploiting the special form of Eq. (43). In particular Eq. (43) implies that the computed functions q must be of the form

$$q=\tilde{q}+Ax \tag{50}$$

where the derivatives of $\tilde{q}$ are given by C and the constant portion of D is produced by the linear term. Because the sparsity pattern of C is even simpler than that of D, it is attractive to construct C by sparse differencing. After constructing index sets based on the sparsity pattern of C, the forward difference estimate is

$$C_{ij}\approx\frac{\tilde{q}_i(x+\Delta)-\tilde{q}_i(x)}{\delta_j}$$

$$=\frac{q_i(x+\Delta)-q_i(x)-(A\Delta)_i}{\delta_j} \tag{51}$$

and the central difference estimate is

$$C_{ij}\approx\frac{q_i(x+\Delta)-q_i(x-\Delta)-2(A\Delta)_i}{2\delta_j} \tag{52}$$

Row i of $A\Delta$ is denoted by $(A\Delta)_i$ in these estimates. Thus, when the constant portion A is specified, finite differencing can be used to construct C, and D follows directly from the definition (43).

Constructing index sets to span the columns of C is especially simple, since the blocks are uncoupled. The easiest approach is to use one variable from each phase, i.e., one column from each block of C. Clearly the block with the maximum number of columns defines the number of index sets needed. Thus, the number of index sets is just equal to the maximum number of variables introduced in any phase, i.e., $\gamma=\max_{k=1}^{\eta}[\nu_k]$. It is especially important to note that the number of index sets does *not* grow with the number of phases! Although the scheme is quite efficient, it may not be the optimal partitioning. Coleman and Moré[19] show that the problem of defining the minimum number of index sets that span the space can be formulated as a graph coloring problem. Application of their algorithms might afford further computational efficiencies.

C. Parallel Sparse Differences

When sparse differences are computed on a serial processor, calculations occur one index set at a time. Thus, the functions are evaluated with the variables in the first index set perturbed, followed by an evaluation with the variables in the second index set perturbed, and so on. On a parallel processor the perturbed function evaluations can be performed simultaneously on different processors. If the trajectory is simulated as described in Sec. V.B, then trajectory parallelism can also be incorporated. In this case the simulation of phase i with perturbed inputs from index set j is accomplished on the ij arithmetic processor. If enough processors are available and

overhead is ignored, the entire Jacobian D can be evaluated in the time it takes to evaluate the longest phase.

VII. Examples

To illustrate the analyses performed, a set of four orbit transfer problems were solved. The main advantage of these problems is that they are easy to implement because they avoid the added complications of atmospheric dynamics. On the other hand, the numerical integration of finite burns requires enough computation to insure that the time spent evaluating the trajectory dominates the time spent inside the optimization operator. Results are obtained for the standard, direct, "engineering" formulation of the problem, and compared with those obtained from the multiple shooting formulation.

The first two examples are typical mission performance optimization problems since the objective is to maximize the payload weight that a given vehicle can take to a geosynchronous equatorial orbit from a Shuttle park orbit. In the first case the upper stage is a two-stage solid rocket with performance similar to the inertial upper stage (IUS), whereas in the second case the upper stage has a single restartable motor with which it can make multiple burns. Both cases have trajectories composed of alternating coast and burn phases.

The last two examples assume that the payload is fixed and the investigation is to determine whether a piecewise, linear tangent steering law can be used to move a low-thrust orbit transfer vehicle from one circular park orbit to another at a possibly different inclination. To make the problem unique, the altitude of the final circular orbit was maximized. Here the phase boundaries are determined by the times at which the linear tangent steering coefficients are changed. These examples are representative of approximate optimal control solutions.

