Real-Time Trajectory Optimization on Parallel Processors

(NASA Research Grant NAG-1-1009)

Final Report

July 10, 1993

Prepared by Mark L. Psiaki,

Principal Investigator

Cornell University

Ithaca, N.Y. 14853-7501

Summary

A parallel algorithm has been developed for rapidly solving trajectory optimization problems. The goal of the work has been to develop an algorithm that is suitable to do real-time, on-line optimal guidance through repeated solution of a trajectory optimization problem. The algorithm has been developed on an INTEL iPSC/860 message passing parallel processor. It uses a zero-order-hold discretization of a continuous-time problem and solves the resulting nonlinear programming problem using a custom-designed augmented Lagrangian nonlinear programming algorithm. The algorithm achieves parallelism of function, derivative, and search direction calculations through the principle of domain decomposition applied along the time axis. It has been encoded and tested on 3 example problems, the Goddard problem, the acceleration-limited, planar minimum-time to the origin problem, and a National Aerospace Plane minimum-fuel ascent guidance problem. Execution times as fast as 118 sec of wall clock time have been achieved for a 128-stage Goddard problem solved on 32 processors. A 32-stage minimum-time problem has been solved in 151 sec on 32 processors. A 32-stage National Aerospace Plane problem required 2 hours when solved on 32 processors. A speed-up factor of 7.2 has been achieved by using 32-nodes instead of 1-node to solve a 64-stage Goddard problem.

1. Introduction

1.1 Review of Project Objective

This grant's goal has been to achieve nonlinear optimal feedback control through repeated, on-line solution of trajectory optimization problems in real-time. Such control could be used for aerospace vehicle guidance, such as National Aerospace Plane (NASP) ascent guidance. The primary effort of this grant has been to develop a fast trajectory optimizer that can solve problems which include inequality constraints. The optimizer should be able to update solutions about once every 5 seconds. General parallel algorithms have been developed to try to meet this goal. They have been developed and tested on INTEL iPSC/2 and iPSC/860 distributed-memory parallel processors, but they are applicable to any highly-parallel distributed-memory machine (e.g. the INTEL Touchstone system or Transputer-based systems).

1.2 Summary of 3 1/2-Year Grant Activities

The activities under this grant can be broken into 5 main categories:

1. Development of a parallel solver for dynamic quadratic programs (QPs).
2. Development of a robust serial nonlinear programming algorithm (NP).
3. Development of a specification and interface for trajectory optimization problem encoding, including approximation of continuous-time phases by discrete-time phases.
4. Integration of the parallel QP algorithm with the serial NP algorithm to produce a parallel trajectory optimization algorithm.
5. Modelling, encoding, and testing of example trajectory optimization problems.

The first year of the project was devoted primarily to developing a parallel dynamic quadratic programming algorithm. The second year's effort was split between developing a refined parallel QP algorithm and developing a robust serial NP algorithm. During these first two years Kihong

Park, the project's graduate research assistant, concentrated on the parallel QP work. Prof. Mark Psiaki, the project's principal investigator, helped guide Park, and he developed the nonlinear programming algorithm.

Park spent the final year and a half of the project working to parallelize the serial nonlinear programming algorithm. This included integration of it with the parallel QP algorithm, which calculates NP search directions, and with parallel function, gradient, and Jacobian software that was developed by Psiaki.

Psiaki spent the last year and a half of the grant developing the specifications and software that allow the submission of different problems to the main algorithm. He also spent time modeling and encoding three test problems: a Goddard problem, a linear tangent steering minimum-time problem, and a NASP minimum-fuel ascent guidance problem.

1.3 Outline of Report

The remainder of this report is divided into 6 Sections plus conclusions and 5 Appendices. Section 2 summarizes the results of work that has been done to parallelize the computation of trajectory optimization search directions based on first and second gradient information. Appendix A contains two papers that give the details of this work. Section 3 summarizes the nonlinear programming algorithm that has been developed to serve as the heart of the trajectory optimization algorithm, and Appendix B contains a paper that describes the algorithm in more detail. Section 4 gives an overview of how the nonlinear programming algorithm has been augmented with parallel derivative calculations and integrated with the parallel search direction algorithm to yield the final parallel nonlinear trajectory optimization algorithm. Appendix C contains a paper that fills in some of the algorithm details omitted in Section 4.

Section 5 presents three example trajectory optimization problems, the Goddard sounding rocket problem, the minimum-time, acceleration-limited particle-in-a-plane problem, and a NASP ascent guidance problem. Section 6 presents parallel computational results for these three problems. Section 7 makes observations about the algorithm based upon computational experience, and it suggests possible improvements. Section 8 presents the conclusions.

Appendix D presents a specification document that explains how to encode of a trajectory optimization problem so that it can be linked to the parallel trajectory optimization algorithm. The code that models the planar minimum-time problem is included at the end of Appendix D as an example of how to conform to the problem encoding specification.

Appendix E outlines the theory of a multi-dimensional spline technique that has been developed as a by-product of this research. Multi-dimensional splines have been used on the NASP problem, and the procedure on Appendix E presents a relatively simple technique to carry out rapid, on-line multi-dimensional spline calculations that have continuous second partial derivatives of the interpolated function.

2. Dynamic Quadratic Programming/Search Direction Calculation on a Parallel Processor

This part of the effort concentrated on developing an efficient parallel solver for a problem of the form:

$$\text{find:} \qquad \boldsymbol{\mathcal{X}} = \left[\mathbf{x}_0^T, \mathbf{x}_1^T, \mathbf{x}_2^T, \ldots, \mathbf{x}_N^T\right]^T \tag{1a}$$

$$\text{to minimize:} \qquad J = \sum_{k=0}^{N} \left\{ \tfrac{1}{2} \mathbf{x}_k^T \mathbf{H}_k \mathbf{x}_k + \mathbf{g}_k^T \mathbf{x}_k \right\} \tag{1b}$$

$$\text{subject to:} \qquad \mathbf{E}_k \mathbf{x}_k + \mathbf{e}_k + \mathbf{F}_{k+1} \mathbf{x}_{k+1} = \mathbf{0} \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{1c}$$

$$\mathbf{D}_k \mathbf{x}_k + \mathbf{d}_k = \mathbf{0} \qquad \text{for } k = 0 \ldots N \tag{1d}$$

The solution vector at a given stage, $\mathbf{x}_k$, is usually a combination of a state vector, $\mathbf{x}_k$, and a control vector, $\mathbf{u}_k$. The vector could also include slack variables if required by the NP algorithm that uses the QP solver.

The constraints in eq. (1c) are dynamic constraints that link stages. Not that the form is general enough to admit the standard linear difference equation model of a linear time-varying discrete-time system. The nonhomogeneous term in eq. (1c), $\mathbf{e}_k$, is included to make the QP

algorithm compatible with NP trajectory optimization algorithms that do not satisfy the dynamics at intermediate stages of the solution process.

The auxiliary constraints in eq. (1d) allow the modeling of state and control constraints that apply point-wise in time, such as bounds on the throttle setting or on the dynamic pressure. Even though the constraints in. eq. (1d) are equality constraints, they are useful for dealing with inequality constraints if an active set strategy is employed. An active set strategy assumes that some of the inequality constraints are active and others are inactive. The active inequalities are enforced as equalities, and the inactive constraints are ignored. Of course, logic must be employed to make changes to the active set assumptions as needed. In the present effort, this logic is part of the overall NP algorithm, not part of the QP algorithm.

The QP algorithm that solves eqs. (1a)-(1d) gets used by the nonlinear trajectory optimization algorithm to determine a search direction in state and control time-history space. It is used by the NP algorithm in a sequential quadratic programs sort of approach: the constraints are linearized about the current solution estimate and the Lagrangian function (cost plus multipliers times constraints) is approximated by a quadratic expansion about the current solution estimate. This yields a quadratic program, which is solved for a search direction. The solution is modified by moving along this search direction in state and control time history space. A merit function is used to determine the step length. This process ensures global convergence in many cases and quadratic convergence near the solution, where the quadratic program is a good approximation of the original nonlinear program.

A nonlinear trajectory optimization algorithm can call for hundreds or thousands of QP solutions during one solution for an optimal trajectory. Therefore, it is extremely important that the QP be solved rapidly. In order to accomplish this, the domain decomposition approach has been used to parallelize the QP solution procedure.

The basic idea of domain decomposition is to split the problem into smaller problems and partially solve the smaller problems. The unsolved parts of the problem are then joined into

progressively bigger portions that admit more partial solution to be accomplished until the entire problem is finally solved.

The domains that are split to solve eqs. (1a)-(1d) are time domains. The problem is split into a number of groups of stages. Each group is a set of contiguous stages. Any such set of stages can be partially optimized independently of any other such set if one assumes that the state vector on the boundary of any two such sets must be held fixed during the partial optimization process. After these partial optimizations, the remaining unsolved parts of contiguous sets of stages are aggregated into a smaller number of larger sets of contiguous stages. This frees some state vectors that had been on set boundaries, allowing them to be optimized in the next partial optimization cycle.

The final parallel QP algorithm achieves good speed-up when run on a parallel processor. It can solve an N-stage dynamic QP problem on p (<N) processors in wall clock time that scales as $(N/p) + \mathrm{Log}_2(p)$. Two versions of the algorithm and their computational timing results are described in detail in the two papers that have been included as Appendix A of this report. The second paper in the Appendix A describes the algorithm that actually has been used as part of the overall nonlinear programming algorithm.

3. Nonlinear Programming Algorithm

Once approximated in a discrete-time form, most trajectory optimization problems can be stated in a general nonlinear programming form:

find:	$\boldsymbol{x}$	(2a)
to minimize:	$J(\boldsymbol{x})$	(2b)
subject to:	$\mathbf{c}_e(\boldsymbol{x}) = \mathbf{0}$	(2c)
	$\mathbf{c}_i(\boldsymbol{x}) \leq \mathbf{0}$	(2d)

where $\boldsymbol{x}$ is a large vector that includes the entire discrete-time state and control time histories, the $J(\boldsymbol{x})$ cost function is a summation over stage-wise costs, $\mathbf{c}_e(\boldsymbol{x})$ is a large vector of equality constraint functions that includes, among other things, the dynamic difference equations, and the

$\mathbf{c}_i(\mathbf{x})$ inequality constraints may include state and control auxiliary constraints at each stage. If N is the number of discrete-time time steps (or stages), then the dimensions of the vectors $\mathbf{x}$, $\mathbf{c}_e(\mathbf{x})$, and $\mathbf{c}_i(\mathbf{x})$ include a factor of N.

While an efficient trajectory optimization algorithm must take advantage of special problem structure that is not apparent in eqs. (2a)-(2d), a good algorithm must also employ generic nonlinear programming techniques that apply to problems of this general form. Therefore, an effort has been made to develop a good algorithm for solving such problems.

The basic NP algorithm that is the heart of the a real-time trajectory optimizer must have several important characteristics. First, it must be robustly convergent. In other words, it must be very likely to converge to a feasible local minimum even from a poor first guess. Second, it must have fast local convergence: it must rapidly determine the solution when it is near the solution. Third, it must have good global convergence speed. It must progress rapidly from being "far away" from the solution to being "near" the solution. The above requirements constitute a "tall order" for any general NP algorithm.

The algorithm developed under this grant is a sequential quadratic programs-type implementation of the augmented Lagrangian nonlinear programming algorithm. It uses a shifted penalty function that is capable of achieving exact satisfaction of equality and inequality constraints without requiring infinite penalty weights. Several features have been added to the algorithm in hopes of speeding convergence. One is the use of constraint curvature directly in the quadratic sub-problem. Another is the use of a special constrained form of the QP subproblem that allows the use of large penalty weights, which speed local convergence. The box trust region technique has been incorporated into the algorithm in order to guaranteed convergence to a local minimum of the augmented Lagrangian function, which translates into global convergence to a problem solution as long as the algorithm does not converge to an infeasible point near a non-zero local minimum of the norm of the constraint violations.

The algorithm was first encoded and tested in MATLAB as a serial algorithm. In the paper that has been included as Appendix B of this report, the NP algorithm is described, and it is

compared with NPSOL on several test problems. It performs well on the test problems, and it has more robust convergence than NPSOL version 4.

Another, slightly different version of this algorithm has been developed which is not described in Appendix B. The modified version has three principal differences from the algorithm described in appendix B. All of these have to do with how the algorithm's "inner" loop solves a quadratically-constrained quadratic program. First, the alternate algorithm uses curved line searches on every search step. Second, it does an inexact one-variable minimization of the augmented Lagrangian function during a parabolic line search using a golden section search with a secant-method-type acceleration of terminal convergence. Third, it uses a different curvature correction than that described in eqs. (11a)-(11c) of the Appendix B. Instead of finding the correction that minimizes $\frac{1}{2}\delta\tilde{\mathbf{x}}^T\delta\tilde{\mathbf{x}}$ as in eq. (11a), it finds the correction that minimizes $\frac{1}{2}\delta\tilde{\mathbf{x}}^T\mathbf{H}\delta\tilde{\mathbf{x}}$ where $\mathbf{H}$ is computed as in eq. (9) of Appendix B using the multipliers from eq. (12) of Appendix B. This $\mathbf{H}$ gets modified by adding positive numbers to its diagonal if it would otherwise yield an indefinite projected Hessian during the solution of the modified version of Appendix B's eqs. (11a)-(11c).

This modified algorithm also has been tried on the test problems described in Appendix B. It yielded significantly more rapid convergence on most of the problems by reducing the total number of line searches required to solve the several QPs that arise during a given NP run.

4. Parallel Nonlinear Trajectory Optimization Algorithm

4.1 Algorithm Overview

The parallel nonlinear trajectory optimization algorithm is a special version of the nonlinear programming algorithm that is described in Section 3 and Appendix B. The special version exploits parallelism and the special dynamic programming problem structure in order to speed up the algorithm when solving trajectory optimization problems.

The algorithm has been encoded to solve continuous-time problems of the form

$$\text{find:} \quad \boldsymbol{u}(t) \text{ and } \boldsymbol{x}(t) \text{ for } t_0 \le t \le t_f \tag{3a}$$

$$\text{to minimize:} \quad J = \int_{t_0}^{t_f} L[\boldsymbol{x}(t),\boldsymbol{u}(t),t]\, dt + V[\boldsymbol{x}(t_f)] \tag{3b}$$

$$\text{subject to:} \quad \boldsymbol{x}(t_0) \text{ given} \tag{3c}$$

$$\dot{\boldsymbol{x}} = \boldsymbol{f}[\boldsymbol{x}(t),\boldsymbol{u}(t),t] \tag{3d}$$

$$\boldsymbol{a}_e[\boldsymbol{x}(t),\boldsymbol{u}(t),t] = \boldsymbol{0} \tag{3e}$$

$$\boldsymbol{a}_i[\boldsymbol{x}(t),\boldsymbol{u}(t),t] \le \boldsymbol{0} \tag{3f}$$

$$\boldsymbol{a}_{e_f}[\boldsymbol{x}(t_f)] = \boldsymbol{0} \tag{3g}$$

$$\boldsymbol{a}_{i_f}[\boldsymbol{x}(t_f)] \le \boldsymbol{0} \tag{3h}$$

which it first approximates as a discrete-time problem through a zero-order-hold approximation of the control time history. The algorithm also has the option of directly solving discrete-time problems of the form

$$\text{find:} \quad \boldsymbol{\chi} = \left[\mathbf{u}_0^T, \mathbf{x}_1^T, \mathbf{u}_1^T, \mathbf{x}_2^T, \ldots, \mathbf{u}_{N-1}^T, \mathbf{x}_N^T \right]^T \tag{4a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{N-1} L_k(\mathbf{x}_k,\mathbf{u}_k) + V[\boldsymbol{x}_N] \tag{4b}$$

$$\text{subject to:} \quad \mathbf{x}_0 \text{ given} \tag{4c}$$

$$\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k) \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{4d}$$

$$\mathbf{a}_{e_k}(\mathbf{x}_k,\mathbf{u}_k) = \mathbf{0} \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{4e}$$

$$\mathbf{a}_{i_k}(\mathbf{x}_k,\mathbf{u}_k) \le \mathbf{0} \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{4f}$$

$$\mathbf{a}_{e_N}(\mathbf{x}_N) = \mathbf{0} \tag{4g}$$

$$\mathbf{a}_{i_N}(\mathbf{x}_N) \le \mathbf{0} \tag{4h}$$

The algorithm also can solve mixed discrete-time/continuous-time problems in which any given phase is either totally discrete-time or totally continuous time. Discrete-time phases are allowed to begin and end with different numbers of state vector elements. Through clever problem modeling

tricks, this feature allows for optimization of initial conditions. Other modeling tricks allow the code to solve free-end-time problems.

The specialization of Section 3's basic NP algorithm takes two main forms. First, the NP algorithm's linear algebra is performed using the parallel dynamic quadratic programming algorithm that is the subject of Section 2 and Appendix A. Second, the cost and constraint functions (including the dynamics model), the cost gradients, the constraint Jacobians, and the cost and constraint Hessians are evaluated in parallel.

Parallel evaluation of problem functions and their derivatives is possible because the problem can be split in a stage-wise manner. The cost, the differential equations, and their derivatives at time step 2 and at time step 56 can be evaluated simultaneously because they do not affect each other at intermediate solution estimates. One feature of the basic NP algorithm that contributes to this separability is its permission of constraint violations at intermediate solutions. In particular, the dynamics need not be exactly satisfied until the final solution is reached, which is why stage-2 functions and derivatives are independent of stage-56 functions and derivatives.

Of course, functions and their derivatives at different stages are related at the final solution. This relationship is achieved by the NP algorithm during its search for the problem solution. The algorithm's mechanism for achieving the necessary relationships is its search direction computation via solution of a dynamic QP. The dynamic QP solver transmits function and derivative information between stages.

The algorithm has been fully encoded and tested using a 32-node INTEL iPSC/860 message-passing parallel processor. The results of this testing are described below in Section 6. The algorithm itself is described in more detail in the paper contained in Appendix C.

4.2 Problem Encoding

The algorithm has been encoded to solve problems that are modeled via two FORTRAN 77 subroutines, each of which has a pre-specified argument list and functionality. One subroutine models the problem's cost function and dynamics equations. The other subroutine models any auxiliary constraints such as bounds on state or control variables. The algorithm needs the names

of these user-defined subroutines along with user-defined data that gets input to the main trajectory optimization algorithm via several array and scalar arguments.

The necessary input data and subroutines are described in a specification document that has been included as the first part of Appendix D. This document makes it possible for people to use the parallel trajectory optimization algorithm even though they are not intimately familiar with its inner workings. Also included in Appendix D is the code that has been used to model the minimum-time to the origin problem, which is described below in Section 5.2.

As proof of the usefulness of the problem modeling specification, Prof. Psiaki was able to do all of the problem modeling and problem encoding for the examples described below in Section 5, despite the fact that he never saw any of the parallel trajectory optimization code. He was able to encode the problems simply by adhering to the specification in Appendix D.

5. Modeling of Three Example Trajectory Optimization Problems

5.1 Goddard Problem (Maximizing the Final Altitude of a Sounding Rocket)

The following problem maximizes the terminal altitude of a sounding rocket in flight over a spherical, non-rotating Earth [1]:

$$\text{find:} \quad t_f \text{ and } T(t) \text{ for } 0 \le t \le t_f \tag{5a}$$

$$\text{to minimize:} \quad J = -h(t_f) \tag{5b}$$

$$\text{subject to} \quad \dot{h} = v \tag{5c}$$

$$\dot{v} = \frac{T - D(v,h)}{m} - \frac{1}{h^2} \tag{5d}$$

$$\dot{m} = -\frac{T}{0.5} \tag{5e}$$

$$h(0) = 1,\ v(0) = 0,\ m(0) = 1 \tag{5f}$$

$$0 \le T(t) \le 3.5 \tag{5g}$$

$$m(t_f) = 0.6 \tag{5h}$$

where h is the distance of the rocket from the center of the Earth measured in Earth radii, v is the velocity in Earth radii per Herg (2π Hergs is the period of a circular orbit at the Earth's surface), m

is the rocket mass nondimensionalized by its initial mass, T is the rocket thrust nondimensionalized by the rocket's initial weight at the Earth's surface, D(v,h) is the velocity- and altitude-dependent aerodynamic drag, and 0.5 is the exhaust velocity of the burned rocket fuel. Note that the two inequalities in (5g) put minimum and maximum limits on the control, T. The drag function assumes a constant, zero-lift drag coefficient and an exponential atmospheric density

$$D(v,h) = 6{,}200\, e^{500(1-h)}\, v\, |v|\, C_D \tag{6}$$

where $C_D = 0.05$.

In order to model this free-end-time problem in a fixed-end-time format, the problem has been split into two phases, an extra state variable has been added, and an artificial problem time has been introduced whose fixed terminal value is 1. Call the artificial problem time τ. Then actual time is $t = t_f\, \tau$.

Given an N+1-stage discrete-time approximation of the problem, stages 0 to N with stage N being the terminal stage, the problem has been re-modeled in the form

find: $T(\tau)$ for $0 \le \tau \le 1$ and u_2 for $0 \le \tau \le (1/N)$ (7a)

to minimize: $J = -h(1)$ (7b)

subject to

$$\frac{dh}{d\tau} = \begin{cases} u_2\, v & \text{for } 0 \le \tau < (1/N) \\ x_4\, v & \text{for } (1/N) \le \tau \le 1 \end{cases} \tag{7c}$$

$$\frac{dv}{d\tau} = \begin{cases} u_2 \left[\dfrac{T - D(v,h)}{m} - \dfrac{1}{h^2}\right] & \text{for } 0 \le \tau < (1/N) \\ x_4 \left[\dfrac{T - D(v,h)}{m} - \dfrac{1}{h^2}\right] & \text{for } (1/N) \le \tau \le 1 \end{cases} \tag{7d}$$

$$\frac{dm}{d\tau} = \begin{cases} u_2 \left[-\dfrac{T}{0.5}\right] & \text{for } 0 \le \tau < (1/N) \\ x_4 \left[-\dfrac{T}{0.5}\right] & \text{for } (1/N) \le \tau \le 1 \end{cases} \tag{7e}$$

$$\frac{dx_4}{d\tau} = \begin{cases} N\, u_2 & \text{for } 0 \le \tau < (1/N) \\ 0 & \text{for } (1/N) \le \tau \le 1 \end{cases} \tag{7f}$$

$$h(0) = 1,\ v(0) = 0,\ m(0) = 1 \tag{7g}$$

$$0 \le T(\tau) \le 3.5 \tag{7h}$$

$$m(1) = 0.6 \tag{7i}$$

One can understand the relationship between this formulation and the original formulation by noting that $x_4(\tau) = t_f$ for $(1/N) \leq \tau \leq 1$. Due to the zero-order-hold approximation of the control time history and the N+1-stage approximation, the algorithm assumes that $u_2(\tau)$ is constant over the interval $0 \leq \tau < (1/N)$. Therefore, $u_2(\tau)$ also equals t_f on this interval. Thus, all of the differential equations are re-scaled by t_f, which correctly accounts for the modified independent variable, τ.

Once in this problem form, it is straight-forward to encode the problem in accordance with the specification in Appendix D. This has been done, and computational results are presented in Section 6.1 below. In all of the runs reported in Section 6.1, the first guess of the optimal trajectory is approximately

$$t_f = 0.2 = x_4 = u_2 \tag{8a}$$

$$T(t) = \begin{cases} 2 & \text{for} \quad 0 \leq t < 0.1 \\ 0 & \text{for} \quad 0.1 \leq t \leq 0.2 \end{cases} \tag{8b}$$

$$h(t) = \begin{cases} 1 + 0.5t^2 & \text{for} \quad 0 \leq t < 0.1 \\ 0.99 + 0.2t - 0.5t^2 & \text{for} \quad 0.1 \leq t \leq 0.2 \end{cases} \tag{8c}$$

$$v(t) = \begin{cases} t & \text{for} \quad 0 \leq t < 0.1 \\ 0.2 - t & \text{for} \quad 0.1 \leq t \leq 0.2 \end{cases} \tag{8d}$$

$$m(t) = \begin{cases} 1 - 4t & \text{for} \quad 0 \leq t < 0.1 \\ 0.6 & \text{for} \quad 0.1 \leq t \leq 0.2 \end{cases} \tag{8e}$$

Which corresponds to a flat Earth, zero drag assumption and a control policy that keeps T at 2 until all of the fuel is used up. Note that this first guess does not satisfy the dynamic constraints because it neglects the drag term and models $1/h^2$ as a constant.

5.2 Acceleration-Limited Minimum Time to the Origin

The following problem minimizes the time to bring a particle to rest at the origin in the plane subject to a magnitude limit on its acceleration:

$$\text{find:} \qquad t_f \text{ and } \mathbf{u}(t) \text{ for } 0 \le t \le t_f \tag{9a}$$

$$\text{to minimize:} \quad J = t_f \tag{9b}$$

$$\text{subject to} \quad \dot{\mathbf{x}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{u} \tag{9c}$$

$$\mathbf{x}(0) \ (\neq \mathbf{0}) \text{ given} \tag{9d}$$

$$\mathbf{u}^T(t)\mathbf{u}(t) \le 1.0 \tag{9e}$$

$$\mathbf{x}(t_f) = \mathbf{0} \tag{9f}$$

where the first two elements of the state vector $\mathbf{x}$ are positions, its third and fourth elements are velocities, and the two components of $\mathbf{u}$ are accelerations.

The solution to this problem is the well-known bilinear tangent steering control law [2]:

$$\mathbf{u}(t) = \begin{bmatrix} \cos \theta(t) \\ \sin \theta(t) \end{bmatrix} \tag{10}$$

where $\theta(t)$ satisfies an equation of the form

$$\tan \theta(t) = \frac{\beta_1 + \beta_2 t}{\beta_3 + \beta_4 t} \tag{11}$$

with unknown constants β_1, β_2, β_3, and β_4. These constants are unique up to a common scale factor, and they must be numerically determined simultaneously with t_f by solving a set of algebraic equations that enforce the terminal constraint, eq. (9f).

An easy way to generate known solutions to this problem is to pick β_1, β_2, β_3, β_4, and t_f and integrate backwards in time from $\mathbf{x}(t_f) = \mathbf{0}$ for t_f seconds to generate $\mathbf{x}(0)$. This has been done to generate an example problem for testing the trajectory optimization algorithm described in this report. The resulting initial condition is

$$\mathbf{x}(0) = \begin{bmatrix} -73.605383 \\ -35.125486 \\ 2.949018 \\ 0.430989 \end{bmatrix} \tag{12}$$

Several modeling tricks have been used to get this problem into a form suitable for encoding in accordance with the specification in Appendix D. It is desirable to have the acceleration time history be piecewise linear rather than piecewise constant, which is what the zero-order-hold assumption would yield. Therefore, it is necessary to augment the optimization state vector to include the two accelerations and to create a new optimization control vector that consists of the two rates of change of the two accelerations rather than the two accelerations themselves.

As in the Goddard problem of Section 5.1, the state vector is further augmented to allow for the solution of this free-end-time problem with the fixed-end-time algorithm. The new state stores the end time t_f.

A further modeling trick is needed to allow optimization of the initial acceleration states in addition to optimization of the rates of change of the accelerations. This is accomplished by modeling the first stage as a discrete-time stage. It does not take up any actual time, but it initializes the state vector for the remaining continuous-time steps. Thus, for an N+1 stage problem, the first stage, stage 0, is modeled by the following difference equation and constraints (for k = 0):

$$\mathbf{x}_{k+1} = \begin{bmatrix} -73.605383 \\ -35.125486 \\ 2.949018 \\ 0.430989 \\ \mathbf{u}_k \end{bmatrix} \tag{13a}$$

$$u_{k_1}^2 + u_{k_2}^2 \leq 1 \tag{13b}$$

$$0.01 \leq u_{k_3} \tag{13c}$$

where $\mathbf{x}_k$ (not shown) is a zero-dimensional vector, $\mathbf{x}_{k+1}$ is a 7-dimensional vector, and $\mathbf{u}_k$ is a 3-dimensional vector. Elements 1 and 2 of $\mathbf{x}_{k+1}$ are the initial positions, elements 3 and 4 are the

initial velocities, and elements 5 and 6 are the initial accelerations. Notice how elements 5 and 6 of $\mathbf{x}_{k+1}$ are set equal to elements 1 and 2 of $\mathbf{u}_k$, which allows optimization of the initial accelerations.

The third element of $\mathbf{u}_k$ is the problem's final time, $u_{k3} = t_f$, which becomes the 7th element of $\mathbf{x}_{k+1}$ at the end of this stage. As with the Goddard problem, the algorithm works in artificial time τ that has a fixed terminal value of $\tau = 1$. Thus, the relationship between this artificial time and the real problem time is $t = t_f \tau$. The extra constraint in eq. (13c) is added for the practical purpose of preventing the algorithm from thinking that it can run time in reverse.

Stages 1 through N-1 of the problem are modeled by the following continuous-time cost integral, differential equation, and constraints

$$J = \int_0^1 x_7 d\tau \tag{14a}$$

$$\frac{d\mathbf{x}}{d\tau} = x_7 \left\{ \begin{bmatrix} 0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0&0 \\ 0&0 \\ 0&0 \\ 0&0 \\ 1&0 \\ 0&1 \\ 0&0 \end{bmatrix} \mathbf{u} \right\} \tag{14b}$$

$$x_5^2(\tau) + x_6^2(\tau) \leq 1 \tag{14c}$$

$$0.01 \leq x_7 \tag{14d}$$

where the constraint in eq. (14d) is added for practical purposes. Even though eq. (14d) is redundant with eq. (13c) because $dx_7/d\tau = 0$, this constraint is useful to help ensure reasonable intermediate solution estimates during the optimization process when the constraint $dx_7/d\tau = 0$ might be violated.

The terminal stage, stage N, is modeled simply by the terminal constraints on the positions and velocities

$$x_i(\tau=1) = 0 \quad \text{for } i = 1, ..., 4 \tag{15}$$

coupled with one more constraint on the acceleration magnitude

$$x_5^2(\tau=1) + x_6^2(\tau=1) \leq 1 \qquad (16)$$

The code that models this problem has been included at the end of Appendix D as an example of how to apply the problem encoding specification at the beginning of Appendix D. The subroutine called FLMNTM encodes the dynamics and cost functions in eqs. (13a), (14a) and (14b) along with their first and second partial derivatives. The subroutine CMNTM encodes the auxiliary constraint functions in eqs. (13b), (13c), (14c), (14d), (15) and (16) along with their first and second partial derivatives.

The first guess used for this problem takes the form

$$t_f = 2.980345 \qquad (17a)$$

$$x_1(t) = -73.605383 + 2.949018t - 0.494744t^2 \qquad (17b)$$

$$x_2(t) = -35.125486 + 0.430989t - 0.072305t^2 \qquad (17c)$$

$$x_3(t) = 2.949018 - 0.989489t \qquad (17d)$$

$$x_4(t) = 0.430989 - 0.144610t \qquad (17e)$$

$$x_5(t) = -0.989489 \qquad (17f)$$

$$x_6(t) = -0.144610 \qquad (17g)$$

$$x_7(t) = 2.980345 \qquad (17h)$$

$$u_1(t) = 0.0 \qquad (17i)$$

$$u_2(t) = 0.0 \qquad (17j)$$

where the notation used corresponds to the definitions associated with eqs. (14a)-(14d). Basically, this guess puts on the "brakes" to bring the system to rest, $x_3(t_f) = x_4(t_f) = 0$, but it fails to bring the system to the origin, $x_1(t_f) \neq 0$ and $x_2(t_f) \neq 0$. It satisfies the dynamics constraints and the inequality constraint on the acceleration magnitude.

The subroutine MNTMIN in Appendix D sets the initial guess of the optimal trajectory. This subroutine also defines all of the other user-definable optimization algorithm inputs that are called for in the specification at the beginning of Appendix D except for N, the number of the terminal problem stage. The quantity N must already be known by the routine that calls MNTMIN. A main

program that calls the parallel optimization code can first make a call to MNTMIN in order to set up the user-defined quantities.

MNTMIN's output arrays NUVEC, NXVEC, NECVEC, and NICVEC and its output scalars NUMAX, NXMAX, and NCMAX define various problem vector dimensions. The arrays U0 and X0 contain the first guess. The arrays IDT and T further define the problem model by indicating whether a stage is continuous-time or discrete-time and by defining the time associated with each problem stage. The arrays KRK and ISECGR provide signals to the optimization algorithm of which numerical integration routine to use for continuous-time stages and of whether or not analytic second derivatives are available for problem functions. The arrays DELTU and DELTX provide finite difference intervals for approximating any second derivatives that are not provided in analytic form by the user.

5.3 NASP Minimum-Fuel Ascent Guidance Problem

A minimum-fuel NASP ascent guidance problem can be posed to take the vehicle from just after lift-off (10 feet off of the ground at Mach 0.4 with a flight-path angle of 2° and a gross weight 300,000 lb) to a circular orbit at 100 miles altitude using the minimum amount of fuel:

find: t_f and $\alpha(t)$, $\phi_a(t)$, and $\phi_r(t)$ for $0 \le t \le t_f$ (18a)

to minimize: $J = -m(t_f)$ (18b)

subject to $\dot{h} = V \sin\gamma$ (18c)

$$\dot{V} = \frac{T\cos\alpha - D}{m} - 32.174\left(\frac{R_E}{h + R_E}\right)^2 \sin\gamma \quad (18d)$$

$$\dot{\gamma} = \frac{\left[\frac{T\sin\alpha + L}{m} - 32.174\left(\frac{R_E}{h + R_E}\right)^2 \cos\gamma\right]}{V} + \left[\frac{V\cos\gamma}{h + R_E}\right] \quad (18e)$$

$$\dot{m} = -\frac{\bar{q}\, C_{T_a}(\phi_a, M)}{32.174\, I_a(\phi_a, M)} - \frac{15000\,\phi_r}{32.174 \bullet 444.0} \quad (18f)$$

$h(0) = 10$, $V(0) = 446.56$, $\gamma(0) = 0.034907$, and $m(0) = 9324.3$ (18g)

$\bar{q} \le 2000$ (18h)

$2.771\text{x}10^{-8}\sqrt{\rho(h)}\, V^3 \le 400.0$ (18i)

$$-0.01745 \leq \alpha \leq 0.20944 \tag{18j}$$

$$-0.34907 \leq \delta E(\alpha,M) \leq 0.34907 \tag{18k}$$

$$\phi_{a_{min}}(M) \leq \phi_a \leq \phi_{a_{max}}(M) \tag{18l}$$

$$0.0 \leq \phi_r \leq 1.0 \tag{18m}$$

$$-4.0 \leq \left[\frac{T\cos\alpha - D}{32.174\ m}\right] \leq 4.0 \tag{18n}$$

$$\left[\frac{T\sin\alpha + L}{32.174\ m}\right] \leq 4.0 \tag{18o}$$

$$0.0 \leq h \tag{18p}$$

$$h(t_f) = 528000 \tag{18q}$$

$$\gamma(t_f) = 0.0 \tag{18r}$$

$$V(t_f) = 25620.39 \tag{18s}$$

The problem model has four state variables; h is the vehicle altitude above the Earth's surface in ft, V is the vehicle's inertial speed in ft/sec, γ is the flight-path angle in rad., and m is the mass in slugs. Equations (18c)-(18e) model motion in a vertical plane over a spherical, non-rotating Earth. They include the aerodynamic forces, lift L and drag D, the net axial thrust force due to the air-breathing and rocket propulsion systems, T, and the usual $1/(h+R_E)^2$ central gravitational force term. Equation (18f) models the vehicle mass decrease due to fuel consumption by the air-breathing and rocket propulsion systems. The control variables in the problem are α, the angle-of-attack in rad., ϕ_a, the non-dimensional fuel equivalence ratio of the air-breathing propulsion system, and $\phi_r(t)$, the rocket throttle setting expressed as a fraction of the maximum available thrust.

Other quantities that appear in the problem model are R_E, the radius of the Earth in ft, $\bar{q}$, the dynamic pressure in lb/ft^2, M, the Mach number, $C_{T_a}(\phi_a,M)$, the thrust coefficient of the air-breathing propulsion system in ft^2, $I_a(\phi_a,M)$, the fuel specific impulse of the air-breathing propulsion system in sec, $\rho(h)$, the atmospheric density in $slugs/ft^3$, $\delta E(\alpha,M)$, the trim elevon angle in rad, and $\phi_{a_{min}}(M)$ and $\phi_{a_{max}}(M)$, the Mach-number dependent limits on the air-breathing propulsion system's fuel equivalence ratio. Some of the numerical constants that appear in the

problem model are the acceleration of gravity at the Earth's surface, 32.174 ft/sec^2, the rocket's maximum in vacuo thrust, 15000 lb, and the rocket's in vacuo fuel specific impulse, 444.0 sec.

The problem's auxiliary constraints enforce various practical limits. Constraint (18h) keeps the dynamic pressure below 2000 psf in order to ensure structural integrity. Constraint (18i) keeps the maximum local heating rate below 400 BTU/sec/ft^2. The angle-of-attack is constrained to lie between -1^o and +12^o by constraints (18j), and the trim elevon setting is constrained to lie between $\pm$ 20^o by eq. (18k). Constraints (18l) enforce practical limits on the air-breathing propulsion system's fuel equivalence ratio. Below Mach 2, ϕ_a is constrained to lie between 0.0 and 2.0. Above Mach 2, ϕ_a is constrained to lie between 0.05 and 5.0. Similarly, rocket thrust is limited by constraints (18m). Constraints (18n) limit propulsive plus aerodynamic acceleration in the velocity direction to be between $\pm$ 4 g's, and constraint (18o) limits the normal propulsive plus aerodynamic acceleration to be below 4 g's. Constraint (18p) keeps the vehicle from flying into the ground shortly after take-off. Constraints (18q)-(18s) enforce the final achievement of a circular orbit at the prescribed altitude.

The air-breathing propulsion system model is an updated version of the model contained in Ref. 3. It has been supplied by the first author of Ref. 3. The model gives tabulated data for the thrust coefficient, $C_{T_a}(\phi_a,M)$, and the fuel specific impulse, $I_a(\phi_a,M)$, on a two-dimensional grid in (ϕ_a,M) space. The model is discontinuous at M = 2 because of a propulsion system mode switch, presumably from a turbojet mode for $M \leq 2$ to a ramjet/scramjet mode for $M \geq 2$. The tabulated data only goes up to $\phi_a = 2$ for $M < 2$, but it goes up to $\phi_a = 5$ for $M > 2$, which is the reason for the Mach-number dependent limits placed on ϕ_a by constraints (18l).

Two-dimensional cubic splines of the tabulated data have been used to generate the functions $C_{T_a}(\phi_a,M)$ and $I_a(\phi_a,M)$ for use by the trajectory optimization code. These splines are continuous with continuous first and second partial derivatives (except at M = 2). Because the underlying functions are discontinuous at M = 2, two separate cubic splines have been generated for each function, one that applies below Mach 2, and one that applies above Mach 2. These models are shown in 2-D form on Figs. 1 and 2.

The discontinuities in these functions could cause problems for the optimization software described in this report. An augmentation of the problem model has been developed to avert such difficulties. It enforces auxiliary constraints on Mach number. At one pre-selected time step, call it step number $N_{mach\text{-}2}$, the equality constraint $M = 2$ is enforced. At all time steps before $N_{mach\text{-}2}$ the inequality constraint $M \leq 2$ is enforced, and at all time steps after $N_{mach\text{-}2}$ the inequality constraint $2 \leq M$ is enforced. The low-Mach-number cubic splines of the functions $C_{T_a}(\phi_a,M)$ and $I_a(\phi_a,M)$ are always used at time steps before $N_{mach\text{-}2}$, and the high-Mach-number cubic splines of the functions $C_{T_a}(\phi_a,M)$ and $I_a(\phi_a,M)$ are always used at time steps greater than or equal to $N_{mach\text{-}2}$. In other words, the propulsion system mode switch is enforced at a particular time step. A clever modeling trick allows the actual time of this transition to remain free for the optimization process to determine. The trick uses an artificial time τ to encode the problem. The Mach 2 transition occurs at a fixed artificial time, but an augmented control quantity allows the actual time of this mode switch to be free.

The rocket model used here is borrowed from Ref. 4. It assumes an in vacuo maximum thrust of 15000 lb, an in vacuo fuel specific impulse of 444 sec., and a nozzle area of 1 ft^2. The thrust inside the atmosphere is

$$T_r = \phi_r[15000 - 1.0\, p(h)] \tag{19}$$

where p(h) is the atmospheric pressure in psf, which is implemented as a cubic spline function of altitude based on tabulated data from the 1976 U.S. Standard Atmosphere [5].

Combining the rocket thrust and the air-breathing system's thrust, the total vehicle thrust is

$$T = \bar{q}\, C_{T_a}(\phi_a,M) + \phi_r[15000 - 1.0\, p(h)] \tag{20}$$

The total rate of fuel use in eq. (18f) consists of two terms: the first term is the fuel used by the air-breathing system, and the second term is the fuel used by rocket. This thrust model assumes that the two propulsion systems can operate simultaneously, and it lets the optimization algorithm determine whether it would be beneficial to do so.

Before defining the aerodynamic model, it is necessary to define several aerodynamic quantities. The dynamic pressure takes the usual form

$$\bar{q} = \frac{1}{2}\rho(h)\, V^2 \tag{21}$$

The $\rho(h)$ density function is modeled as a cubic spline of tabulated data from Ref. 5. The Mach number is given by

$$M = V/a(h) \tag{22}$$

where a(h) is the speed of sound in ft/sec. It is modeled as a function of the air pressure and the density

$$a(h) = \sqrt{\frac{1.4\ p(h)}{\rho(h)}} \tag{23}$$

which agrees well with the tabulated a(h) data in Ref. 5 for values of h up to 280,000 ft. Above this altitude the concept of a speed of sound starts to break down, but eq. (23) is still used to define a(h) for determination of M. This should not cause great errors because the aerodynamic and propulsive forces that depend on M are very small above h = 280,000 ft.

The aerodynamic model gives the lift and drag forces, L and D, and the trim elevon setting, $\delta E(\alpha,M)$. Lift and drag are determined from the nondimensional trimmed lift and drag coefficients, $C_{L_{tr}}(\alpha,M)$ and $C_{D_{tr}}(\alpha,M)$:

$$L = \bar{q}SC_{L_{tr}}(\alpha,M) \tag{24a}$$

$$D = \bar{q}SC_{D_{tr}}(\alpha,M) \tag{24b}$$

where S is the wing reference area, 3603 ft^2. Reference 3 provides untrimmed lift, drag, and pitching moment coefficients along with increments to these coefficients that depend on α, M and δE. Given a c.g. location expressed as a fraction of the mean aerodynamic chord, x/c, one can solve the pitch trim equation for the equilibrium elevon setting:

$$0 = C_{M_a}(\alpha,M) + 2C_{M_{\delta E}}(\alpha,M,\delta E) + [x/c][C_{L_a}(\alpha,M) + 2C_{L_{\delta E}}(\alpha,M,\delta E)] \tag{25}$$

where $C_{M_a}(\alpha,M)$ and $C_{L_a}(\alpha,M)$ are the untrimmed pitching moment and lift coefficients, and $C_{M_{\delta E}}(\alpha,M,\delta E)$ and $C_{L_{\delta E}}(\alpha,M,\delta E)$ are the increments to these coefficients due to just one elevon (left or right).

The trim elevon setting can be calculated as a function purely of α and M, $\delta E = \delta E(\alpha,M)$, because the c.g. location has been modeled as being a pre-determined function of M. The function

x(M)/c has been chosen to make the vehicle nearly neutrally stable over much of its flight envelope when trimmed at lift = weight. This is true at high Mach numbers, but the aerodynamic data from Ref. 3 has a discontinuity at M = 1 due to a canard retraction as M increases through 1. In order to preserve continuity of the x(M)/c function, the vehicle is allowed to be statically unstable below Mach 1. The pre-programmed c.g. function in tabulated form is given in Table 1.

Given the trim elevon setting from eq. (25), the trimmed lift and drag coefficients become

$$C_{L_{tr}}(\alpha,M) = C_{L_a}(\alpha,M) + 2C_{L_{\delta E}}[\alpha,M,\delta E(\alpha,M)] \quad (26a)$$

$$C_{D_{tr}}(\alpha,M) = C_{D_a}(\alpha,M) + 2C_{D_{\delta E}}[\alpha,M,\delta E(\alpha,M)] \quad (26b)$$

where C_{D_a} is the untrimmed drag coefficient, and $C_{D_{\delta E}}$ is the drag increment due to one elevon being deflected.

The functions $C_{L_{tr}}(\alpha,M)$, $C_{D_{tr}}(\alpha,M)$, and $\delta E(\alpha,M)$ are implemented as 2-dimensional cubic splines that interpolate between grid points in (α,M) space. As mentioned above, each of these functions has a discontinuity at M = 1 due to the assumption of a canard retraction as M increases through this value. Therefore, each of these functions has two separate cubic splines, one that applies for $M \leq 1$ and one that applies for $M \geq 1$. This leaves a discontinuity, which can be seen clearly on the plots of $C_{L_{tr}}(\alpha,M)$ and $C_{D_{tr}}(\alpha,M)$ that appear in Figs. 3 and 4.

As with the propulsion model's discontinuities, the aerodynamic discontinuities could cause problems for the optimization software. Further auxiliary constraints on Mach number have been added to the problem statement in order to avoid such problems. At a pre-selected time step numbered $N_{mach\text{-}1}$, the equality constraint M = 1 is enforced. At all time steps before $N_{mach\text{-}1}$ the inequality constraint $M \leq 1$ is enforced, and at all time steps after $N_{mach\text{-}1}$ the inequality constraint $1 \leq M$ is enforced. The low-Mach-number cubic splines of the functions $C_{L_{tr}}(\alpha,M)$, $C_{D_{tr}}(\alpha,M)$, and $\delta E(\alpha,M)$ are always used at time steps before $N_{mach\text{-}1}$, and the high-Mach-number cubic splines of these functions are always used at time steps greater than or equal to $N_{mach\text{-}1}$. As with the propulsion mode switch, a modeling trick allows the actual problem time of the switch to be left free despite that fact that it occurs at a fixed stage number. Of course, the stage number $N_{mach\text{-}1}$ is chosen to be less than $N_{mach\text{-}2}$.

Several modeling tricks have been used to put the problem in a form compatible with the specification given in Appendix D. The state vector gets augmented to store $dt/d\tau$, the rate of change of actual time with problem time, and some auxiliary control variables, or constraints, or both are required at stages 0, $N_{mach\text{-}1}$, and $N_{mach\text{-}2}$ in order to allow adjustment of $dt/d\tau$ so that it is independently definable for the three main problem phases, $M < 1$, $1 \leq M < 2$, and $2 \leq M$.

In an attempt to improve the numerical conditioning of the problem, the inequality constraints have been re-scaled so that the maximum or minimum limit, if nonzero, becomes ± 1. Similarly, the optimization algorithm works with the following units in order to deal with **x** and **u** elements nearly on the order of 1 to 10. Time is in 1,000 sec. units, altitude is in 10,000 ft. units, speed is in units of 1,000 ft/sec., mass is in 1,000 slug units, flight-path angle is in rad., and angle-of-attack is in deg.

The first guess used is for this optimization roughly approximates two features found in optimal trajectories that appear in Ref. 4: an arc along the $\bar{q} = 2000$ psf limit followed by an arc along the heating-rate limit. The guess starts at $M = .4$ and $h = 10$ ft. It assumes a linear increase of h and M until $M = 0.85$ and h equals the altitude at with $M = 2$ would yield $\bar{q} = 2000$ psf. Next, the guess accelerates in level flight to increase M from 0.85 to 2, which brings the trajectory onto the $\bar{q} = 2000$ psf contour. Now M and h increase along this contour until the heating-rate limit in eq. (18i) is reached. The guess continues increasing M and h along this contour until V = 23,000 fps. It then does a pull-up to put the vehicle into a transfer orbit with apogee = 528,000 ft., the target orbital altitude. Finally, the guess does a long burn near apogee to approximately circularize the final orbit.

The air-breathing propulsion system is run at $\phi_a = 1$ up until the pull-up into a transfer orbit, and the rocket throttle setting is kept at zero until after the pull-up. During the circularizing burn, $\phi_r = 1$ is used, and the time of the burn is based on the velocity increment needed at apogee to circularize the orbit.

Initially, this guess procedure yields a plot of h vs. V for the atmospheric phase. The time parameterization of the h-V profile and the guessed time histories $\gamma(t)$ and m(t) are determined by

satisfying Euler approximations of 3 of the state differential equations, eqs. (18c), (18d), and (18f). The $\alpha(t)$ guess is determined to minimize the error in the Euler approximation of eq. (18e) subject to the minimum and maximum bounds on α in eq. (18j).

Of course, this first guess violates many of the problem constraints. It violates all of the differential equations slightly, and it may violate any of the auxiliary constraints that have not been specifically used in this first guess procedure (e.g., the bounds on $\delta E(\alpha,M)$).

6. Trajectory Optimization Computational Results

All of the results reported in this section are for work done on a 32-node INTEL iPSC/860. Encoding has been done in parallel FORTRAN. This is a message-passing parallel processor. Each node of the iPSC/860 is a "super" scalar processor. Simple tests indicate that each node's floating-point processor speed is about 1.5 Mflops, which is comparable to an HP Apollo 9000 model 720 work station. The interconnections between nodes are on a rectangular grid, but the effect of the "distance" between nodes in this grid is supposed to be minimal due to advanced message routing hardware.

6.1 Results for the Goddard Problem

The Goddard problem has been solved on this machine using a variety of numbers of discrete-time problem stages to approximate the problem and using a variety of numbers of processors. Problems with 16, 32, 64, and 128 stages have been solved. Processor numbers ranging from 1 to 32 have been used when appropriate. The results reported here use the alternate NP algorithm outlined at the end of Section 3, the one that always uses curved line searches (except during its feasibility phase).

Solution time histories for a 128-stage problem are plotted on Figs. 5 and 6. They compare well with the exact solution given in Ref. 1. Figure 3 of Ref. 1 indicates that the fixed-end-time solution with $t_f = 0.198$ is also the optimal free-end-time solution. The $t_f = 0.198$ altitude plot on Fig. 5 of Ref. 1 is very similar to the altitude plot in the upper left-hand corner of Fig. 5 of this

report, and the $t_f = 0.198$ thrust solution on Fig. 6 of Ref. 1 is very similar to the thrust plot on Fig. 6 of this report; the slight differences are due, most likely, to the problem discretization.

An important point about the present algorithm is how well it captures the singular arc that appears on the thrust plot between $t = 0.024$ and $t = 0.07$. This is accomplished without having to tell the algorithm in advance that it must differentiate the optimality condition in order to get the right equation for determining the singular arc. In essence, this differentiation is done automatically by the QP solution procedure described in Section 2 and in Appendix A.

Computational timing results for various sized problems running on various numbers of processors are presented on Fig. 7. This figure shows how the addition of more processors can speed up the solution procedure significantly. For a 64-stage problem, the algorithm terminates 7.2 times faster when running on 32 nodes than when running on one node. The speediest solutions for a given number of problem stages are achieved when the number of processors is greater than or equal to one half the number of problem stages. Of course the number of processors must not exceed the number or problem stages or the ability to parallelize under the present scheme breaks down.

The times on Fig. 7 are for solution of the problem from a cold start, a relatively poor first guess. If the algorithm were to be used in a real-time guidance loop, then it would start with a good first guess on each update. The guess would be the remainder of the optimal trajectory that had been computed for the previous guidance update. This would be identical to the current optimal trajectory if not for disturbances, sensor noise, and model uncertainty. It is hoped that these contributions to solution uncertainty would be relatively small. If such were case, then the real-time version of the guidance algorithm would need to perform only a very few of its major iterations: computation of functions and their first and second partial derivatives and solution of a quadratic program. Furthermore, the solution of each quadratic program would require only one or a very small number of minor iterations due to the goodness of the solution guess. Thus, the algorithm might quickly re-optimize trajectories to provide real-time optimal commands.

With this in mind, Fig. 8 presents the wall clock time per major iteration when the algorithm is near the solution. These results are for various numbers of processors, all of them solving the 64-stage Goddard problem. Note how the time per solution is split about evenly between the derivative calculations and the QP solution procedure for the case of 1 processor, but the QP procedure takes about 5 times as long as the derivative calculations when using 32 processors. This happens because the derivative calculations are inherently more parallelizable than the QP solution algorithm. Also, note that the 32-processor result is 12 times faster than the 1-processor result.

If the algorithm could terminate in 10 such iterations during a real-time guidance update, then guidance updates could occur about once every 2.5 sec. Noting that time is expressed in Hergs, the actual time per zero-order-hold interval is 2.54 sec for the 64-stage problem. Therefore, this scheme could work in a real-time loop under the assumption of 10 terminal-type algorithm iterations per guidance update. Note that the ability to perform such updates in the required number of iterations has not, as yet, been studied.

The dependence of the number of major and minor algorithm iterations on the number of processors is shown in Figs. 9 and 10, respectively. Major iterations are those that calculate derivatives and start solving a new QP; they are termed "middle-loop" iterations in the paper in Appendix B. Minor iterations are those that perform one set of matrix factorizations and a line search; they are termed "inner-loop" iterations in Appendix B.

Figures 9 and 10 demonstrate that the solution time is moderately influenced by a change in the required number of algorithm iterations with a change in the number of processors. This change is thought to be due primarily to the way in which the algorithm deals with directions of negative curvature in the QP projected Hessian. It can calculate different directions of negative curvature for the same indefinite QP when using different numbers of processors.

6.2 Results for the Minimum-Time to the Origin Problem

Both 32-stage and 64-stage approximations of the acceleration-limited minimum-time to origin problem have been solved. Figure 11 plots several different time histories associated with

the 64-stage solution. On the upper left-hand plot of the figure a1 and a2 are the accelerations. They correspond to x_5 and x_6, respectively, in the model presented in eqs. (14a)-(14d) of Section 5.2. Similarly, v1 and v2 on the lower-left plot of Fig. 11 correspond to x_3 and x_4, respectively, in eqs. (14a)-(14d). These plots agree very well with the exact solution computed from the linear tangent steering law, eqs. (10) and (11).

One noteworthy aspect of the solution is that an auxiliary state-variable inequality constraint, eq. (14c) is active at all times. The sum of the squares of the curves for a1 and a2 is equal to 1 at each time point on the upper left-hand plot of Fig. 11. The algorithm is able to compute the optimal solution without the typical requirement that eq. (14c) be differentiated in order to introduce a control variable. In fact, just the opposite is the case in this problem.

The state variable constraint in eq. (14c) was originally a control variable constraint, eq. (9e), in the original problem form, eqs. (9a)-(9f). The model has been modified by making the original controls into states and creating new controls that are the time derivatives of the original controls. This has been done in order to smooth out the zero-order-hold effects and achieve a better approximate solution for a given number of discrete-time steps. This modeling trick would be counter-productive were it not for the algorithm's ability to automatically handle pure state constraints.

The 32-stage minimum-time problem has been solved on 1, 2, 4, 8, 16, and 32 processors. Figure 12 plots the solution times for this problem versus the number of processors used to solve the problem. As with the Goddard problem, an increase in the number of processors decreases the wall-clock time required for solution. The 5-fold decrease in the solution time as the number of processors increases from 1 to 32 is about the same as for the 32-stage Goddard problem.

The variation of the number of major algorithm iterations with the number of processors for the 32-stage minimum-time problem is plotted on Fig. 13. Similarly, Fig. 14 plots the number of minor algorithm iterations vs. the number of problem stages. Both of these curves increase in comparison to the corresponding plots for the 64-stage Goddard problem, review Figs. 9 and 10. It is not clear why the iteration counts are higher for this problem. The cause may have something

to do with the increased number of states in the final form of the problem model (7 in this problem vs. 4 in the Goddard problem), or it may have to do with the presence of an active state inequality constraint in this problem.

The variations of the iteration counts with the number of processors are more pronounced than for the Goddard problem. The 1-processor case requires significantly fewer of each type of iteration. Again, these discrepancies are attributable to the presence of an indefinite projected Hessian during many (the majority!) of the QP/line-search minor iterations.

The time per major iteration as it varies with the number of processors is plotted on Fig. 15. These times are for major iterations near successful algorithm termination, when only one QP search direction calculation is required per major iteration. Major iterations far from the solution generally require multiple QP search direction calculations and, therefore, take significantly longer. Plotted along with the total time per iteration are its principal components, similar to Fig. 8 for the Goddard problem.

The total iteration time is roughly the same as for the Goddard problem (review Fig. 8), but the proportion of time spent solving the QP is larger than for the Goddard problem. This makes sense because the QP algorithm's scalar operation count goes as $[n^3 + O(n^2)][(N/p) + \log_2(p)]$ when solving a problem with n state vector elements and N stages on p processors. For the present problem n = 7 and N = 32, whereas n = 4 and N = 64 for the Goddard problem on Fig. 8. The plots show that the QP solution procedure for the minimum-time problem is slower for all cases considered, but not quite as slow in comparison to the corresponding Goddard problem procedure as is indicated strictly by the leading terms of the scalar operation count.

The iteration speed for the 32-stage minimum-time problem operating on 16 or 32 processors is under 0.5 sec. This is a fast execution time and may indicate suitability for use in real-time guidance. Any serious consideration of the real-time guidance issue will have to determine a reliable estimate of the typical number of iterations required to re-converge the solution after one guidance update interval.

6.3 Results for the NASP Ascent Guidance Problem

A 32-stage NASP ascent guidance problem has been solved using the parallel algorithm developed under this grant. The solved problem models the Mach 1 transition, and the concomitant change in the aerodynamic model due to canard retraction, as occurring at stage $N_{mach\text{-}1} = 6$. It models the Mach 2 transition and concomitant propulsion model switch as occurring at stage $N_{mach\text{-}2} = 9$. The initial stage is stage 0, and the terminal stage is stage N = 31, which leaves 22 zero-order-hold intervals for flight above Mach 2. These hold intervals have been modeled as all being of equal real-time duration.

The algorithm solved the problem in 2 hours of wall clock time using all 32 of the iPSC/860's processors. It executed 2035 major algorithm iterations to solve the problem, about 100 of which were used to achieve initial approximate feasibility of the trajectory; the NP algorithm described in Appendix B has an initial phase in which it enforces feasibility.

The alternate NP algorithm described at the end of Section 3 is the one that has been used to solve this problem. An attempt was made to solve the NASP problem with the first NP algorithm of Section 3, but it did not get near convergence after 2000 major algorithm iterations.

Plots of the solution are presented on Figs. 16, 17, and 18. The mass time history, the lower-right plot on Fig. 16, shows that the final on-orbit mass is about 56% of the original post-lift-off weight, which seems reasonable. The altitude and flight-path angle time histories, the two left-hand plots on Fig. 16, show an initial steep climb to about 150,000 ft. After this the vehicle climbs more slowly until it reaches its orbital velocity at an altitude of just below 200,000 ft. Finally, it executes a pull-up into a transfer orbit to reach its target orbital altitude.

The control time histories on Fig. 17 show that the air-breathing engine provides most of the acceleration. Its fuel equivalence ratio, shown on the upper right-hand plot, is non-zero throughout the ascent. It seems a little bit silly that ϕ_a remains nonzero after the exit of the atmosphere at about t = 3,000 sec, but this does not cause much loss of final mass because so little actual fuel flow is involved. The rocket throttle setting, the lower left-hand plot, remains at zero until just before the terminal time when it helps to circularize the final orbit.

The angle of attack time history, the upper left-hand plot on Fig. 17, shows some jitteriness coupled with a slow downward trend as speed increases toward the orbital velocity. There is a pull-up just before t = 3,000 sec to initiate the transfer orbit, then α is held negative during the transfer orbit and part of the terminal burn, presumably to help circularize at the proper altitude.

The jitteriness of the angle-of-attack plot points up a shortcoming of this solution. It seems that the zero-order-hold intervals are too long during the flight phase above Mach 2. Each interval is about 200 sec long. If V = 2,900 fps, then the approximate Phugoid period is 400 sec. The lightly-damped Phugoid mode will alias if it has a shorter period (a higher frequency), which it will have for $V \leq 2{,}900$ fps. This aliasing of a lightly-damped open-loop mode could easily account for the jitters apparent on all of the plots, especially those showing angle-of-attack and flight-path angle.

The altitude vs. Mach number plot, the left-hand plot on Fig. 18, shows related problems†. The vehicle accelerates from Mach 2 to Mach 17 in just one zero-order-hold interval -- compare this plot with the velocity time history in the upper right-hand corner of Fig. 16. During this hold interval the trajectory may violate the dynamic pressure constraint because the auxiliary constraints are enforced only at the boundaries of the hold intervals. (Note that the straight-line approximation of the trajectory shown on the figure is probably not the actual trajectory computed via numerical integration.) The two spikes that appear on the h vs. M plot also may be due to the long hold interval.

There are two possible fixes for the aliasing problem. One is to model the controls as being piecewise linear or piecewise parabolic. This could be done by augmenting the state vector with the original controls and making new control variables out of the first or second time derivatives of the original controls. The smoothed-out control signals would be less likely to excite the lightly-

---

† On a different note, the h vs. M curve changes from solid to dashed above about 280,000 ft because Mach number is not very meaningful at such extreme altitudes.

damped phugoid mode even when aliasing occurred. Another, simpler fix would be to greatly increase the number of problem stages. Attempts have been made to solve a 64-stage problem, but the algorithm took too long to converge in the tests conducted to date.

The average time per major iteration of the algorithm is 3.5 sec. of wall clock time for this problem. Iteration times near successful algorithm termination are probably even shorter than 3.5 sec. This is in the ball park of the kind of iteration speed that might make real-time optimal guidance practical for the NASP ascent problem, but other difficulties with the algorithm need to be worked out before it can be used. In particular, a solution to the aliasing problem must be found.

Also, algorithm modifications are needed in order to speed global convergence. The slow global convergence of the algorithm on the NASP problem implies that standard guidance updates, despite having relatively good first guesses, would probably require many iterations of this algorithm. This would probably require too much time to make on-line solution practical.

7. Discussion of Algorithm Performance and Suggestions for Improvement

Application of the parallel trajectory optimization algorithm to the 3 test problems has shown some strengths and some weaknesses of the approach. One of the two most significant strengths is the ability of parallelism to greatly speed up each major algorithm iteration, which consists of derivative calculations, a search direction calculation, and a search step. These operations are common to almost all optimization algorithms, and the results of this work could be applied to parallelize trajectory optimization algorithms other than the one presented here.

Another significant strength of the algorithm is the rapidity with which it is able to determine a feasible solution to a problem. Such a solution obeys the dynamics and all of the auxiliary constraints, including any terminal constraints, but it may be far from the optimum of the given cost function. If the most important guidance objectives are stated in terms of terminal constraints, such as the constraint that the NASP achieve a circular orbit of a certain altitude, then the algorithm can rapidly design sub-optimal trajectories that meet the most important guidance objectives.

The main weakness of the algorithm is its slowness to achieve convergence to the optimal solution. It can require many major iterations, most of which require a number of minor iterations. Each minor iteration involves solution of an equality-constrained dynamic QP. While this process has been parallelized, it is inherently difficult to achieve good parallel efficiency for this process. The algorithm would do well to cut down on the number of equality-constrained QP solutions that it computes on the way to a solution.

One possible cause of much of the slowness revolves around the algorithm's actions when it encounters an indefinite projected Hessian during computation of a search direction. Presently, it attempts to search in a direction of negative curvature every other time that it encounters an indefinite Hessian, but it cannot guarantee that it is searching in the direction of most negative curvature.

The presence of quadratic constraint terms in the middle-loop QP problem (see the paper in Appendix B) may cause more harm than good. These terms, when coupled with the algorithm's logic for dealing with negative cost function curvature, can cause the algorithm to cycle between several search directions, none of which makes much progress in reducing the cost function.

Another problem may be the augmented Lagrangian NP algorithm itself. The current implementation, as described in Appendix B, is designed to work well with large penalty weights in order to be able to assure rapid local convergence. Unfortunately, large penalty weights can cause global convergence problems that are not addressed by the current algorithm.

Yet another problem with the algorithm may be its use of the box trust region method of assuring global convergence. This method tends to lean heavily on equality-constrained QP factorizations and to rely only very lightly on function evaluations. In the parallel environment, QP solutions are relatively expensive, and function evaluations are relatively cheap. A more efficient parallel algorithm might be one that ensures global convergence by using line searches to determine the search step length.

The rapidity of the constraint satisfaction phase of the algorithm suggests an alternate approach to develop a rapid parallel trajectory optimization algorithm. Instead of using a standard

NP type of algorithm, one could develop a parallel equation solving algorithm for the necessary conditions. While this would pose some difficulties in that the Kuhn-Tucker necessary conditions involve inequalities and the complementarity condition, these difficulties could probably be surmounted. The algorithm would use the norm of the necessary condition violation as its merit function. It could use this merit function in a Levenberg-Marquardt-type of approach. Parallel solution times might be on the order of the times required by the current algorithm to find an initial feasible trajectory during its first phase. Such performance, if realized, would make the necessary condition solver 20 times faster than the present algorithm when solving the NASP problem.

The equation solving approach has the advantage that many of the parallel ideas developed under this grant could be directly applied to it, but it has the disadvantage of possible convergence to a saddle point or to a local maximum; the necessary conditions only ensure that a stationary point has been found. In any event, one of the parallel QP factorization algorithms of Appendix A could be used to check whether or not the stationary point was a local minimum. The algorithm would form and Cholesky factorize the projected Hessian at the solution. If the Cholesky factorization terminated successfully without producing a negative diagonal element, then the solution could be certified to be a local minimum.

8. Conclusions

An effort has been made to develop a fast parallel trajectory optimization algorithm suitable for use in on-line, real-time guidance of an aerospace vehicle. The parallel algorithm approximates continuous-time phases of a problem via the zero-order-hold control discretization and Runge-Kutta numerical integration over the hold intervals. This gives rise to a large nonlinear program with the specialized dynamic programming structure. The parallel algorithm uses a specialized version of the augmented Lagrangian nonlinear programming algorithm to solve this problem.

The algorithm starts with a guessed solution that need not satisfy the dynamics; state and control time histories are guessed. Next, the algorithm computes the cost, dynamics, and auxiliary constraint functions and their first and second partial derivatives. This is accomplished in parallel

by having different processors simultaneously compute these quantities for different time steps. The algorithm then uses this information to set up a quadratic approximation of the original trajectory optimization problem. It solves this quadratic approximation, subject to trust region bounds, to compute an improved guess of the solution. It uses a special parallel dynamic quadratic programming algorithm to solve this problem. Given an improved guess, this process repeats itself until suitable termination criteria are satisfied.

Global convergence to a local minimum is assured by adaptively adjusting the trust region size to enforce descent of the augmented Lagrangian function. Satisfaction of the dynamic constraints and the auxiliary constraints is enforced via an outer loop in which multiplier guesses are updated to bias the penalty terms in the augmented Lagrangian function.

The algorithm has been tested off-line on three problems, the Goddard problem of maximizing a sounding rocket's peak altitude, the planar, acceleration-limited minimum-time to the origin problem, and a lift-off-to-orbit National Aerospace Plane (NASP) minimum-fuel ascent guidance problem. The algorithm is able to accurately solve problems with singular arcs or with active state-variable inequality constraints without placing any special burden on the problem modeling process.

Good solution speeds have been achieved on the Goddard and minimum-time problems, even with fairly poor first guesses. A 128-stage Goddard problem has been solved in just 118 sec using all 32 nodes of an INTEL iPSC/860 parallel processor. This problem has 4 state variables and one control variable after special modeling tricks have been applied. A 32-stage minimum-time problem has been solved in 151 sec. The problem model that was submitted to the algorithm has 7 state variables and 2 control variables.

The use of fewer processors results in increased solution times. The solution of a 64-stage Goddard problem takes 7 times longer on 1 processor than on 32 processors. The optimum number of processors seems to be about half the number of problem stages for typical problems.

Performance on the NASP problem has not been as good. A 32-stage problem with a 5-dimensional state vector required 2 hours (7200 sec) to reach a solution when using 32 processors.

The time per major algorithm iteration is low for all three problems, especially when the algorithm is near successful termination. Each major iteration calculates functions and derivatives, solves a QP, and updates the solution estimate. When using all 32 processors, the iteration time on the Goddard problem is 0.26 sec near algorithm termination, but this time increases to 3.08 sec when using just 1 processor. Similarly, a 32-stage minimum-time problem's iteration time near algorithm termination is just 0.38 sec on 32 processors, but it is 3.09 sec on 1 processor. A 32-stage NASP problem has an average major iteration time of 3.5 sec when run on 32 processors, and its iteration time near algorithm termination is even smaller, but it required 2035 such iterations to converge from the first guess tried in this study, which accounts for the long time required to solve that problem.

The suitability of this algorithm for on-line, real-time guidance purposes depends on two things, its iteration speed and the number of iterations that it must execute to re-converge to the solution between guidance updates. The algorithm developed under this grant has a proven ability to iterate rapidly, but its ability to converge in few iterations seems to be problem-dependent. In summary, the algorithm is not suitable for all on-line guidance applications in its present form, but it may be suitable for some on-line applications.

9. References

1. Tsiotras, P. and Kelley, H.J., "Goddard Problem with Constrained Time of Flight", **Journal of Guidance, Control, and Dynamics**, Vol. 15, No. 2, March-April 1992, pp. 289-296.
2. Bryson, A.E. Jr. and Ho, Y.C., **Applied Optimal Control**, Ginn and Co., (Waltham, Mass., 1969).
3. Shaughnessy, J.D., Pinckney, S.Z., McMinn, J.D., Cruz, C.I., and Kelly, M.-L., "Hypersonic Vehicle Simulation Model: Winged-Cone Configuration", NASA TM-102610, November 1990.

4. Corban J.E., "Real-Time Guidance and Propulsion Control for Single-Stage-to-Orbit Airbreathing Vehicles", Ph.D. Dissertation, Georgia Institute of Technology, Nov., 1989.
5. **U.S. Standard Atmosphere, 1976**, U.S. Government Printing Office, 1976.
6. Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., **Numerical Recipes, The Art of Scientific Computing**, Cambridge Univ. Press, (New York, 1989).

Table 1

NASP Center-of-Gravity Location

as a Function of Mach Number

M	x/c
0.3	0.204000
0.7	0.217000
0.9	0.230000
1.5	0.243292
2.5	0.234107
4.0	0.137814
6.0	0.068927
10.0	0.040987
15.0	0.030701
20.0	0.031225
24.2	0.030864

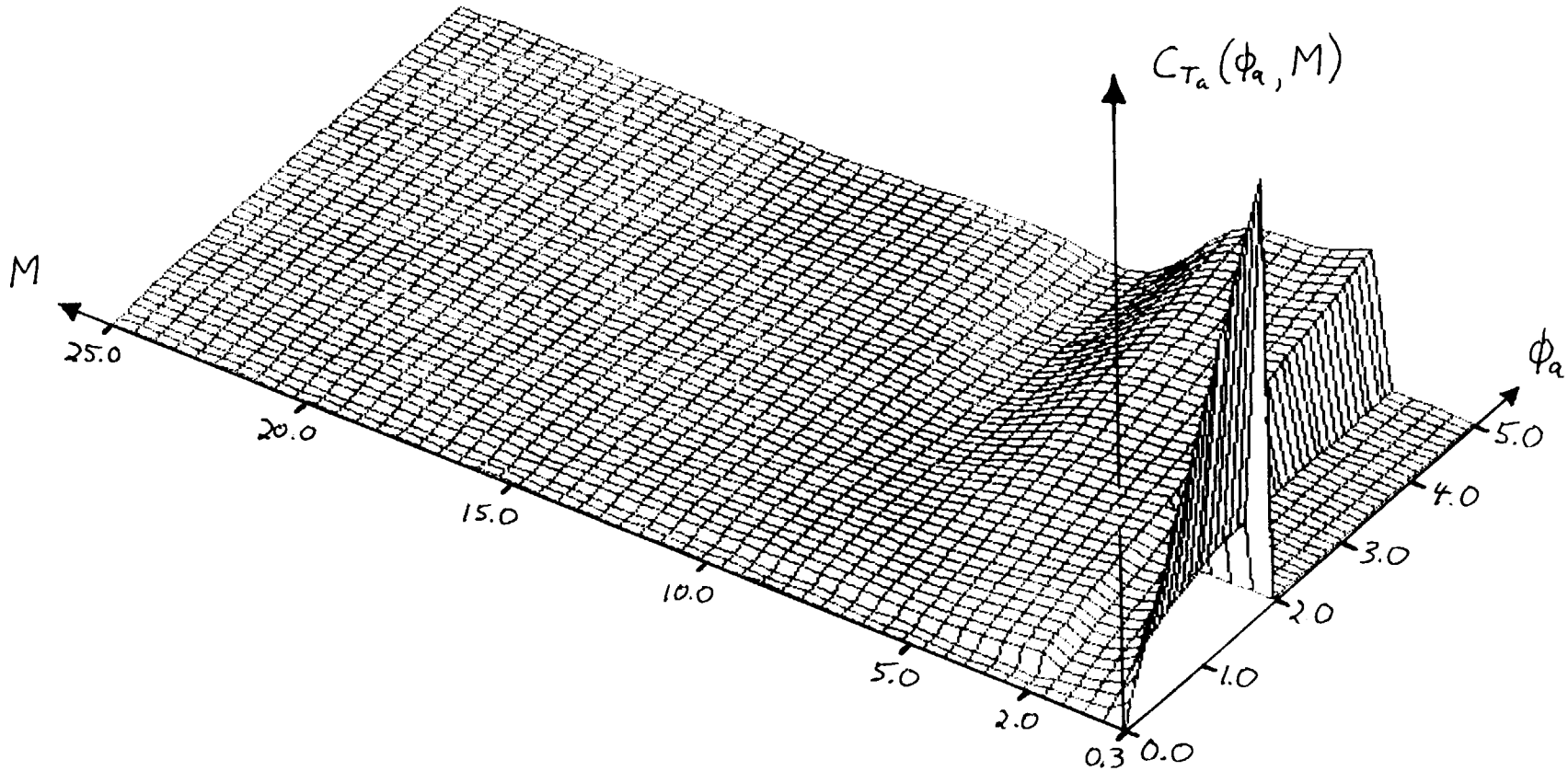


Fig. 1. Air-breathing propulsion system's thrust coefficient vs. Mach number and fuel equivalence ratio. (Mach number scale changes at Mach 2.)

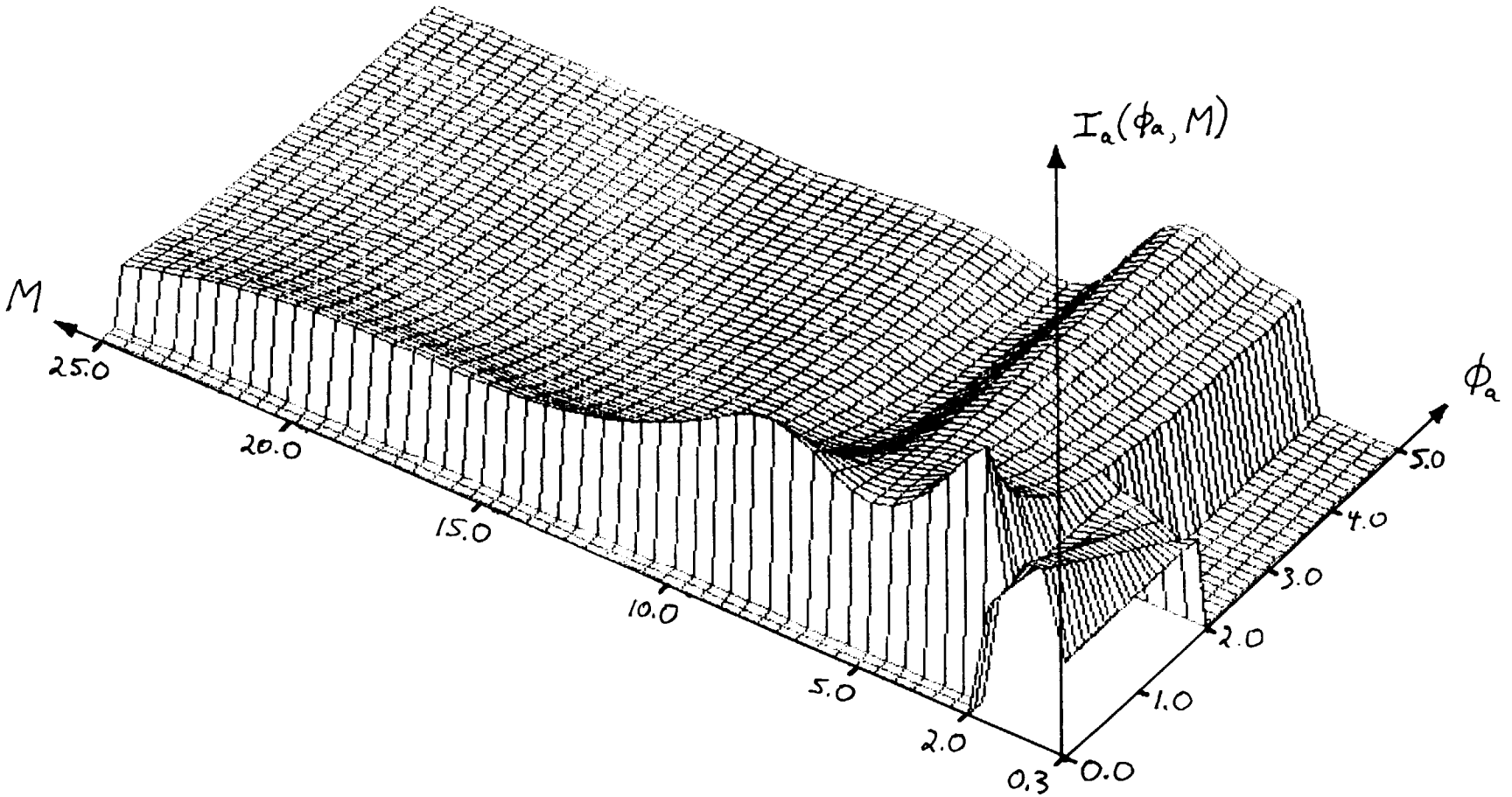


Fig. 2. Air-breathing propulsion system's specific impulse vs. Mach number and fuel equivalence ratio. (Mach number scale changes at Mach 2.)

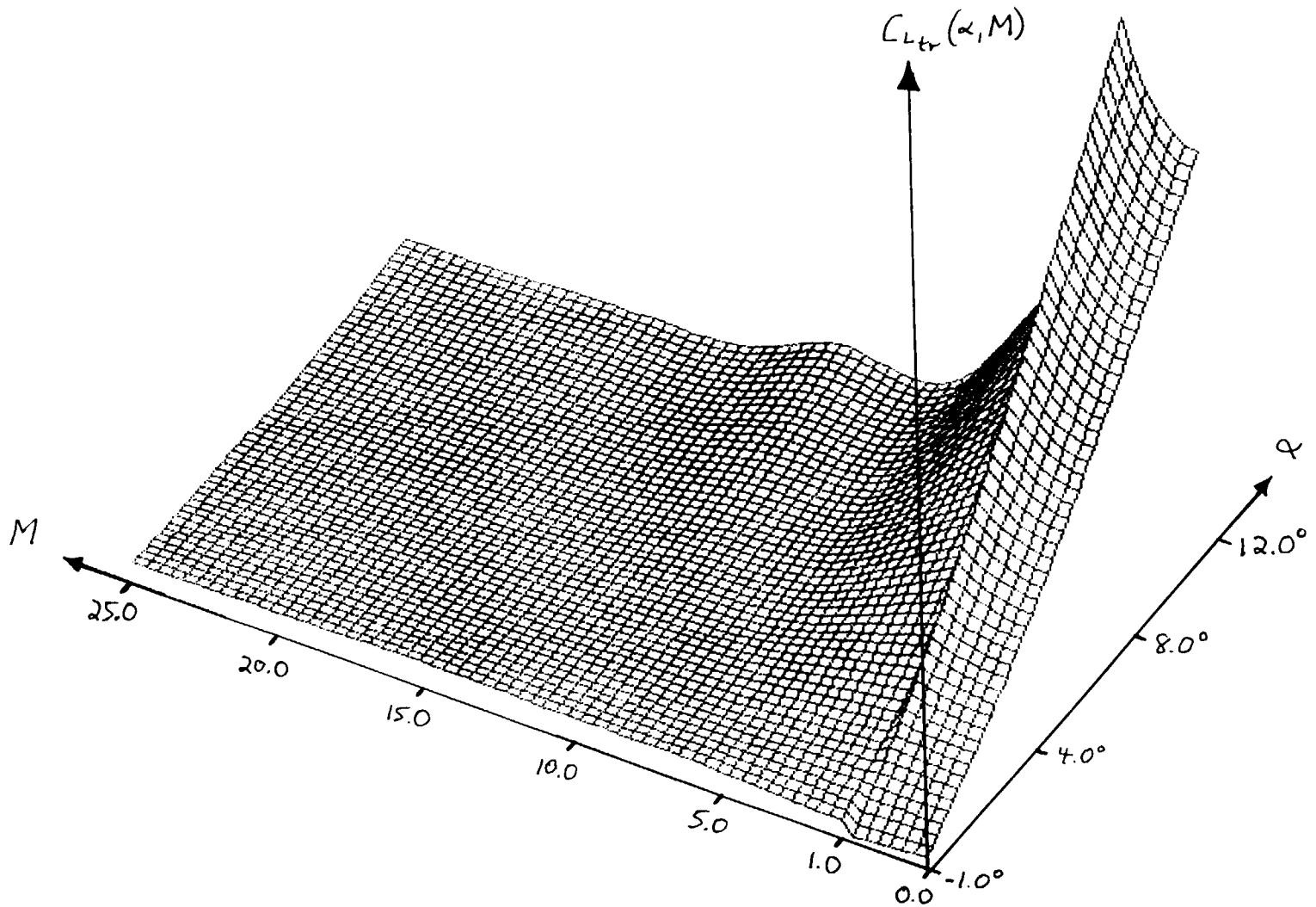


Fig. 3. Trimmed lift coefficient vs. Mach number and angle of attack. (Mach number scale changes at Mach 1.)

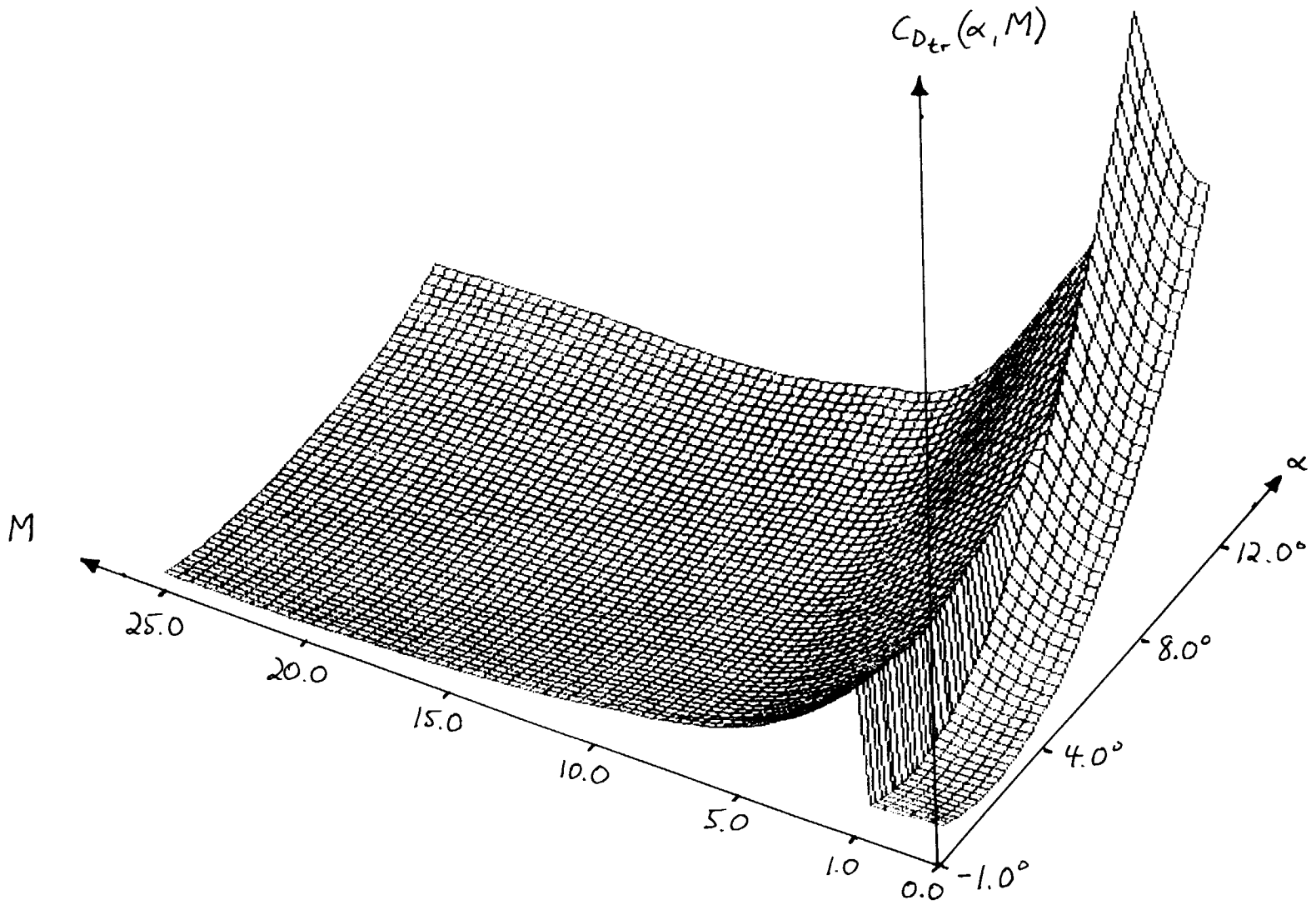


Fig. 4. Trimmed drag coefficient vs. Mach number and angle of attack. (Mach number scale changes at Mach 1.)

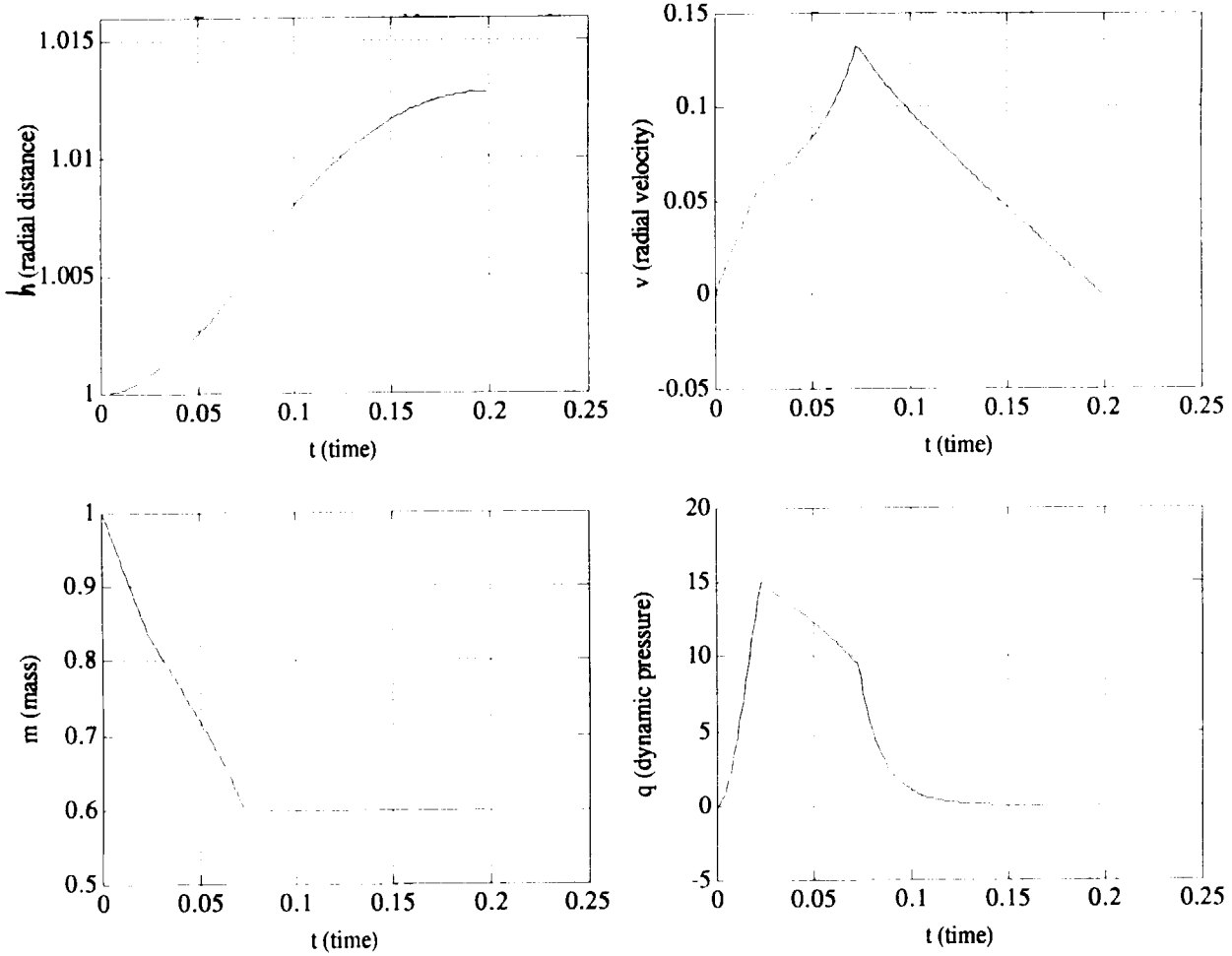


Fig. 5. Four time histories of the 128-stage solution to the Goddard problem.