A. Two-Burn Transfer

The first example involves the calculation of the maximum payload that can be transferred from a 160 n.mi. circular park orbit at an inclination of 28.8 deg, to a geosynchronous equatorial orbit. The simulation begins at the descending node. A variable length coast is integrated during phase 1, followed by an inertially fixed finite burn during phase 2, which depletes the propellant in the stage. Phase 3 is the coast in the transfer orbit, and phase 4 is the inertially fixed finite burn of the second stage. The direct approach to the problem has eight variables—namely, two coast times, two orientation angles for each burn, the propellant weight for the second burn, and the payload weight. There are six constraints for the direct approach—on the final altitude, velocity, flight-path angle, azimuth, latitude, and propellant expended. The multiple shooting version of this problem has 26 variables, and 24 constraints, since the original problem is augmented to include six state variables and constraints at three phase boundaries.

Table 1 Optimization results – maximum payload orbit transfer

Item	Two-burn		Three-burn	
	Dir.	M.S.	Dir.	M.S.
No. opt. requested FE[a]	63	131	158	561
No. Jacobian evaluation	6	6	12	14
No. FE per Jacobian	16	20	26	22
No. FE for Jacobians	96	120	312	308
Total FE	159	251	470	869
% FE for Jacobians	60.4	47.8	66.4	35.4
FE time,	394.7	164.1	1651.1	559.6
Average seconds per FE	2.482	0.654	3.513	0.644
Opt. time,	2.2	28.6	7.4	313.7
Total time,	396.9	192.7	1658.5	873.3
% Opt. time	0.6	14.8	0.4	35.9
No. of processors	1	4	1	6

[a]Function evaluation.

Table 1 presents results obtained by solving the direct formulation (DIR) on a single processor, compared with those obtained using the multiple shooting (MS) method executed on four processors. For simplicity, the number of processors was chosen to be the same as the number of phases. The first line of the table gives the number of function evaluations requested by the optimization algorithm, i.e., the number of different estimates for the optimization variables. For this example each function evaluation required simulation of four phases. The second line of the table presents the number of Jacobian evaluations required to solve the optimization problem. Since a Jacobian is computed using finite differences, the third line presents the number of function evaluations per Jacobian. The fourth line presents the total number of function evaluations needed for Jacobian evaluations. The total number of function evaluations (the sum of lines one and four) is then given on line five, with the percentage of the total used in Jacobians listed on the next line. The total time required to perform the function evaluations is given on line seven. Line eight presents the average time per function evaluation, and is obtained directly by dividing line seven by line five. The total time required by the optimization algorithm is given in line nine, and as a percentage of the total time in eleven. The total computation time required to solve the problem, line ten, is the sum of lines seven and nine.

Table 2 presents a summary of the computational speedup as a function of the number of arithmetic processors available. The first four lines of the table summarize the results for the direct method and the second four lines summarize results for the multiple shooting method. The first column of the table contains the number of processors used, and the second column gives the total computation time required for solution of the problem. The third column is the speedup relative to the computation time for the same method on a single processor. Thus, the speedup on line six is obtained by dividing the computation time shown on line five, by the time shown on line six ($3.38 = 651.6/192.7$). The next column of the table presents the utilization efficiency of the multiple processors, and is obtained by dividing the speedup in column three by the number of processors. Finally, the last column displays the total speedup over the one processor direct method case for the specified method and number of processors. Marked quantities are estimated directly from the computed results.

Table 2 Speedups – max payload orbit transfer

No. CPU	Time, s	Relative speed	Relative util.	Total speed
Two-burn direct				
1	396.9	1.00	1.000	1.00
4	218.1	1.82	0.455	1.82
8	187.1	2.12	0.265	2.12
16[a]	173.5	2.29	0.143	2.29
Two-burn multiple shooting				
1[a]	651.6	1.00	1.000	0.61
4	192.7	3.38	0.845	2.06
8	154.3	4.22	0.528	2.57
80[a]	118.2	5.51	0.069	3.36
Three-burn direct				
1	1658.5	1.00	1.000	1.00
6[a]	773.2	2.14	0.357	2.14
12[a]	688.9	2.41	0.201	2.41
26[a]	604.6	2.74	0.106	2.74
Three-burn multiple shooting				
1[a]	3366.5	1.00	1.000	0.49
6	873.3	3.85	0.642	1.90
12[a]	774.2	4.35	0.363	2.14
132[a]	684.0	4.92	0.037	2.43

[a]Estimates.