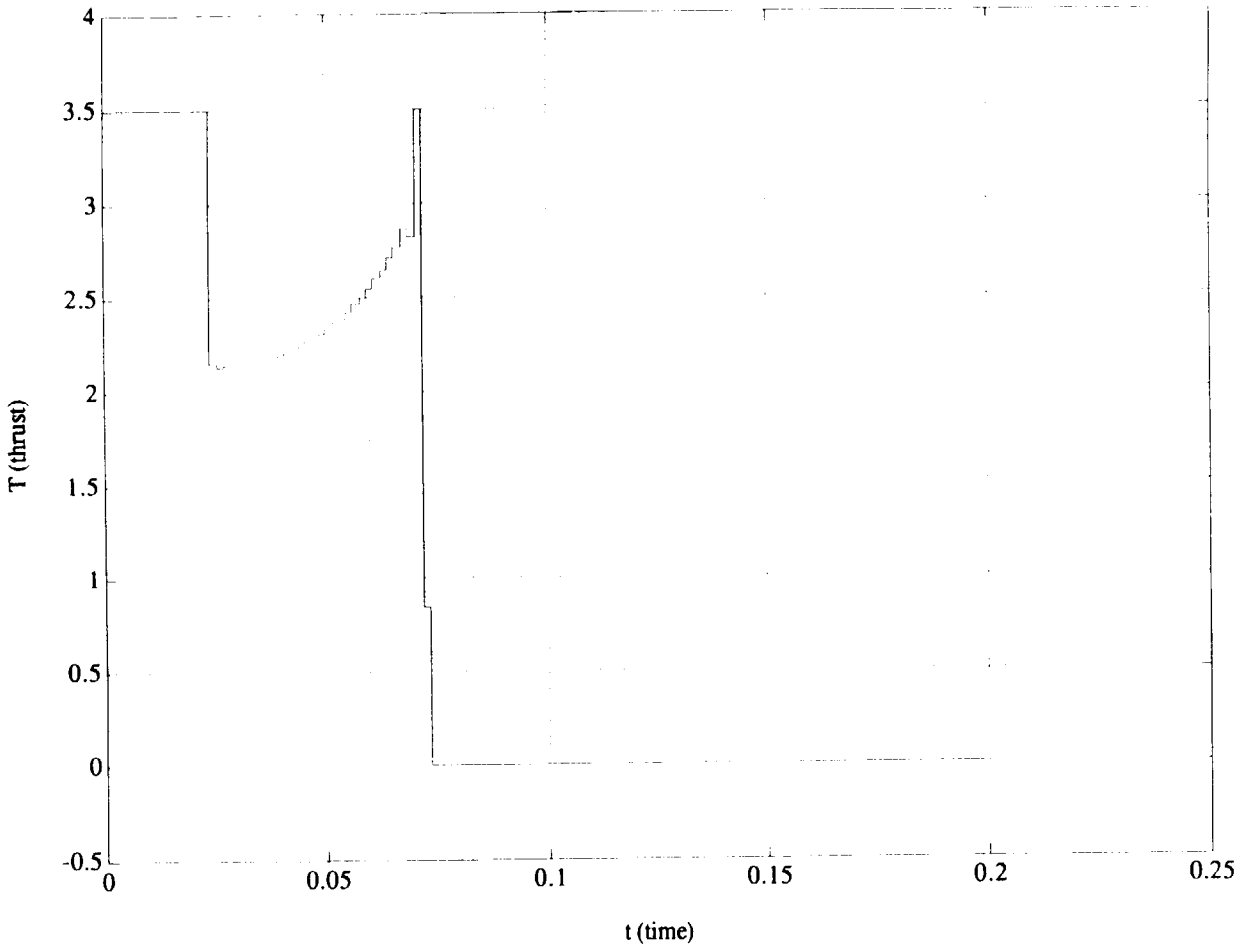


Fig. 6. The thrust (control) time history for the 128-stage Goddard problem.

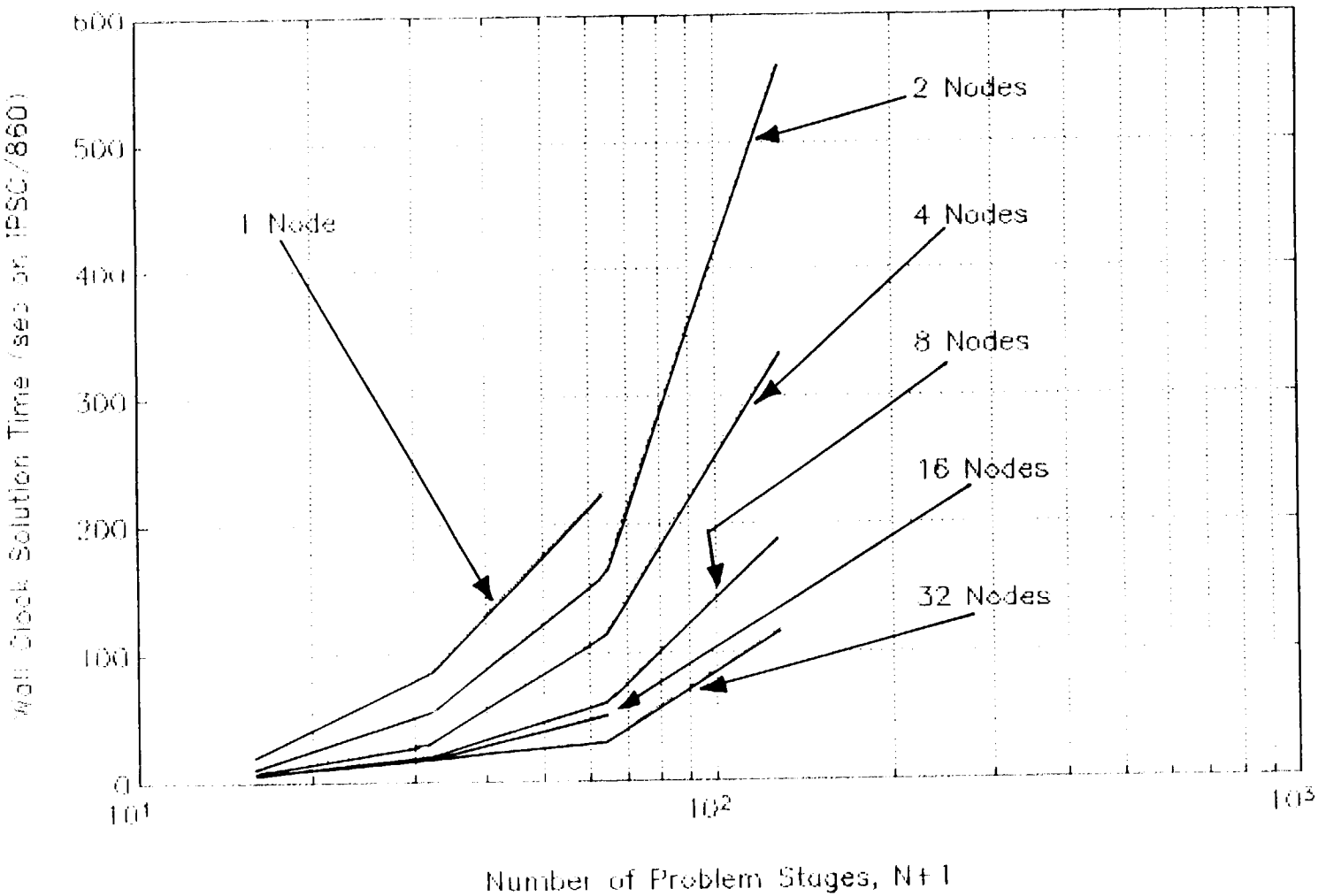


Fig. 7. Solution time vs. the number of problem stages for the Goddard problem run on various numbers of processors.

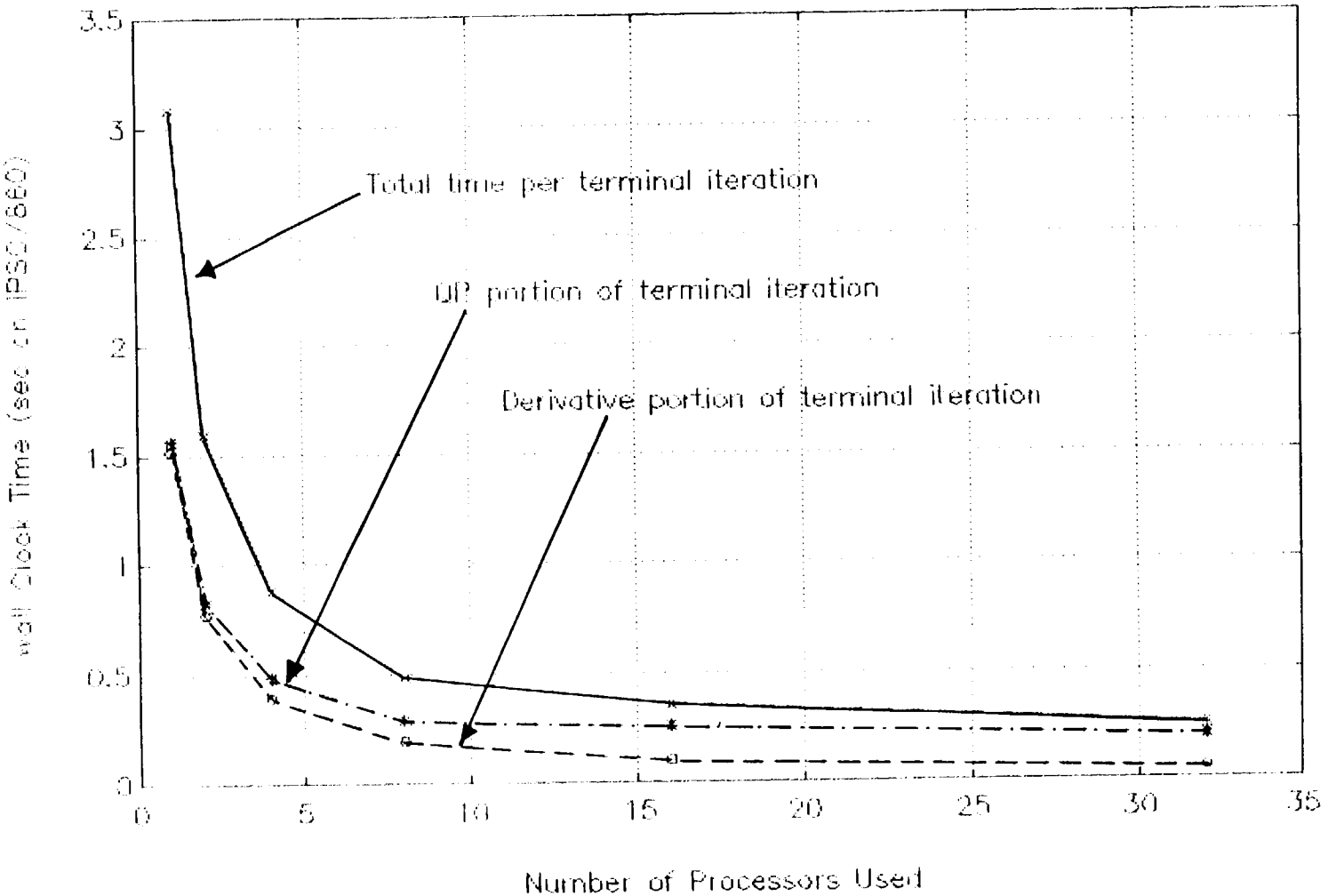


Fig. 8. Time per major iteration and its components when near the solution of the 64-stage Goddard problem.

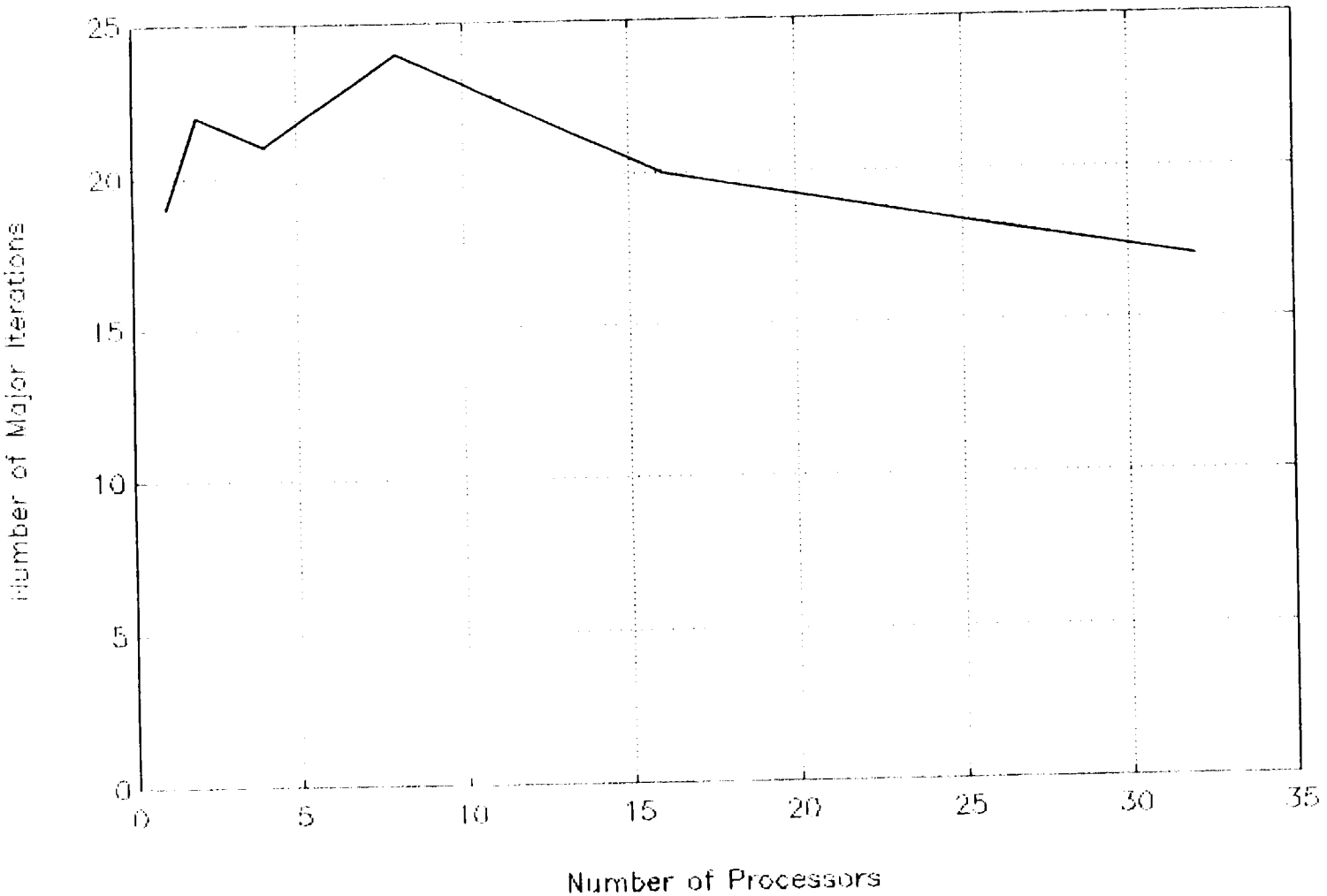


Fig. 9. The number of major algorithm iterations vs. the number of processors for the 64-stage Goddard problem.

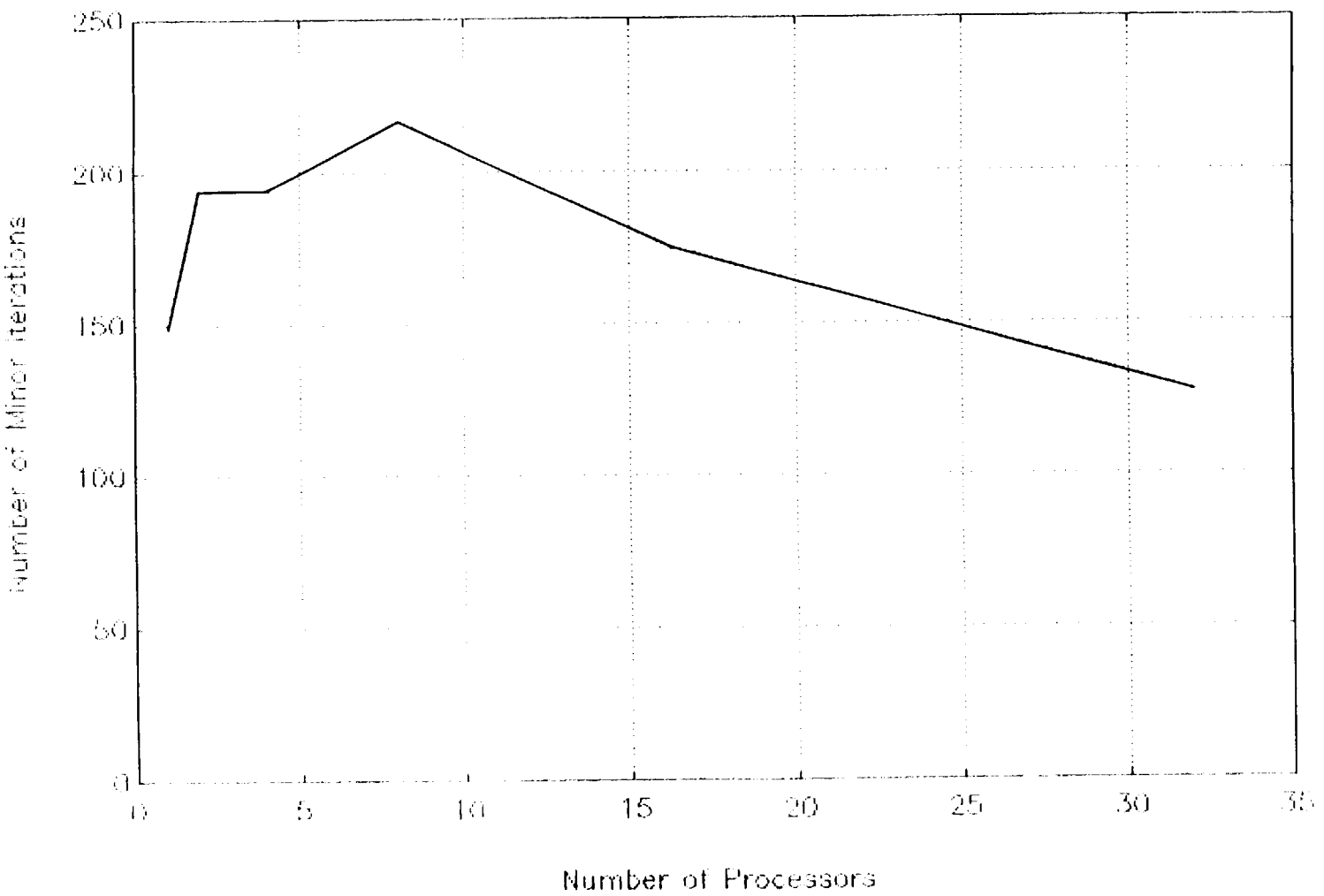


Fig. 10. The number of minor QP iterations vs. the number of processors for the 64-stage Goddard problem.

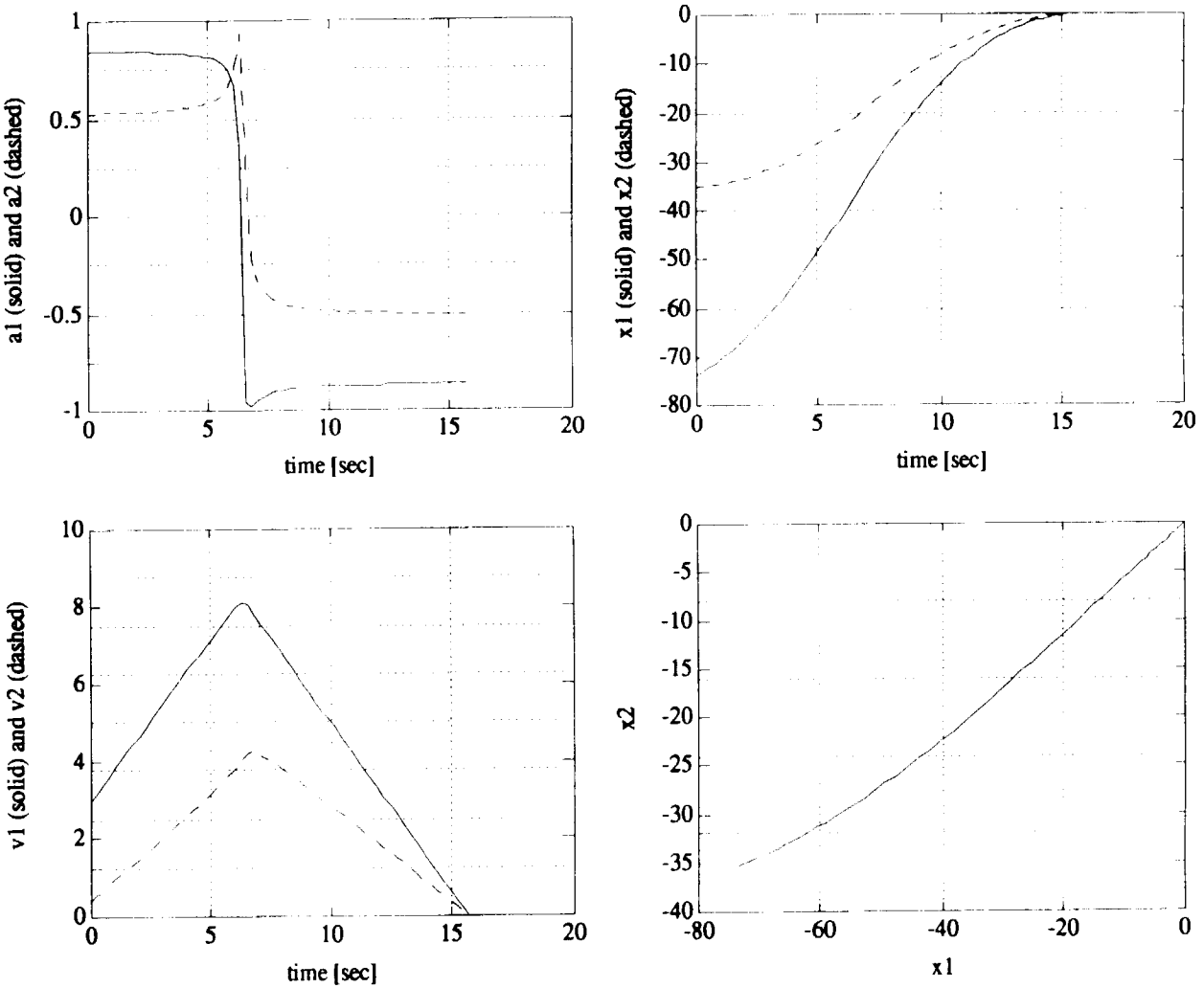


Fig. 11. Solution time histories and phase plot for the 64-stage minimum-time to the origin problem.

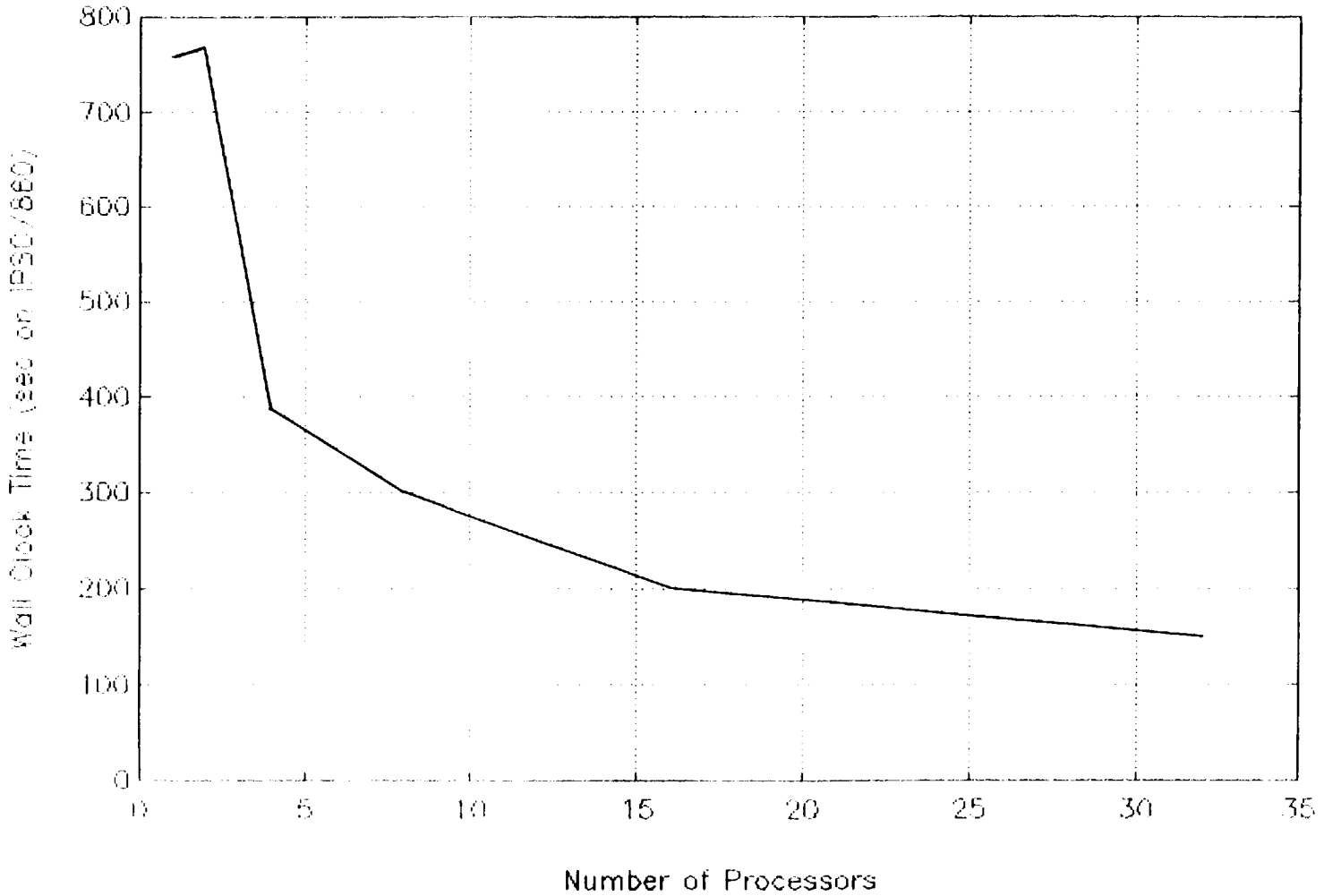


Fig. 12. Solution time vs. the number of processors for the 32-stage minimum-time to the origin problem.

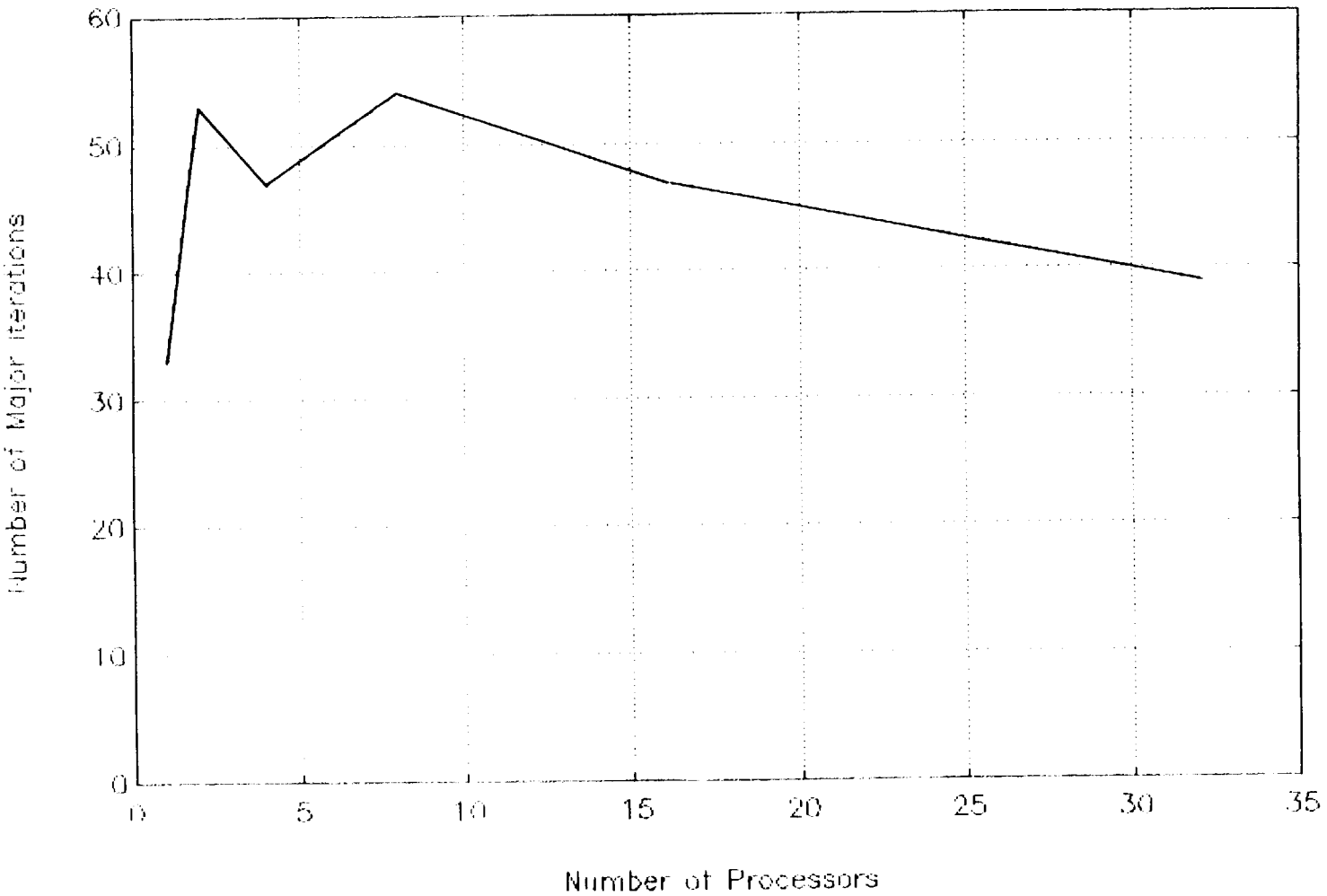


Fig. 13. The number of major algorithm iterations vs. the number of processors the the 32-stage minimum-time to the origin problem.

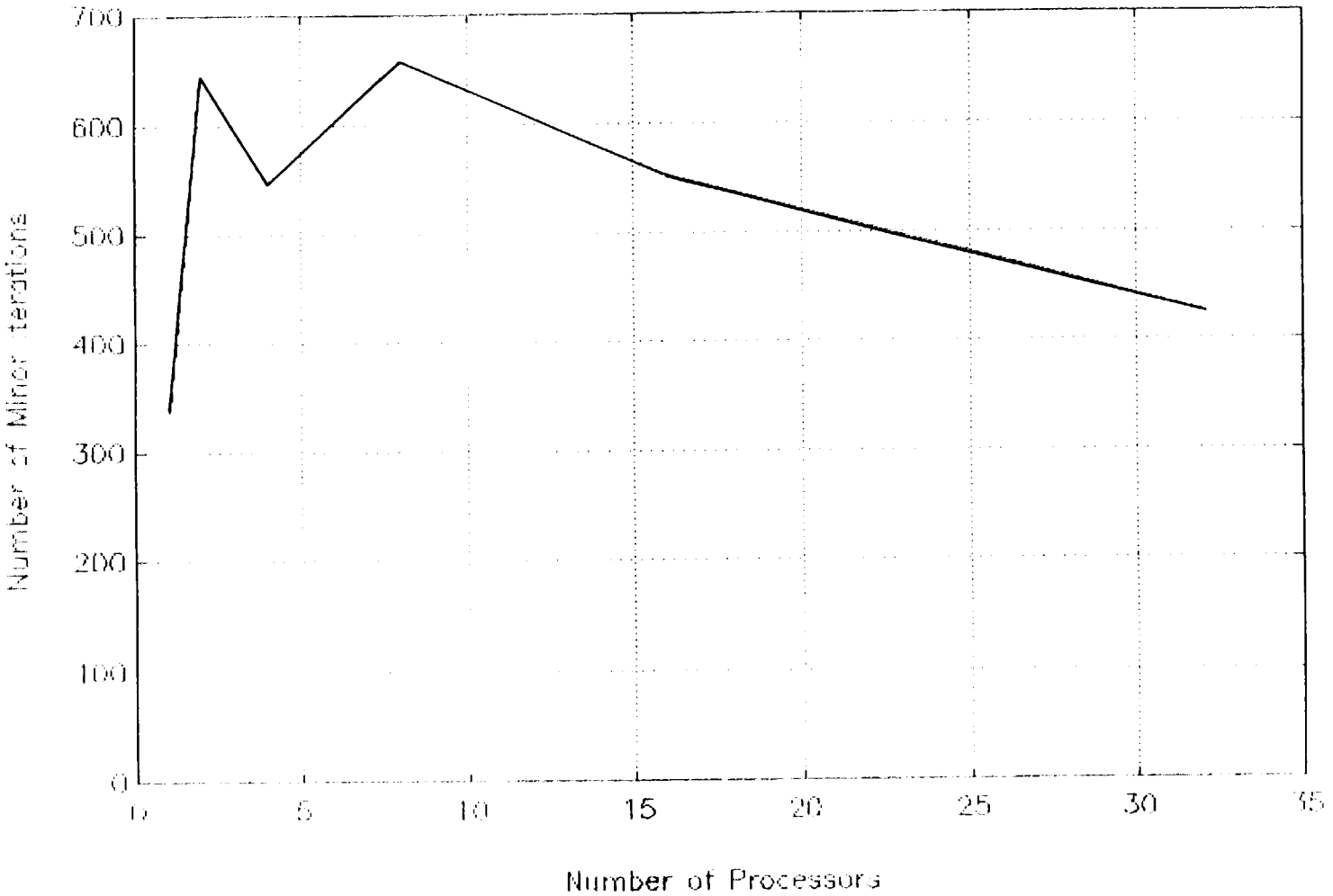


Fig. 14. The number of minor QP iterations vs. the number of processors the the 32-stage minimum-time to the origin problem.

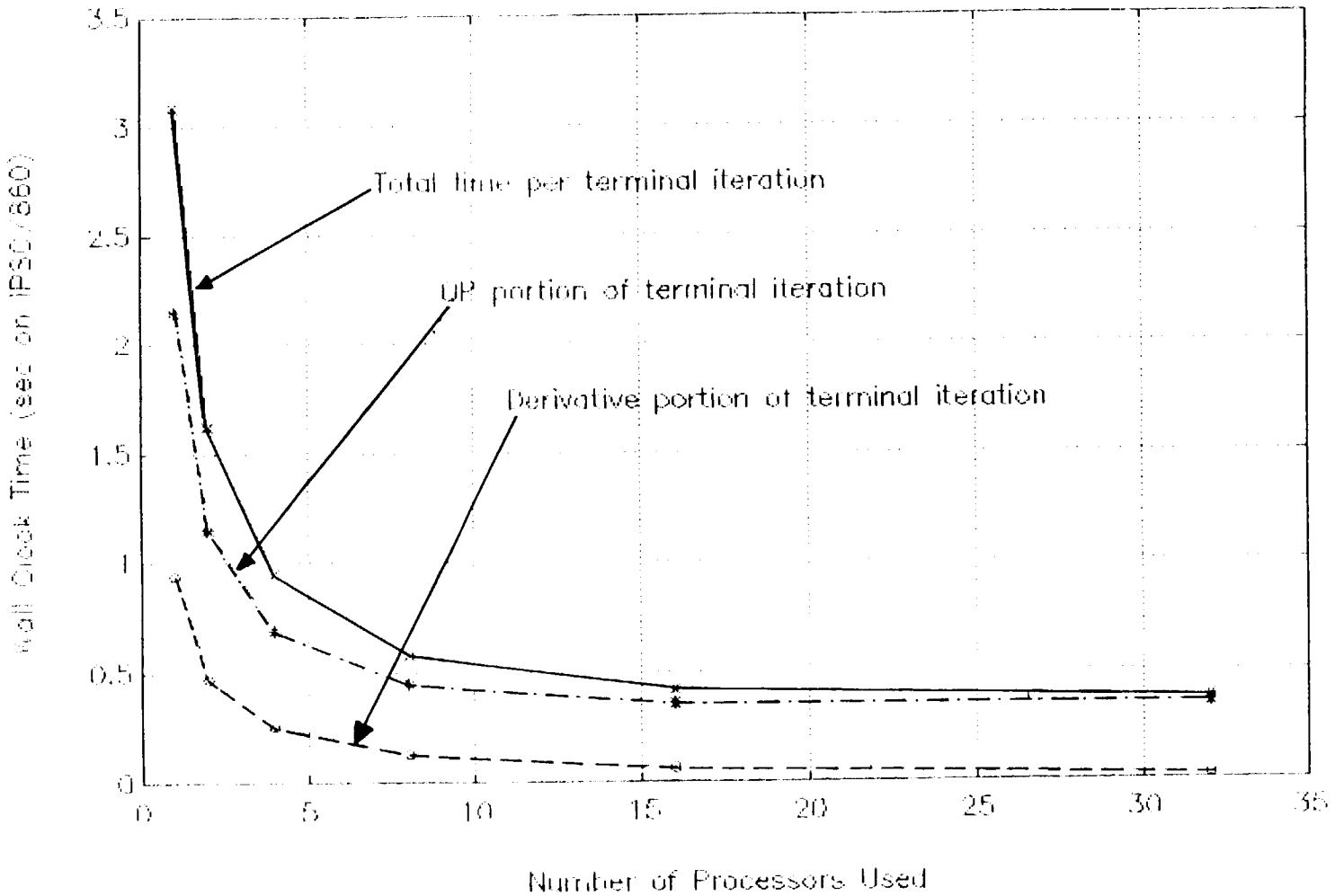


Fig. 15. Time per major iteration and its components when near the solution of the 32-stage minimum-time to the origin problem.

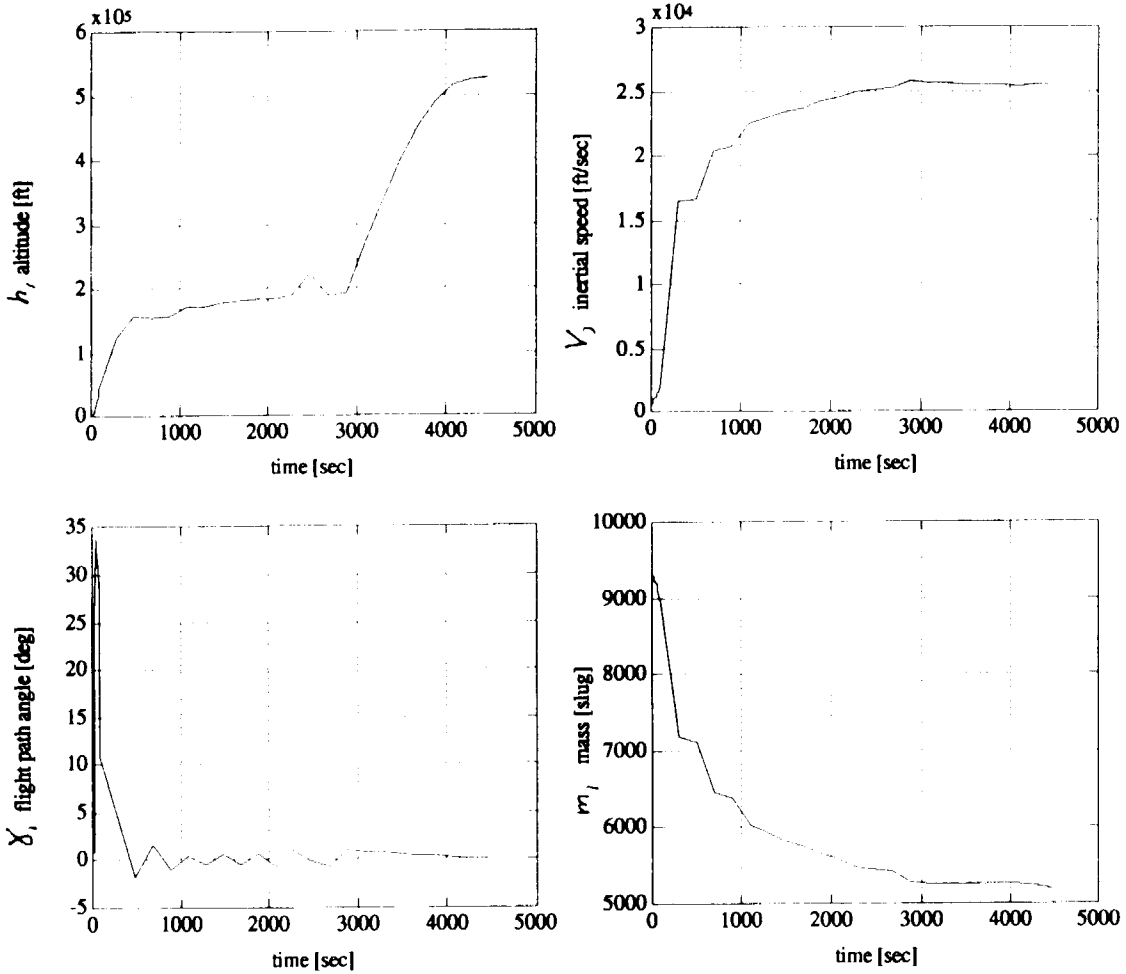


Fig. 16. State time histories for a 32-stage NASP ascent guidance problem.

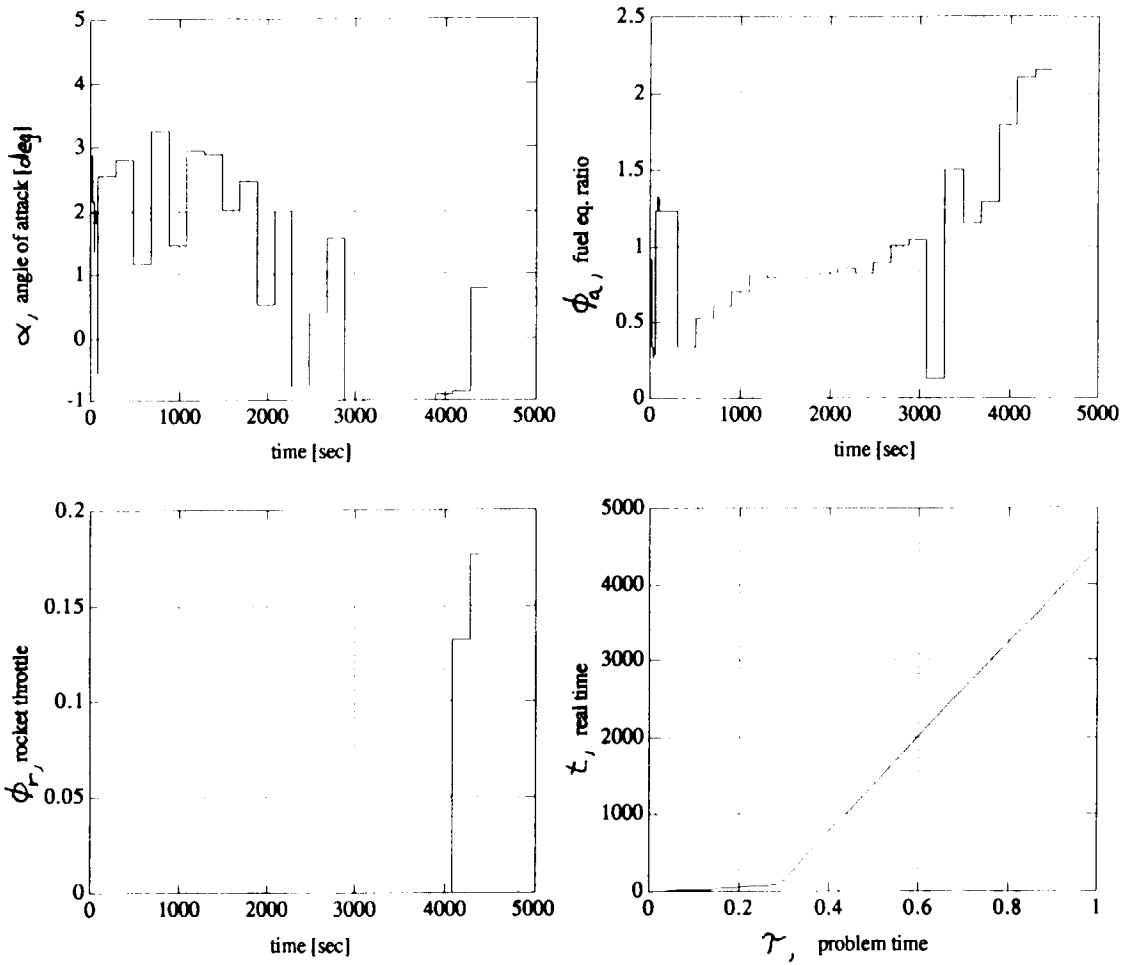


Fig. 17. Control time histories and the real-time/problem-time relationship for a 32-stage NASP ascent guidance problem.

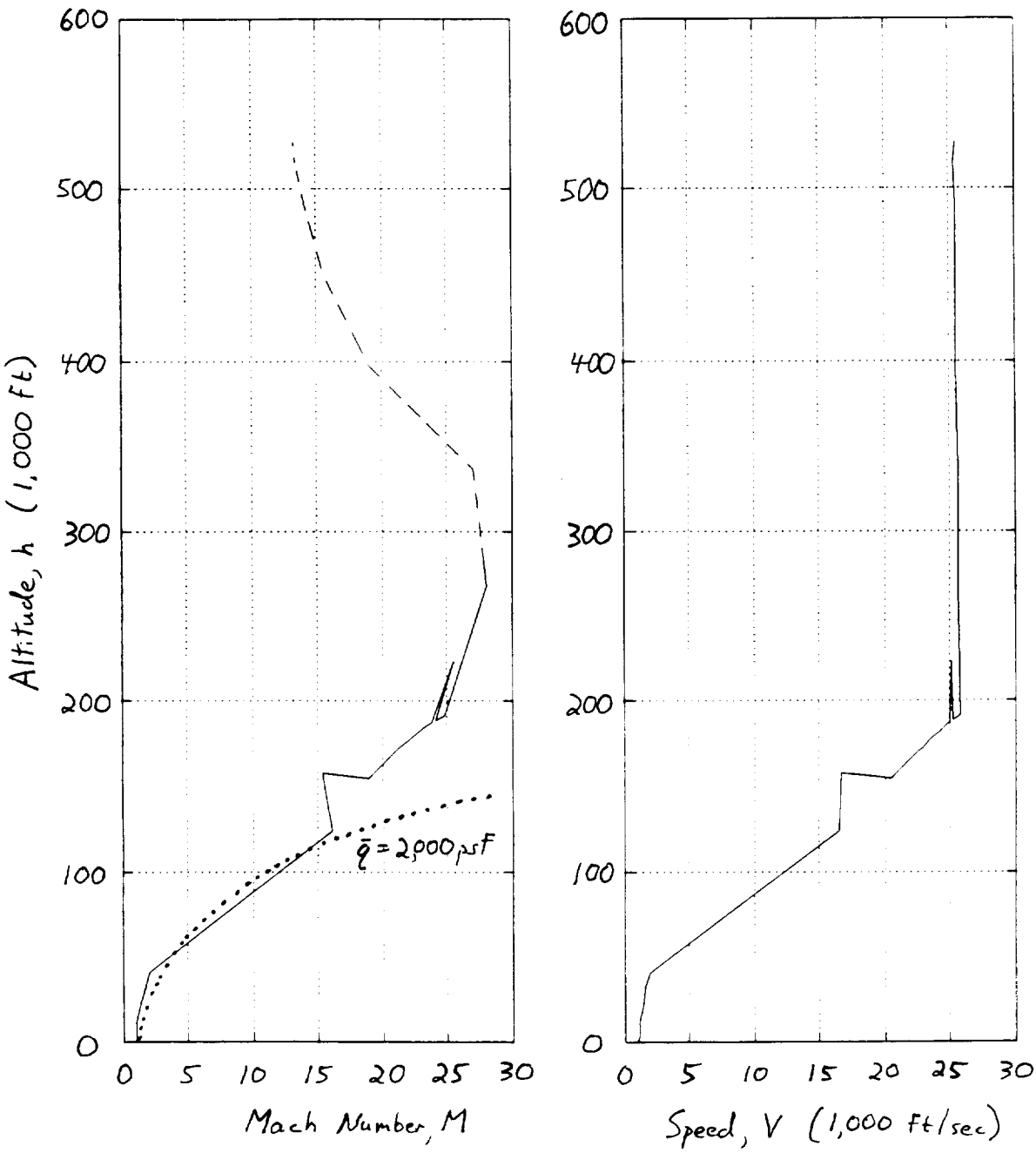


Fig. 18. Altitude vs. Mach number and altitude vs. speed contours for a 32-stage NASP ascent guidance problem.

Appendix A

Two papers on parallel dynamic quadratic programming.

A Parallel Solver for Trajectory Optimization Search Directions[1]

M. L. PSIAKI [2] and K. PARK[3]

[1] This research was supported in part by the National Aeronautics and Space Administration under Grant No. NAG-1-1009.

[2] Assistant Professor, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

[3] Graduate Research Assistant, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

Abstract. A key algorithmic element of a real-time trajectory optimization hardware/software implementation is presented, the search step solver. This is one piece of an algorithm whose overall goal is to make nonlinear trajectory optimization fast enough to provide real-time commands during guidance of a vehicle such as an aero-maneuvering orbiter or the National Aerospace Plane. Many methods of nonlinear programming require the solution of a quadratic program (QP) at each iteration to determine the search step. In the trajectory optimization case the QP has a special dynamic programming structure, an LQR-like structure. The algorithm exploits this special structure with a divide and conquer type of parallel implementation. A hypercube message-passing parallel machine, the INTEL iPSC/2, has been used. The algorithm solves a $(p \cdot N)$-stage problem on N processors in $O(p+\log_2 N)$ operations. The algorithm yields a factor of 8 speed-up over the fastest known serial algorithm when solving a 1024-step test problem on 32 processors.

Key Words. Trajectory optimization, parallel processing, quadratic programming, dynamic programming, linear quadratic regulator.

1. Introduction

The present work is part of an effort to do real-time optimal guidance by repeatedly solving trajectory optimization problems on line. The form of the nonlinear problem that must get solved is

$$\text{find:} \qquad \underline{\mathbf{x}} = \left[\underline{u}_0^T, \underline{x}_1^T, \underline{u}_1^T, \underline{x}_2^T, \ldots, \underline{u}_{N-1}^T, \underline{x}_N^T, \underline{u}_N^T\right]^T \tag{1a}$$

$$\text{to minimize:} \qquad J = \sum_{k=0}^{N} L(\underline{x}_k, \underline{u}_k, k) \tag{1b}$$

$$\text{subject to:} \qquad \underline{x}_0 \text{ given} \tag{1c}$$

$$\underline{x}_{k+1} = \underline{f}(\underline{x}_k, \underline{u}_k, k) \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{1d}$$

$$\underline{a}(\underline{x}_k, \underline{u}_k, k) \left\{ \begin{matrix} = \\ \leq \end{matrix} \right\} 0 \qquad \text{for } k = 0 \ldots N \tag{1e}$$

where $\underline{x}_k$ is the state vector at stage k, and $\underline{u}_k$ is the control vector. The nonlinear difference equation [Eq. (1d)] defines the system's dynamics, and Eq. (1e) defines any auxiliary constraints. Note that control vector $\underline{u}_N$ is associated with the terminal stage N, which is unusual, but possible. Also, the number of elements in the state vector and control vector may vary from stage to stage. The word stage is used to refer to a single discrete time step, not to a system staging process such as booster separation on a launch vehicle.

Trajectory optimization is a mature discipline, and algorithms exist for solving such problems (e.g., Refs. 1-7), but none can be used for real-time guidance. This paper is part of a research effort aimed at developing a parallel trajectory optimization algorithm that can qualify for real-time use via improved speed and convergence reliability. This paper partly addresses the issue of algorithm speed by developing a parallel implementation that reduces the time required to determine the search direction for a single iteration of a trajectory optimization algorithm. It is applicable to any trajectory optimization algorithm that generates search steps and that is formulated to work on a discrete-time system.

Many search-based trajectory optimization algorithms compute the search direction by solving a linear-equality-constrained, quadratic-cost problem. Such algorithms use active set strategies, which guess that all of the equality constraints and a subset of the inequality constraints are satisfied as exact equalities. The linear equality constraints of the search-step subproblem are

just the linearizations of the constraints in the active set [Eq. (1d) and a subset of the rows of Eq. (1e)]. In Newton's method, the Hamiltonian is approximated by retaining Taylor series terms out to the quadratics; this results in a problem with a quadratic cost. Even the steepest-descent method effectively uses a quadratic cost model: the Hamiltonian is expanded out to the linear terms, and its Hessian is approximated by the identity matrix in the control diagonal block and zeros elsewhere. Such problems are similar to time-varying LQR problems. The present paper concentrates on the development of an efficient means of solving such problems on a distributed-memory message-passing parallel processor with a hypercube connection topology.

A number of other researchers have attempted to speed up the solution of problems like Eqs. (1a)-(1e) via use of parallel processors. Larson and Tse tried to speed up Bellman's state/control discretization approach to dynamic programming (Ref. 8). Menon and Lehman proposed a method based on integrating matrices and general parallel matrix solvers (Ref. 9). Travassos and Kaufman proposed methods based on the TPBVP approach and a general parallel optimization package applied to enforce satisfaction of the terminal boundary condition (Ref. 10). Chang et. al. decompose the problem into shorter problems of several stages each (Ref. 11), first optimizing each group separately, in parallel, then optimizing the interconnected problem in an outer optimization loop. Betts and Huffman have calculated cost gradients and constraint Jacobians in parallel (Ref. 12). Wright proposes calculating the gradients, Jacobians, and Hessians in parallel and develops an algorithm to solve for the Newton step with a parallel factorization of the linearized necessary conditions (Ref. 13).

The present algorithm is a divide-and-conquer type algorithm and is closely related to Wright's method of factorizing the linear necessary conditions. Both methods are, essentially, special factorizations of the Kuhn-Tucker matrix. The present method has capabilities that Wright's method lacks: it can handle auxiliary state constraints, and it can detect and correct for an indefinite projected Hessian. The present factorization algorithm is interpreted as a series of parallel partial solutions of single-stage optimization problems followed by grouping of pairs of stages into single longer stages. Thus, parallelization is accomplished on a stage-wise basis.

Repeated application eventually reduces the problem to a single stage. This parallel reduction of the number of problem stages is similar to the approach that Chang et. al. use for the nonlinear problem (Ref. 11), though they only go through one reduction cycle. Two versions of the basic algorithm are presented: the "pure" algorithm, which assumes that the number of stages equals the number of processors, and a "hybrid" algorithm for use when the number of stages exceeds the number of processors.

The next section of the paper reviews general equality-constrained QP procedures and the usual backwards sweep method of solving such problems on a serial processor. Section 3 presents the outline and details of the parallel QP algorithm. It also gives details of a benchmark serial algorithm that has been used to determine the speed-up due to parallelism. Section 4 describes the test problem for the algorithm and gives timing results for the parallel and serial algorithms. Section 5 includes further discussion of the parallel algorithm. Section 6 concludes the paper.

2. Generalized Time-Varying LQR Problem with Equality Constraints

Algorithms for solving Eqs. (1a)-(1e) start with a guessed solution time history $\underline{x}^{(0)}$ and generate a sequence of time histories $\underline{x}^{(1)},\underline{x}^{(2)},\underline{x}^{(3)},\ldots$ that converges to the optimum. In Newton's method (with an active-set strategy for the inequalities), the search direction $\delta\underline{x}$ is computed at each iteration by solving a problem that looks like a time-varying discrete-time LQR problem, with some extra elements:

$$\text{find:} \qquad \delta\underline{x} = \left[\delta\underline{u}_0^T, \delta\underline{x}_1^T, \delta\underline{u}_1^T, \delta\underline{x}_2^T, \ldots, \delta\underline{u}_{N-1}^T, \delta\underline{x}_N^T, \delta\underline{u}_N^T\right]^T \tag{2a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{N} \left\{ \frac{1}{2} \begin{bmatrix} \delta\underline{x}_k^T, \delta\underline{u}_k^T \end{bmatrix} \begin{bmatrix} H_{xx_k} & H_{xu_k} \\ H_{xu_k}^T & H_{uu_k} \end{bmatrix} \begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_k \end{bmatrix} + \begin{bmatrix} g_{x_k}^T, g_{u_k}^T \end{bmatrix} \begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_k \end{bmatrix} \right\} \tag{2b}$$

$$\text{subject to:} \quad \delta\underline{x}_0 \text{ given} \tag{2c}$$

$$\delta\underline{x}_{k+1} = F_k\,\delta\underline{x}_k + G_k\,\delta\underline{u}_k + \underline{z}_k \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{2d}$$

$$A_k \, \delta \underline{x}_k + B_k \, \delta \underline{u}_k + \underline{c}_k = 0 \qquad \text{for } k = 0 \ldots N \qquad (2e)$$

This is a minimization problem with a quadratic cost and linear equality constraints. The quadratic cost in Eq. (2b) comes from the second order approximation of the Hamiltonian about the current iterate. The H_{xx_k}, H_{xu_k}, and H_{uu_k} matrices correspond to the quadratic terms and the g_{x_k} and g_{u_k} vectors correspond to the linear terms. Equation (2d) is a linearization of Eq. (1d) about the current iterate. The F_k matrix is the state transition matrix, the G_k matrix is the control effectiveness matrix, and the $\underline{z}_k$ vector represents the amount by which Eq. (1d) is not satisfied at the current iterate. At each iteration, a set of inequality constraints in Eq. (1e) are assumed active and are enforced as equalities. They are linearized about the current iterate point to yield some of the linear equality constraints in Eq. (2e). The remainder of the linearized equality constraints in Eq. (2e) arise from the linearizations of any equality constraints in Eq. (1e). The A_k and B_k matrices make up the Jacobian of the active constraints, and the $\underline{c}_k$ vector's elements represent the amounts by which active constraints in Eq. (1e) are not satisfied at the current iterate.

The differences between the problem of Eqs. (2a)-(2e) and a standard time-varying, discrete-time LQR problem are the following: the presence of auxiliary constraints [Eq. (2e)], the presence of nonhomogeneous terms in the state difference equation, $\underline{z}_k$, and the presence of the linear cost terms. Also, the quadratic cost matrices need not satisfy the conditions necessary for LQR stability. The search step calculation problems of most trajectory optimization algorithms can be transformed into this special form (e.g., those of Refs. 5 and 7).

2.1. Variable Reduction Method for General Equality-Constrained Quadratic Programming. The variable reduction method uses the constraints to solve for some of the problem variables in terms of the remaining variables (Ref. 14). If the stagewise quantities in Eqs. (2a)-(2e) are lumped together, the following problem results:

$$\text{find:} \qquad \delta \underline{\chi} \qquad (3a)$$

$$\text{to minimize:} \qquad J = \tfrac{1}{2} \delta \underline{\chi}^T \mathcal{H} \, \delta \underline{\chi} + \underline{\mathcal{g}}^T \delta \underline{\chi} \qquad (3b)$$

subject to: $\mathcal{A}\,\delta\underline{x} + \underline{c} = 0$ (3c)

The cost in Eq. (3b) includes all the terms in Eq. (2b) combined into a large Hessian matrix and gradient vector. The constraints in Eq. (3c) correspond to all of the constraints in Eqs. (2c)-(2e) for all of the stages (including the difference equations), which get lumped into the large $\mathcal{A}$ matrix and the large $\underline{c}$ vector.

In the variable reduction method, the $\mathcal{A}$ matrix in Eq. (3c) is split into $[\mathcal{A}_1, \mathcal{A}_2]$ such that $\mathcal{A}_1$ is a square matrix, and all the remaining quantities except $\underline{c}$ in Eqs. (3a)-(3c) are split in accordance with this:

find: $\delta\underline{x}_1, \delta\underline{x}_2$ (4a)

to minimize: $$J = \frac{1}{2}\begin{bmatrix}\delta\underline{x}_1^T, \delta\underline{x}_2^T\end{bmatrix}\begin{bmatrix}\mathcal{H}_{11} & \mathcal{H}_{12} \\ \mathcal{H}_{12}^T & \mathcal{H}_{22}\end{bmatrix}\begin{bmatrix}\delta\underline{x}_1 \\ \delta\underline{x}_2\end{bmatrix} + \begin{bmatrix}\underline{g}_1^T, \underline{g}_2^T\end{bmatrix}\begin{bmatrix}\delta\underline{x}_1 \\ \delta\underline{x}_2\end{bmatrix}$$ (4b)

subject to: $$[\mathcal{A}_1, \mathcal{A}_2]\begin{bmatrix}\delta\underline{x}_1 \\ \delta\underline{x}_2\end{bmatrix} + \underline{c} = 0$$ (4c)

If $\mathcal{A}_1$ is nonsingular, the variable reduction technique eliminates $\delta\underline{x}_1$ and the constraints from the problem. The vector $\delta\underline{x}_1$ is replaced by its constrained value:

$$\delta\underline{x}_1 = -\mathcal{A}_1^{-1}[\mathcal{A}_2\delta\underline{x}_2 + \underline{c}]$$ (5)

Substitution of Eq. (5) into Eq. (4b) leaves a cost that depends only upon $\delta\underline{x}_2$, and the constraint in Eq. (4c) is satisfied for all values of $\delta\underline{x}_2$ by virtue of Eq. (5). Therefore, unconstrained minimization of the resulting cost function can be performed with respect to $\delta\underline{x}_2$.

2.2. Sequential Variable Reduction/Partial Solution to Generate the Matrix-Riccati Equation. The time-varying matrix Riccati Equation of discrete-time LQR theory is the result of a QP solution procedure that is akin to the variable reduction technique. An appropriate name would be the sequential variable reduction/partial solution technique. It allows exploitation

of the special structure of the cost and constraints in Eqs. (2a)-(2e). It involves repeated application of a two-stage process. First, it uses some of the constraints to eliminate some of the variables, as in the variable reduction technique. Second, it optimizes any variables that become unconstrained after the first stage. This optimization process involves writing down the gradients of the cost with respect to the unconstrained variables and solving the resulting equations for the unconstrained variables. These unconstrained variables are expressed in terms of the remaining problem variables. Substitution of this expression for the unconstrained variables back into the cost eliminates these variables.

Assuming no control variable at the terminal stage and no auxiliary equality constraints [no Eq. (2e)] the matrix-Riccati technique performs a variable reduction by using the stage N-1 difference equation to eliminate $\delta\underline{x}_N$ from the problem. This results in the stage N-1/stage N cost expression:

$$J_{N-1,N} = \frac{1}{2}\left[\delta\underline{x}_{N-1}^T, \delta\underline{u}_{N-1}^T\right]\begin{bmatrix} (H_{xx_{N-1}}+F_{N-1}^T H_{xx_N} F_{N-1}) & (H_{xu_{N-1}}+F_{N-1}^T H_{xx_N} G_{N-1}) \\ (H_{xu_{N-1}}+F_{N-1}^T H_{xx_N} G_{N-1})^T & (H_{uu_{N-1}}+G_{N-1}^T H_{xx_N} G_{N-1}) \end{bmatrix}\begin{bmatrix}\delta\underline{x}_{N-1} \\ \delta\underline{u}_{N-1}\end{bmatrix}$$
$$+\left[\{g_{x_{N-1}}+F_{N-1}^T(g_{x_N}+H_{xx_N}\underline{z}_N)\}^T, \{g_{u_{N-1}}+G_{N-1}^T(g_{x_N}+H_{xx_N}\underline{z}_N)\}^T\right]\begin{bmatrix}\delta\underline{x}_{N-1} \\ \delta\underline{u}_{N-1}\end{bmatrix}$$
$$+\frac{1}{2}\underline{z}_N^T H_{xx_N}\underline{z}_N + g_{x_N}^T\underline{z}_N \qquad (6)$$

The stage N-1 control no longer appears in any of the constraints, and it only appears in the portion of the cost expressed in Eq. (6). Therefore, $\delta\underline{u}_{N-1}$ can be optimized out of the problem. Optimization of Eq. (6) with respect to $\delta\underline{u}_{N-1}$ yields:

$$\delta\underline{u}_{N-1} = -(H_{uu_{N-1}}+G_{N-1}^T H_{xx_N} G_{N-1})^{-1}\{(H_{xu_{N-1}}+F_{N-1}^T H_{xx_N} G_{N-1})^T\delta\underline{x}_{N-1}$$
$$+ g_{u_{N-1}} + G_{N-1}^T(g_{x_N}+H_{xx_N}\underline{z}_N)\} \qquad (7)$$

Equation (7) is then used to eliminate $\delta \underline{u}_{N-1}$ from Eq. (6). This results in a stage N-1/stage N cost that depends only on $\delta \underline{x}_{N-1}$. The resulting formula for the Hessian of that cost is just the discrete-time matrix Riccati equation.

3. Parallel Stage-Halving Algorithm

The principle of variable reduction followed by partial optimization is also used in the parallel factorization algorithms of this paper. The algorithms also include the auxiliary equality constraints, Eq. (2e). Two algorithms will be described. The first, described below and in section 3.2, is for the situation where the number of processors equals the number of stages. Section 3.3 describes a hybrid between section 3.2's stage halving algorithm and a serial backwards sweep algorithm. It is useful when the number of problem stages exceeds the number of processors.

Three main differences distinguish the stage-halving algorithm from the backwards sweeping method. First, the auxiliary constraints are used to reduce the number of control variables, if possible. Second, QR factorization is used, when possible, to free control variables from dependence on difference equation constraints. This allows partial optimization of control variables at early stages. Both of these operations can be carried out in parallel. Neither of these operations changes the number of stages or the number of state variables at each stage, but they both can reduce the number of controls at each stage, and the first operation can reduce the number of auxiliary constraints at each stage.

The third main deviation from the backwards sweep algorithm is that this algorithm follows its control vector/constraint reduction with a halving of the number of stages. This is accomplished by a standard variable reduction. State vectors $\delta \underline{x}_1$, $\delta \underline{x}_3$, ..., $\delta \underline{x}_N$ (assume N is odd) are eliminated by substitution of state difference equations 0,2,...,N-1 into the cost and the remaining auxiliary constraints. This is done in parallel and yields a problem of the same form as Eqs. (2a)-(2e), but with half as many stages; each stage corresponds to a doubled discrete-time interval. The sizes of the state vectors at the remaining stages are unchanged. The number of control vectors and the number of auxiliary constraints per stage both grow, but are bounded -- recall that the operations

just prior to stage halving both serve to reduce the number of constraints and controls. In both cases, the bound equals twice the dimension of the state vector.

The algorithm then repeats its cycle of reducing the number of control vector elements and constraints followed by a halving of the number of stages. This goes on until only one stage is left. That stage is completely solved, and the result is back-substituted into the preceding 2-stage problem and so forth until the entire solution is generated. The time to perform these operations on N processors goes as $n^3\log_2(N)$, where n is the (average) number of state vector elements.

The only message passing of the above-outlined algorithm occurs during the stage-halving cycle and during the backwards substitution. The ideal architecture for such operations is a binary tree. Such a tree can be implemented on a generic hypercube architecture. The algorithm has been implemented on a 32-node hypercube, the INTEL iPSC/2 at the Cornell Theory Center.

Figure 1 displays the binary tree and the initial distribution of problem stages on 16 processors (nodes of the tree) for a 16-stage problem. The arrows indicate the message passing that occurs during the first stage-number halving. The bottom plot indicates the processor locations of the remains of the problem stages after this halving process.

3.1. Stage-Having Algorithm on an Example Problem. In this section, a simple 4-stage QP problem is solved using the above algorithmic idea. This should help understand the detailed algorithm that is described in the next section.

The example problem to be solved here is:

$$\text{find:} \quad [u_0, x_1, u_1, x_2, u_2, x_3]^T \tag{8a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{2} \frac{1}{2}(x_k^2 + u_k^2) \tag{8b}$$

$$\text{subject to:} \quad x_0 = 1 \tag{8c}$$

$$x_{k+1} = x_k + u_k \qquad \text{for } k = 0,1,2 \tag{8d}$$

$$x_3 = 4 \tag{8e}$$

where all of the variables are scalar quantities. Table 1 shows this problem in a stagewise manner.

Table 1. Stages of original example problem.

Stage	Dynamic equations(D/E)	Auxiliary constraints(A/C)	Cost
0	$x_1 = 1 + u_0$	none	$\frac{1}{2}(1 + u_0^2)$
1	$x_2 = x_1 + u_1$	none	$\frac{1}{2}(x_1^2 + u_1^2)$
2	$x_3 = x_2 + u_2$	none	$\frac{1}{2}(x_2^2 + u_2^2)$
3	none	$x_3 = 4$	none

The first step in this example is to halve the number of stages by joining stages in pairs using the dynamic equations, i.e., x_1 is eliminated from the problem using the dynamic equation for stage 0, and x_3 is eliminated from the problem using the dynamic equation for stage 2. Table 2 shows what the problem becomes after this operation. Stage 0' in Table 2 represents the {0,1} stage pair of the original problem and stage 1' represents the {2,3} pair. The new variables in Table 2 are defined as follows:

$$\underline{u}_{0'} = \begin{bmatrix} u_0 \\ u_1 \end{bmatrix}, \qquad x_{1'} = x_2, \qquad u_{1'} = u_2 \tag{9}$$

Table 2. Example problem after stage halving.

Stage	D/E	A/C	Cost
0'	$x_{1'} = 1 + [1, 1]\,\underline{u}_{0'}$	none	$\frac{1}{2}\underline{u}_{0'}^T \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \underline{u}_{0'} + [1, 0]\,\underline{u}_{0'} + 1$
1'	none	$x_{1'} + u_{1'} = 4$	$\frac{1}{2}(x_{1'}^2 + u_{1'}^2)$

The next step is to remove $u_{1'}$ from the problem using the auxiliary equation, $x_{1'} + u_{1'} = 4$, which yields $u_{1'} = 4 - x_{1'}$. This expression is used to eliminate $u_{1'}$ from the cost at stage 1'. Table 3 shows the resulting problem.

Table 3. Example problem after elimination of auxiliary constraints.

Stage	D/E	A/C	Cost
0'	$x_{1'} = 1 + [1, 1]\, \underline{u}_{0'}$	none	$\frac{1}{2}\underline{u}_{0'}^T \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \underline{u}_{0'} + [1, 0]\, \underline{u}_{0'} + 1$
1'	none	none	$x_{1'}^2 - 4\, x_{1'} + 8$

The next step is to QR factorize the [1, 1] matrix in the dynamic equation of stage 0'. This right QR factorization transforms $\underline{u}_{0'}$ so that one of the transformed control vector elements does not enter the dynamic equation.

$$[1, 1]\, Q = [\sqrt{2}, 0] \qquad \text{where } Q = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \text{ an orthogonal matrix.} \tag{10a}$$

$$Q \begin{bmatrix} u_{3_{0'}} \\ u_{4_{0'}} \end{bmatrix} = \underline{u}_{0'} \tag{10b}$$

where $u_{3_{0'}}$ and $u_{4_{0'}}$ are the elements of the transformed control vector. Table 4 shows the resulting problem.

Table 4. Example problem after orthogonal transformation of control vector.

Stage	D/E	A/C	Cost
0'	$x_{1'} = 1 + \sqrt{2}\, u_{3_{0'}}$	none	$\frac{1}{4}\begin{bmatrix} u_{3_{0'}} & u_{4_{0'}} \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} u_{3_{0'}} \\ u_{4_{0'}} \end{bmatrix} + \frac{1}{\sqrt{2}} [1, 1] \begin{bmatrix} u_{3_{0'}} \\ u_{4_{0'}} \end{bmatrix} + 1$
1'	none	none	$x_{1'}^2 - 4\, x_{1'} + 8$

The next step eliminates $u_{4_{0'}}$ from the problem. Since it does not enter the constraints, the cost can be optimized with respect to $u_{4_{0'}}$. By setting to zero the partial derivative of the cost with respect to $u_{4_{0'}}$, one can derive the formula

$$u_{4_{0'}} = -\frac{1}{3} u_{3_{0'}} - \frac{\sqrt{2}}{3} \tag{11}$$

Equation (11) can be used to eliminate $u_{4_{0'}}$ from the problem. The resulting problem appears in Table 5.

Table 5. Example problem after partial optimization of controls.

Stage	D/E	A/C	Cost
0'	$x_{1'} = 1 + \sqrt{2}\, u_{3_{0'}}$	none	$\frac{2}{3} u_{3_{0'}}^2 + \frac{\sqrt{2}}{3} u_{3_{0'}} + \frac{5}{6}$
1'	none	none	$x_{1'}^2 - 4\, x_{1'} + 8$

The next step joins stage 0' and 1', eliminating $x_{1'}$ from the problem using the dynamic equation from stage 0'. At this point, the original constrained 4-stage problem has been reduced to the following unconstrained 1-stage problem:

$$\text{find:} \quad u_{3_{0'}} \text{ to minimize } J = \frac{8}{3} u_{3_{0'}}^2 - \frac{5\sqrt{2}}{3} u_{3_{0'}} + \frac{35}{6} \tag{12}$$

The solution is easily found, and it is $u_{3_{0'}} = \frac{5\sqrt{2}}{16}$.

Back Substitution

Once $u_{3_{0'}}$ is known, the optimal values of the original variables in (8a) are found by back substitution. For example, u_0 is found as follows: first, $u_{4_{0'}}$ is computed by Eq. (11), and $u_{4_{0'}}$

together with u_{3_0}, yields $\underline{u}_0$, by Eq. (10b), u_0 is just the first entry of $\underline{u}_0$, from Eq. (9). It is $u_0 = -1/8$ in this case. Other variables are found in a similar way.

3.2. Details of General Algorithm.

Steps and Detailed Operations for One Stage-Halving Cycle

Step 1. Completely QR factorize the B_k auxiliary constraint matrix (in parallel):

$$Q_{1_k} B_k Q_{2_k} = \begin{bmatrix} R_{1_k} & 0 \\ 0 & 0 \end{bmatrix} \tag{13}$$

where Q_{1_k} and Q_{2_k} are orthogonal and R_{1_k} is square and upper triangular.

This effectively transforms the control coordinates and the constraint equations, splitting the controls into those that are in the auxiliary constraints and those that are not, and splitting the constraints into those that involve controls and those that do not. Note, the matrix on the right may not always take the most general form shown above. The factorization effort can be reduced by taking advantage of any rows and columns of B_k that are already zero.

This factorization necessitates a reformulation of other problem matrices to account for the reformulated control variables and constraints:

$$\begin{bmatrix} \delta\underline{u}_{1_k} \\ \delta\underline{u}_{2_k} \end{bmatrix} = Q_{2_k}^T \delta\underline{u}_k \tag{14a}$$

$$\begin{bmatrix} A_{1_k} \\ A_{2_k} \end{bmatrix} = Q_{1_k} A_k, \quad \begin{bmatrix} \underline{c}_{1_k} \\ \underline{c}_{2_k} \end{bmatrix} = Q_{1_k} \underline{c}_k, \quad [G_{1_k}, G_{2_k}] = G_k Q_{2_k} \tag{14b}$$

$$\begin{bmatrix} H_{11_k} & H_{12_k} \\ H_{12_k}^T & H_{22_k} \end{bmatrix} = Q_{2_k}^T H_{uu_k} Q_{2_k}, \quad [H_{x1_k}, H_{x2_k}] = H_{xu_k} Q_{2_k}, \quad \begin{bmatrix} g_{1_k} \\ g_{2_k} \end{bmatrix} = Q_{2_k}^T g_{u_k} \tag{14c}$$

The subscripts 1 and 2 on the new A matrices and $\underline{c}$ vectors refer to the transformed constraints; the 1 subscript corresponds to constraints that depend on controls, and the 2 subscript refers to constraints that are independent of controls. The 1 and 2 subscripts on the H and G matrices and on the g vectors refer to the index of the transformed control vector that gets multiplied by these matrices and vectors in the cost and in the state difference equations.

Step 2. Use the first auxiliary constraint to solve for $\delta\underline{u}_{1_k}$ in terms of $\delta\underline{x}_k$, and eliminate the first auxiliary constraint and $\delta\underline{u}_{1_k}$ from the problem (in parallel):

$$\delta\underline{u}_{1_k} = - R_{1_k}^{-1}[A_{1_k}\delta\underline{x}_k + \underline{c}_{1_k}] \tag{15}$$

This takes the form of a feedback control law. This is the necessary feedback to stay on the $(\,)_{1_k}$ transformed constraint. After elimination of $\delta\underline{u}_{1_k}$ and the $(\,)_{1_k}$ auxiliary constraint, the problem takes the form

find: $$\delta\underline{x} = \left[\delta\underline{u}_{2_0}^T, \delta\underline{x}_1^T, \delta\underline{u}_{2_1}^T, \delta\underline{x}_2^T, \ldots, \delta\underline{u}_{2_{N-1}}^T, \delta\underline{x}_N^T, \delta\underline{u}_{2_N}^T\right]^T \tag{16a}$$

to minimize: $$J = \sum_{k=0}^{N} \{ \frac{1}{2}\left[\delta\underline{x}_k^T, \delta\underline{u}_{2_k}^T\right]\begin{bmatrix} \tilde{H}_{xx_k} & \tilde{H}_{x2_k} \\ \tilde{H}_{x2_k}^T & H_{22_k} \end{bmatrix}\begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_{2_k} \end{bmatrix} + \left[\tilde{g}_{x_k}^T, \tilde{g}_{2_k}^T\right]\begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_{2_k} \end{bmatrix} + \text{constant}\} \tag{16b}$$

subject to: $$\delta\underline{x}_0 \text{ given} \tag{16c}$$

$$\delta\underline{x}_{k+1} = \tilde{F}_k\,\delta\underline{x}_k + G_{2_k}\,\delta\underline{u}_{2_k} + \tilde{\underline{z}}_k \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{16d}$$

$$A_{2_k}\,\delta\underline{x}_k + \underline{c}_{2_k} = 0 \qquad \text{for } k = 0 \ldots N \tag{16e}$$

which has the same form as the problem in Eqs. (2a)-(2e). The matrices and vectors with the (~) overstrike are derived by substitution of equations (14a)-(14c) and (15) into problem (2a)-(2e). They are

$$\tilde{H}_{xx_k} = H_{xx_k} - H_{x1_k}R_{1_k}^{-1}A_{1_k} - [H_{x1_k}R_{1_k}^{-1}A_{1_k}]^T + [R_{1_k}^{-1}A_{1_k}]^T H_{11_k}R_{1_k}^{-1}A_{1_k} \tag{17a}$$

$$\tilde{H}_{x2_k} = H_{x2_k} - [R_{1_k}^{-1}A_{1_k}]^T H_{12_k} \tag{17b}$$

$$\tilde{g}_{x_k} = g_{x_k} - \{H_{x1_k} - [R_{1_k}^{-1}A_{1_k}]^T H_{11_k}\}R_{1_k}^{-1}\underline{c}_{1_k} - [R_{1_k}^{-1}A_{1_k}]^T g_{1_k} \tag{17c}$$

$$\tilde{g}_{2_k} = g_{2_k} - H_{12_k}^T R_{1_k}^{-1}\underline{c}_{1_k} \tag{17d}$$

$$\tilde{F}_k = F_k - G_{1_k}R_{1_k}^{-1}A_{1_k}, \quad \tilde{\underline{z}}_k = \underline{z}_k - G_{1_k}R_{1_k}^{-1}\underline{c}_{1_k} \tag{17e}$$

Step 3. Right QR factorize G_{2_k} to eliminate extra controls from the state difference equation (in parallel).

$$G_{2_k}Q_{3_k} = [L_{3_k}, \ 0] \tag{18}$$

where Q_{3_k} is orthogonal and L_{3_k} is square and lower triangular.

This step can be done only if G_{2_k} has more columns than rows. Such may not be the case initially, but after several halvings of the number of stages, this will generally be true. Again, this factorization necessitates a reformulation of other problem matrices to account for the reformulated control variables:

$$\begin{bmatrix} \delta\underline{u}_{3_k} \\ \delta\underline{u}_{4_k} \end{bmatrix} = Q_{3_k}^T \delta\underline{u}_{2_k} \tag{19a}$$

$$\begin{bmatrix} H_{33_k} & H_{34_k} \\ H_{34_k}^T & H_{44_k} \end{bmatrix} = Q_{3_k}^T H_{22_k} Q_{3_k}, \quad [H_{x3_k}, H_{x4_k}] = \tilde{H}_{x2_k} Q_{3_k}, \quad \begin{bmatrix} \underline{g}_{3_k} \\ \underline{g}_{4_k} \end{bmatrix} = Q_{3_k}^T \tilde{\underline{g}}_{2_k} \tag{19b}$$

The 3 and 4 subscripts on the H matrices and on the $\underline{g}$ vectors refer to the index of the transformed control vector that gets multiplied by these matrices and vectors in the cost.

This step is one of the most costly parts of the whole algorithm due to the large size of the Q_3 matrix -- it is usually 2n x 2n where n is the state space dimension. The operation time can be reduced by exploiting the special structure of the G_2 matrix -- after several stage-halvings, the rightmost part of G_2 is already lower-triangular -- and by working with individual Householder vectors rather than explicitly forming the Q_3 matrix.

Step 4. Partially optimize by computing the optimal $\delta\underline{u}_{4_k}$ as a function of $\delta\underline{u}_{3_k}$ and $\delta\underline{x}_k$ (in parallel for all stages where step 3 has been carried out). Solve the equation

$$H_{44_k}\delta\underline{u}_{4_k} = -[H^T_{34_k}\delta\underline{u}_{3_k} + H^T_{x4_k}\delta\underline{x}_k + \underline{g}_{4_k}] \tag{20}$$

then eliminate $\underline{u}_{4_k}$ from the cost function by substituting this solution for it.

This equation can be solved using a Cholesky factorization of $H_{44_k} = R^T_{4_k}R_{4_k}$ and back substitution; R_{4_k} is upper triangular. This implies that the H_{44_k} matrix is positive definite. If H_{44_k} is not positive definite, then the search-step problem [Eqs. (2a)-(2e)] is ill-defined; it has an infinite minimum.

An important feature of this algorithm is that it can be used to detect whether or not the projected Hessian is indefinite. This is done during the Cholesky factorization of H_{44_k}. In the event of an indefinite H_{44_k}, the Cholesky factorization procedure can be modified to determine a diagonal modification to H_{44_k} which ensures that the solution of the (modified) Eqs. (2a)-(2e) is a descent direction of the original problem in Eqs. (1a)-(1e). Alternatively, the modified Cholesky

factorization procedure can be used to determine a direction of negative curvature (Ref. 14, pp. 108-111), which could be used as a search direction.

The solution of Eq. (20) looks somewhat like a feedback control law, in that $\delta\underline{u}_{4_k}$ depends on $\delta\underline{x}_k$, but the "control law" is peculiar in that $\delta\underline{u}_{4_k}$ also depends on $\delta\underline{u}_{3_k}$. Upon substitution of the solution into the cost, the problem becomes

$$\text{find:} \qquad \delta\underline{x} = \left[\delta\underline{u}_{3_0}^T, \delta\underline{x}_1^T, \delta\underline{u}_{3_1}^T, \delta\underline{x}_2^T, \ldots, \delta\underline{u}_{3_{N-1}}^T, \delta\underline{x}_N^T\right]^T \tag{21a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{N} \left\{ \frac{1}{2} \begin{bmatrix} \delta\underline{x}_k^T, \delta\underline{u}_{3_k}^T \end{bmatrix} \begin{bmatrix} \bar{H}_{xx_k} & \bar{H}_{x3_k} \\ \bar{H}_{x3_k}^T & \bar{H}_{33_k} \end{bmatrix} \begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_{3_k} \end{bmatrix} + \begin{bmatrix} \bar{g}_{x_k}^T, \bar{g}_{3_k}^T \end{bmatrix} \begin{bmatrix} \delta\underline{x}_k \\ \delta\underline{u}_{3_k} \end{bmatrix} + \text{constant} \right\} \tag{21b}$$

$$\text{subject to:} \quad \delta\underline{x}_0 \text{ given} \tag{21c}$$

$$\delta\underline{x}_{k+1} = \tilde{F}_k \, \delta\underline{x}_k + L_{3_k} \, \delta\underline{u}_{3_k} + \tilde{\underline{z}}_k \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{21d}$$

$$A_{2_k} \, \delta\underline{x}_k + \underline{c}_{2_k} = 0 \qquad \text{for } k = 0 \ldots N \tag{21e}$$

which has the same form as the problem in Eqs. (2a)-(2e). Note that there is no longer a control at the terminal stage -- $\delta\underline{u}_{3_N}$ is a vector of zero dimension. The matrices and vectors with the (-) overstrike are derived by substitution of the solution of Eq. (20) and Eqs. (19a)-(19b) into problem (16a)-(16e). They are

$$\bar{H}_{xx_k} = \tilde{H}_{xx_k} - [H_{x4_k} R_{4_k}^{-1}][H_{x4_k} R_{4_k}^{-1}]^T \tag{22a}$$

$$\bar{H}_{x3_k} = H_{x3_k} - [H_{x4_k} R_{4_k}^{-1}][H_{34_k} R_{4_k}^{-1}]^T \tag{22b}$$

$$\bar{H}_{33_k} = H_{33_k} - [H_{34_k} R_{4_k}^{-1}][H_{34_k} R_{4_k}^{-1}]^T \tag{22c}$$

$$\bar{g}_{x_k} = \tilde{g}_{x_k} - [H_{x4_k} R_{4_k}^{-1}][R_{4_k}^{-T} g_{4_k}] \tag{22d}$$

$$\bar{g}_{3_k} = g_{3_k} - [H_{34_k} R_{4_k}^{-1}][R_{4_k}^{-T} g_{4_k}] \tag{22e}$$

Step 5. Halve the number of stages by using the difference equations [Eq. (21d)] for stages k = 0, 2, 4,..., N-1 to eliminate state vectors $\delta\underline{x}_1$, $\delta\underline{x}_3$, $\delta\underline{x}_5$,..., $\delta\underline{x}_N$ from the cost and from the auxiliary constraints. Assume that N is odd (N = 2M+1). This is done in parallel and includes parallel message passing between M+1 pairs of processors.

This is the concluding step in the process of halving the number of problem stages. After this step the problem is back in the form of Eqs. (2a)-(2e) only with half as many stages -- M+1 stages instead of N+1:

find: $$\delta\underline{\hat{x}} = \left[\delta\hat{\underline{u}}_0^T, \delta\hat{\underline{x}}_1^T, \delta\hat{\underline{u}}_1^T, \delta\hat{\underline{x}}_2^T, \ldots, \delta\hat{\underline{u}}_{M-1}^T, \delta\hat{\underline{x}}_M^T, \delta\hat{\underline{u}}_M^T\right]^T \tag{23a}$$

to minimize: $$J = \sum_{k=0}^{M} \left\{ \frac{1}{2} \left[\delta\hat{\underline{x}}_k^T, \delta\hat{\underline{u}}_k^T\right] \begin{bmatrix} \hat{H}_{xx_k} & \hat{H}_{xu_k} \\ \hat{H}_{xu_k}^T & \hat{H}_{uu_k} \end{bmatrix} \begin{bmatrix} \delta\hat{\underline{x}}_k \\ \delta\hat{\underline{u}}_k \end{bmatrix} + \left[\hat{\underline{g}}_{x_k}^T, \hat{\underline{g}}_{u_k}^T\right] \begin{bmatrix} \delta\hat{\underline{x}}_k \\ \delta\hat{\underline{u}}_k \end{bmatrix} \right.$$

$$\left. + \text{constant} \right\} \tag{23b}$$

subject to: $$\delta\hat{\underline{x}}_0 \text{ given} \tag{23c}$$

$$\delta\hat{\underline{x}}_{k+1} = \hat{F}_k \delta\hat{\underline{x}}_k + \hat{G}_k \delta\hat{\underline{u}}_k + \hat{\underline{z}}_k \qquad \text{for } k = 0 \ldots M\text{-}1 \tag{23d}$$

$$\hat{A}_k \delta\hat{\underline{x}}_k + \hat{B}_k \delta\hat{\underline{u}}_k + \hat{\underline{c}}_k = 0 \qquad \text{for } k = 0 \ldots M \tag{23e}$$

where the vectors and matrices with the (^) overstrike are derived from substitution of alternate difference equations into Eqs. (21a)-(21e):

$$\delta\hat{\underline{u}}_k = \begin{bmatrix} \delta\underline{u}_{3_{2k}} \\ \delta\underline{u}_{3_{2k+1}} \end{bmatrix} \quad \text{for } k = 0,\ldots,M\text{-}1, \quad \delta\hat{\underline{u}}_M = \delta\underline{u}_{3_{2M}}, \quad \delta\hat{\underline{x}}_k = \delta\underline{x}_{2k} \quad \text{for } k = 0,\ldots,M \tag{24a}$$

$$\hat{F}_k = \tilde{F}_{2k+1}\tilde{F}_{2k}, \quad \hat{G}_k = \left[\tilde{F}_{2k+1}L_{3_{2k}},\ L_{3_{2k+1}}\right], \quad \hat{\underline{z}}_k = [\tilde{F}_{2k+1}\tilde{\underline{z}}_{2k} + \tilde{\underline{z}}_{2k+1}]$$

$$\text{for } k = 0,\ldots,M\text{-}1 \tag{24b}$$

$$\hat{A}_k = \begin{bmatrix} A_{2_{2k+1}}\tilde{F}_{2k} \\ A_{2_{2k}} \end{bmatrix}, \quad \hat{\underline{c}}_k = \begin{bmatrix} A_{2_{2k+1}}\tilde{\underline{z}}_{2k} + \underline{c}_{2_{2k+1}} \\ \underline{c}_{2_{2k}} \end{bmatrix} \quad \text{for } k = 0,\ldots,M \tag{24c}$$

$$\hat{B}_k = \begin{bmatrix} A_{2_{2k+1}}L_{3_{2k}} & 0 \\ 0 & 0 \end{bmatrix} \quad \text{for } k = 0,\ldots,M\text{-}1, \quad \hat{B}_M = \begin{bmatrix} A_{2_{2M+1}}L_{3_{2M}} \\ 0 \end{bmatrix} \tag{24d}$$

$$\hat{H}_{xx_k} = \bar{H}_{xx_{2k}} + \tilde{F}_{2k}^T \bar{H}_{xx_{2k+1}}\tilde{F}_{2k} \quad \text{for } k = 0,\ldots,M \tag{24e}$$

$$\hat{H}_{xu_k} = \left[[\bar{H}_{x3_{2k}} + \tilde{F}_{2k}^T \bar{H}_{xx_{2k+1}}L_{3_{2k}}],\ [\tilde{F}_{2k}^T \bar{H}_{x3_{2k+1}}]\right] \quad \text{for } k = 0,\ldots,M\text{-}1 \tag{24f}$$

$$\hat{H}_{xu_M} = [\bar{H}_{x3_{2M}} + \tilde{F}_{2M}^T \bar{H}_{xx_{2M+1}}L_{3_{2M}}] \tag{24g}$$

$$\hat{H}_{uu_k} = \begin{bmatrix} [\bar{H}_{33_{2k}} + L_{3_{2k}}^T \bar{H}_{xx_{2k+1}}L_{3_{2k}}] & [L_{3_{2k}}^T \bar{H}_{x3_{2k+1}}] \\ [L_{3_{2k}}^T \bar{H}_{x3_{2k+1}}]^T & \bar{H}_{33_{2k+1}} \end{bmatrix} \quad \text{for } k = 0,\ldots,M\text{-}1 \tag{24h}$$

$$\hat{H}_{uu_M} = [\bar{H}_{33_{2M}} + L_{3_{2M}}^T \bar{H}_{xx_{2M+1}}L_{3_{2M}}] \tag{24i}$$

$$\hat{g}_{x_k} = \bar{g}_{x_{2k}} + \tilde{F}_{2k}^T [\bar{g}_{x_{2k+1}} + \bar{H}_{xx_{2k+1}}\tilde{\underline{z}}_{2k}] \quad \text{for } k = 0,\ldots,M \tag{24j}$$

$$\hat{g}_{u_k} = \begin{bmatrix} \bar{g}_{3_{2k}} + L_{3_{2k}}^T[\bar{g}_{x_{2k+1}} + \bar{H}_{xx_{2k+1}}\tilde{\underline{z}}_{2k}] \\ \bar{g}_{3_{2k+1}} + \bar{H}_{x3_{2k+1}}^T\tilde{\underline{z}}_{2k} \end{bmatrix} \quad \text{for } k = 0,\ldots,M\text{-}1 \tag{24k}$$

$$\hat{g}_{u_M} = \bar{g}_{3_{2M}} + L_{3_{2M}}^T[\bar{g}_{x_{2M+1}} + \bar{H}_{xx_{2M+1}}\tilde{\underline{z}}_{2M}] \tag{24m}$$

This step is the only step in the entire stage-number halving cycle that involves message passing. The quantities that get computed in equations (24b)-(24m) must all reside on a single processor at the end of this step, one processor for each index k. At the beginning of this step, however, the information associated with index 2k resides on one processor, while the information associated with index 2k+1 resides on another processor. The binary tree structure of Fig. 1 ensures that all such pairs of processors will have a direct connection. Thus, the M+1 messages that must get passed can get passed in parallel. Repeated application of this 5-step stage-number halving procedure results in information fan-in on the binary tree -- information moves up the tree in Fig. 1.