If the computational expense of the phases had been perfectly balanced for this four-phase trajectory, one would expect the multiple shooting approach to reduce the cost of a function evaluation by a factor of four. By looking at the time it takes to fly each trajectory on line eight of Table 1, it is seen that the phases are fairly evenly balanced since the sequential trajectory took 3.85 times longer than the parallel trajectory. Thus, if the optimization had been identical in both cases, the speedup would have been exactly the ratio of the total trajectory to its longest phase when using the same number of processors as phases. As can be seen in the table, this was not the case.

The final speedup factor is actually determined by three competing effects. The simplest is the factor obtained by breaking a trajectory up into its corresponding phases. In this case, the factor of 3.85 is nearly equal to the number of phases. The other two effects are related to the optimization solution process.

The first optimization effect is the computational overhead of the optimization algorithm itself. Because the multiple shooting formulation involves more variables and constraints, operations performed by the optimization algorithm involve larger matrices, and hence, a greater number of arithmetic operations. As a rule of thumb, this extra overhead can be assumed proportional to the ratio of the number of elements in the Jacobian matrices for the two methods. In the current example, the multiple shooting Jacobian has 624 ($=26\times24$) elements, whereas the direct Jacobian has 48 ($=8\times6$) elements. This would indicate that the optimization overhead should be greater by a factor of 13 (624/48). If one now takes the 28.6 s spent in the optimization for the multiple shooting approach, and divides by the 2.2 s for the direct approach (see Table 1), the ratio is in fact 13. The accuracy of this comparison is partially coincidental, and partially because the optimization iteration histories were identical, i.e., same number of Jacobian evaluations, constraint solving iterations, and optimization iterations.

The other effect that contributes to the speedup is related to the ratio of the number of trajectories that must be flown in the two cases. While this factor is also dependent on the relative size of the problem, it primarily reflects the fact that the optimization problems in the multiple shooting and direct cases are different, and will generally have somewhat different iteration histories. In the current example, 92 (58%) more function evaluations were required in the optimization of the multiple shooting formulation. Of the 92 extra function evaluations, 24 occurred while evaluating Jacobian matrices and 68 occurred in the constraint solving steps.

B. Three-Burn Transfer

Another maximum payload performance problem was chosen as a second example. Starting from the same 160 n.mi. circular park orbit at 28.8 deg, a hypothetical single stage liquid fueled upper stage with 30,000 lb of propellant is to transfer the maximum possible amount of payload to a geosynchronous equatorial orbit using two perigee burns and one apogee burn. The engine is assumed to have an I_{sp} of 320 s and a thrust of 6400 lb.

The constraints for this problem are the same as the preceding example except the weight is constrained such that the total amount of propellant burned in the three burns must be less than or equal to 30,000 lb. The set of variables has been increased to include the extra coast time, the pointing angles of the extra burn, and the amount of propellant used on each burn. The direct approach requires 13 variables and 6 constraints, whereas the multiple shooting version has 48 variables and 41 constraints.

Inspection of the optimization results in Table 1 and the speedups computed in Table 2 shows that increasing the number of phases to increase the trajectory speedup factor did not demonstrate the expected gain over example 1. In fact, the multiple shooting formulation is slightly worse in this case than the preceding one, and the standard direct problem makes more efficient use of the processors available.

In analyzing the components of the speedup it is seen that the gain due to parallel integration of phases is 5.46, or just about the value expected for a trajectory with six phases, i.e., the ratio of the entire trajectory to the longest phase. Unfortunately, the optimization overhead increased tremendously. Since the multiple shooting formulation uses Jacobian matrices with approximately 25 times as many elements as the direct formulation, the time required for the linear algebra should be about $7.4\times25=185$ s. Additional overhead is due to the extra optimization steps and constraint solving steps taken by the multiple shooting formulation during the optimization process.