Nesting and Back Substitution

The foregoing 5-step process can be repeated until the entire problem is solved: the halved problem is again halved in a nested cycle. Thus, the hatted vectors and matrices at the end of a cycle become the un-hatted vectors and matrices at the beginning of the next cycle -- note the exact replication of the problem (2a)-(2e) form in problem (23a)-(23e). Suppose that, initially, $N = 2^j-1$. That is, suppose the problem starts with 2^j stages. After j times through the halving cycle, the problem will have only one stage, stage 0. During subsequent application, the cycle will terminate at step 4. The dimension of $\delta\underline{u}_{3_0}$ will be zero, and the state, $\delta\underline{x}_0$, is known: so, $\delta\underline{u}_{4_0}$ is completely determined.

Back substitution is used to reconstruct the entire state and control time history after the problem halving sequence has yielded a problem with a single stage. The back substitution process is the reverse of the problem halving process, it is a stage-number doubling process. It involves repeated application of a two-step process.

Given a certain level of problem halving, and given the $\delta\underline{x}_k$ and $\delta\underline{u}_{3_k}$ vectors at each stage corresponding to that level, the first step is to determine the $\delta\underline{u}_k$ vector at each stage for that level. This can be done in parallel by successive application of Eqs. (20) [to determine $\delta\underline{u}_{4_k}$], (19a) [to determine $\delta\underline{u}_{2_k}$], (15) [to determine $\delta\underline{u}_{1_k}$], and (14a) [to determine $\delta\underline{u}_k$]. Knowledge of the $\delta\underline{u}_k$ vector at each stage for the current level of halving translates into knowledge of the

$\delta\underline{u}3_{2k}$ and $\delta\underline{u}3_{2k+1}$ vectors at each pair of stages for the next previous level, the level with twice as many stages [see Eq. (24a)].

The second step is to determine the $\delta\underline{x}_k$ vector at each stage for the next previous level of halving. The state time history at the current level already contains half of the desired vectors [Eq. (24a)]. The unknown state vectors at the alternate stages are determined in parallel from the state difference equations at alternate stages of the previous problem level [Eq. (21d) for k = 0, 2, 4, ..., N-1]. This can be done because each stage's $\delta\underline{u}3_k$ vector at this previous level has already been determined.

Message passing occurs during this back substitution procedure. The pattern of the information flow is a fan-out along a binary tree (traveling downward on Fig. 1). After each $\delta\underline{u}_k$ has been calculated at a given level -- each stage on a different processor, it is broken into its two components, $\delta\underline{u}3_{2k}$, $\delta\underline{u}3_{2k+1}$, and one of the components is sent to a neighboring processor. The other component remains on the current processor, which will handle the corresponding stage at the next level of doubling.

3.3. Hybrid Multi-Stage-per-Processor Parallel QP Algorithm. The algorithm in the foregoing section requires as many processors as the number of problem stages. A small additional procedure enables the algorithm to handle problems with more stages than the given number of processors. Two benefits come from the improvement. First, it allows solution of a greater range of problems on a given hypercube, a 32-node hypercube in this research. Second, for typical problems, significant problem speed-up can be achieved even with more than one problem stage per processor. For a given number of processors, the speed-up over the best known one-processor algorithm grows with the number of stages in the problem.

The working of the multi-stage-per-processor algorithm is slightly more complex than the single-stage-per-processor algorithm. When a single processor has more than one stage, the algorithm starts by working only with the last stage of this set of stages -- it must have a contiguous set of problem stages. It starts by performing steps 1-4 of section 3.2 on this last stage. Then it performs step 5 to join the last stage on the processor to the second-to-last stage. At

the end of this cycle, each processor has one less stage. Figure 2 shows how a 24-stage problem on 8 processors becomes a 16-stage problem on these same 8 processors. This goes on until each processor has only one stage. At this point, the algorithm can do stage halving in the manner presented in section 3.2. Effectively, a backwards sweep is performed on each processor before the stage halving process begins. Thus, this algorithm is termed a hybrid algorithm.

The back substitution process described in the foregoing section undergoes a similar modification. It starts with the algorithm's original stage-doubling back substitution process. This terminates when each processor has received a control vector and a state vector corresponding to the first of its set of stages. A forward sweep of the state equation and the "feedback" control equations is then performed. This sweep extends to all of the stages on the given processor.

3.4. Bench-Mark (Best Known) Serial QP Algorithm. In order to determine the benefits due to parallelism, a good serial algorithm is needed for comparison purposes. To make an honest comparison, the bench-mark algorithm has to solve the same problem as the parallel algorithm in the fastest possible serial way. The fastest known serial QP algorithms for Dynamic Quadratic programs are those based upon the backwards sweep -- their serial execution time is O(N), where N is the number of discrete-time problem stages. Unfortunately, the standard backwards-sweep algorithm, the discrete-time matrix Riccati equation, does not handle auxiliary equality constraints.

The principles in section 3.2 have been used to develop a backwards sweeping algorithm that overcomes these difficulties. The algorithm starts with the last problem stage. It applies steps 1, 2, and 4 of section 3.2 to this stage. Step 3 need not be performed because there is no state difference equation for propagation to a next stage. Step 5 is then performed to just the last two stages. The result is a problem with just one less stage. This process is repeated until the first problem stage is reached. The resulting one-stage problem is completely solved. The entire state and control time histories are then reconstructed using a forward sweep.

As a further complication, assume that derivatives are calculated in parallel as in Ref. 12. Then this bench-mark serial algorithm must operate with gradients, Hessians, and Jacobians that

are distributed on different processors. If this serial algorithm runs on a single processor, the communication time to transfer the distributed QP information onto the single processor would greatly slow the algorithm.

A better way to use the serial algorithm is to have the backwards sweep transferred from one processor to the next as it needs information for a different stage. For the case when there are more stages than the number of available processors, each processor deals with a set of contiguous stages whose gradients, Hessians, and Jacobian have been calculated on it. Figure 3 depicts this backwards sweep as it transfers between processors that have derivative information about different sets of problem stages. In this case, there are more problem stages than nodes. On a given node, the serial algorithm sweeps backwards through all of the stages (arrows between numbers). After sweeping through all of the stages on a given node, it transfers to the node with the next previous stage (arrows between nodes).

4. Computational Test and Results

4.1. Aeromaneuvering Test Problem. In order to test the algorithm, a specific nonlinear trajectory optimization problem and a linear-quadraticization of the problem about a guessed solution are needed. An aeromaneuvering problem from Miele and Lee (Ref. 15) has been used.

The maneuver involves transfer from a geosynchronous Earth orbit (GEO) of 0° inclination to a circular low Earth orbit of $+ 28^{\circ}$ inclination (LEO) (see Fig. 4). The maneuver that Miele and Lee call type 2 has been used: three propulsive impulses with plane changes and one atmospheric maneuvering arc. The first impulse (point 1 in Fig. 4) is nontangential; it changes the inclination, and it puts the spacecraft (S/C) into an elliptical transfer orbit that takes it toward the Earth. Next, the S/C enters the upper atmosphere[4] (point 2) and performs a deceleration and further plane change using aerodynamic drag and lift. No thrusting is used during this phase. Next, it exits the

[4] Atmospheric entry and exit are defined as occurring at an altitude of 120 km.

atmosphere (point 3), simultaneously performing its second nontangential propulsive burn. The maneuver ends when the S/C reaches the correct LEO altitude (point 4) and performs its final nontangential burn to circularize the orbit and achieve the correct final inclination. The LEO altitude is set at 300 km.

State propagation has been performed in two ways. Outside of the atmosphere, the state has been propagated using Kepler's laws coupled with the transformations between Kepler's elements and the state vector elements. Inside the atmosphere, Miele and Lee's simple models of the lift and drag have been used along with a table look-up of the atmospheric density (Ref. 16, p.820) to get the aerodynamic forces. The gravitational force is based on the spherical Earth model. The resulting system involves six coupled scalar ordinary differential equations -- three dynamics equations and three kinematics equations. Inside the atmosphere, 6th-order Runge-Kutta numerical integration of these ordinary differential equations from time t_k to time t_{k+1} effectively defines the function $\underline{f}(\underline{x}_k,\underline{u}_k,k)$ on the right hand side of the system's nonlinear difference equation [Eq. (1d), the discrete-time system dynamics]. The control is held fixed at $\underline{u}_k$ during this integration, which gives a zero-order hold. The state vector at all stages is:

$$\underline{x} = \begin{bmatrix} V \\ \gamma \\ \psi \\ r \\ \phi \\ \theta \end{bmatrix} \tag{25}$$

where V is inertial speed, γ is flight path angle, ψ is heading angle (0^o = east, $+90^o$ = north), r is radial distance to Earth's center, ϕ is latitude (positive north), and θ is longitude (positive east). This is the same state vector as in Miele and Lee (Ref. 15), where the equations of motion are presented.

The definitions and lengths of the control vector and constraint vector vary with different problem discrete-time steps. The cost function also varies with time step. This is necessary for

efficient modeling of the problem. Suppose the problem is modeled by 2^J (= N+1) steps, labeled 0, ..., N. Table 6 defines the steps, the controls, the constraints, and the cost. Note that the time duration of step zero has been automatically adjusted to ensure that the S/C altitude is 120 km at the end of the step. Similarly, V_c and γ_c have been eliminated from step N by requiring them to yield a circular orbit. The controls during the atmospheric portion are C_L, the lift coefficient, σ the bank angle, and τ, the actual time duration of the discrete-time step. The control Δv after the second impulsive burn is the change in the true anomaly until the third burn (the coast distance).

Table 6. Activities, controls, constraints, and cost modeling at different time steps of the aeromaneuvering example.

Step No.	Portion of Flight	Control vector	Constraints	Cost
0	First burn and coast to upper atmosphere	(V_c,γ_c,ψ_c)[5]	Perigee altitude of transfer orbit less than atmosphere height.	Magnitude of ΔV impulse[6]
1	Nonthrusting atmospheric flight with controlled lift and drag.	(C_L,σ,τ)	$\tau \geq 0.25$ sec; $-0.9 \leq C_L \leq 0.9$; Heating rate $\leq 1 \times 10^6$ watts/m^2.	0
:				
N-2	"	"	"	"
N-1	Second burn at atmosphere exit and coast to LEO altitude	$(V_c,\gamma_c,\psi_c,\Delta v)$	r = atmosphere edge $= r_E + 120$km; $\gamma \geq 0$; $\gamma_c \geq 0$.	Magnitude of ΔV impulse
N	Third burn to circularize and correct inclination at LEO altitude.	ψ_c	r = LEO radius $= r_E + 300$km; Inclination = 28°.	Magnitude of ΔV impulse

5 Defines the velocity vector after the impulsive burn.

6 This definition is used by Miele and Lee and reflects fuel use. Total fuel use is a monotonic function of the sum over all stages of the impulsive velocity changes.

A guessed optimal solution is needed in order to obtain a linear-quadratic problem for solution by the parallel QP solver. This guessed solution has been generated as follows. The guessed solution makes a first burn to achieve the correct inclination and to reduce the perigee altitude to 60 km. The guess then follows this transfer orbit to perigee neglecting atmospheric effects. It lumps all of the aerodynamic forces into an assumed velocity reduction at perigee that lowers the apogee altitude to 300 km. It then propagates this orbit to apogee where it circularizes it with a burn. It does no burn at atmospheric exit. All but stages 0, N-1, and N have been assumed to occur within the atmosphere during later linear-quadraticization.

The controls for the burn/coast arcs are well defined in the first-guess procedure. The controls for the atmospheric portions have been guessed to be $C_L = .05$, $\sigma = 45^\circ$, and τ = time to traverse a fixed change in true anomaly along the guessed trajectory.

The guessed active constraints include those inequalities that are violated by the guessed solution. All of the equality constraints -- both the auxiliary constraints listed in the above table and the state difference equation equality constraints -- have been considered active. The multipliers associated with each active constraint have been guessed to be a positive constant times the signed constraint violation; this is akin to the penalty function approach.

The cost gradients, Hessians of the Hamiltonians, and Jacobians of the active constraints are needed by the parallel QP solver. All of these have been generated numerically by finite differencing. In addition, a positive multiple of the identity matrix has been added to the Hessian before QP solution. This assures a positive definite problem for the QP procedure as would occur if this search step procedure were part of a nonlinear programming algorithm that used the L_2 trust-region method. The authors do not intend to use the L_2-trust region method when they eventually develop the full nonlinear optimization algorithm. Rather, the addition of a multiple of the identity matrix to the Hessian has been done because the Hessian modification capability, to which section 3.2 alludes, has not been implemented yet.

All of the calculations that produced the necessary QP have been done off-line on a serial processor, an IBM PC-XT. The results have been loaded onto the iPSC/2 nodes to test the parallel

QP algorithm. The timing results presented below do not reflect the times for off-line generation and node loading. In the future, these operations will also be done in parallel on the iPSC/2 nodes, but such complexity has not been necessary to the testing of the parallel QP algorithm.

4.2. Computational Timing Results. The parallel and serial QP algorithms have been used to solve 8, 16, 32, 64, 128, 256, 512 and 1024 stage cases of the aero-maneuvering test problem. With the parallel algorithm, different numbers of processors (8, 16 and 32) have been used to solve the same problem. The serial algorithm of section 3.4 has been swept over 8 processors for the 8 stage problem, 16 processors for the 16 stage problem, and 32 processors for all problems with 32 or more stages. Figure 5 gives timing results for both algorithms with the various combinations of processors and numbers of problem stages mentioned above.

For the parallel algorithm, although the difference is minute, the 16 stage problem is solved faster with 8 processors than with 16 processors; the 32 stage problem is solved faster with 16 processors than with 32 processors. This means that having 2 stages per node initially and one less overall stage-halving cycle is better than starting with a single stage per node. Although this improvement might not occur for different problems, a saving of half as many nodes as the number of stages could be a significant economy for problems with many stages.

The speed-up of the parallel algorithm with 32 processors over the serial algorithm is shown in Fig. 6. For the serial algorithm, a problem with twice as many stages takes twice the time (see Fig. 5). For the parallel algorithm, as the number of stages becomes large, the early work of backwards sweeping through multiple stages on a single processor dominates the computation time; the amount of work doubles as the number of stages doubles. Thus, the curve in Fig. 6 will eventually level out as the number of problem stages increases. The slope of the curve is already decreasing after 256 stages. Note that the speed-up reported on Fig. 6 would be slightly less if the serial algorithm could work with information that was already concentrated on a single processor.

The achieved algorithm speed-up is significant. Figure 6 shows an 8-fold speed increase for a 1024-stage problem when solved on 32 processors. In considering this speed-up, recall that the parallel algorithm is being compared to the best known serial algorithm. This is the most

conservative way of determining speed-up. A less conservative measure of speed-up due to parallelism would compare the parallel algorithm executed in parallel with itself executed serially. In this case, the speed-up factor would be 29, a 91% efficiency, on the 1024-stage problem.

Efficiency aside, an important characteristic of this algorithm is its ability to reduce wall-clock time for problems with a large number of stages. The lowest curve on Fig. 5 illustrates this ability to maintain low wall-clock time by increasing the number of processors as the number of problem stages grows.

The only other known parallel algorithm for similar problems is that developed by Wright (Ref. 13). Wright's algorithm scales with problem size the same way that the present algorithm scales with problem size. Wright's algorithm probably is faster than the present algorithm by a scale factor that is independent of problem size. This is because it is a simpler algorithm: it "divides" in the same way, but it "conquers" each sub-problem more efficiently. Wright's algorithm is probably about 4 times faster than the present algorithm on problems that can be solved by both algorithms. Unfortunately, Wright's algorithm cannot handle auxiliary state constraints; therefore, it could not be used to solve this example problem to provide a comparison between the two parallel algorithms.

5. Additional Discussion of Algorithm

5.1. Relationship of Stage-Halving to Control Concepts. The stage-halving algorithm of section 3.2 can be further illuminated by considering it in the control context. Suppose that the states at certain points of time before the end time are totally prescribed. In this case, the original optimal control problem divides neatly into independent sub-problems, which can be solved in parallel. After these are solved, the original cost of the total problem can be computed. This cost will depend on the values prescribed for the states at the interim points where the states are fixed. These states can then be varied to optimize the total cost. The resulting solution will solve the original problem. This idea, coupled with the idea of algorithm nesting, yields the approach of section 3.2. This concept is also found in Ref. 11.

5.2. Bottleneck Performance. In general, a parallel algorithm can be subject to bottlenecks. For instance, suppose one stage has more auxiliary constraints to factorize than does another stage. This will delay the other stage at some point by making it wait to combine (step 5) with a slower executing stage. One might conjecture that such effects get compounded to slow the overall stage-halving algorithm.

This issue has been investigated. The time to execute each step of the algorithm on each processor has been determined for example problems. These results show that delays occur, but they do not get compounded. The total execution time of the algorithm is no slower than the sum over all of the stage halvings of the execution time of the slowest stage at each level of stage halving.

5.3. Compatibility with Parallel Gradient, Jacobian, and Hessian Calculations. The algorithm of section 3 brings an additional benefit (beyond its real-time speed-up factor) to the solution of nonlinear trajectory optimization problems. It is well suited for use in conjunction with parallel gradient and Jacobian calculation. References 12 and 13 both point out the possible advantage of calculating the gradients of the cost function and the Jacobians of the constraints in parallel. This also turns out to be a stage-wise parallelization. After calculation, these matrices and vectors get used by the current algorithm in the same distributed manner as the manner in which they are generated -- no additional message passing is needed. This is the reason that the bench-mark serial algorithm has been constrained to use distributed derivative information.

5.4. Numerical Stability. One point of caution concerns the numerical stability of the algorithm. If the F_k matrices for k = 0, 1, 2, ..., N-1 are large compared to 1 (if the open-loop system is wildly unstable), then numerical problems can occur. The algorithm includes repeated multiplication by these matrices. Other trajectory optimization algorithms seem to suffer from this same drawback. The authors do not know of any examples where this becomes a problem.

Alternatively, if A_{1_k} in Eq. (15) is very large compared to R_{1_k}, then numerical instability can occur. This can happen when the overall nonlinear programming algorithm, of which this

algorithm is a part, allows constraint violations in the form of slack variables that later get penalized in the cost.

If either of these problems occur, then steps 1, 2, and 5 can be altered to yield a parallel version of the standard null space method. Such an algorithm would be guaranteed to have numerical stability. This alteration would result in a slower algorithm, but its execution time would still scale as $n^3\log_2 N$ on N processors.

6. Conclusions

An algorithm has been presented that can solve an equality-constrained time-varying discrete-time LQR problem. It uses a hypercube message-passing parallel processor to solve the problem in $O(n^3\log_2 N)$ operations on N processors, where N is the number of problem stages and n is the number of state vector elements. It can also solve problems with more stages than the number of processors -- it solves an N stage problem on p processors in $O(n^3([N/p]+\log_2 p))$ operations. It uses a stage-wise parallelization, divide and conquer approach. It is a specialized form of a technique known as domain decomposition. It can handle auxiliary equality constraints. Such auxiliary constraints would arise from state or control equality or inequality constraints if the present algorithm were used to calculate the search direction as part of an active-set nonlinear trajectory optimization algorithm.

In order to test the algorithm, an aeromaneuvering problem has been solved. The problem, being nonlinear, has been linearized about a guessed solution. A good serial algorithm that can handle auxiliary equality constraints also has been developed for fair evaluation of the parallel algorithm's speed. A significant speed-up of the parallel algorithm over the serial algorithm has been achieved -- a factor of 8 for a 1024-stage problem on 32 processors. With more processors available, a higher speed-up should be possible.

The algorithm has been developed as part of an approach to the problem of doing nonlinear trajectory optimization on line. It reduces the wall clock time that a trajectory optimization algorithm must spend to solve for a search direction.

References

1. POLAK, E., *An Historical Survey of Computation Methods in Optimal Control*, SIAM Review, Vol. 15, pp. 553-584, 1973.
2. BREAKWELL, J.V., *The Optimization of Trajectories*, SIAM Journal on Applied Mathematics, Vol. 7, pp. 215-247, 1959.
3. BRYSON, A.E. and HO, Y.C., *Applied Optimal Control*, Hemisphere Publishing, Washington, DC, 1975.
4. KELLEY, H.J., *Method of Gradients*, Optimization Techniques, Edited by G. Leitmann, Academic Press, New York, New York, pp. 206-254, 1962.
5. MIELE, A., PRITCHARD, R.E., and DAMOULAKIS, J.N., *Sequential Gradient-Restoration Algorithm for Optimal Control Problems*, Journal of Optimization Theory and Applications, Vol. 5, pp. 235-282, 1970.
6. YAKOWITZ, S.J., *The Stagewise Kuhn-Tucker Condition and Differential Dynamic Programming*, IEEE Transactions on Automatic Control, Vol. AC-31, pp. 25-30, 1986.
7. HARGRAVES, C.R., and PARIS, S.W., *Direct Trajectory Optimization Using Nonlinear Programming and Collocation*, Journal of Guidance, Control, and Dynamics, Vol. 10, pp. 338-342, 1987.
8. LARSON, R.E., and TSE, E., *Parallel Processing Algorithms for the Optimal Control of Nonlinear Dynamic Systems*, IEEE Transactions on Computers, Vol. C-22, pp. 777-786, 1973.
9. MENON, P.K.A., and LEHMAN, L.L., *A Parallel Quasilinearization Algorithm for Air Vehicle Trajectory Optimization*, Journal of Guidance, Control, and Dynamics, Vol. 9, pp. 119-121, 1986.
10. TRAVASSOS, R., and KAUFMAN, H., *Parallel Algorithms for Solving Nonlinear Two-Point Boundary-Value Problems Which Arise in Optimal Control*, Journal of Optimization Theory and Applications, Vol. 30, pp. 53-71, 1980.

11. CHANG, S.C., CHANG, T.S., and LUH, P.B., *A Hierarchical Decomposition for Large-scale Optimal Control Problems with Parallel Processing Structure*, Automatica, Vol. 25, pp. 77-86, 1989.

12. BETTS, J.T., and HUFFMAN, W.P., *Trajectory Optimization on a Parallel Processor*, Journal of Guidance, Control, and Dynamics, Vol. 14, pp. 431-439, 1991.

13. WRIGHT, S.J., *Solution of Discrete-Time Optimal Control Problems on Parallel Computers*, Report No. MCS-P89-0789, Argonne National Laboratory, Chicago, Illinois, 1989.

14. GILL, P.E., MURRAY, W., and WRIGHT, M.H., *Practical Optimization*, Academic Press, New York, New York, 1981.

15. MIELE, A., and LEE, W.Y., *Optimal Trajectories for Hypervelocity Flight*, Proceedings of the 1989 American Control Conference, Pittsburgh, Vol. 3, pp. 2017-2023, 1989.

16. WERTZ, J.R., Editor, *Spacecraft Attitude Determination and Control*, D. Reidel Publishing Company, Boston, Massachusetts, 1978.

List of Figures

Fig. 1. Mapping of trajectory optimization problem stages to processor nodes on a binary tree message passing machine.

Fig. 2. Stage locations during part of the multi-stage-per-processor parallel QP algorithm.

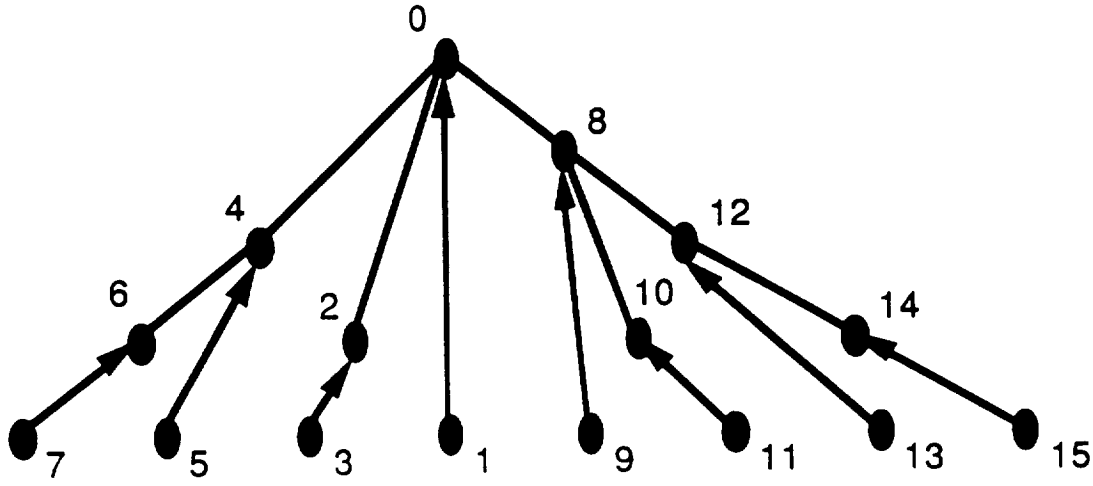
Fig. 3. Stage locations and message passing during a serial backwards sweep with distributed derivative information, a 24-stage problem on 8 processors.

Fig. 4. Orbit transfer of an aeromaneuvering spacecraft (not to scale).

Fig. 5. Time to solve the test problem on the INTEL iPSC/2 as a function of the number of problem stages.

Fig. 6. Speed-up of the (32-processor) parallel algorithm compared to the serial algorithm as a function of the number of problem stages.

Stage Locations at Nodes
Before Stage Number Halving



Combined-Stage Locations at Nodes
After Stage Number Halving

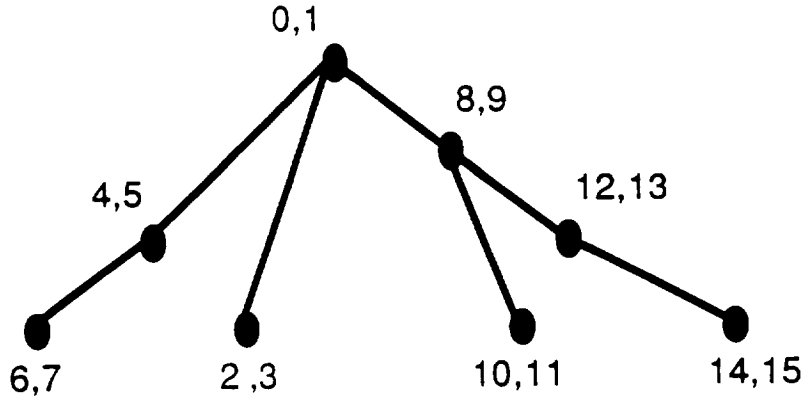
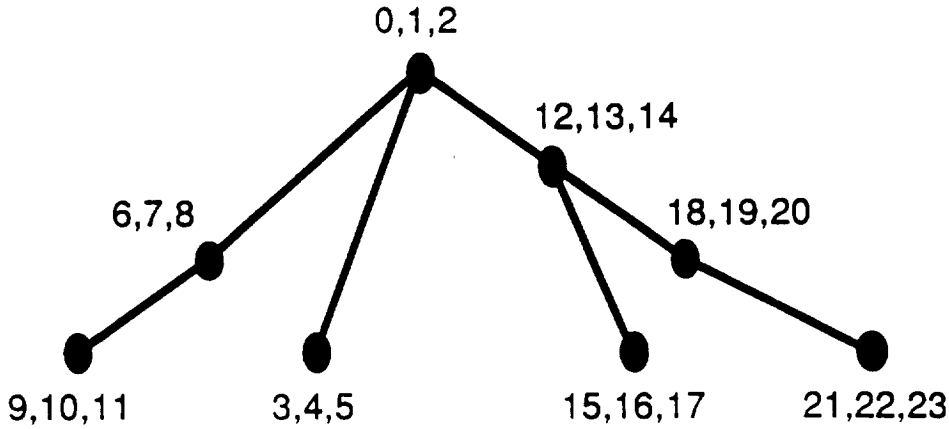


Fig. 1. Psiaki & Park

Stage Locations at Nodes Before
Elimination of Last Stage at Each Node



Stage Locations at Nodes After
Elimination of Last Stage at Each Node
(Parentheses indicate two stages that
have been joined into a single stage)

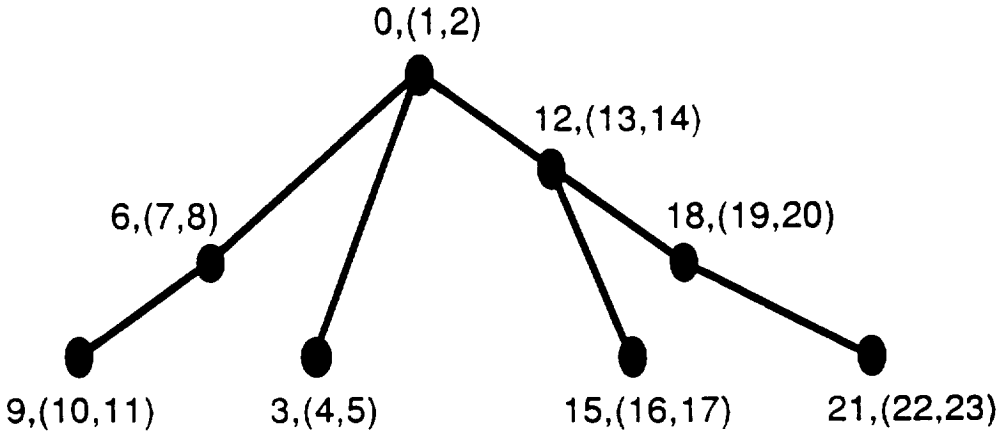


Fig. 2. Psiaki & Park

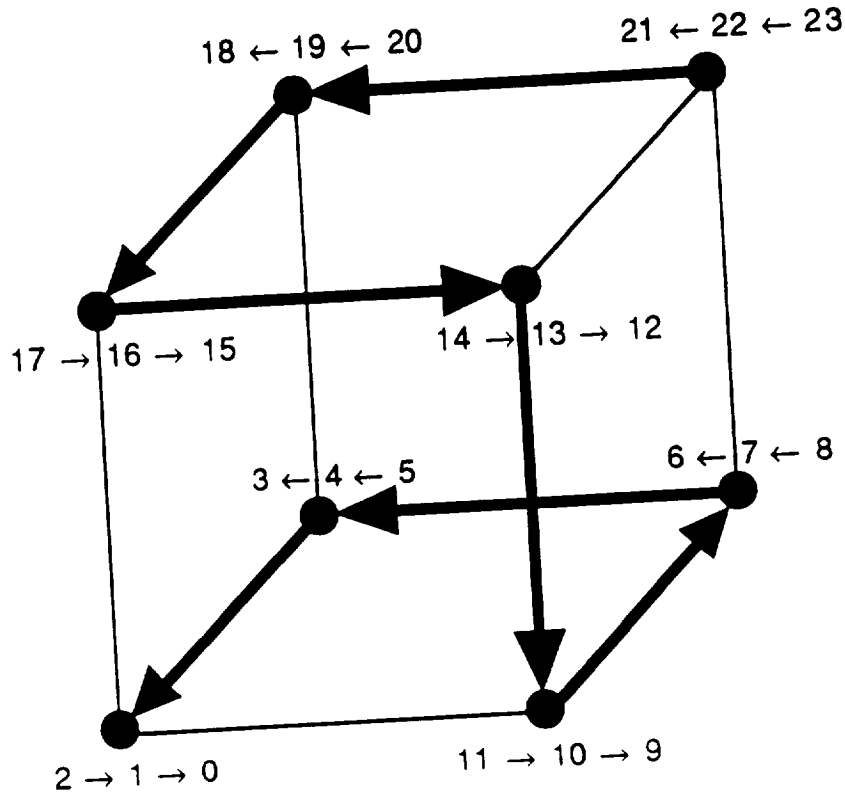


Fig. 3 Psiaki & Park

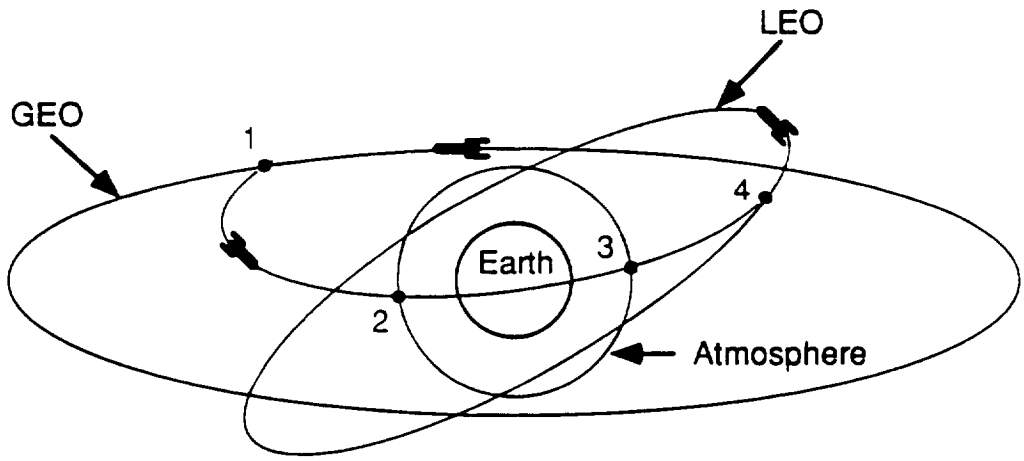


Fig. 4. Psiaki & Park

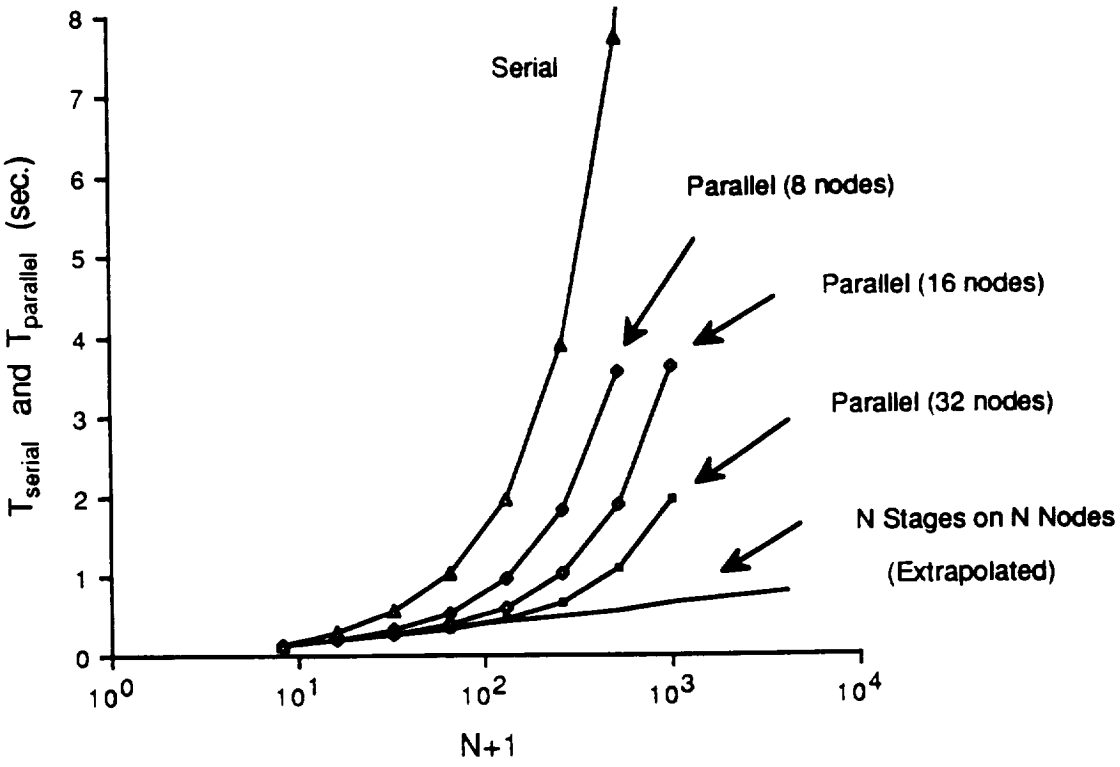


Fig. 5. Psiaki & Park

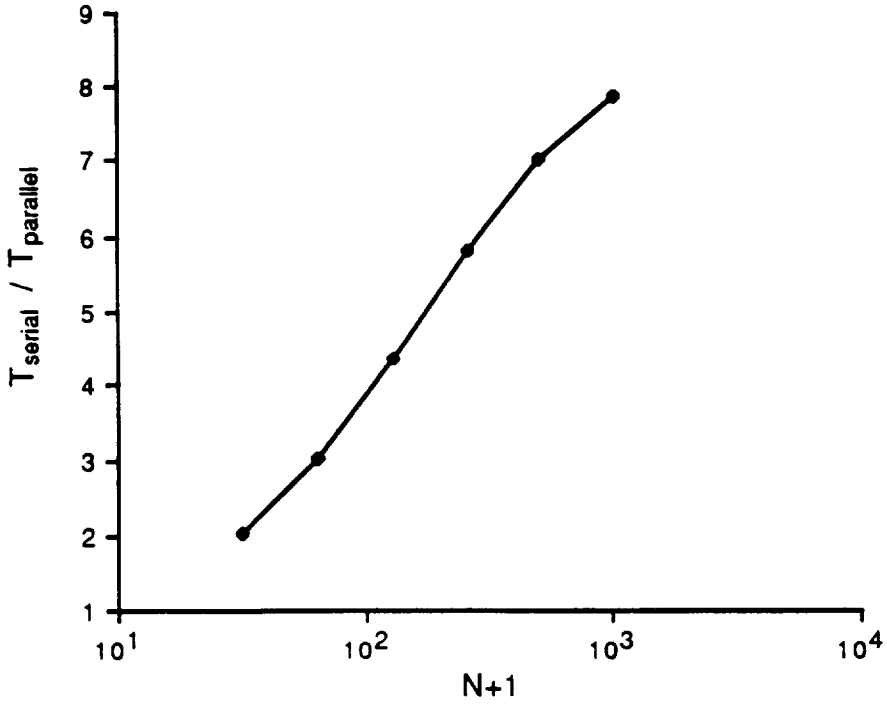


Fig. 6. Psiaki & Park.

A Parallel Orthogonal Factorization Null-Space Method for Dynamic Quadratic Programming[1]

M. L. PSIAKI [2] and K. PARK[3]

1 This research was supported in part by the National Aeronautics and Space Administration under Grant No. NAG-1-1009.

2 Assistant Professor, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

3 Graduate Research Assistant, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

Abstract. An algorithm has been developed to solve quadratic programs that have a dynamic programming structure. It is being developed for use as part of a parallel trajectory optimization algorithm. The dynamic quadratic programming algorithm has been developed to achieve significant speed without sacrificing numerical stability. The algorithm makes use of the dynamic programming problem structure and the domain decomposition approach. It parallelizes the orthogonal factorization null-space method of quadratic programming by developing a parallel orthogonal factorization and a parallel Cholesky factorization. Tests of the algorithm on a 32-node INTEL iPSC/2 hypercube demonstrate speed-up factors as large as 10 in comparison to the fastest known equivalent serial algorithm.

Key Words. Quadratic programming, dynamic programming, orthogonal factorization, parallel algorithms, domain decomposition.

1. Introduction

The objective of this paper is to develop and test a parallel algorithm for solving equality-constrained quadratic programs that have a dynamic programming problem structure. The algorithm must be rapid, it must have good numerical stability, and it must be able to detect an indefinite projected Hessian. Also, the algorithm must be able to determine a feasible direction of negative curvature when the projected Hessian is indefinite.

This algorithm has been designed to be part of a sequential quadratic programming-type nonlinear trajectory optimization algorithm (Refs. 1 and 2). The dynamic quadratic programming (DQP) algorithm must execute very rapidly because the parent nonlinear trajectory optimization algorithm may be used to provide real-time guidance commands to an aerospace vehicle.

The nonlinear programming (NP) core of the parent trajectory optimization algorithm requires the solution of an equality-constrained quadratic program (QP) each time it generates a search direction (Ref. 2). The characteristics of the core NP algorithm dictate the required good numerical stability of this paper's QP algorithm. A general variable reduction null-space method may lead to extreme ill-conditioning when used in conjunction with the core NP algorithm of Ref. 2, but an orthogonal factorization null-space method will have satisfactory numerical stability. The NP core also dictates the required ability to detect and determine feasible directions of negative curvature when they exist.

A general form of a dynamic quadratic program is

$$\text{find:} \qquad \boldsymbol{\chi} = \left[\boldsymbol{\chi}_0^T, \boldsymbol{\chi}_1^T, \boldsymbol{\chi}_2^T, \ldots, \boldsymbol{\chi}_N^T\right]^T \tag{1a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{N} \left\{ \tfrac{1}{2} \boldsymbol{\chi}_k^T \mathbf{H}_k \boldsymbol{\chi}_k + \mathbf{g}_k^T \boldsymbol{\chi}_k \right\} \tag{1b}$$

$$\text{subject to:} \quad \mathbf{E}_k \boldsymbol{\chi}_k + \mathbf{e}_k + \mathbf{F}_{k+1} \boldsymbol{\chi}_{k+1} + \mathbf{f}_{k+1} = \mathbf{0} \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{1c}$$

$$\mathbf{D}_k \boldsymbol{\chi}_k + \mathbf{d}_k = \mathbf{0} \qquad \text{for } k = 0 \ldots N \tag{1d}$$

The index k refers to the stage or time-step number. There are a total of N stages. The large solution vector, $\boldsymbol{\chi}$, is a composition of components from each stage, $\boldsymbol{\chi}_0$, $\boldsymbol{\chi}_1$, $\boldsymbol{\chi}_2$, ..., and $\boldsymbol{\chi}_N$.

The dimension of the solution vector $\mathbf{x}_k$ at a given stage is n_k. This dimension may vary from stage to stage. In an optimal control problem, the vector $\mathbf{x}_k$ will be a composite of the "state" and "control" vectors at time-step k.