Finally, the number of function evaluations has almost doubled. Part of the increase is due to the extra Jacobian evaluations and iterations in the optimization algorithm, and part to the extra function calls on each constraint solving step. In particular, an excess of 245 trajectories were flown during four specific constraint solving iterations. This appears to be an algorithm peculiarity that is related to the size of the problem and to the way the algorithm responds when the constraint elimination process is difficult. Although the optimization overhead expenses are necessarily algorithm specific, one can expect similar trends for other nonlinear programming methods.

C. Linear Tangent Steering with Plane Change

The third example is a four phase trajectory that addresses the efficiency of using a spiral burn to make an orbit transfer between circular orbits with plane change. To be somewhat realistic, the engine thrust was assumed to be 640 lb with an I_{sp} of 320 s, and the tank was assumed to hold a total of 10,000 lb of propellant. The problem is to maximize the final circular orbit altitude for a vehicle with a fixed payload weight of 5,000 lb assuming a final orbit inclination of 20 deg. For this type of problem, the engine pointing is critical to the determination of the final optimal altitude. The thrust pointing vector is defined by a linear tangent steering law, which is optimal for a transfer over a flat Earth with constant gravity. Since these assumptions are not valid for the duration of the 5,000 s burn, the entire burn is divided into four phases with each phase having a separate set of steering coefficients. Thus for phases $k=1\ldots4$, the desired thrust pointing direction u can be expressed as a linear function of the time from the beginning of the phase Δt,

$$u_{1k}=a_{1k}-b_{1k}\Delta t \tag{53}$$

$$u_{2k}=a_{2k}-b_{2k}\Delta t \tag{54}$$

$$u_{3k}=a_{3k}-b_{3k}\Delta t \tag{55}$$

To fix the arbitrary scale factor inherent in the preceding formulation, the coefficient a_{1k} was set to 1. The remaining five coefficients on each phase were used as optimization variables. The resulting problem requires choosing the steering coefficients to maximize the final orbit altitude and constrain the final flight-path angle to zero, the final velocity to the circular velocity at the corresponding altitude, and the final orbit inclination to 20 deg. The trajectory consists of a single burn with the steering coefficients changed every 1250 s. The resulting problem has 17 degrees of freedom with 20 variables and 3 constraints in the direct approach, and 38 variables and 21 constraints in the multiple shooting approach.

The sparsity pattern of the Jacobian for the multiple shooting formulation of this problem is displayed in Fig. 1. As predicted, the matrix displays the block diagonal form with a super-diagonal set of constant blocks. The number of blocks is just equal to the number of phases. By counting the maximum

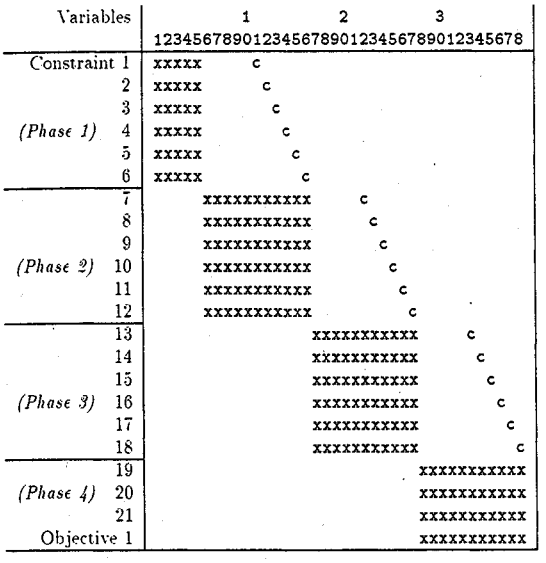

Fig. 1 Linear tangent steering multiple shooting Jacobian sparsity pattern; direct approach: 4 × 20, 80 nonzero elements, multiple shooting: 22 × 38, 224 nonzero elements ($c=-1$, x = general nonzero).