The cost function J in Eq. (1b) is a summation of stage-wise costs, each of which includes both linear and quadratic terms. The cost is de-coupled: the cost at any given stage depends only the solution vector at that stage. The $n_k x n_k$ matrix $\mathbf{H}_k$ is the cost Hessian for stage k and the $n_k x 1$ vector $\mathbf{g}_k$ is the cost gradient for stage k when $\mathbf{x}_k = \mathbf{0}$.

Two types of linear constraints enter the problem, constraints that link neighboring stages, Eq. (1c), and constraints that involve only the solution vector of a given stage, Eq. (1d). The former constraints usually arise from dynamic models, and they provide the only coupling between the stages. The matrices $\mathbf{E}_k$ and $\mathbf{F}_{k+1}$ are the Jacobian matrices of the dynamic constraints. The $\mathbf{D}_k$ matrices are the Jacobians of the single-stage constraints. The fixed vectors $\mathbf{d}_k$, $\mathbf{e}_k$, and $\mathbf{f}_{k+1}$ are the nonhomogeneous constraint terms. The vectors $\mathbf{e}_k$ and $\mathbf{f}_{k+1}$ could be added to form a single nonhomogeneous term for each dynamic constraint, but the more general form in Eq. (1c) has been chosen to facilitate bookkeeping for the parallel domain decomposition algorithm that is developed in this paper. The number of constraint equations may vary from stage to stage, but the total number of constraints will normally be less than the dimension of $\mathbf{X}$ (the total number of unknowns).

The cost function in Eq. (1b) can be aggregated into two large terms, one quadratic and one linear:

$$J = \frac{1}{2}\mathbf{X}^T \begin{bmatrix} \mathbf{H}_0 & & & & \\ & \mathbf{H}_1 & & \mathbf{0} & \\ & & \mathbf{H}_2 & & \\ & & & \ddots & \\ & \mathbf{0} & & & \mathbf{H}_{N-1} \\ & & & & & \mathbf{H}_N \end{bmatrix} \mathbf{X} + \left[\mathbf{g}_0^T, \mathbf{g}_1^T, \mathbf{g}_2^T, \ldots, \mathbf{g}_{N-1}^T, \mathbf{g}_N^T\right] \mathbf{X} \qquad (2)$$

where cost de-coupling is reflected in the block-diagonal sparsity structure of the Hessian. Similarly, the constraints in Eqs. (1c) and (1d) can be formed into one large vector of equalities:

$$\begin{bmatrix} D_0 & & & \\ E_0 & F_1 & & 0 \\ & D_1 & & \\ & E_1 & F_2 & \\ & & \ddots & \\ & 0 & E_{N-1} & F_N \\ & & & D_N \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} d_0 \\ e_0+f_1 \\ d_1 \\ e_1+f_2 \\ \vdots \\ e_{N-1}+f_N \\ d_N \end{bmatrix} = 0 \qquad (3)$$

The sparsity structure of this matrix is called a staircase structure for obvious reasons. Much research has already been done on how best to exploit this structure, especially for the linear programming case (e.g., Refs. 3 and 4).

Wright (Ref. 5) and Psiaki and Park (Ref. 6) have both developed parallel algorithms for solving time-varying Linear Quadratic Regulator-type control problems. They are applicable to a special form of problem (1a)-(1d) where $\mathbf{F}_k = [\mathbf{I}, \mathbf{0}]$ for k = 1, ..., N and where the solution vector consists of a state and control vector, $\boldsymbol{x}_k = [\mathbf{x}_k^T, \mathbf{u}_k^T]^T$. When run on a distributed memory parallel processor, these algorithms offer significant savings in wall clock time in comparison to the fastest known serial algorithms. Wright's algorithms, although faster than the algorithm of Psiaki and Park, have the further restriction that they cannot handle single-stage state constraints: they are restricted to single-stage constraints of the form $\mathbf{D}_k = [\mathbf{0}, \mathbf{D}_{u_k}]$. Also, none of Wright's algorithms can detect an indefinite projected Hessian, and they may yield a cost ascent direction when the Hessian is indefinite.

Unfortunately, the previously-developed algorithms are unsuitable for use with the core NP algorithm in Ref. 2. Wright's algorithms' inability to deal with general $\mathbf{D}_k$ matrices and their inability to guarantee descent make them unsuitable. Psiaki and Park's algorithm does not have good enough numerical stability because parts of it use the variable-reduction null-space method. Therefore, a new parallel algorithm is needed.

The parallel algorithm developed in this paper is based on the domain decomposition approach developed by Psiaki and Park (Ref. 6). It is designed to run on a distributed-memory, message-passing parallel computer. The new algorithm is very similar in spirit to the algorithm of

Ref. 6. The main difference is in the exclusive use of the orthogonal factorization version of the null-space method. This provides the requisite numerical stability, but makes the algorithm use more memory. Additionally, this paper defines a method of determining directions of negative curvature when the projected Hessian is indefinite.

The remainder of this paper is divided into two sections plus conclusions. Section 2 develops the new DQP algorithm. It starts by developing an algorithm for the case where the number of processors equals the number of stages, N. Later, it generalizes to the case of fewer processors than problem stages. Section 3 presents a test problem and computational timing results using an INTEL iPSC/2 32-node hypercube. These results demonstrate the scaling of the wall clock execution time with the problem size.

2. Algorithm Design

2.1 Review of Orthogonal Factorization Null-Space Method. The algorithm to solve problem (1a)-(1d) is a special parallel form of the orthogonal factorization null-space method described in Ref. 7. When solving dense problems on a serial processor, this method begins by performing a right (or LQ) orthogonal factorization of the large constraint matrix in Eq. (3):

$$\begin{bmatrix} D_0 & & & \\ E_0 & F_1 & & 0 \\ & D_1 & & \\ & E_1 & F_2 & \\ & & \ddots & \\ & 0 & E_{N-1} & F_N \\ & & & D_N \end{bmatrix} Q = [L, 0] \qquad (4)$$

where Q is an orthogonal matrix and L is a square lower-triangular matrix. This is done using Householder transformations, and the form in Eq. (4) can always be achieved when the dimension of x exceeds the number of problem constraints. The Q matrix transforms x into two components:

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = Q^T x \qquad (5)$$

where $\mathbf{z}_2$ is in the null space of the constraints.

After transformation, the original problem takes the form

find: $\mathbf{z}_1$ and $\mathbf{z}_2$ (6a)

to minimize: $$J = \frac{1}{2}\begin{bmatrix}\mathbf{z}_1^T, \mathbf{z}_2^T\end{bmatrix}\begin{bmatrix}\mathcal{H}_{11} & \mathcal{H}_{12} \\ \mathcal{H}_{12}^T & \mathcal{H}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{z}_1 \\ \mathbf{z}_2\end{bmatrix} + \begin{bmatrix}\mathcal{g}_1^T, \mathcal{g}_2^T\end{bmatrix}\begin{bmatrix}\mathbf{z}_1 \\ \mathbf{z}_2\end{bmatrix} \quad (6b)$$

subject to: $$\mathcal{L}\mathbf{z}_1 + \begin{bmatrix} d_0 \\ e_0+f_1 \\ d_1 \\ \vdots \\ e_{N-1}+f_N \\ d_N \end{bmatrix} = 0 \quad (6c)$$

where the transformed Hessian and gradient are

$$\begin{bmatrix}\mathcal{H}_{11} & \mathcal{H}_{12} \\ \mathcal{H}_{12}^T & \mathcal{H}_{22}\end{bmatrix} = \mathbf{Q}^T\begin{bmatrix} H_0 & & & \\ & H_1 & & 0 \\ & & \ddots & \\ & 0 & & H_{N-1} \\ & & & & H_N \end{bmatrix}\mathbf{Q} \quad (7a)$$

$$\begin{bmatrix}\mathcal{g}_1 \\ \mathcal{g}_2\end{bmatrix} = \mathbf{Q}^T\begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{N-1} \\ g_N \end{bmatrix} \quad (7b)$$

If the constraints are non-degenerate, then the transformed problem in Eqs (6a)-(6c) is easy to solve. $\mathcal{L}$ is nonsingular in this case, and $\mathbf{z}_1$ is determined uniquely by the constraints:

$$\mathbf{z}_1 = -\mathcal{L}^{-1}\begin{bmatrix} d_0 \\ e_0+f_1 \\ d_1 \\ \vdots \\ e_{N-1}+f_N \\ d_N \end{bmatrix} \quad (8)$$

The null-space coordinate $\mathbf{z}_2$, which does not enter the constraints, can be optimized by setting $\partial J/\partial \mathbf{z}_2 = \mathbf{0}$. Solution of this expression for $\mathbf{z}_2$ yields

$$\mathbf{z}_2 = -\mathcal{H}_{22}^{-1}\left[\mathcal{H}_{12}^{T}\mathbf{z}_1 + \mathbf{g}_2\right] \tag{9}$$

where $\mathcal{H}_{22}$ must be positive definite for the original problem to have a unique local minimum. In this case, $\mathcal{H}_{22}^{-1}$ can be computed using the Cholesky factorization of $\mathcal{H}_{22}$. The final problem solution $\mathbf{x}$ can be computed from $\mathbf{z}_1$ and $\mathbf{z}_2$ by inverting the orthogonal transformation in Eq. (5).

The technique just described does not account for the sparsity or special structure of the problem matrices in Eqs. (2) and (3). A straight-forward application of this technique is likely to result in the generation of dense matrices $\mathbf{Q}$, $\mathbf{L}$, $\mathcal{H}_{11}$, $\mathcal{H}_{12}$, and $\mathcal{H}_{22}$. This would lead to $O(N^2)$ storage requirements, and the orthogonal and Cholesky factorizations would require $O(N^3)$ operations. As N, the number of problem stages, becomes large, a standard technique becomes inefficient or even infeasible.

2.2 Overview of New Algorithm. The technique of this paper uses the domain decomposition principle to develop a sparse, parallel orthogonal factorization technique for solving problem (1a)-(1d). It's memory requirements and operation count are both O(N) for an N-stage problem. When run on p (<N) processors with distributed memory and message passing capabilities, the wall clock time to execute the algorithm is $O[(N/p) + \log_2(p)]$.

The general idea of the algorithm is to perform the steps of the standard LQ null-space method in a piecemeal, stage-wise manner, where possible, to avoid creating fill in the resulting matrix factors. Different processors deal with different stages at the same time. First, it LQ factorizes all of the single-stage constraints, Eq (1d) for k = 0,...,N. This can be done in parallel and results in a transformation of the $\mathbf{x}_k$ at each stage. Part of the transformed $\mathbf{x}_k$ is completely determined by the constraints, and the single-stage constraints along with some of the unknowns are eliminated from the problem. At this point the only remaining constraints are those that link neighboring stages. These can be LQ factorized at each stage to further transform the remaining unknowns. Some of the transformed unknowns do not enter any constraints. All of the preceding

operations can be done without disturbing the block-diagonal structure of the large Hessian matrix in Eq. (2).

The next operation optimizes the unconstrained variables. The partial derivatives of the cost with respect to these variables are taken. These partial derivatives are set equal to zero, and the resulting equations are solved for the unconstrained variables in terms of the remaining constrained variables. This solution is substituted back into the cost expression to eliminate the unconstrained variables from the problem. This can be done at each stage in parallel. At the end of this operation the dynamic QP has been transformed into one that has the same number of stages as the original QP, but it has no single-stage constraints, and it has fewer unknowns at each stage.

One final operation is performed; then, the above steps are repeated. The final operation is to reduce the number of stages by a factor of 2 by combining adjacent stages into a single larger stage. This operation increases the number of variables per stage and makes some of the remaining constraints into single-stage constraints. Message passing must occur between pairs of processors at this point. The resulting problem is in the form of Eqs. (1a)-(1d), and the algorithm can be re-applied in a nested fashion. The resulting flow of data between processors in the nested application of the algorithm is a fan-in along a binary tree, with a processor at each node (Ref. 6).

As in Ref. 6, the problem eventually boils down to a single-stage problem, which gets completely solved. The solution of the single-stage problem then gets back-substituted into a double stage problem and so on in a stage-doubling operation. This back substitution process completes the determination of the variables that have been eliminated during parallel, stage-wise partial optimization. This process includes message passing in a fan-out along a binary tree of processors.

2.3 Algorithm for Solving N-Stage Problem on N Processors. The following algorithm solves problem (1a)-(1d), and it also determines the Lagrange multipliers for the constraints in Eqs. (1c) and (1d). The algorithm assumes that the constraints are not degenerate and that the projected Hessian is positive definite. Section 2.4 describes modifications that allow

determination of feasible directions of negative curvature when the projected Hessian is indefinite. The necessary modifications to handle degenerate constraints are discussed in Section 2.5.

Steps and Detailed Operations for One Stage Halving

Step 1. LQ factorize the $\mathbf{D}_k$ constraint matrices (in parallel):

$$\mathbf{D}_k \mathbf{Q}_{1_k} = [\mathbf{L}_{1_k} \; 0] \qquad \text{for } k = 0,\ldots,N \qquad (10)$$

where $\mathbf{Q}_{1_k}$ is orthogonal and $\mathbf{L}_{1_k}$ is square, lower triangular, and nonsingular.

This effectively transforms the $\mathbf{x}_k$ vectors, splitting them into components that are in the single-stage constraints, call these $\mathbf{x}_{1_k}$, and components that are not, call these $\mathbf{x}_{2_k}$:

$$\begin{bmatrix} \mathbf{x}_{1_k} \\ \mathbf{x}_{2_k} \end{bmatrix} \equiv \mathbf{Q}_{1_k}^T \mathbf{x}_k \qquad \text{for } k = 0,\ldots,N \qquad (11)$$

This factorization necessitates a transformation of other problem matrices and vectors consistent with the stage-wise variable transformations:

$$[\mathbf{F}_{1_k} \; \mathbf{F}_{2_k}] \equiv \mathbf{F}_k \mathbf{Q}_{1_k} \qquad \text{for } k = 1,\ldots,N \qquad (12a)$$

$$[\mathbf{E}_{1_k} \; \mathbf{E}_{2_k}] \equiv \mathbf{E}_k \mathbf{Q}_{1_k} \qquad \text{for } k = 0,\ldots,N\text{-}1 \qquad (12b)$$

$$\begin{bmatrix} \mathbf{H}_{11_k} & \mathbf{H}_{12_k} \\ \mathbf{H}_{12_k}^T & \mathbf{H}_{22_k} \end{bmatrix} \equiv \mathbf{Q}_{1_k}^T \mathbf{H}_k \mathbf{Q}_{1_k}, \quad \begin{bmatrix} \mathbf{g}_{1_k} \\ \mathbf{g}_{2_k} \end{bmatrix} \equiv \mathbf{Q}_{1_k}^T \mathbf{g}_k \qquad \text{for } k = 0,\ldots,N \qquad (12c)$$

The 1 and 2 subscripts on the F, E, and H matrices and on the g vectors refer to the index of the transformed $\mathbf{x}$ vector that gets multiplied by these matrices and vectors in the cost and in the constraint equations. In large sparse matrix form, the transformed constraint Jacobian matrix becomes

$$\begin{bmatrix} L_{1_0} & 0 & & & & & & & & \\ E_{1_0} & E_{2_0} & F_{1_1} & F_{2_1} & & & & & 0 & \\ & & L_{1_1} & 0 & & & & & & \\ & & E_{1_1} & E_{2_1} & F_{1_2} & F_{2_2} & & & & \\ & & & & & & \ddots & & & \\ & & & & & & & E_{1_{N-1}} & E_{2_{N-1}} & F_{1_N} & F_{2_N} \\ & & 0 & & & & & & & L_{1_N} & 0 \end{bmatrix} \tag{13}$$

which still has staircase structure. The transformed Hessian becomes

$$\begin{bmatrix} H_{11_0} & H_{12_0} & & & & \\ H_{12_0}^T & H_{22_0} & & & 0 & \\ & & H_{11_1} & H_{12_1} & & \\ & & H_{12_1}^T & H_{22_1} & & \\ & & & & \ddots & \\ & & & & H_{11_N} & H_{12_N} \\ & 0 & & & H_{12_N}^T & H_{22_N} \end{bmatrix} \tag{14}$$

which still has the block-diagonal structure.

Step 2. Use the single-stage constraints to solve for the constrained x_{1_k} and eliminate the single-stage constraints and the x_{1_k} from the problem (in parallel):

$$x_{1_k} = - L_{1_k}^{-1} d_k \qquad \text{for } k = 0,\ldots,N \tag{15}$$

After elimination of the x_{1_k} and the single-stage constraints, the problem takes the form

$$\text{find:} \qquad x_{2_0}, x_{2_1}, x_{2_2}, \ldots, x_{2_N} \tag{16a}$$

$$\text{to minimize:} \quad J = \sum_{k=0}^{N} \{ \tfrac{1}{2} x_{2_k}^T H_{22_k} x_{2_k} + \tilde{g}_{2_k}^T x_{2_k} \} \tag{16b}$$

$$\text{subject to:} \quad E_{2_k} x_{2_k} + \tilde{e}_k + F_{2_{k+1}} x_{2_{k+1}} + \tilde{f}_{k+1} = 0 \qquad \text{for } k = 0 \ldots N\text{-}1 \tag{16c}$$

which is similar in form to the problem in Eqs. (1a)-(1d) only without any single-stage constraints. The new problem is derived by substitution of Eqs. (10), (11), (12a)-(12c), and (15) into problem (1a)-(1d). The vectors with the (~) overstrike are

$$\tilde{g}_{2_k} = g_{2_k} - H_{12_k}^T L_{1_k}^{-1} d_k \qquad \text{for } k = 0,\ldots,N \tag{17a}$$

$$\tilde{e}_k = e_k - E_{1_k} L_{1_k}^{-1} d_k, \quad \tilde{f}_{k+1} = f_{k+1} - F_{1_{k+1}} L_{1_{k+1}}^{-1} d_{k+1} \quad \text{for } k = 0,\ldots,N-1 \tag{17b}$$

Step 2 eliminates all of the rows and columns that contain an L_{1_k} matrix from the constraints in Eq. (13). Also, it eliminates all of the rows and columns that contain an H_{11_k} matrix from the Hessian in Eq. (14). In large sparse matrix form, the new constraint Jacobian and Hessian for the problem in Eqs. (16a)-(16c) are, respectively,

$$\begin{bmatrix} E_{2_0} & F_{2_1} & & 0 \\ & E_{2_1} & F_{2_2} & \\ & & \ddots & \\ 0 & & E_{2_{N-1}} & F_{2_N} \end{bmatrix} \tag{18a}$$

$$\begin{bmatrix} H_{22_0} & & & 0 \\ & H_{22_1} & & \\ & & H_{22_2} & \\ & & & \ddots \\ 0 & & & H_{22_N} \end{bmatrix} \tag{18b}$$

Step 3. LQ factorize $\begin{bmatrix} F_{2_k} \\ E_{2_k} \end{bmatrix}$ to eliminate variables from the remaining constraints by forming zero columns in the transformed Jacobian matrix:

$$E_{2_0} Q_{2_0} = \begin{bmatrix} L_{4_0} & 0 \end{bmatrix} \tag{19a}$$

$$\begin{bmatrix} F_{2_k} \\ E_{2_k} \end{bmatrix} Q_{2_k} = \begin{bmatrix} L_{3_k} & 0 & 0 \\ E_{3_k} & L_{4_k} & 0 \end{bmatrix} \qquad \text{for } k = 1,\ldots,N-1 \tag{19b}$$

$$F_{2_N} Q_{2_N} = \begin{bmatrix} L_{3_N} & 0 \end{bmatrix} \tag{19c}$$

where the Q_{2_k} are orthogonal and the L_{3_k} and L_{4_k} are square and lower triangular (in parallel).

This step can be done only if the matrix to be factorized has more columns than rows. For general dynamic QPs this may not be true initially, but this factorization will become possible after several halvings of the number of stages.

Step 3 effectively transforms each x_{2_k} vector and splits it into three components. One component enters two constraints, the constraint that couples the current stage (k) to the preceding

stage (k-1) and the constraint that couples the current stage (k) to the following stage (k+1); call this component $\varkappa_{3_k}$. It is a "state"-type component in the terminology of control theory. Another component enters only the constraint that couples the current stage (k) to the following stage (k+1); call this component $\varkappa_{4_k}$. It is a "control"-type component. The third component, $\varkappa_{5_k}$, enters none of the constraints. One might call it an "ineffective control". The transformations are

$$\begin{bmatrix} \varkappa_{4_0} \\ \varkappa_{5_0} \end{bmatrix} \equiv Q_{2_0}^T \varkappa_{2_0} \tag{20a}$$

$$\begin{bmatrix} \varkappa_{3_k} \\ \varkappa_{4_k} \\ \varkappa_{5_k} \end{bmatrix} \equiv Q_{2_k}^T \varkappa_{2_k} \qquad \text{for } k = 1,\ldots,N\text{-}1 \tag{20b}$$

$$\begin{bmatrix} \varkappa_{3_N} \\ \varkappa_{5_N} \end{bmatrix} \equiv Q_{2_N}^T \varkappa_{2_N} \tag{20c}$$

Again, this step necessitates a transformation of other problem matrices and vectors to account for the transformed $\varkappa$ variables:

$$\begin{bmatrix} H_{44_0} & H_{45_0} \\ H_{45_0}^T & H_{55_0} \end{bmatrix} \equiv Q_{2_0}^T H_{22_0} Q_{2_0}, \quad \begin{bmatrix} g_{4_0} \\ g_{5_0} \end{bmatrix} \equiv Q_{2_0}^T \tilde{g}_{2_0} \tag{21a}$$

$$\begin{bmatrix} H_{33_k} & H_{34_k} & H_{35_k} \\ H_{34_k}^T & H_{44_k} & H_{45_k} \\ H_{35_k}^T & H_{45_k}^T & H_{55_k} \end{bmatrix} \equiv Q_{2_k}^T H_{22_k} Q_{2_k}, \quad \begin{bmatrix} g_{3_k} \\ g_{4_k} \\ g_{5_k} \end{bmatrix} \equiv Q_{2_k}^T \tilde{g}_{2_k} \quad \text{for } k = 1,\ldots,N\text{-}1 \tag{21b}$$

$$\begin{bmatrix} H_{33_N} & H_{35_N} \\ H_{35_N}^T & H_{55_N} \end{bmatrix} \equiv Q_{2_N}^T H_{22_N} Q_{2_N}, \quad \begin{bmatrix} g_{3_N} \\ g_{5_N} \end{bmatrix} \equiv Q_{2_N}^T \tilde{g}_{2_N} \tag{21c}$$

The 3, 4, and 5 subscripts on the **H** matrices and on the **g** vectors refer to the index of the transformed $\varkappa$ vector that gets multiplied by these matrices and vectors in the cost. After the step-3 orthogonal transformations, the large sparse form of the Jacobian becomes more sparse, but it retains its staircase form:

$$\begin{bmatrix} L_{4_0} & 0 & L_{3_1} & 0 & 0 & & & & & \\ & & E_{3_1} & L_{4_1} & 0 & L_{3_2} & 0 & 0 & & 0 \\ & & & & & E_{3_2} & L_{4_2} & 0 & & \\ & & & & & & & & \ddots & \\ & & & & & & & & L_{3_{N-1}} & 0 & 0 \\ & & & 0 & & & & & E_{3_{N-1}} & L_{4_{N-1}} & 0 & L_{3_N} & 0 \end{bmatrix} \tag{22}$$

and the transformed Hessian retains its block-diagonal form

$$\begin{bmatrix} H_{44_0} & H_{45_0} & & & & & & & & & & 0 \\ H_{45_0}^T & H_{55_0} & H_{33_1} & H_{34_1} & H_{35_1} & & & & & & & \\ & & H_{34_1}^T & H_{44_1} & H_{45_1} & & & & & & & \\ & & H_{35_1}^T & H_{45_1}^T & H_{55_1} & & & & & & & \\ & & & & & \ddots & & & & & & \\ & & & & & & H_{33_{N-1}} & H_{34_{N-1}} & H_{35_{N-1}} & & & \\ & & & & & & H_{34_{N-1}}^T & H_{44_{N-1}} & H_{45_{N-1}} & & & \\ & & & & & & H_{35_{N-1}}^T & H_{45_{N-1}}^T & H_{55_{N-1}} & H_{33_N} & H_{35_N} & \\ & 0 & & & & & & & & H_{35_N}^T & H_{55_N} & \end{bmatrix} \tag{23}$$

Step 4. Partially optimize by computing the optimal x_{5_k} as a function of x_{3_k} and x_{4_k} (in parallel for all stages where step 3 has been carried out). Solve the equation

$$H_{55_k} x_{5_k} = -[H_{35_k}^T x_{3_k} + H_{45_k}^T x_{4_k} + g_{5_k}] \tag{24}$$

then eliminate x_{5_k} from the cost function by substituting this solution for it.

This equation can be solved using a Cholesky factorization, $H_{55_k} = L_{5_k} L_{5_k}^T$, and back substitution; L_{5_k} is square and lower triangular. The H_{55_k} matrix constitutes a block on the diagonal of the projection of the Hessian of the original problem onto the null space of the constraints. It must be positive definite for the original QP to have a finite, unique solution. Given the assumption of a positive definite projected Hessian for the overall problem, the Cholesky factor L_{5_k} exists and is nonsingular, and Eq. (24) has a unique solution. (Section 2.4 explains useful

calculations that can be performed in the case of an indefinite or semi-definite H_{55_k}.) Upon substitution of the solution x_{5_k} into the cost, the problem becomes

$$\text{find:} \quad x_{4_0}, x_{3_1}, x_{4_1}, x_{3_2}, x_{4_2}, \ldots, x_{4_{N-1}}, \text{ and } x_{3_N} \tag{25a}$$

$$\text{to minimize:} \quad J = \tfrac{1}{2} x_{4_0}^T \bar{H}_{44_0} x_{4_0} + \bar{g}_{4_0}^T x_{4_0}$$

$$+ \sum_{k=1}^{N-1} \left\{ \tfrac{1}{2} \begin{bmatrix} x_{3_k}^T & x_{4_k}^T \end{bmatrix} \begin{bmatrix} \bar{H}_{33_k} & \bar{H}_{34_k} \\ \bar{H}_{34_k}^T & \bar{H}_{44_k} \end{bmatrix} \begin{bmatrix} x_{3_k} \\ x_{4_k} \end{bmatrix} + \begin{bmatrix} \bar{g}_{3_k}^T, \bar{g}_{4_k}^T \end{bmatrix} \begin{bmatrix} x_{3_k} \\ x_{4_k} \end{bmatrix} \right\}$$

$$+ \tfrac{1}{2} x_{3_N}^T \bar{H}_{33_N} x_{3_N} + \bar{g}_{3_N}^T x_{3_N} \tag{25b}$$

$$\text{subject to:} \quad L_{4_0} x_{4_0} + \tilde{e}_0 + L_{3_1} x_{3_1} + \tilde{f}_1 = 0 \tag{25c}$$

$$E_{3_k} x_{3_k} + L_{4_k} x_{4_k} + \tilde{e}_k + L_{3_{k+1}} x_{3_{k+1}} + \tilde{f}_{k+1} = 0 \qquad \text{for } k = 1 \ldots N\text{-}1 \tag{25d}$$

This problem is derived by first applying the transformations in Eqs. (19a) through (21c) to the problem in Eqs. (16a)-(16c) followed by substitution of the solution of Eq. (24) into the resulting QP. The resulting matrices and vectors with the (-) overstrike are

$$\bar{H}_{33_k} = H_{33_k} - H_{35_k} L_{5_k}^{-T} L_{5_k}^{-1} H_{35_k}^T \qquad \text{for } k = 1,\ldots,N \tag{26a}$$

$$\bar{H}_{34_k} = H_{34_k} - H_{35_k} L_{5_k}^{-T} L_{5_k}^{-1} H_{45_k}^T \qquad \text{for } k = 1,\ldots,N\text{-}1 \tag{26b}$$

$$\bar{H}_{44_k} = H_{44_k} - H_{45_k} L_{5_k}^{-T} L_{5_k}^{-1} H_{45_k}^T \qquad \text{for } k = 0,\ldots,N\text{-}1 \tag{26c}$$

$$\bar{g}_{3_k} = g_{3_k} - H_{35_k} L_{5_k}^{-T} L_{5_k}^{-1} g_{5_k} \qquad \text{for } k = 1,\ldots,N \tag{26d}$$

$$\bar{g}_{4_k} = g_{4_k} - H_{45_k} L_{5_k}^{-T} L_{5_k}^{-1} g_{5_k} \qquad \text{for } k = 0,\ldots,N\text{-}1 \tag{26e}$$

Step 4 eliminates from the constraint Jacobian in Eq. (22) all those columns that contain only zeros. Also, it eliminates all of the corresponding rows and columns from the Hessian matrix in Eq. (23), which are the rows and columns that contain an H_{55_k} matrix. Additionally, the remaining nonzero blocks of the large sparse Hessian get modified. The new large sparse forms of the constraint Jacobian and cost Hessian for the problem in Eqs. (25a)-(25d) are, respectively,

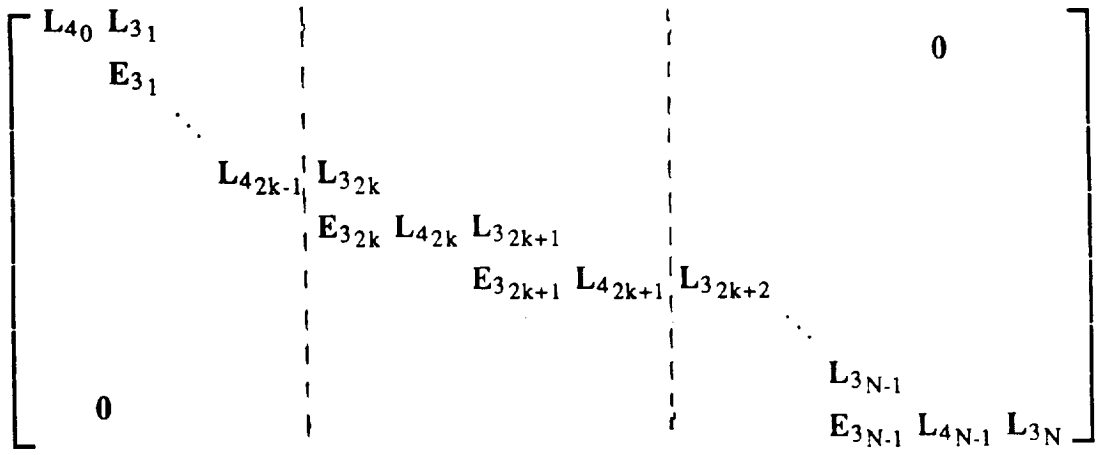
$$\begin{bmatrix} L_{4_0} & L_{3_1} & & & & & \\ E_{3_1} & L_{4_1} & L_{3_2} & & & 0 & \\ & E_{3_2} & L_{4_2} & \ddots & & & \\ & & & & L_{3_{N-1}} & & \\ & 0 & & & E_{3_{N-1}} & L_{4_{N-1}} & L_{3_N} \end{bmatrix} \qquad (27a)$$

$$\begin{bmatrix} \bar{H}_{44_0} & & & & & 0 & \\ & \bar{H}_{33_1} & \bar{H}_{34_1} & & & & \\ & \bar{H}_{34_1}^T & \bar{H}_{44_1} & & & & \\ & & & \ddots & & & \\ & & & & \bar{H}_{33_{N-1}} & \bar{H}_{34_{N-1}} & \\ & & & & \bar{H}_{34_{N-1}}^T & \bar{H}_{44_{N-1}} & \\ 0 & & & & & & \bar{H}_{33_N} \end{bmatrix} \qquad (27b)$$

The Jacobian is still a staircase matrix, and the Hessian is still a block diagonal matrix. Steps 1 through 4 do not involve any calculations which require data to be passed between stages, nor do they create any new coupling in the problem matrices.

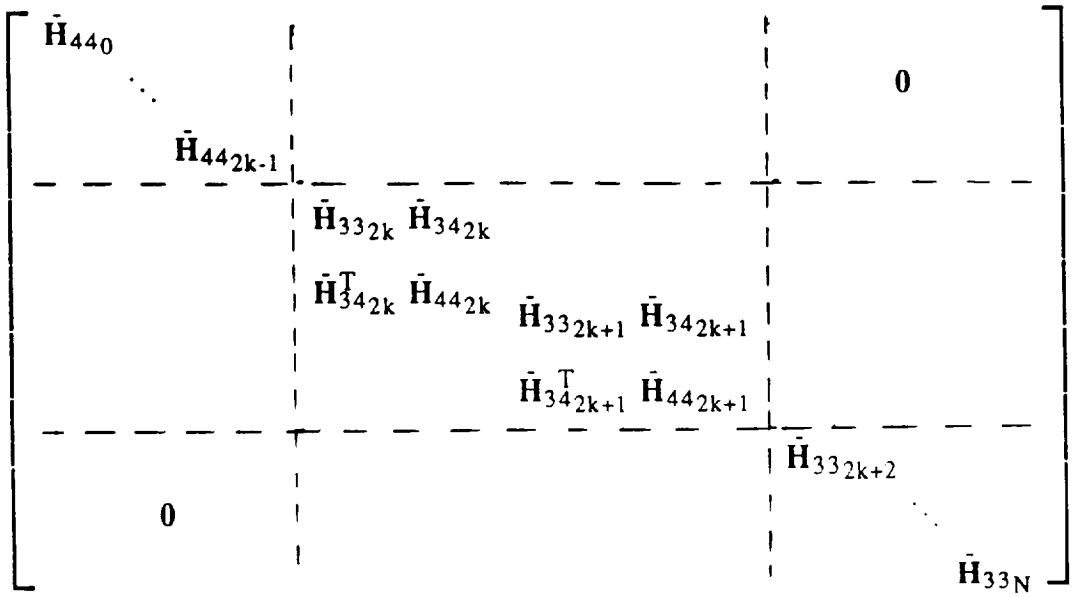
Step 5. Halve the number of stages by joining each even-numbered stage, stages 0,2,4,...,N-1, to the odd numbered stage that follows it. Assume that N is odd (N = 2M+1). This is done in parallel and includes parallel message passing between M+1 pairs of processors.

This aggregation of adjoining stages amounts to a re-definition of the boundaries between stages in the large sparse problem matrices. Stages 2k and 2k+1 get joined together into a "hyperstage". The dashed vertical lines in the following large sparse Jacobian show how step 5 super-imposes on the Jacobian of Eq. (27a) boundaries between a new "hyperstage" and its neighboring "hyperstages".



(28)

Similar boundaries between new "hyperstages" can be drawn to split up the Hessian matrix of Eq. (27b):



(29)

Step 5 is the concluding step in the process of halving the number of problem stages. After this step the problem is back in the form of Eqs. (1a)-(1d) only with half as many stages, M+1 stages instead of N+1:

find: $\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \ldots,$ and $\hat{\mathbf{x}}_M$ (30a)

to minimize: $J = \sum_{k=0}^{M} \left(\frac{1}{2} \hat{\mathbf{x}}_k^T \hat{\mathbf{H}}_k \hat{\mathbf{x}}_k + \hat{\mathbf{g}}_k^T \hat{\mathbf{x}}_k \right)$ (30b)

subject to: $\hat{E}_k \hat{x}_k + \hat{e}_k + \hat{F}_{k+1} \hat{x}_{k+1} + \hat{f}_{k+1} = 0$ for k = 0 ... M-1 (30c)

$\hat{D}_k \hat{x}_k + \hat{d}_k = 0$ for k = 0 ... M (30d)

The vectors and matrices with the (^) overstrike are derived by grouping of neighboring pairs of stages:

$$\hat{x}_0 \equiv \begin{bmatrix} x_{4_0} \\ x_{3_1} \\ x_{4_1} \end{bmatrix}, \quad \hat{x}_k \equiv \begin{bmatrix} x_{3_{2k}} \\ x_{4_{2k}} \\ x_{3_{2k+1}} \\ x_{4_{2k+1}} \end{bmatrix} \text{ for } k = 1,...,M-1, \quad \hat{x}_M \equiv \begin{bmatrix} x_{3_{2M}} \\ x_{4_{2M}} \\ x_{3_{2M+1}} \end{bmatrix} \tag{31a}$$

$$\hat{F}_k \equiv \left[L_{3_{2k}}, 0, 0, 0\right] \text{ for } k = 1,...,M-1, \quad \hat{F}_M \equiv \left[L_{3_{2M}}, 0, 0\right] \tag{31b}$$

$$\hat{D}_0 \equiv \left[L_{4_0}, L_{3_1}, 0\right], \quad \hat{D}_k \equiv \left[E_{3_{2k}}, L_{4_{2k}}, L_{3_{2k+1}}, 0\right] \text{ for } k = 1,...,M-1,$$

$$\hat{D}_M \equiv \left[E_{3_{2M}}, L_{4_{2M}}, L_{3_{2M+1}}\right] \tag{31c}$$

$$\hat{E}_0 \equiv \left[0, E_{3_1}, L_{4_1}\right], \quad \hat{E}_k \equiv \left[0, 0, E_{3_{2k+1}}, L_{4_{2k+1}}\right] \text{ for } k = 1,...,M-1 \tag{31d}$$

$$\hat{d}_k \equiv \tilde{e}_{2k} + \tilde{f}_{2k+1} \text{ for } k = 0,...,M, \qquad \hat{e}_k \equiv \tilde{e}_{2k+1} \text{ for } k = 0,...,M-1,$$

$$\hat{f}_k \equiv \tilde{f}_{2k} \text{ for } k = 1,...,M \tag{31e}$$

$$\hat{H}_0 \equiv \begin{bmatrix} \bar{H}_{44_0} & 0 & 0 \\ 0 & \bar{H}_{33_1} & \bar{H}_{34_1} \\ 0 & \bar{H}_{34_1}^T & \bar{H}_{44_1} \end{bmatrix} \tag{31f}$$

$$\hat{H}_k \equiv \begin{bmatrix} \bar{H}_{33_{2k}} & \bar{H}_{34_{2k}} & 0 & 0 \\ \bar{H}_{34_{2k}}^T & \bar{H}_{44_{2k}} & 0 & 0 \\ 0 & 0 & \bar{H}_{33_{2k+1}} & \bar{H}_{34_{2k+1}} \\ 0 & 0 & \bar{H}_{34_{2k+1}}^T & \bar{H}_{44_{2k+1}} \end{bmatrix} \text{ for } k = 1,...,M-1 \tag{31g}$$

$$\hat{H}_M \equiv \begin{bmatrix} \bar{H}_{33_{2M}} & \bar{H}_{34_{2M}} & 0 \\ \bar{H}_{34_{2M}}^T & \bar{H}_{44_{2M}} & 0 \\ 0 & 0 & \bar{H}_{33_{2M+1}} \end{bmatrix} \tag{31h}$$

$$\hat{g}_0 \equiv \begin{bmatrix} \bar{g}_{4_0} \\ \bar{g}_{3_1} \\ \bar{g}_{4_1} \end{bmatrix}, \quad \hat{g}_k \equiv \begin{bmatrix} \bar{g}_{3_{2k}} \\ \bar{g}_{4_{2k}} \\ \bar{g}_{3_{2k+1}} \\ \bar{g}_{4_{2k+1}} \end{bmatrix} \text{ for } k = 1,...,M-1, \quad \hat{g}_M \equiv \begin{bmatrix} \bar{g}_{3_{2M}} \\ \bar{g}_{4_{2M}} \\ \bar{g}_{3_{2M+1}} \end{bmatrix} \tag{31i}$$

As in the algorithm of Ref. 6, this step is the only step in the entire stage-number halving process that involves message passing. The quantities that get computed in Eqs. (31b)-(31i) must all reside on a single processor at the end of this step, one processor for each index k. At the beginning of this step, however, the information on the right-hand side of the equations associated with index 2k resides on one processor, while the information associated with index 2k+1 resides on another processor. A binary tree structure of inter-processor connections ensures that all such pairs of processors will have a direct connection, and the M+1 messages can get passed in parallel (Ref. 6). A binary tree can be imbedded in any hypercube processor.

Nesting. The foregoing 5-step process can be repeated until the entire problem is solved: the halved problem is again halved in a nested cycle. Thus, the hatted vectors and matrices at the end of a cycle become the un-hatted vectors and matrices at the beginning of the next cycle -- note the exact replication of the problem (1a)-(1d) form in problem (30a)-(30d). Suppose that, initially, $N = 2^j-1$. That is, suppose the problem starts with 2^j stages. After j times through the halving cycle, the problem will have only one stage, stage 0. Data will have fanned in to a single processor along a binary tree with j levels of branches (Ref. 6).

The final single-stage problem can be solved by a similar technique. Steps 1, 2, and 4 of the above cycle can be applied. There is no need for step 3 because there cannot be any multi-stage constraints. The vectors $\mathbf{x}_{2_0}$ and $\mathbf{x}_{5_0}$ are equivalent, and the vectors $\mathbf{x}_{3_0}$ and $\mathbf{x}_{4_0}$ have zero dimension. The vector $\mathbf{x}_{5_0}$ can be determined from Eq. (24), and the vector $\mathbf{x}_{1_0}$ can be determined from Eq. (15). Finally, the vector $\mathbf{x}_0$ can be determined from Eq. (11).

After the first stage-halving cycle, further computational efficiency can be realized if attention is paid to the structure of $\hat{\mathbf{F}}_k$ in Eq. (31b) and of $\hat{\mathbf{D}}_k$ in Eq. (31c). They both contain zeros, and these zeros can translate into savings during the LQ factorizations of steps 1 and 3 in the above cycle. The timing results reported in Section 3 are for an algorithm that includes this time-saving feature.

Back Substitution and Stage Doubling. Back substitution is used to reconstruct the entire solution history after the nested problem-halving sequence has yielded a problem with a single

stage and after that single-stage problem has been solved. The back substitution process is the reverse of the problem-halving process, it is a stage-number doubling process. Given a certain level of problem halving, and given the $\mathbf{x}_{3_k}$ and $\mathbf{x}_{4_k}$ time histories corresponding to that level, the first step is to determine the $\mathbf{x}_k$ time history for that level -- recall that $\mathbf{x}_k$ denotes the solution at stage k prior to the two orthogonal transformations and partitionings that occur in steps 1 and 3. The vector $\mathbf{x}_k$ can be determined in parallel by successive application of Eq. (24) [to determine $\mathbf{x}_{5_k}$], Eq. (20) [to determine $\mathbf{x}_{2_k}$], and Eq. (11) [to determine $\mathbf{x}_k$ -- $\mathbf{x}_{1_k}$ has already been determined during the stage-halving process via Eq. (15)]. Knowledge of the $\mathbf{x}_k$ time history for the current level of problem halving translates into knowledge of the $\mathbf{x}_{3_{2k}}$, $\mathbf{x}_{4_{2k}}$, $\mathbf{x}_{3_{2k+1}}$, and $\mathbf{x}_{4_{2k+1}}$ time histories for the next previous level -- the level with twice as many stages -- [see Eq. (31a)]. The stage-doubling algorithm can then repeat itself.

Message passing occurs during this back substitution procedure. The pattern of the information flow is a fan-out along a binary tree. After each $\mathbf{x}_k$ has been calculated at a given level -- each stage on a different processor, it is broken into it two pairs, $(\mathbf{x}_{3_{2k}},\mathbf{x}_{4_{2k}})$ and $(\mathbf{x}_{3_{2k+1}},\mathbf{x}_{4_{2k+1}})$. One of the pairs is sent to a neighboring processor. The other pair remains on the current processor, which will perform computations for the corresponding stage at the next level of doubling.

Multiplier Computation. One often needs to determine Lagrange multipliers for the original problem constraints in Eqs. (1c) and (1d). The nonlinear programming algorithm of Ref. 2 sometimes requires these multipliers. Alternatively, an active-set algorithm for solving inequality-constrained QPs needs to determine multipliers in order to decide which constraints to drop from the active set.

The multipliers can be calculated during the same stage-doubling process that calculates the $\mathbf{x}_k$ vectors. Suppose that the multiplier vectors associated with Eq. (1c) are $\underline{\alpha}_k$ for k = 0,...,N-1 and that the multipliers associated with Eq. (1d) are $\underline{\beta}_k$ for k = 0,...,N. Given all of the $\underline{\alpha}_k$ vectors at a given level of stage doubling, the following process determines the $\underline{\beta}_k$ vectors. First, adjoin the constraints to the cost in Eqs. (1a)-(1d) using the $\underline{\alpha}_k$ and $\underline{\beta}_k$ multipliers. Second, perform the

Q_{1_k} transformations defined in Eqs. (11)-(12c). Third, set the partial derivative with respect to $\mathbf{x}_{1_k}$ of the resulting expression equal to zero. Last, solve the resultant equation for $\underline{\beta}_k$ by back substitution with the $\mathbf{L}_{1_k}^T$ matrix.

Knowledge of the $\underline{\alpha}_k$ and $\underline{\beta}_k$ time histories at a given level of stage doubling translates into knowledge of the $\underline{\alpha}_k$ time history at the next higher level of stage doubling. Thus, the process can be repeated. The single-stage problem that terminates the stage-halving process has no constraints like Eq. (1d); therefore, it has no $\underline{\alpha}_k$ vector that needs to be known ahead of time. This fact allows the multiplier computation algorithm to initialize at the beginning of the stage-doubling process.