Table 3 Optimization results – linear tangent steering

Item	Four-phase		Seven-phase	
	Dir.	M.S.	Dir.	M.S.
No. opt. requested FE[a]	390	551	129	226
No. Jacobian evaluation	27	31	16	16
No. FE per Jacobian	40	22	42	14
No. FE for Jacobians	1080	682	672	224
Total FE	1470	1233	470	869
% FE for Jacobians	73.5	55.3	83.9	49.8
FE time,	8044.0	1719.6	3794.4	312.6
Average seconds per FE	5.473	1.395	4.737	0.695
Opt. time, s	18.6	105.3	9.2	70.1
Total time, s	8062.6	1824.9	3803.6	382.7
% opt. time	0.2	5.8	0.2	18.3
No. of processors	1	4	1	7

[a]Function evaluation.

number of general nonzero elements in any row, the number of index sets needed to compute the matrix by perturbations is determined to be 11.

Table 3 contains the optimization summary for both the direct approach to the problem and the multiple shooting formulation. As noted in the preceding two cases, the multiple shooting formulation required more (1.4 times) function calls than the direct approach, and a similar number (+4) of Jacobian evaluations.

This problem demonstrates one of the strengths of the multiple shooting approach in taking advantage of sparse index sets when computing the Jacobian matrix by perturbations. For this problem the direct approach requires 40 perturbations in order to generate the central difference derivatives for the Jacobian at each gradient evaluation. On the other hand, by breaking up the trajectory into independent phases, a maximum of 11 index sets (22 perturbations) are required to generate the corresponding multiple shooting Jacobian. This is because the maximum number of variables introduced on any phase is just the five linear tangent steering coefficients plus the six state variables. In fact, 11 index sets would suffice regardless of the number of steering phases. Even the use of partial trajectories,[2] where only that part of the trajectory occurring after the introduction of the variable is simulated, would yield a growth in the time spent computing the Jacobian that is proportional to the number of phases. In fact, if we

Table 4 Speedups – linear tangent steering

No. CPU	Time, s	Relative speed	Relative util.	Total speed
Four-phase direct				
1	8062.6	1.00	1.000	1.00
4	3630.8	2.22	0.555	2.22
8[a]	2891.9	2.79	0.349	2.79
40[a]	2299.6	3.51	0.088	3.51
Four-phase multiple shooting				
1[a]	6853.5	1.00	1.000	1.18
4	1824.9	3.76	0.940	4.42
8[a]	1349.6	5.08	0.635	5.97
88[a]	917.2	7.47	0.085	8.79
Seven-phase direct				
1	3803.6	1.00	1.000	1.00
7[a]	1075.0	3.54	0.506	3.54
14[a]	847.6	4.49	0.321	4.49
42[a]	696.1	5.46	0.130	5.46
Seven-phase multiple shooting				
1[a]	2201.8	1.00	1.000	1.73
7	382.7	5.75	0.821	9.94
14[a]	305.0	7.22	0.516	12.47
98[a]	238.3	9.24	0.094	15.96

[a]Estimates.

define T as the computation time for a full trajectory, η as the number of equal length phases, and ν_k as the number of variables in phase k, then the time required to compute a central difference Jacobian using the direct method is just

$$T_d = 2T \sum_{k=1}^{\eta} \nu_k = 2TN \tag{56}$$

When the partial trajectory mechanism is employed, the time required for a Jacobian is

$$T_p = 2T \sum_{k=1}^{\eta} \nu_k \frac{\eta - k + 1}{\eta} \tag{57}$$

In contrast the time required for a Jacobian evaluation using sparse differences is given by

$$T_s = 2T \max_k (\nu_k) \tag{58}$$

In general, both sparse differences and partial trajectories require less computation than the direct method, that is, $T_p \leq T_d$ and $T_s \leq T_d$. Furthermore, for most applications sparse differences are cheaper than partial trajectories, i.e., $T_s \leq T_p$. In this example, the 18 fewer trajectories that are required by the multiple shooting formulation on each Jacobian evaluation more than compensate for the extra function evaluations that are required to solve the larger constrained problem. This leads to fewer function evaluations being required by the multiple shooting formulation, and the conclusion that this method is more efficient, even on a single processor, than the direct method as is shown in Tables 3 and 4.

D. Coplanar Linear Tangent Steering

The final example is based on increasing the number of steering arcs from four to seven, and performing the maneuver in the equatorial plane with no plane change. Because the transfer is coplanar, there are only two coordinates describing position and two for velocity instead of three each. Thus, the addition of new phases adds only four variables and constraints per phase instead of six. Also, by doing no plane change, the number of linear tangent coefficients is reduced to three per phase. The direct formulation of the problem requires 21 variables and two constraints, while the multiple

shooting formulation requires 45 variables and 26 constraints. However, the multiple shooting formulation requires only seven index sets.

Tables 3 and 4 demonstrate results for an initial guess very near the solution, such as one might encounter for a real time guidance application. The multiple shooting formulation benefits greatly because the Jacobian matrix for the direct formulation requires 42 perturbations, or three times as many as the 14 required by multiple shooting.

VIII. Summary and Conclusions

The computational expense of a trajectory optimization problem has traditionally been dominated by two factors, namely the cost of simulating a single trajectory and the cost of computing gradient information for use by an optimization algorithm. The cost of a single trajectory is reduced by breaking the trajectory into "phases" that can be simulated in parallel on different arithmetic processors. When a trajectory is modeled as a sequence of phases, the Jacobian information needed for optimization is characterized by a simple sparsity structure. A sparse finite difference technique permits computing gradients with respect to many variables on a single perturbation. The total number of perturbed trajectories is equal to the number of index sets. On a parallel processor the perturbed trajectories for each index set can be performed independently. When sparse differences are combined with the decomposition of a trajectory into phases and executed on a parallel processor, it is possible to reduce the computational cost of 1) a single trajectory to the cost of the longest phase and 2) a Jacobian evaluation to the cost of the longest phase. To completely realize the savings one must implement the procedure on a computer with enough arithmetic processors to perform the calculations. Furthermore, the computational expense of the trajectory must dominate the computational expense of the optimization algorithm itself. Two conclusions appear obvious: 1) computationally expensive trajectory simulations will benefit the most from parallelization, and 2) very large applications will require parallelization of the optimization algorithm itself in order that its overhead does not become dominant.

Four orbit transfer problems have been used to explore the potential speedup when obtaining solutions with a parallel, multiple shooting approach. Speedup factors have ranged from 1.9 to 9.9 using between four and seven processors. In three of four cases the multiple shooting approach made more efficient use of the available processors than using them solely during the computation of Jacobian matrices. The use of extra processors in computing the Jacobian will always speed up the problem solution over the single processor case for both the direct and multiple shooting methods. Larger speedup factors seem to be limited primarily by the growth in the optimization overhead due to the larger size of the optimization problem associated with the multiple shooting technique. This is especially the case in the constraint solving portion of the optimization code.

Use of sparse differences to generate the Jacobian matrices for the multiple shooting formulation has drastically reduced the number of perturbations required in some cases. This in turn has allowed the multiple shooting formulation to outperform the engineering formulation without the use of parallel processors on these cases.

The computational results presented do not exploit sparsity and/or parallel processing in the nonlinear programming algorithm itself. Use of the sparsity information directly in the optimization algorithm should reduce the growth rate of the overhead by making it proportional to the number of nonzero elements in the Jacobian matrix rather than the total number of elements. Parallel computation in the optimization algorithm could be exploited in the multiple shooting formulation with no increase in processors since the processors used to compute a parallel trajectory are idle during the execution of the optimization.

References

[1]Brauer, G. L., Cornick, D. E., and Stevenson, R., "Capabilities and Applications of the Program to Optimize Simulated Trajectories (POST)," NASA CR-2770, Feb. 1977.