2.4 Procedures for an Indefinite or Positive Semi-Definite Projected Hessian. Section 2.3's equality-constrained QP solution procedure breaks down when the projected Hessian is indefinite or positive semi-definite. Step 4 fails for some stage number and level of problem halving. This situation is signaled by a break-down in the Cholesky factorization process. Either a negative square root or a divide-by-zero occurs, and the real Cholesky factor $\mathbf{L}_{5_k}$ cannot be computed. In the indefinite case, the equality-constrained QP cost function has an infinite minimum at infinity. In the positive semi-definite case, the QP has a non-unique minimum.

Two useful pieces of information can be derived in the indefinite and semi-definite cases if this algorithm is used as part of an SQP-type NP algorithm or as part of an active-set inequality-constrained QP algorithm. First, the parent algorithm usually will want to know whether the equality-constrained QP's projected Hessian is positive definite, positive semi-definite, or indefinite[4]. Second, the parent algorithm may want to know a feasible direction of negative (zero) curvature in the indefinite (positive semi-definite) case.

The method of determining these things is based on the modified Cholesky factorization process presented on pp. 108-110 of Ref. 7. When Cholesky factorizing a matrix that is not sufficiently positive definite, this procedure adds positive values to some of the matrix's diagonal

[4] Negative definite and negative semi-definite are considered to be synonymous with indefinite for purposes of this paper.

elements during factorization. This modification ensures that the matrix is sufficiently positive definite. For stage i, whose matrix $\mathbf{H}_{55_i}$ is not sufficiently positive definite, the modified process would compute Cholesky factors of $\mathbf{H}_{55_i} + \mathbf{E}_i$:

$$\mathbf{L}_{5_i}\mathbf{L}_{5_i}^T = \mathbf{H}_{55_i} + \mathbf{E}_i \tag{32}$$

where $\mathbf{E}_i$ is a positive semi-definite diagonal matrix that gets generated during the modified Cholesky factorization process. Nonzero diagonal elements of $\mathbf{E}_i$ get generated if a diagonal element of $\mathbf{L}_{5_i}$ would otherwise be imaginary, zero, or too small or if any off-diagonal elements of $\mathbf{L}_{5_i}$ would be too large. All of these situations correspond to an $\mathbf{H}_{55_i}$ matrix that is either indefinite, positive semi-definite, or positive definite but poorly conditioned. The latter two cases are effectively equivalent.

The stage-halving and stage-doubling processes of Section 2.3 can still be carried to completion by using the modified $\mathbf{L}_{5_i}$ Cholesky factor where necessary in Eqs. (26a)-(26e) and by replacing $\mathbf{H}_{55_i}$ in the corresponding Eq. (24) with $\mathbf{H}_{55_i}+\mathbf{E}_i$. If the constraints are homogeneous, then this technique is guaranteed to produce a feasible descent direction for the original problem.

If the $\mathbf{E}_i$ matrix has any nonzero elements, then some of these elements correspond to directions of zero or negative curvature. When the modified Cholesky factorization process produces a nonzero diagonal element of $\mathbf{E}_i$ to avert an imaginary diagonal element of $\mathbf{L}_{5_i}$, this corresponds to a direction of negative curvature. A nonzero diagonal element of $\mathbf{E}_i$ that has been produced to avert a zero (or small) diagonal element of $\mathbf{L}_{5_i}$ corresponds to a direction of zero (or low) curvature. If, on the other hand, a nonzero diagonal element of $\mathbf{E}_i$ gets produced to keep the off-diagonal terms of $\mathbf{L}_{5_i}$ from becoming too large, then another nonzero diagonal element of of $\mathbf{E}_i$ will get produced later in the process. This latter element will correspond to a direction of negative or zero curvature.

The modified Cholesky factor $\mathbf{L}_{5_i}$ can be used to compute an actual direction of negative, zero, or low curvature. Suppose that the jth diagonal element of $\mathbf{E}_i$ is nonzero and that this element was produced to avoid an imaginary diagonal element of $\mathbf{L}_{5_i}$. The calculation of a negative curvature direction begins with solution of the equation

$$\mathbf{L}_{5_i}^{T}\,\mathbf{p}_j = \mathbf{e}_j \tag{33}$$

for $\mathbf{p}_j$, where $\mathbf{e}_j$ is a unit vector with a 1 in the jth row and zeros elsewhere. By referring to Ref. 7, it is easy to prove that $\mathbf{p}_j^T\mathbf{H}_{55_i}\mathbf{p}_j/\mathbf{p}_j^T\mathbf{p}_j < 0$. The vector $\mathbf{p}_j$ is a direction of negative curvature for the $\mathbf{x}_{5_i}$ vector. Similarly, if the jth diagonal element of $\mathbf{E}_i$ had been produced to avoid a small or zero diagonal element of $\mathbf{L}_{5_i}$, then $\mathbf{p}_j^T\mathbf{H}_{55_i}\mathbf{p}_j/\mathbf{p}_j^T\mathbf{p}_j$ would be small or zero.

Additional calculations are needed in order to determine the corresponding direction of negative curvature for the original problem in Eqs. (1a)-(1d). These calculations are similar to the stage-doubling back substitution process defined in the Section 2.3. The process starts by setting $\mathbf{x}_{5_i} = \mathbf{p}_j$ and $\mathbf{x}_{5_k} = \mathbf{0}$ for all $k \neq i$ at the current level of stage halving. It also sets all of the $\mathbf{x}_{1_k}$, $\mathbf{x}_{3_k}$, and $\mathbf{x}_{4_k}$ vectors to zero at this level of stage halving. The feasible direction of negative curvature is calculated under the assumption that all of the constraints are homogeneous (i.e., $\mathbf{d}_k = \mathbf{e}_k = \mathbf{f}_k = \mathbf{0}$ for all k); it is a feasible direction in the null space of the constraints.

Next, the process determines the negative-curvature $\mathbf{x}_k$ time history for the current level of problem halving. This time history can be determined in parallel by application of Eq. (20) [to determine the $\mathbf{x}_{2_k}$] followed by application of Eq. (11) [to determine the $\mathbf{x}_k$]. Knowledge of the $\mathbf{x}_k$ time history for the current level translates into knowledge of the $\mathbf{x}_{3_{2k}}$, $\mathbf{x}_{4_{2k}}$, $\mathbf{x}_{3_{2k+1}}$, and $\mathbf{x}_{4_{2k+1}}$ time histories for the next previous level, and the stage-doubling back-substitution algorithm of Section 2.3 can be applied. During the ensuing stage-doubling cycles, the homogeneous constraints assumption is maintained, i.e. $\mathbf{x}_{1_k} = \mathbf{d}_k = \mathbf{0}$.

When the projected Hessian matrix of the original problem in Eqs. (1a)-(1d) is sufficiently positive definite, then all of the $\mathbf{E}_k$ will be identically zero for all of the levels of stage halving. If any of the $\mathbf{E}_k$ matrices have a nonzero diagonal element, then the projected Hessian is either indefinite, positive semi-definite, or positive definite but almost positive semi-definite. In the latter case, the projected Hessian is treated as being positive semi-definite to within the precision of the calculation.

The indefinite case can be distinguished from the positive semi-definite case. If not one of the nonzero elements of the $\mathbf{E}_k$ matrices was needed to avert an imaginary element of the

corresponding L_{5_k} matrix, then the projected Hessian of the entire original problem is positive semi-definite. Otherwise, it is indefinite.

In the indefinite case, several directions of negative curvature may be calculable by the Eq.-(33) technique. These may correspond to different stages and to different levels of problem halving. A parent algorithm that uses this QP algorithm may want to calculate all of these directions in order to do something like pick the one with the most negative curvature as a search direction. Note that none of these directions is guaranteed to be the direction of most negative curvature for the projected Hessian. This should not be a problem for most parent algorithms, so long as at least one direction of negative curvature can be calculated.

The use of a parallel algorithm can compromise numerical stability when calculating directions of negative curvature. The modified Cholesky algorithm of Ref. 7 enforces a maximum limit on off-diagonal elements of the Cholesky factors. This maximum limits the size of the $\mathbf{E}_k$ matrices and ensures numerical stability when the Hessian is indefinite.

The parallel algorithm uses the modified Cholesky procedure to ensure that the off-diagonal elements of all of the L_{5_k} matrices remain small, but elements of matrices such as $L_{5_k}^{-1}H_3S_k^T$ and $L_{5_k}^{-1}H_4S_k^T$, which are used in Eqs. (26a)-(26e), can grow without bound. Numerical experiments indicate that this growth causes no problem when the original problem's projected Hessian is positive definite, but it can lead to round-off error problems in the indefinite case. The ultimate result of this instability is that calculated directions of negative curvature may not have as much negative curvature as they would have had if growth in matrices such as $L_{5_k}^{-1}H_3S_k^T$ and $L_{5_k}^{-1}H_4S_k^T$ could have been limited in a sensible way. Unfortunately, there seems to be no obvious way to limit this growth without incurring adverse side effects in the positive-definite case.

2.5 Modifications to Deal with Degenerate Constraints. If the constraints in Eqs. (1c) and (1d) are degenerate, then the algorithm of Section 2.3 breaks down at step 2 for some level of stage halving. One of the L_{1_k} matrices will be singular.

When the constraints are degenerate, they may or may not admit a feasible solution. If they do not admit a feasible solution, then a sensible approach is to find the optimal $\mathbf{x}_k$ time history that

also minimizes the mean square error in Eqs. (1c) and (1d). Whether or not the residual mean square error is zero, the approach to solving this modified problem is the same. It is a variation to the method in Section 2.3. The algorithm must be modified at steps 1 and 2 in order to handle degenerate constraints.

In the modified step 1, the LQ factorization of the single-stage constraint Jacobian gets replaced by a complete LQ factorization; equation (10) gets replaced by

$$\mathbf{Q}_{3_k}\,\mathbf{D}_k\,\mathbf{Q}_{1_k} = \begin{bmatrix} \mathbf{L}_{1_k} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \qquad \text{for } k = 0,\ldots,N \qquad (34)$$

where $\mathbf{Q}_{1_k}$ and $\mathbf{Q}_{3_k}$ are orthogonal and $\mathbf{L}_{1_k}$ is square, lower triangular, and nonsingular. Any matrix $\mathbf{D}_k$ can always be factored in this manner. The new orthogonal matrix $\mathbf{Q}_{3_k}$ transforms the single-stage constraints, splitting them into two types:

$$\begin{bmatrix} \mathbf{L}_{1_k} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{\chi}_{1_k} \\ \boldsymbol{\chi}_{2_k} \end{bmatrix} = \begin{bmatrix} \mathbf{d}_{1_k} \\ \mathbf{d}_{2_k} \end{bmatrix} = \mathbf{Q}_{3_k}\,\mathbf{d}_k \qquad \text{for } k = 0,\ldots,N \qquad (35)$$

where $\boldsymbol{\chi}_{1_k}$ and $\boldsymbol{\chi}_{2_k}$ have been defined in Eq. (11). The first row of the transformed constraints in Eq. (35) is non-degenerate. The second row in Eq. (35) is a set of degenerate constraints. In the degenerate case, Eqs. (1c) and (1d) of the original problem can be exactly satisfied if and only if $\mathbf{d}_{2_k} = \mathbf{0}$ for all k and for all levels of stage halving.

In the modified step 2, the solution for $\boldsymbol{\chi}_{1_k}$ from the single-stage constraints gets replaced by a least-square solution of the single-stage constraints. In other words, Eq. (15) gets replaced by

$$\boldsymbol{\chi}_{1_k} = -\,\mathbf{L}_{1_k}^{-1}\mathbf{d}_{1_k} \qquad \text{for } k = 0,\ldots,N \qquad (36)$$

Except for multiplier determination, all of the other parts of the algorithm remain the same, including the stage-doubling back-substitution process. The multiplier vector is under-determined when the constraints are degenerate. A sensible thing to do is to determine the minimum-norm multiplier vector. Suppose, as in Section 2.3, that β_k for k = 0, ..., N are the multipliers associated with the constraints in Eq. (1d) at a given level of stage halving. These multipliers can be transformed and split to correspond to the transformed and split constraints in Eq. (35):

$$\begin{bmatrix} \beta_{1_k} \\ \beta_{2_k} \end{bmatrix} = Q_{3_k} \beta_k \qquad \text{for } k = 0,\ldots,N \qquad (37)$$

The minimum-norm multipliers can be determined by a procedure similar to the multiplier determination procedure of Section 2.3. The cost is augmented by adjoining the transformed constraints using the transformed multipliers, and the partial derivative of the augmented stage-wise cost with respect to the x_{1_k} vector is set equal to zero. This yields an equation for β_{1_k}, which can be solved by back substitution with the $L_{1_k}^T$ matrix. The β_{2_k} vector does not affect any such partial derivatives. It is set equal to **0**. The β_k vector is then determined by inverting Eq. (37).

2.6 Modifications to Handle Multiple Stages per Processor. A modified algorithm can solve problems that have more stages than the number of available processors. It is similar in spirit to the multi-stage-per-processor algorithm described in Ref. 6. When a single processor has more than one stage, the algorithm starts by working only with the last stage of this set of stages -- it must have a contiguous set of problem stages. It starts by performing steps 1-4 of Section 2.3 on this last stage. Next, it performs a modified version of step 5 to join the last stage on the processor to the second-to-last stage. Step 5 gets modified to account for the fact that the second-to-last stage may have some single-stage constraints. At the end of these steps, each processor has one less stage. This sequence of steps is repeated until each processor has only one stage.

The modified algorithm performs a sort of backwards sweep through the stages on a given processor. After the sweep reaches the first stage on the processor, the algorithm can continue in the stage-halving manner described in Section 2.3.

This backwards sweep is related to but different from the matrix Riccati backward sweep of LQR theory. First, the current backwards sweep operates on a more general problem form. Second, it uses LQ factorization rather than variable reduction to eliminate the constraints that join stages. Third, this sweep can handle arbitrary single-stage constraints such as "state" constraints.

Figure 1, borrowed from Ref. 6, shows how a 24-stage problem would be mapped onto an 8-processor binary tree (which can be formed on an 8-processor hypercube). The upper graph shows the original 24 stages (0 to 23). The lower graph shows how 16 stages remain after one stage-joining cycle has been executed on each node. Eight of the remaining stages are original problem stages, 0, 3, 6, 9, 12, 15, 18, and 21. Each of the other eight stages is the remains of two of the original problem's stages that have been joined, (1,2), (4,5), (7,8), (10,11), (13,14), (16,17), (19,20), and (22,23).

The back substitution process described in Section 2.3 undergoes a similar modification. It starts with the algorithm's original stage-doubling back substitution process. This terminates when each processor has received an $\mathbf{x}_k$ vector corresponding to the first of its set of stages. A forward sweep is then made through the remaining stages to determine the remaining $\mathbf{x}_k$ vectors.

2.7 Expected Scaling of Wall Clock Time. The expensive operations of Section 2.3's algorithm are the orthogonal matrix factorizations and matrix multiplications in steps 1 and 3 and the Cholesky factorizations and matrix multiplications of step 4. If n is the average number of elements of an $\mathbf{x}_k$ vector, then each of these operations requires $O(n^3)$ flops or less. The actual number of operations depends upon the number of constraints in Eqs. (1c) and (1d), and this number is limited if the constraints are not degenerate. At later levels of stage halving, the average number of elements of $\mathbf{x}_k$ may increase, possibly becoming as large as 4n. The expensive steps then require $O(64n^3)$ flops or less, which is still $O(n^3)$.

Step 2 and the back-substitution steps are relatively inexpensive. Both require $O(n^2)$ flops because they only involve matrix-vector multiplications and back substitution solutions of triangular linear systems. Step 5 is also inexpensive. It involves only message passing between processors and movement of data within the local memory of some processors.

Analysis of the stage-halving procedure of Section 2.3 indicates that its wall-clock time is $O[n^3\log_2(N)]$ for an N stage problem executing on N processors. The time is $O(n^3)$ per stage-halving cycle, and the entire process requires $\log_2(N)$ stage-halving cycles.

When the number of stages, N, is greater than the number of processors, p, then additional time is required for the initial backwards sweep that simultaneously occurs on each processor. If each processor has N/p stages, then this time is $O[n^3(N/p)]$. The entire solution procedure, which includes the backwards sweep followed by the stage-halving process, requires $O\{n^3[(N/p) + \log_2(p)]\}$ This scales just as the algorithm in Ref. 6, although the constant in front of the scaling law differs between the two algorithms.

The INTEL iPSC/2 is the machine on which this algorithm has been tested. It has a vector processor attached to each node. The vector processor time for doing such things as an inner product is approximately independent of n when n is small. The presence of a vector processor at each node affects the actual scaling of the wall clock time as a function of n when n is small: the algorithm's wall clock time is $O\{n^2[(N/p) + \log_2(p)]\}$ when solving an N-stage problem on p processors in this case.

3. Performance Evaluation on a Test Problem

3.1 Aero-maneuvering Test Problem. An aero-maneuvering problem described in Refs. 6 and 8 has been used to test the algorithm. It is a linear-quadraticization about a guessed solution of a nonlinear trajectory optimization problem. The problem has a 6-element state vector and a 3- or 4-element control vector, depending on the problem stage. The linear-quadratic problem form from Ref. 6 is

find: $\mathbf{u}_0, \mathbf{x}_1, \mathbf{u}_1, \mathbf{x}_2, \ldots, \mathbf{u}_{N-1}, \mathbf{x}_N$, and $\mathbf{u}_N$ (38a)

to minimize: $$J = \sum_{k=0}^{N} \{\frac{1}{2}\begin{bmatrix}\mathbf{x}_k^T, \mathbf{u}_k^T\end{bmatrix}\begin{bmatrix}\mathbf{H}_{xx_k} & \mathbf{H}_{xu_k} \\ \mathbf{H}_{xu_k}^T & \mathbf{H}_{uu_k}\end{bmatrix}\begin{bmatrix}\mathbf{x}_k \\ \mathbf{u}_k\end{bmatrix} + \begin{bmatrix}\mathbf{g}_{x_k}^T, \mathbf{g}_{u_k}^T\end{bmatrix}\begin{bmatrix}\mathbf{x}_k \\ \mathbf{u}_k\end{bmatrix}\} \quad (38b)$$

subject to: $\mathbf{x}_0$ given (38c)

$$\mathbf{x}_{k+1} = \mathbf{A}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{c}_k \qquad \text{for } k = 0 \ldots N-1 \quad (38d)$$

$$\mathbf{R}_k \mathbf{x}_k + \mathbf{S}_k \mathbf{u}_k + \mathbf{t}_k = 0 \qquad \text{for } k = 0 \ldots N \quad (38e)$$

where $\mathbf{x}_k$ is the state vector at stage k and $\mathbf{u}_k$ is the control vector.

Equation (38d) is a linearized dynamic system model, linearized about a guessed trajectory. It is the linearization of a model based upon Newtons laws for 3-degree-of-freedom translational motion of a point mass subject to gravity, and thrust or aero-maneuvering forces. A zero-order-hold assumption for the control inputs yields a discrete-time model. The presence of the nonhomogeneous c_k vector in the linearized dynamic model indicates that the guessed solution is infeasible; it does not obey the dynamic model. This situation is allowed at intermediate solution guesses by the NP algorithm of Ref. 2.

The auxiliary single-stage constraints in Eq. (38e) are present only for some stages. The matrices $\mathbf{R}_k$ and $\mathbf{S}_k$ form the Jacobian for each stage's constraints, and the vector t_k is the nonhomogeneous term. Constraints of this form are used at intermediate stages to define entry and exit of the atmosphere. For k = N, the Eq. (38e) constraints ensure that the desired terminal orbit is achieved. At stages where the peak allowable heating rate is exceeded by the guessed solution, additional Eq. (38e) constraints are included to enforce the maximum heating rate limit.

Slack variables have been added to the problem before testing the algorithm. Slack variables are not necessary. They have been added in order to put the QP into the form that would be used by the NP algorithm of Ref. 2 if that algorithm were being used to solve the nonlinear optimal trajectory. In that situation, the QP algorithm of this paper would be calculating an NP search direction. Slack variables can be added to each of the constraints to penalize violations. Then the QP becomes

find: $u_0, y_0, z_0, x_1, u_1, y_1, z_1, x_2, \ldots, u_{N-1}, y_{N-1}, z_{N-1}, x_N, u_N$, and z_N (39a)

to minimize:

$$J = \sum_{k=0}^{N} \left\{ \frac{1}{2} \begin{bmatrix} x_k^T, u_k^T \end{bmatrix} \begin{bmatrix} H_{xx_k} & H_{xu_k} \\ H_{xu_k}^T & H_{uu_k} \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} + \begin{bmatrix} g_{x_k}^T, g_{u_k}^T \end{bmatrix} \begin{bmatrix} x_k \\ u_k \end{bmatrix} \right\}$$

$$+ \sum_{k=0}^{N-1} \frac{h}{2} \left\{ y_k + \sqrt{\frac{\rho}{h}}\, c_k \right\}^T \left\{ y_k + \sqrt{\frac{\rho}{h}}\, c_k \right\}$$

$$+ \sum_{k=0}^{N} \frac{h}{2} \left\{ z_k + \sqrt{\frac{\rho}{h}}\, t_k \right\}^T \left\{ z_k + \sqrt{\frac{\rho}{h}}\, t_k \right\} \qquad (39b)$$

subject to: x_0 given (39c)

$$x_{k+1} = A_k x_k + B_k u_k - \sqrt{\frac{h}{\rho}}\, y_k \qquad \text{for } k = 0 \ldots N-1 \tag{39d}$$

$$R_k x_k + S_k u_k - \sqrt{\frac{h}{\rho}}\, z_k = 0 \qquad \text{for } k = 0 \ldots N \tag{39e}$$

The QP in Eqs. (39a)-(39e) can be restated in the form of Eqs. (1a)-(1d), given the following definitions of the solution vector at each stage:

$$\mathbf{x}_0 \equiv \begin{bmatrix} u_0 \\ y_0 \\ z_0 \end{bmatrix} \tag{40a}$$

$$\mathbf{x}_k \equiv \begin{bmatrix} x_k \\ u_k \\ y_k \\ z_k \end{bmatrix} \qquad \text{for } k = 1,\ldots,N-1 \tag{40b}$$

$$\mathbf{x}_N \equiv \begin{bmatrix} x_N \\ u_N \\ z_N \end{bmatrix} \tag{40c}$$

The corresponding constraint Jacobians, constraint nonhomogeneous vectors, cost Hessians, and cost gradients are easy to determine. For example,

$$\mathbf{D}_k \equiv \left[R_k,\ S_k,\ 0,\ -\sqrt{\frac{h}{\rho}}\ \mathbf{I} \right] \qquad \text{for } k = 1,\ldots,N-1 \tag{41a}$$

$$\mathbf{E}_k \equiv \left[A_k,\ B_k,\ -\sqrt{\frac{h}{\rho}}\ \mathbf{I},\ 0 \right] \qquad \text{for } k = 1,\ldots,N-1 \tag{41b}$$

$$\mathbf{F}_k \equiv [-\mathbf{I},\ 0,\ 0,\ 0] \qquad \text{for } k = 1,\ldots,N-1 \tag{41c}$$

$$\mathbf{H}_k \equiv \begin{bmatrix} H_{xx_k} & H_{xu_k} & 0 & 0 \\ H_{xu_k}^T & H_{uu_k} & 0 & 0 \\ 0 & 0 & hI & 0 \\ 0 & 0 & 0 & hI \end{bmatrix} \qquad \text{for } k = 1,\ldots,N-1 \tag{41d}$$

For the sake of brevity, the remaining definitions of the equivalent problem (1a)-(1d) matrices and vectors have been omitted.

3.2 Computational Timing Results. Different size versions of this problem have been solved ranging from 8 to 128 stages. The storage requirements of the 128 stage problem take up more than half of the available 4 Mbytes of memory per node on a 32-node INTEL iPSC/2.

A serial version of this algorithm has also been tested in order to determine the speed-up due to parallelism. The serial version solves the same problem, Eqs. (1a)-(1d), but it does it using the

best known serial technique: a single backwards sweep, which is similar to the serial algorithm of Ref. 6.

The timing results for these runs are reported on Fig. 2. The horizontal axis gives the number of problem stages on a log scale. The vertical axis gives the wall clock time on the INTEL iPSC/2. Each curve corresponds to a particular algorithm running on a particular number of nodes. As can be seen, all of the parallel algorithms are significantly faster than the serial algorithm. Figure 3 displays the speed-up due to parallelism by plotting the ratio of the serial algorithm time to the parallel algorithm time for the different cases.

These graphs clearly display the benefits of parallelism. For the 128-stage problem, the new parallel algorithm running on 32 nodes is more than 10 times faster than the serial algorithm (the serial algorithm takes 12.4 sec in this case, which puts it well off of the scale of Fig. 2). This speed-up increases with increased problem size and vice versa. The efficiency of the parallel algorithm is not great, a speed-up of 10 for 32 processors translates into a 32 % efficiency.

Note that this comparison is between the parallel algorithm and the best known serial algorithm of equivalent numerical stability. If the parallel algorithm were run on a single processor, it would be slower than the serial algorithm used in this study. Some authors in the parallel computation field use the latter type of "serial" algorithm as the benchmark to determine their speed-up due to parallelism. In that case, the parallel algorithm's speed-up would be about a factor of 16 for a 50% efficiency. This is a measure of the average busy time of each node during the execution of the parallel algorithm. This efficiency rises with (N/p), the number of problem stages per node.

Though neither efficiency is impressive, efficiency is not the ultimate goal of this work. The ultimate goal is a reduced wall clock time so that real-time applications will be feasible. The extrapolated "N-stages-on-N-nodes" line on Fig. 2 shows that a problem with many stages can be solved in a reasonably small amount of wall clock time if enough processors are available.

For comparison purposes, results from the earlier study of Psiaki and Park are included on Fig. 2 (Ref. 6). These are the two lowest curves on the graph; they do not apply to the algorithm

presented in this paper. These curves are the timing curves for the algorithm of Ref. 6 applied to aero-maneuvering problems in the form of Eqs. (38a)-(38e). This is lower-dimensional problem because it does not include the slack variables y_k and z_k. The new parallel algorithm takes between 2.5 and 3 times longer to solve the Eqs. (39a)-(39e) version of a given problem as compared to the parallel algorithm of Ref. 6 operating on the Eqs. (38a)-(38e) version of the same problem.

In order to make a sensible comparison between the two algorithms, the scaling results of section 2.7 must be used. The average number of unknowns at a given stage is 9 in the smaller problem, Eqs. (38a)-(38e), and it is 15 in the larger problem, Eqs. (39a)-(39e). One would expect the algorithm of this paper to execute about $(15/9)^2 = 2.8$ times faster when solving the smaller problem because the wall clock time scales as n^2. Thus, scaling analysis predicts that the new algorithm's speed on the smaller problem would be about equal to the speed achieved on that problem by the algorithm of Ref. 6. No comparison has been made with any of Wright's algorithms because they cannot solve the state-constrained aero-maneuvering problem.

4. Conclusions

This paper has presented an algorithm for the solution of dynamic quadratic programming problems on a parallel computer. The algorithm is an adaptation of the orthogonal factorization/null-space method to the parallel/dynamic programming framework. It takes advantage of the sparse "staircase" structure, and it uses domain decomposition techniques to achieve efficiency and parallelism. The use of a structured orthogonal factorization to determine the null space of the constraints ensures numerical stability of the null space determination. A structured block Cholesky factorization of the projected Hessian ensures numerical stability of the null-space optimization procedure when the projected Hessian is positive definite.

Solution time scales as $n^3[(N/p) + \log_2(p)]$ for an N-stage problem on p processors with an average of n unknowns per stage. On a 32-node INTEL iPSC/2, the algorithm achieves solution times as low as 1.2 sec for a 128 stage problem with 6 state vector elements, 3 control vector

elements, and 6 slack variables. It is faster than the best known equivalent serial algorithm by a factor of 10 or more when solving large problems.

References

1. PSIAKI, M.L. and PARK, K., *A Parallel Trajectory Optimization Tool for Aerospace Plane Guidance*, AIAA Paper No. 91-5069, presented at the AIAA Third International Aerospace Planes Conf., Orlando, FL, 1991.
2. PSIAKI, M.L. and PARK, K., *An Augmented Lagrangian Nonlinear Programming Algorithm that uses SQP and Trust Region Techniques*, submitted to the Journal of Optimization Theory and Applications, in review.
3. FOURER, R., *Solving Staircase Linear Programs by the Simplex Method*, Proceedings of the IIASA Workshop on Large-Scale Linear Programming, Laxenburg, Austria, pp. 179-259, 1981.
4. PSIAKI, M.L., *An Algorithm for the Solution of Dynamic Linear Programs*, Proceedings of the 3rd Annual Conference on Aerospace Computational Control, Oxnard, California, pp. 327-341, 1989.
5. WRIGHT, S.J., *Partitioned Dynamic Programming for Optimal Control*, SIAM Journal on Optimization, Vol. 1, pp. 620-642, 1991.
6. PSIAKI, M.L. and PARK, K., *A Parallel Solver for Trajectory Optimization Search Directions*, Journal of Optimization Theory and Applications, Vol. 73, pp. 533-560, 1992.
7. GILL, P.E., MURRAY, W., and WRIGHT, M.H., *Practical Optimization*, Academic Press, New York, New York, 1981.
8. MIELE, A., and LEE, W.Y., *Optimal Trajectories for Hypervelocity Flight*, Proceedings of the 1989 American Control Conference, Pittsburgh, Pennsylvania, Vol. 3, pp. 2017-2023, 1989.

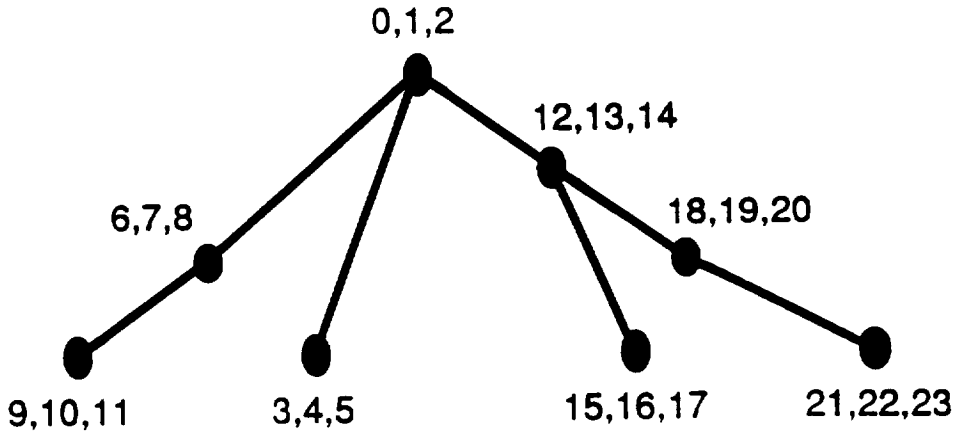
List of Figures

Fig. 1. Stage locations for a 24-stage problem on an 8-processor binary tree before (top) and after (bottom) one step of the backwards sweep (Ref. 6).

Fig. 2. Wall-clock solution time on the INTEL iPSC/2 as a function of the number of problem stages.

Fig. 3. Speed-up of parallel algorithm solutions compared to serial solutions as a function of the number of problem stages.

Stage Locations at Nodes Before
Elimination of Last Stage at Each Node



Stage Locations at Nodes After
Elimination of Last Stage at Each Node
(Parentheses indicate two stages that
have been joined into a single stage)

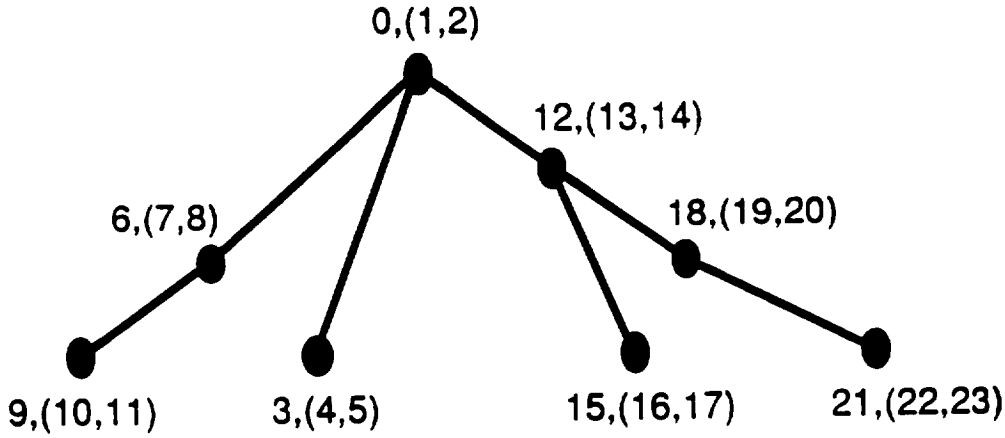


Fig. 1 Psiaki & Park

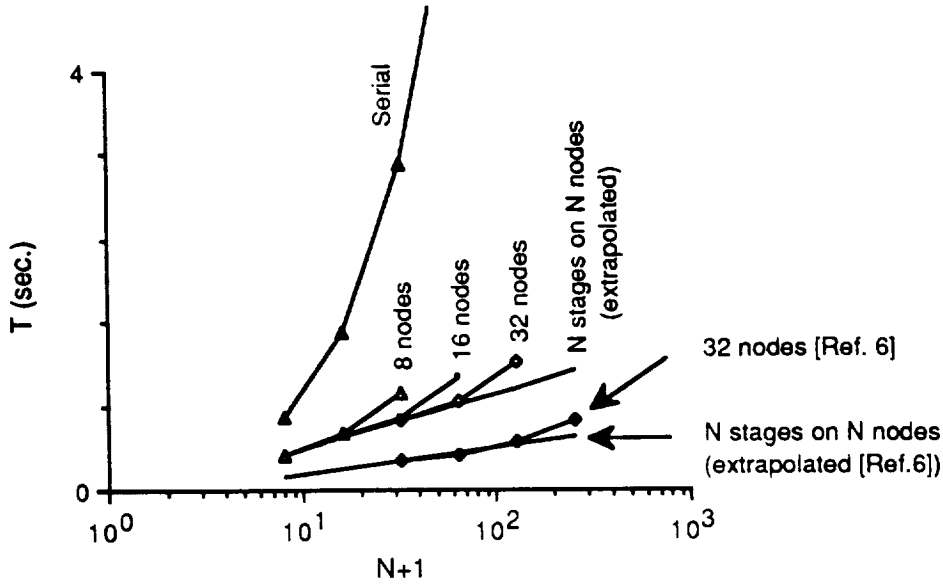


Fig. 2 Psiaki & Park

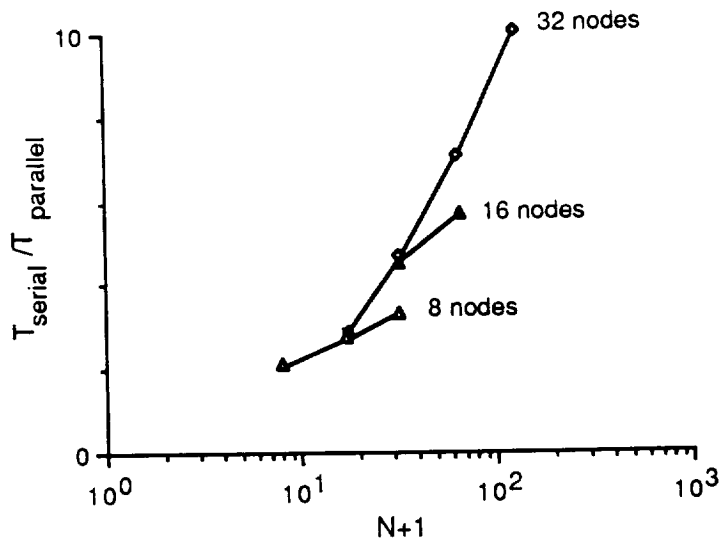


Fig. 3 Psiaki & Park

Appendix B

A paper describing a static
nonlinear programming algorithm.

An Augmented Lagrangian Nonlinear Programming Algorithm that uses SQP and Trust Region Techniques[1]

M. L. PSIAKI [2] and K. PARK[3]

---

1 This research was supported in part by the National Aeronautics and Space Administration under Grant No. NAG-1-1009.

2 Assistant Professor, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

3 Graduate Research Assistant, Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, 14853-7501.

Abstract. An augmented Lagrangian nonlinear programming algorithm has been developed. Its goals are to achieve robust global convergence and fast local convergence. Several unique strategies help the algorithm achieve these dual goals. The algorithm consists of three nested loops. The outer-loop estimates the Kuhn-Tucker multipliers at a rapid linear rate of convergence. The middle-loop minimizes the augmented Lagrangian function for fixed multipliers. This loop uses the sequential quadratic programming technique with a box trust region step-size restriction. The inner-loop solves a single quadratic program. Slack variables and a constrained form of the fixed-multiplier middle-loop problem work together with curved "line" searches in the inner-loop problem to allow large penalty weights for rapid outer-loop convergence. The inner-loop quadratic programs include quadratic constraint terms, which complicate the inner-loop, but speed the middle-loop's progress when the constraint curvature is large.

The new algorithm compares favorably with a commercial sequential quadratic programming algorithm on five low-order test problems. Its convergence is more robust, and its speed is not much slower.

Key Words. Nonlinear programming, sequential quadratic programming, augmented Lagrangian, trust region, constraint curvature.

1. Introduction

An algorithm that solves a typical nonlinear programming problem is presented in this paper. It is a variant of the augmented Lagrangian algorithm proposed by Hestenes (Ref. 1) and by Powell (Ref. 2) and described by Fletcher (Ref. 3). This paper adds several special features to the basic augmented Lagrangian algorithm to make it faster and more likely to converge. One such feature is the solution of a constrained problem that includes slack variables for each estimate of the multipliers. Another feature is the use of a sub-problem that has quadratic terms in the constraints in addition to the usual quadratic cost terms. The algorithm solves a sequence of such sub-problems, each subjected to box trust region constraints.

This algorithm has been developed to serve as the core of a new nonlinear trajectory optimization algorithm (Ref. 4). The algorithm might be used to do real-time guidance of aerospace vehicles. This creates the need for a high degree of convergence reliability and speed.

The algorithm of this paper has been designed to solve nonlinear programs of the form

find:	$\mathbf{x}$		(1a)
to minimize:	$J(\mathbf{x})$		(1b)
subject to:	$c_i(\mathbf{x}) = 0$	for $i = 1, ..., m_e$	(1c)
	$c_i(\mathbf{x}) \le 0$	for $i = (m_e+1), ..., m$	(1d)

where $\mathbf{x}$ is the n-dimensional vector of quantities to be optimized, $J(\mathbf{x})$ is the scalar cost function, and the $c_i(\mathbf{x})$ for $i = 1, ..., m$ are the scalar constraint functions with the first m_e constraints being equality constraints and the last m_i ($=m-m_e$) constraints being inequalities. The functions $J(\mathbf{x})$ and $c_i(\mathbf{x})$ for $i = 1, ..., m$ are assumed to be continuous and to have continuous first and second derivatives.

The remainder of this paper is divided into 2 sections plus conclusions. Section 2 describes the algorithm, which consists of 3 nested iterative loops. Section 3 describes some test problems and compares the performance of the algorithm on these problems to that of NPSOL (Ref. 5).

2. Nonlinear Programming Algorithm Description[4]

2.1. Outer-Loop Augmented Lagrangian Algorithm. The basic outer-loop algorithm for applying the augmented Lagrangian method to problem (1a)-(1d) is given by Fletcher (Ref. 3). Reference 3 also gives a proof of global convergence when a feasible point exists and when the global minimum of each fixed-multiplier problem can be determined. The algorithm is repeated with minor modifications here. The method works with guessed values for the optimal Kuhn-Tucker multipliers, λ_i for i = 1, ..., m, and penalty parameters, ρ_i for i = 1, ..., m, to define the augmented Lagrangian function:

$$L_{aug}(\mathbf{x};\underline{\lambda},\underline{\rho}) = J(\mathbf{x}) + \sum_{i=1}^{m_e} \frac{\rho_i}{2} \{ c_i(\mathbf{x}) + \frac{\lambda_i}{\rho_i} \}^2 + \sum_{i=m_e+1}^{m} \frac{\rho_i}{2} \{ [c_i(\mathbf{x}) + \frac{\lambda_i}{\rho_i}]^+ \}^2 \qquad (2)$$

where $\rho_i > 0$ for i = 1, ..., m and $\lambda_i \geq 0$ for i = (m_e+1), ..., m, where the operation $[\ ]^+$ returns the quantity in the brackets if it is non-negative and zero otherwise, and where the vectors $\underline{\lambda}$ and $\underline{\rho}$ refer to $[\lambda_1, ..., \lambda_m]^T$ and $[\rho_1, ..., \rho_m]^T$, respectively.

Outer-Loop Augmented Lagrangian Algorithm [from Ref. 3 with modifications]:

1. Start with guesses λ_i, and ρ_i for i = 1, ..., m. Set $\|c^{(old)}\|_\infty = \infty$. Be sure $\lambda_i \geq 0$ for i = (m_e+1), ..., m. Define constraint violation limits, c_i^{max} (> 0) for i = 1, ..., m.
2. Minimize $L_{aug}(\mathbf{x};\underline{\lambda},\underline{\rho})$ with respect to $\mathbf{x}$ to find $\mathbf{x}^*(\underline{\lambda},\underline{\rho})$. Increase ρ_i if necessary to ensure that $|c_i(\mathbf{x}^*)| \leq c_i^{max}$ for i = 1, ..., m_e and that $c_i(\mathbf{x}^*) \leq c_i^{max}$ for i = (m_e+1), ..., m to keep intermediate guesses within reasonable limits.
3. Let $c_i \equiv c_i\{\mathbf{x}^*(\underline{\lambda},\underline{\rho})\}$ for i = 1, ..., m_e

   and

$$c_i \equiv \left[c_i\{\mathbf{x}^*(\underline{\lambda},\underline{\rho})\} + \frac{\lambda_i}{\rho_i} \right]^+ - \frac{\lambda_i}{\rho_i} \qquad \text{for } i = (m_e+1), ..., m.$$

---

[4] Reference 6 gives many algorithmic details that have been omitted from this section for the sake of brevity.

4. If all $|c_i|$ for i = 1, ..., m are small enough, then terminate.

5. If $|c_i| > \frac{1}{100}\|c^{(old)}\|_\infty$ for any i = 1, ..., m, then set $\rho_i = 100\rho_i$ for all such i and go to step 2.

6. Set $c^{(old)} = [c_1, ..., c_m]^T$

7. Set $\lambda_i = \lambda_i + \rho_i c_i$ for i = 1, ..., m and go to step 2.

The quantities c_i for i = 1, ..., m are, nominally, the constraint violations.

Step 5 of the above algorithm is different from the algorithm given by Powell (Ref. 2) and repeated in Fletcher (Ref. 3). It raises penalty weights more rapidly to enforce a faster linear rate of convergence of the multiplier estimates, 1/100 rather than 1/4 as in Ref. 2. The enforcement of c_i^{max} limits in step 2 is an added feature that precludes the possibility that extreme descent of the J(x) function would cause the algorithm to diverge in a nonfeasible region.

2.2. Transformation and Quadratic Approximation of Augmented Lagrangian Sub-Problem. An equivalent constrained form of the fixed-multiplier sub-problem in step 2 can be developed, one that remains numerically well-conditioned even as the $\rho_i \to \infty$:

find: $\mathbf{x}$ and $\mathbf{y}$ $(= [y_1, ..., y_m]^T)$ (3a)

to minimize: $J_t(\mathbf{x},\mathbf{y}) = J(\mathbf{x}) + \frac{h}{2}\sum_{i=1}^{m_e} y_i^2 + \frac{h}{2}\sum_{i=m_e+1}^{m}\{[y_i]^+\}^2$ (3b)

subject to: $c_i(\mathbf{x}) - \sqrt{h/\rho_i}\, y_i + (\lambda_i/\rho_i) = 0$ for i = 1, ..., m (3c)

where J_t is the transformed cost function, the y_i for i = 1, ..., m are added slack variables, and h is an arbitrary positive constant that allows adjustment of scaling. The slack variables are added to equality constraints and inequality constraints to penalize constraint violations.