[2]Meder, D. S., and Searcy, J. L., "Generalized Trajectory Simulation (GTS)," SAMSO-TR-75-255, Vol. 1-5, Jan. 1975.

[3]Hargraves, C. R., and Paris, S. W., "Direct Trajectory Optimization Using Nonlinear Programming and Collocation," *Journal of Guidance, Control, and Dynamics,* Vol. 10, No. 4, 1987, pp. 338-342.

[4]Bauer, T., Betts, J., Hallman, W., Huffman, W., and Zondervan, K., "Solving the Optimal Control Problem Using a Nonlinear Programming Technique Part 2: Optimal Shuttle Ascent Trajectories," *Proceedings of the AIAA/AAS Astrodynamics Conference,* AIAA, New York, Aug. 1984.

[5]Betts, J. T., Bauer, T., Huffman, W., and Zondervan, K., "Solving the Optimal Control Problem Using a Nonlinear Programming Technique Part 1: General Formulation," *Proceedings of the AIAA/AAS Astrodynamics Conference,* AIAA, New York, Aug. 1984.

[6]Betts, J. T., "Sparse Jacobian Updates in the Collocation Method for Optimal Control Problems," *Journal of Guidance, Control, and Dynamics,* Vol. 13, No. 3, June 1990, pp. 409-415.

[7]Zondervan, K. P., Bauer, T., Betts, J., and Huffman, W., "Solving the Optimal Control Problem Using a Nonlinear Programming Technique Part 3: Optimal Shuttle Reentry Trajectories," *Proceedings of the AIAA/AAS Astrodynamics Conference,* AIAA, New York, Aug. 1984.

[8]Keller, H. B., *Numerical Methods for Two-Point Boundary Value Problems,* Blaisdell, London, England, UK, 1968.

[9]Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis,* Springer, New York, 1980.

[10]Betts, J. T. "Nonlinear Programming Algorithm, NLP2," Aerospace Corp., Los Angeles, CA, TOR-0088(3464-06)-1, June 1987.

[11]Huffman, W. P., "An Analytic J_2 Propagation Method for Low Earth Orbits," Aerospace Corp., Los Angeles, CA, IOC A81-5422.5-23, Nov. 1981.

[12]Gear, C. W., *Numerical Initial Value Problems in Ordinary Differential Equations,* Prentice-Hall, Englewood Cliffs, NJ, 1971.

[13]Betts, J. T., "Frontiers in Engineering Optimization," *Journal of Mechanisms, Transmissions, and Automation in Design,* Vol. 105, June 1983, pp. 151-154.

[14]Brenan, K., "Engineering Methods Report: The Design and Development of a Consistent Integrator/Interpolator for Optimization Problems," Aerospace Corp., Los Angeles, CA, ATM 88(9975)-52, Sept. 1988.

[15]Gear, C. W., and Vu, T. V., "Smooth Numerical Solutions of Ordinary Differential Equations," *Proceedings of the Workshop on Numerical Treatment of Inverse Problems for Differential and Integral Equations,* 1982.

[16]Vu, T. V., "Numerical Methods for Smooth Solutions of Ordinary Differential Equations," Dept. of Computer Science, Univ. of Illinois, Urbana, IL, Rept. UIUCDCS-R-83-1130, May 1983.

[17]Betts, J. T., "Optimal Three-Burn Orbit Transfer," *AIAA Journal,* Vol. 15, No. 6, 1977, pp. 861-864.

[18]Curtis, A., Powell, M. J. D., and Reid, J. K., "On the Estimation of Sparse Jacobian Matrices," *Journal of Institutional Mathematical Applications,* Vol. 13, 1974, pp. 117-120.

[19]Coleman, T. F., and Moré, J. J., "Estimation of Sparse Jacobian Matrices and Graph Coloring Problems," *SIAM Journal of Numerical Analysis,* Vol. 20, 1983, pp. 187-209.