A quadratic problem (QP) that approximates problem (3a)-(3c) is

find: $\Delta\mathbf{x}$ and $\Delta\mathbf{y}$ (4a)

to minimize: $\Delta J_t^{(j)}(\Delta \mathbf{x}, \Delta \mathbf{y}) = \frac{1}{2}\Delta \mathbf{x}^T \mathbf{H}^{(j)} \Delta \mathbf{x} + \mathbf{g}^{(j)T} \Delta \mathbf{x}$

$$+ \sum_{i=1}^{m_e} \left\{ \frac{h}{2} \Delta y_i^2 + \sqrt{\rho_i h}\, \tilde{c}_i^{(j)}\, \Delta y_i \right\}$$

$$+ \sum_{i=m_e+1}^{m} \left(\frac{h}{2} \left\{ [\Delta y_i + \sqrt{\rho_i / h}\, \tilde{c}_i^{(j)}]^+ \right\}^2 - \frac{\rho_i}{2} \left\{ [\tilde{c}_i^{(j)}]^+ \right\}^2 \right) \qquad (4b)$$

subject to: $\mathbf{a}_i^{(j)T} \Delta \mathbf{x} + \frac{1}{2}\Delta \mathbf{x}^T \mathbf{B}_i^{(j)} \Delta \mathbf{x} - \sqrt{h/\rho_i}\, \Delta y_i = 0 \quad \text{for } i = 1, \ldots, m \qquad (4c)$

$$-\mathbf{d}^{(j)} \le \Delta \mathbf{x} \le \mathbf{d}^{(j)} \qquad (4d)$$

where $\Delta \mathbf{x}$ and $\Delta \mathbf{y}$ are, respectively, deviations of $\mathbf{x}$ and $\mathbf{y}$ from current guesses, $\mathbf{x}^{(j)}$ and $\mathbf{y}^{(j)}$:

$$\Delta \mathbf{x} \equiv \mathbf{x} - \mathbf{x}^{(j)} \qquad (5a)$$

$$\Delta y_i \equiv y_i - y_i^{(j)} = y_i - \sqrt{\rho_i / h}\, \{ c_i(\mathbf{x}^{(j)}) + (\lambda_i / \rho_i) \} \qquad \text{for } i = 1, \ldots, m \qquad (5b)$$

and where $\mathbf{H}^{(j)}$ is the cost function Hessian evaluated at $\mathbf{x}^{(j)}$, $\mathbf{B}_i^{(j)}$ is the Hessian of constraint i, $\mathbf{g}^{(j)}$ is the cost function gradient, $\mathbf{a}_i^{(j)}$ is the gradient of constraint i, and

$$\tilde{c}_i^{(j)} \equiv c_i(\mathbf{x}^{(j)}) + (\lambda_i / \rho_i) = \sqrt{h/\rho_i}\, y_i^{(j)} \qquad \text{for } i = 1, \ldots, m \qquad (6)$$

The quantities $\mathbf{d}^{(j)}$ ($> \mathbf{0}$) are box trust region bounds. The algorithm assumes that the Hessians of the cost and constraint functions are available either analytically or via finite-difference approximation.

QP (4a)-(4c) is similar to the QP of an SQP technique in both form and function; the middle-loop solves a sequence of such problems in order to perform step-2 of the outer-loop. However, the inclusion of quadratic constraint terms in the problem, the $\Delta \mathbf{x}^T \mathbf{B}_i^{(j)} \Delta \mathbf{x}$ terms in Eq. (4c), is a significant deviation from the standard SQP technique. They make the approximate problem valid over a larger region, but complicate the solution procedure for Eqs. (4a)-(4d). The box trust region bounds in Eq. (4d) ensure that a solution to the approximate problem produces a decrease of the original augmented Lagrangian function.

2.3. Middle-Loop SQP/Trust Region Algorithm for Minimizing the Fixed ρ/λ Augmented Lagrangian Function. This section defines the middle-loop algorithm to accomplish step 2 of the outer-loop augmented Lagrangian algorithm.

Middle-Loop SQP/Trust-Region Algorithm for Minimizing an Augmented Lagrangian Function:

2.1. Start with a guess of the solution to problem (3a)-(3c), $\mathbf{x}^{(0)}$, and a guess of the box trust region bounds, $\mathbf{d}^{(0)}$ (> 0). Set j = 0. Compute $L_{aug}^{(0)} = L_{aug}(\mathbf{x}^{(0)};\underline{\lambda},\underline{\rho})$.

2.2. Calculate $\mathbf{y}^{(j)}$, $\mathbf{H}^{(j)}$, $\mathbf{B}_i^{(j)}$, $\mathbf{a}_i^{(j)}$, and $\tilde{c}_i^{(j)}$ for i = 1, ..., m.

2.3. (Approximately) solve problem (4a)-(4d) to determine $\Delta\mathbf{x}$, $\Delta\mathbf{y}$, and $\Delta J_t^{(j)}$.

2.4. Compute $L_{aug}^{(j)+\Delta} = L_{aug}(\mathbf{x}^{(j)}+\Delta\mathbf{x};\underline{\lambda},\underline{\rho})$, compute $r = (L_{aug}^{(j)+\Delta} - L_{aug}^{(j)})/\Delta J_t^{(j)}$, and compute $\gamma = \max_{1\le i\le n} |\Delta x_i|/d_i^{(j)}$.

2.5 If $r \le 0.25$, then set $\mathbf{d}^{(j+1)} = 0.25 \bullet \gamma \bullet \mathbf{d}^{(j)}$.

If $r > 0.75$ and $\gamma = 1$, then set $\mathbf{d}^{(j+1)} = \min(2\bullet\mathbf{d}^{(j)},\mathbf{d}^{(0)})$.

Otherwise, set $\mathbf{d}^{(j+1)} = \mathbf{d}^{(j)}$.

2.6 If $r \le 0$, set $\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)}$ and $L_{aug}^{(j+1)} = L_{aug}^{(j)}$.

Otherwise, set $\mathbf{x}^{(j+1)} = \mathbf{x}^{(j)}+ \Delta\mathbf{x}$ and $L_{aug}^{(j+1)} = L_{aug}^{(j)+\Delta}$.

2.7 Test termination conditions for $\mathbf{x}^{(j+1)}$ and terminate successfully if they are satisfied.

2.8 Set j = j + 1. If r > 0, go to step 2.2; otherwise, go to step 2.3.

This algorithm adopts the SQP philosophy of solving a quadratic approximation to the original problem to determine the next solution iterate. The use of quadratic terms in both the constraints and the cost makes the quadratic sub-problem a very accurate approximation of the problem in Eqs. (3a)-(3c) when near a local minimum of the latter problem that satisfies second-order sufficient conditions. Steps 2.5 and 2.6 of the algorithm implement the trust region ideas found in Ref. 3 (p. 96).

2.4 Inner-Loop Solution of the Quadratically-Constrained QP

The quadratic constraint terms make problem (4a)-(4d) much more difficult than a standard QP. An iterative Newton-type procedure is used to solve this sub-problem. This inner-loop generates a sequence of solution estimates $\Delta\mathbf{x}^{(k)}$ and associated slack variables

$$\Delta y_i^{(k)} = \sqrt{\rho_i/h}\ \left\{ a_i^T \Delta x^{(k)} + \tfrac{1}{2}\Delta x^{(k)T} B_i \Delta x^{(k)} \right\} \qquad \text{for } i = 1, \ldots, m \tag{7}$$

where the superscripts (j) that appear in Eqs. (4a)-(4d) have been dropped for the sake of convenience. Equation (7) guarantees feasibility of Eq. (4c).

Each successive iterate of $\Delta x^{(k)}$ is generated by a line search along a search direction δx. Search directions are determined by solving a linear-equality-constrained QP of the form

$$\text{find} \qquad \delta x \text{ and } \delta y_a \tag{8a}$$

$$\text{to minimize:} \quad \delta J(\delta x, \delta y_a) = \tfrac{1}{2}[\delta x^T, \delta y_a^T] \begin{bmatrix} \mathcal{H} & 0 \\ 0 & hI \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y_a \end{bmatrix} + [g_x^T, g_{y_a}^T] \begin{bmatrix} \delta x \\ \delta y_a \end{bmatrix} \tag{8b}$$

$$\text{subject to:} \quad \begin{bmatrix} A^{(k)} & D^{(k)} \\ E^{(k)} & 0 \end{bmatrix} \begin{bmatrix} \delta x \\ \delta y_a \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{8c}$$

where δy_a is a vector of increments to the slack variables associated with the active subset of the problem constraints in Eq. (4c), $[A^{(k)}\ D^{(k)}]$ is the Jacobian of these active constraints evaluated at $\Delta x^{(k)}$, $E^{(k)}$ is the Jacobian of the active box trust region constraints, $[g_x^T, g_{y_a}^T]$ is the gradient of the Eq.-(4b) cost at $\Delta x^{(k)}$, $\Delta y_a^{(k)}$, and $\mathcal{H}$ is a Hessian matrix of the form

$$\mathcal{H} = H + \sum_{i \in \text{Active Set}} \beta_i B_i \tag{9}$$

Given a positive definite projected Hessian, the QP in Eqs. (8a)-(8c) remains well-conditioned as penalty weights approach infinity. The corresponding elements of $D^{(k)}$ approach 0 in this case. When the projected Hessian is indefinite, the algorithm detects this condition and toggles back and forth in its choice of δx between choosing a feasible direction of negative curvature and a projected steepest descent direction.

The algorithm chooses one of two alternate updates for $\Delta x^{(k)}$. One update makes a straight-line search in the δx direction:

$$\Delta x^{(k+1)} = \Delta x^{(k)} + \alpha\, \delta x \tag{10a}$$

where α is the step length. The other update makes a curved search

$$\Delta x^{(k+1)} = \Delta x^{(k)} + \alpha\, \delta x + \delta\tilde{x}(\alpha) \tag{10b}$$

where $\delta\tilde{x}(\alpha)$ corrects for the curvature in the active problem constraints. A similar correction has been used by Coleman and Conn (Ref. 7), Fletcher (Ref. 8 and Ref. 3, pp. 393-396), and Betts

and Huffman (Ref. 9). It helps the algorithm take larger steps when high penalty weights would limit α in Eq. (10a)-type steps. The curvature correction is a solution of

$$\text{find} \qquad \delta\tilde{\mathbf{x}} \tag{11a}$$

$$\text{to minimize} \qquad \tfrac{1}{2}\delta\tilde{\mathbf{x}}^T\delta\tilde{\mathbf{x}} \tag{11b}$$

$$\text{subject to} \qquad \mathbf{A}^{(k)}\,\delta\tilde{\mathbf{x}} + \frac{1}{2}\begin{bmatrix} (\alpha\delta\mathbf{x})^T\mathbf{B}_{i_1}(\alpha\delta\mathbf{x}) \\ (\alpha\delta\mathbf{x})^T\mathbf{B}_{i_2}(\alpha\delta\mathbf{x}) \\ \vdots \\ (\alpha\delta\mathbf{x})^T\mathbf{B}_{i_s}(\alpha\delta\mathbf{x}) \end{bmatrix} = \mathbf{0} \tag{11c}$$

where $i_1, \ldots, i_s$ are the indices of the active subset of the Eq.-(4c) constraints.

The β_i multipliers used to form the Hessian in Eq. (9) are calculated in two different ways, depending on which type of line search is to be performed. They must make the cost in Eq. (8b) a valid approximation, to second-order in $\alpha\,\delta\mathbf{x}$, of the cost in Eq. (4b). If the straight-line search of Eq. (10a) is used, then the correct multipliers are

$$\beta_i = \rho_i\,\bar{c}_i + \sqrt{\rho_i h}\;\Delta y_i \qquad \text{for } i = i_1, \ldots, i_s \tag{12}$$

If the curved search of Eq. (10b) is to be used, then the correct multiplier vector is the solution of

$$\text{find} \qquad \underline{\beta} = [\beta_{i_1}, \ldots, \beta_{i_s}]^T \tag{13a}$$

$$\text{to minimize} \qquad \tfrac{1}{2}(\mathbf{A}^{(k)T}\,\underline{\beta} + \mathbf{g}_x)^T(\mathbf{A}^{(k)T}\,\underline{\beta} + \mathbf{g}_x) \tag{13b}$$

The step length α is determined by a univariate minimization. When the straight-line step in Eq. (10a) is used, the cost function in Eq. (4b) is used as the step-length merit function, subject to the box trust region bounds in Eq. (4d). Equation (7) determines feasible values of $\Delta\mathbf{y}^{(k+1)}$ as a function of $\Delta\mathbf{x}^{(k+1)}$ for use in this merit function. The merit function is piecewise-quartic in α. Exact univariate minimization can be performed in a finite number of operations. When the curved step in Eq. (10b) is used, a local approximation of the Eq.-(4b) cost is used as the merit function. It yields a piecewise-quadratic univariate minimization problem that also can be solved exactly [6].

Inner-Loop Minimization Algorithm for the Quadratically-Constrained, Piecewise-Quadratic-Cost Sub-problem:

2.3.1 Start with the solution guess $\Delta\mathbf{x}^{(0)} = \mathbf{0}$ and $\Delta\mathbf{y}^{(0)} = \mathbf{0}$. Set k = 0, and initialize the active set of problem constraints and the active set of box trust region constraints.

2.3.2 Solve the linear-equality-constrained QP of Eqs. (8a)-(8c) for the search direction, $[\delta\mathbf{x},\delta\mathbf{y}_a]$. Use Eqs. (13a)-(13b) to determine the β_i multipliers for the Hessian.

2.3.3 Determine α by performing a line search. Use the merit function associated with the curved step. Update the active set of problem constraints and the active set of box trust region constraints.

2.3.4 Test termination conditions for $\Delta\mathbf{x}^{(k+1)} = \Delta\mathbf{x}^{(k)} + \alpha\delta\mathbf{x} + \delta\tilde{\mathbf{x}}(\alpha)$ and stop with solution $\Delta\mathbf{x}^{(k+1)}$ if conditions are satisfied. Otherwise, set $\Delta\mathbf{x}^{(k+1)} = \Delta\mathbf{x}^{(k)} + \alpha\delta\mathbf{x}$ and set k = k + 1

2.3.5 Test stationary point conditions. If they are satisfied, then go to step 2.3.11.

2.3.6 If $k \geq 7$ and $\Delta J_t(\Delta\mathbf{x}^{(k)},\Delta\mathbf{y}^{(k)}) \geq 0$, then set $\Delta\mathbf{x}^{(k)} = \mathbf{0}$, $\Delta\mathbf{y}^{(k)} = \mathbf{0}$, and re-initialize the active set of problem constraints and the active set of box trust region constraints.

2.3.7 If $k \geq 15$, then terminate before reaching the optimal solution.

2.3.8 Solve the linear-equality-constrained QP of Eqs. (8a)-(8c) for the search direction, $[\delta\mathbf{x},\delta\mathbf{y}_a]$. Use Eqs. (12) to determine the β_i multipliers for the Hessian.

2.3.9 Determine α by performing a line search. Use the merit function associated with the straight-line step. Update the active set of problem constraints and the active set of box trust region constraints.

2.3.10 Set $\Delta\mathbf{x}^{(k+1)} = \Delta\mathbf{x}^{(k)} + \alpha\delta\mathbf{x}$, set k = k + 1, and go to step 2.3.5.

Steps that drop active trust region bounds if non-optimal.

2.3.11 If $\Delta J_t(\Delta x^{(k)},\Delta y^{(k)})$ can be reduced by moving off of some active trust region bounds, then determine a feasible descent direction δx that moves away from the non-optimal bounds, drop these bounds from the active set, and go to step 2.3.9.

2.3.12 If $\Delta J_t(\Delta x^{(k)},\Delta y^{(k)}) \geq 0$, then set $\Delta x^{(k)} = 0$, $\Delta y^{(k)} = 0$, re-initialize the active set of problem constraints and the active set of box trust region constraints, and go to step 2.3.8.

2.3.13 Terminate successfully.

This algorithm should terminate after just one execution of steps 2.3.1-2.3.4 when the middle-loop algorithm is near its optimum. The active set will have been correctly identified, and the δx direction calculated in this case will be the Newton direction for the middle-loop. Figure 1 depicts a typical scenario for termination at step 2.3.4.

When the algorithm enters the loop in steps 2.3.5-2.3.10, it searches for the optimum via a series of steps that use straight-line searches. Figure 2 depicts such a scenario. The heavy arrow marked $\alpha\delta x^{(0)}$ is the step-2.3.3 increment. The curvature correction $\delta\tilde{x}(\alpha)$ is rejected in step 2.3.4 because it does not come near enough to the optimal solution or to the constraint. The two increments $\alpha\delta x^{(1)}$ and $\alpha\delta x^{(2)}$ are the search steps taken in the two subsequent iterations of steps 2.3.5-2.3.10. The reason for trying the curvature correction $\delta\tilde{x}(\alpha)$ before resorting to the straight-line searches $\alpha\delta x^{(1)}$ and $\alpha\delta x^{(2)}$ is that $\delta\tilde{x}(\alpha)$ is relatively inexpensive to calculate.

One danger of this technique is that a cost increase may occur during the $\alpha\delta x^{(0)}$ step. The approximate nature of the merit function used in step 2.3.3 could allow this to happen. Steps 2.3.6 and 2.3.12 provide logic for recovering from such a problem.

The inner-loop terminates after 15 iterations at step 2.3.7 even if problem (4a)-(4d) has not been completely solved. It is unwise to take too many inner-loop search steps per middle-loop search step because each inner-loop iteration involves a significant amount of linear algebra.

Changes to the active constraint sets can occur in two places. Problem inequality constraints, which are enforced via penalty terms, can be added or dropped during the line searches of steps 2.3.3 and 2.3.9. Box trust region constraints can be added to the active set in these same steps, but step 2.3.11 is the only place where box trust region bounds can be dropped from the active set.

2.5. Phase-I Constraint Satisfaction Algorithm. The convergence proof of the outer-loop algorithm assumes that the middle-loop can find a global minimum (Ref. 3), which is not guaranteed for this paper's algorithm. This limitation could cause the algorithm to converge to a point near an infeasible local minimum of the norm of the constraint violations. The algorithm tries to avoid this situation by using a Phase-I procedure to move from an arbitrary first guess to an almost-feasible first guess, one that does not violate any problem constraint, i, by more than $0.01c_i^{max}$. Phase-I executes the middle- and inner-loops with a multiplier guess $\underline{\lambda} = \mathbf{0}$, with very large penalty weights, and with the assumption that $J(\mathbf{x}) = 0$. In order to make the inner-loop QP problem well posed, Phase-I uses $\mathbf{H}^{(j)} = \varepsilon\mathbf{I}$ and $\mathbf{g}^{(j)} = \mathbf{0}$ in Eq. (4b).

2.6. Discussion of Algorithm. The algorithm has several advantages as compared to other nonlinear programming techniques. It has a strong likelihood of global convergence because the augmented Lagrangian outer-loop has good global convergence. Even with highly-curved constraints, the algorithm has reasonably fast global convergence because of the explicit use of quadratic constraint approximations in the QP sub-problem. Rapid local convergence is assured because the inner- and middle-loops converge quadratically under "normal" conditions, and the outer-loop has a very rapid linear convergence rate of .01.

The algorithm also has several significant disadvantages. It must re-do full matrix factorizations for each iteration of the inner-loop. In a general context this is a serious deficiency, but in the intended application, trajectory optimization problems, a parallel factorization algorithm is available (Ref. 10), which decreases the impact of this weakness. Another disadvantage is the algorithm's reliance on analytic or finite-difference Hessians. The algorithm requires many more gradient-type evaluations per search step than a quasi-Newton method. Again, this weakness is

not serious in the context of trajectory optimization, where the Hessian is sparse but the quasi-Newton approximation is not, and where Hessian calculations are highly parallelizable (Ref. 11).

3. Performance Evaluation on 5 Test Problems

This section reports the results of tests of the algorithm's speed and convergence robustness, and it compares the algorithm with NPSOL version 4.02 (Ref. 5). Each NPSOL iteration is one in which it calculates a gradient, does a BFGS quasi-Newton update to the projected Hessian, solves a QP, and performs a line search. For comparison purposes, each iteration of the augmented Lagrangian algorithm's middle-loop is defined as one "iteration". Five test problems are described in the appendix. Table 1 compares the performance of the two algorithms on the five problems.

Table 1.

Iteration Count Comparison of New Algorithm with NPSOL on 5 Test Problems.

Problem	NPSOL	New Algorithm Loops				
		Middle Total	Middle Phase-I	Outer Total	Inner Total	Middle Optimality
1	6	6	1	3	15	5
2	Quits at 34	73	46	4	285	27
3	Quits at 1	15	3	5	71	12
4A	Quits	Quits	-	-	-	-
4B	Quits at 7	14	3	6	89	11
5	9	13	2	5	70	11

Obviously, the new algorithm is more robust. It converged in all but one of the cases listed, but NPSOL converged in only two of the cases. (Note that NPSOL's poorer robustness should not be construed as evidence that all SQP techniques are inherently less robust.) When NPSOL

does converge, it is as fast or slightly faster in terms of the number of QPs that get solved; compare the second and third columns in Table 1 for problems 1 and 5. For static problems on a serial processor, NPSOL's speed advantage is more pronounced than indicated by the iteration counts because its QP sub-problems are cheaper to derive and cheaper to solve.

A comparison of the last two columns in Table 1 shows that the average number of inner-loop iterations per middle-loop iteration ranges from 3 to 10.5 for the new algorithm. For each successful run, the early middle-loop iterations require multiple inner-loop iterations, and the latter middle-loop iterations each require only one inner-loop iteration, consistent with the second-order nature of the inner-loop.

Variations in the choice of initial penalty weights can affect the performance of the new algorithm. Another solution run for problem 5 has been tried in which the initial penalty weights ρ_i are set at 100 (they are 1000 for the run listed in Table 5). This latter run is slower, requiring a total of 22 middle-loop iterations to solve the problem.

4. Conclusions

An augmented Lagrangian nonlinear programming algorithm has been developed. It consists of 3 nested loops. The outer-loop is an augmented Lagrangian algorithm with a first-order multiplier update and a rapid rate of linear convergence. The middle-loop algorithm is an SQP-type minimization of the augmented Lagrangian function. To alleviate ill-conditioning, the middle-loop problem is recast into a constrained form with slack variables that get penalized. The inner-loop algorithm solves a quadratically-constrained piecewise quadratic program with box trust region bounds. The algorithm has been compared with NPSOL on 5 low-order test problems. The algorithm has much more robust convergence than NPSOL, and its speed is comparable to NPSOL.

Appendix, 5 Test Problems

Test problem 1:

find: $\mathbf{x} = [x_1, x_2]^T$

to minimize: $J(\mathbf{x}) = - x_1 - x_2$

subject to: $x_1^2 + x_2^2 - 1 \le 0$

$(x_1 + 1)^2 + x_2^2 - 4 \le 0$

$-2x_1 + x_2 - 2 \le 0$

First guess: $\mathbf{x} = [2,0]^T$

Optimum: $\mathbf{x} = [.7071,.7071]^T$

Test problem 2 (very difficult constraint curvature case):

find: $\mathbf{x} = [x_1, x_2]^T$

to minimize: $J(\mathbf{x}) = x_2$

subject to: $(x_1 - 1)^2 + x_2^2 + 10000\,(x_1^2 + x_2^2 - 1)^2 - .0625 \le 0$

First guess: $\mathbf{x} = [0,.99]^T$

Optimum: $\mathbf{x} = [.9688,-.2480]^T$

Test problem 3 [Ref. 3, p. 330]:

find: $\mathbf{x} = [x_1, x_2, x_3, x_4]^T$

to minimize: $J(\mathbf{x}) = x_1 x_2$

subject to: $\dfrac{(x_1x_3 + x_2x_4)^2}{x_1^2 + x_2^2} - x_3^2 - x_4^2 + 1 = 0$

$- x_1 + x_3 + 1 \le 0$

$- x_2 + x_4 + 1 \le 0$

$- x_3 + x_4 \le 0$

$- x_4 + 1 \le 0$

First guess: $\mathbf{x} = [1,1,1,1]^T$

Optimum: $\mathbf{x} = [3.41,3.41,2.41,1]^T$

Test problem 4:

find: $\mathbf{x} = [x_1, x_2, x_3, x_4, x_5, x_6, x_7]^T$

to minimize: $J(\mathbf{x}) = \frac{1}{2}(x_4 - x_2)(x_3 - x_1)$

subject to: $(x_6 - x_4)(x_3 - x_1) - (x_5 - x_1)(x_2 - x_4) = 0$

$x_5 - x_7(x_4 - x_2) = 0$

$x_6 - x_7(x_3 - x_1) = 0$

$-x_5^2 - x_6^2 + 1 \le 0$

$x_j + 1 \le 0 \quad$ for j = 1,2

$-x_j \le 0 \quad$ for j = 3,4,5,6,7

First guess 4A: $\mathbf{x} = [-2, -3, 6, 6, 6, 6, 6]^T$

First guess 4B: $\mathbf{x} = [-2, -3, 6, 6, 6, 6, .7]^T$

Optimum: $\mathbf{x} = [-1, -1, 2.41, 2.41, .71, .71, .21]^T$

Test problem 5: Minimum-fuel 2-burn impulsive orbit transfer from equatorial geosynchronous Earth orbit to 28°-inclination low Earth orbit. This problem is described in Ref. 6. It has 8 elements in its x decision vector, a linear cost, and 9 nonlinear inequality constraints.

References

1. HESTENES, M.R., *Multiplier and Gradient Methods*, Journal of Optimization Theory and Applications, Vol. 4, pp. 303-320, 1969.
2. POWELL, M.J.D., *A Method for Nonlinear Constraints in Minimization Problems*, in Optimization, Edited by R. Fletcher, Academic Press, London, pp. 283-298, 1969.
3. FLETCHER, R., *Practical Methods of Optimization, 2nd Ed.*, J Wiley & Sons, New York, New York, 1987.
4. PSIAKI, M.L. and PARK, K., *A Parallel Trajectory Optimization Tool for Aerospace Plane Guidance*, AIAA Paper No. 91-5069, presented at the AIAA Third International Aerospace Planes Conf., Orlando, FL, 1991.

5. GILL, P.E., MURRAY, W., SAUNDERS, M.A., and WRIGHT, M.H., *User's Guide for NPSOL (Version 4.0): A FORTRAN Package for Nonlinear Programming*, Stanford University, Systems Optimization Laboratory Report No. SOL 86-2, 1986.

6. PSIAKI, M.L. and PARK, K., *Detailed Description of an Augmented Lagrangian Nonlinear Programming Algorithm that uses SQP and Trust Region Techniques*, Mechanical and Aerospace Engineering Report No. MSD-92-02, 1992.

7. COLEMAN, T.F. and CONN, A.R., *Nonlinear Programming via an Exact Penalty Function: Asymptotic Analysis*, Mathematical Programming, Vol. 24, pp. 123-136, 1982.

8. FLETCHER, R., *Second Order Corrections for Non-differentiable Optimization*, Numerical Analysis, Dundee 1981, Lecture Notes in Mathematics 912, Edited by G.A. Watson, Springer-Verlag, New York, New York, pp. 85-114, 1982.

9. BETTS, J.T., AND HUFFMAN, W.P., *Application of Sparse Nonlinear Programming to Trajectory Optimization*, Journal of Guidance, Control, and Dynamics, Vol. 15, pp. 198-206, 1992.

10. PSIAKI, M.L. and PARK, K., *A Parallel Orthogonal Factorization Null-Space Method for Dynamic Quadratic Programming*, submitted to the Journal of Optimization Theory and Applications, in review.

11. BETTS, J.T., AND HUFFMAN, W.P., *Trajectory Optimization on a Parallel Processor*, Journal of Guidance, Control, and Dynamics, Vol. 14, pp. 431-439, 1991.

List of Figures

Fig. 1. Rapid termination of the inner-loop after one optimization step and one curvature-correction step.

Fig. 2. Search steps taken by the inner-loop when it fails to terminate in one iteration.

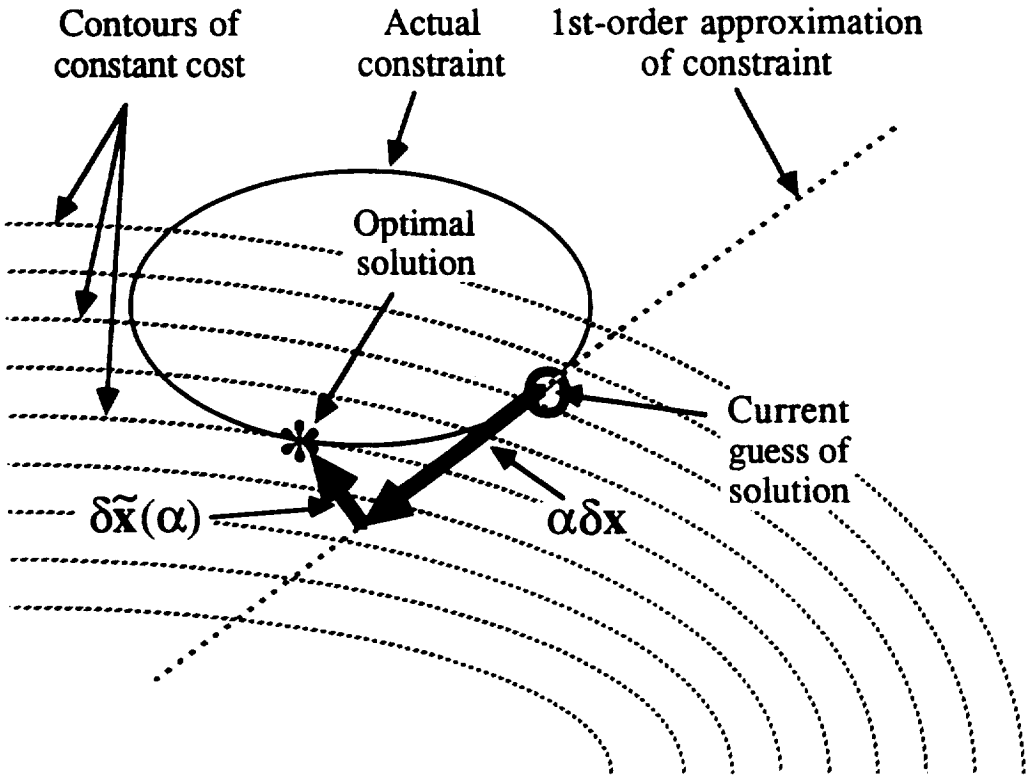


Fig. 1 Psiaki & Park

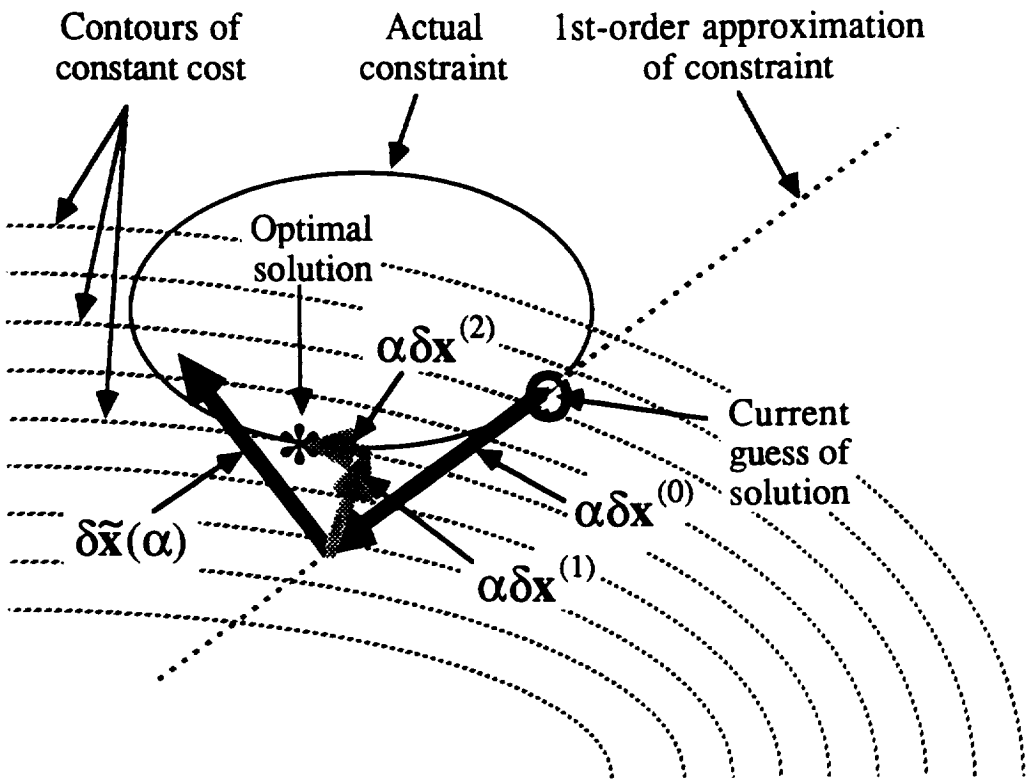


Fig. 2 Psiaki & Park

Appendix C

A paper about the integration of component algorithms into a parallel trajectory optimization algorithm.

A PARALLEL TRAJECTORY OPTIMIZATION TOOL FOR AEROSPACE PLANE GUIDANCE

Mark L. Psiaki* and Kihong Park**
Cornell University
Ithaca, N.Y. 14853-7501

Abstract

A parallel trajectory optimization algorithm is being developed. One possible mission is to provide real-time, on-line guidance for the National Aerospace Plane. The algorithm solves a discrete-time problem via the augmented Lagrangian nonlinear programming algorithm. The algorithm exploits the dynamic programming structure of the problem to achieve parallelism in calculating cost functions, gradients, constraints, Jacobians, Hessian approximations, search directions, and merit functions. Special additions to the augmented Lagrangian algorithm achieve robust convergence, achieve (almost) superlinear local convergence, and deal with constraint curvature efficiently. The algorithm can handle control and state inequality constraints such as angle-of-attack and dynamic pressure constraints. Portions of the algorithm have been tested. The nonlinear programming core algorithm performs well on a variety of static test problems and on an orbit transfer problem. The parallel search direction algorithm can reduce wall clock time by a factor of 10 for this part of the computation task.

1. Introduction

1.1. Objectives.

This effort aims to develop a fast trajectory optimization algorithm that converges with high reliability. One possible application of such a tool is for real-time guidance of an aerospace vehicle such as an aerospace plane. Other applications would be to off-line trajectory planning or to off-line vehicle/design studies.

The algorithm is designed to solve a fixed-time Bolza-type optimal control problem. Other problem types can be stated in this form, e.g., the minimax problem can be approximated by a Bolza problem, and the free-end-time problem can be transformed into a fixed-end-time problem. The problem under consideration is

$$\text{find:} \quad \boldsymbol{u}(t) \text{ and } \boldsymbol{x}(t) \quad \text{for } t_0 \le t \le t_f \tag{1a}$$

$$\text{to minimize:} \quad J = \int_{t_0}^{t_f} L[\boldsymbol{x}(t),\boldsymbol{u}(t),t]\,dt + V[\boldsymbol{x}(t_f)] \tag{1b}$$

$$\text{subject to:} \quad \boldsymbol{x}(t_0) \text{ given} \tag{1c}$$

$$\dot{\boldsymbol{x}} = \boldsymbol{f}[\boldsymbol{x}(t),\boldsymbol{u}(t),t] \tag{1d}$$

$$\boldsymbol{a}_e[\boldsymbol{x}(t),\boldsymbol{u}(t),t] = \boldsymbol{0} \tag{1e}$$

$$\boldsymbol{a}_i[\boldsymbol{x}(t),\boldsymbol{u}(t),t] \le \boldsymbol{0} \tag{1f}$$

$$\boldsymbol{a}_{ef}[\boldsymbol{x}(t_f)] = \boldsymbol{0} \tag{1g}$$

$$\boldsymbol{a}_{if}[\boldsymbol{x}(t_f)] \le \boldsymbol{0} \tag{1h}$$

where $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ are the state and control time histories, respectively. Equation (1b) is a Bolza type cost, and Eq. (1d) defines the system's dynamics. Equations (1e) and (1f) are auxiliary constraints and may be pure control constraints, such as maximum angle of attack, pure state constraints, such as heating rate of an aerospace plane, or mixed control and state constraints, such as normal load factor. Equations (1g) and (1h) are terminal state constraints.

Use For real-Time Guidance. A block diagram of how a trajectory optimization algorithm might be used for real-time control is shown in Fig. 1. It approximates $\boldsymbol{u}(t)$ by a sequence $\boldsymbol{u}(t_k)$, ..., $\boldsymbol{u}(t_{N-1})$ [$t_N = t_f$]. The algorithm takes a sensor-based estimate of the initial state, $\hat{\boldsymbol{x}}(t_k)$. It uses that state as the initial condition in a trajectory optimization problem to compute the entire optimal control and state time histories, $\boldsymbol{u}(t_k)$, ..., $\boldsymbol{u}(t_{N-1})$ and $\boldsymbol{x}(t_{k+1})$, ..., $\boldsymbol{x}(t_N)$, and it sends the control for the current time, $\boldsymbol{u}(t_k)$, to the controlled system.

The proposed guidance system uses feedback by discarding its prediction of $\boldsymbol{x}(t_{k+1})$ at time t_{k+1} in favor of the sensor-based estimate $\hat{\boldsymbol{x}}(t_{k+1})$. Any discrepancy between these values forces the algorithm to re-optimize the trajectory. Such a system should be more robust than programmed optimal control. A similar system has been used successfully on the Boeing Inertial Upper Stage [1].

Use for Off-line Studies. A faster algorithm with more robust convergence would also be helpful to off-line trajectory optimization. When used for traditional mission planning, increased speed and robustness would reduce the amount of engineering time required to do a job. The increased speed might make exotic design studies more practical. For example, robust trajectory optimization, in which the sensitivity of the solution to poorly known parameters is also constrained or penalized, could be made more practical.

The present algorithm uses parallelism to achieve faster solution speeds. It is compatible with a new parallel supercomputer that NASA and a consortium of other agencies have purchased. This machine is a 500+ node

---

* Assistant Prof., Mech. and Aero. Eng.; Member, AIAA.

** Graduate Student.

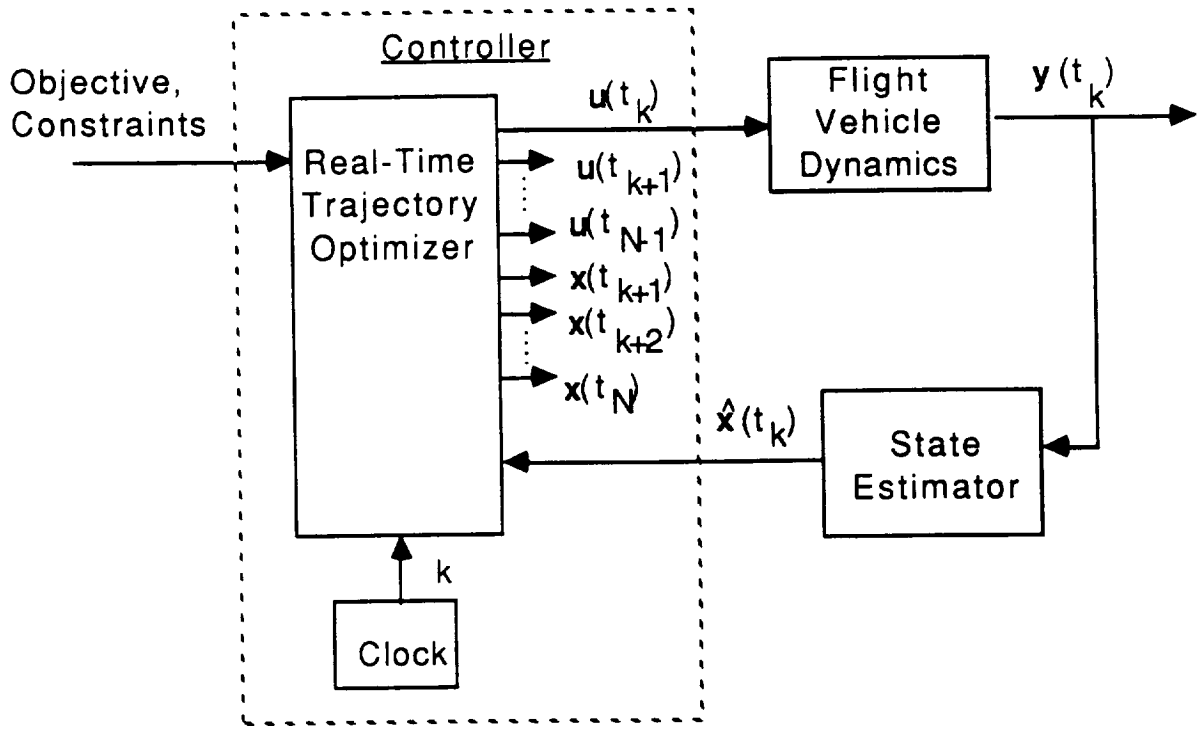


Fig. 1. Block Diagram of a Guidance System that uses a Trajectory Optimization Algorithm in the Loop.

INTEL touchstone machine. Each node is rated at 18 Mflops or more. Thus, this algorithm could take advantage of a highly parallel machine with about 9 Gflops of speed.

1.2. Related Research.

Many algorithms have been developed for off-line trajectory optimization, e.g. see Refs. 2-7. These are numerical gradient-based search algorithms. Generally, such algorithms are unsuited for on-line applications because they are too slow or because they do not converge reliably enough.

Another approach to real-time optimization is the perturbation technique. It expands the problem in terms of a small parameter in such a way that the zeroth-order solution has a closed-form solution or one that is easily computed numerically. The higher-order terms are computed by solving a series of linear, nonhomogeneous problems [8]. Such designs may be very effective for a given application, but they will only work when a natural perturbation parameter exists and when the zeroth-order problem is easy to solve. Furthermore, each new application requires a new perturbation analysis to design the approximate optimal controller.

Several researchers have explored the possibility of using parallelism to speed up numerical search trajectory optimization. Betts and Huffman were the first to implement a complete numerical trajectory optimization algorithm that makes use of parallelism [9]. Their work concentrated on parallelizing the calculation of cost derivatives and constraint Jacobians. Their algorithm realizes significant improvements until the problem size becomes too large, at which point their serial nonlinear programming algorithm becomes a significant bottleneck.

Wright [10] and Psiaki and Park [11,12] have both worked on algorithms for parallelizing the linear algebra in the nonlinear programming portion of a trajectory optimization algorithm. Both approaches use a divide-and-conquer approach that does a stage-wise domain decomposition of the linear algebra. Both algorithms are faster than serial backwards sweep algorithms. Wright's algorithm is faster than that of Psiaki and Park, but it cannot handle auxiliary state constraints or an indefinite Hessian.

The present algorithm uses the numerical search trajectory optimization approach. It defines a transformation to a discrete-time problem in order to make use of ideas from the field of general nonlinear programming (parameter optimization), but it retains the structure and benefits of the dynamic programming form. The parallel linear algebra and the augmented Lagrangian nonlinear programming algorithm developed in previous work are used [12]. Parallel gradient and Jacobian calculations are used, similar to Betts and Huffman [9].

Parallel discrete-Newton Hessian approximation is used also.

1.3. Outline of Paper.

Section 2 describes the trajectory optimization algorithm. It gives transformations to a discrete-time problem form and then to a parameter optimization problem form. The section defines and discusses the nonlinear programming algorithm in the parameter optimization form. Section 2 finishes with explanations of how parallelism is used to speed up the calculation of derivatives, to perform matrix factorizations, and to evaluate a merit function. Section 3 describes the performance of sections of the code on test problems, the general nonlinear programming algorithm and the parallel linear algebra algorithm --- the entire algorithm is not yet running. Section 4 explains the planned development sequence for completion of the entire algorithm, and section 5 gives conclusions about the algorithm's design and about the performance of some algorithm components.

2. Algorithm Design.

2.1. Transformation to Discrete Time.

The problem in Eqs. (1a)-(1h) will be solved on a digital computer via a numerical technique. Some form of problem discretization must be used simply to represent the solution. A number of researchers have developed algorithms directly for the continuous-time problem, which they implement approximately. Our approach is to first approximate the problem in Eqs. (1a)-(1h) by a finite-dimensional numerical optimization technique. Then we develop an algorithm for the finite problem that can be exactly implemented (neglecting round-off error). Such a problem becomes a parameter optimization problem.

Approximation of a continuous-time problem by a parameter optimization problem has several advantages. A parameter optimization algorithm can use many of the sophisticated techniques that have been developed by researchers in the general field of nonlinear programming. For example, the continuous-time problem of determining switching times and transversality conditions for state inequality constraints is conceptually simpler in discrete-time: it reduces to a matter of determining the active set of inequality constraints. With a good linear algebra package there is no need to differentiate the state constraints in order to get the controls to appear in them.

The parameter optimization technique sometimes gets criticized because it can destroy problem structure and cause the resulting algorithm to be very slow. The particular form of parameter optimization problem used in this paper is a discrete-time optimal control problem. This preserves the dynamic programming structure.

A special-purpose algorithm has been designed to solve the resulting nonlinear program (NP). General-purpose NP algorithms can be used [7,9], but they take no advantage of the dynamic programming problem structure. Such algorithms run very slowly for a large number of discrete-time steps†. The special algorithms of this paper exploit the sparsity and structure of the dynamic programming form.

The zero-order-hold discretization of the control time history splits the interval $[t_0,t_f]$ into N intervals $[t_k,t_{k+1}]$ for k = 0, ..., N-1, with $t_N = t_f$. The control is modeled as being constant at the value u_k for each interval $[t_k,t_{k+1})$:

$$u(t) = \begin{cases} \vdots & \\ u_{k-1} & \text{for } t_{k-1} \le t < t_k \\ u_k & \text{for } t_k \le t < t_{k+1} \\ u_{k+1} & \text{for } t_{k+1} \le t < t_{k+2} \\ \vdots & \end{cases} \qquad (2)$$

Then, the problem in Eqs. (1a)-(1h) may be transformed into the discrete-time form

$$\text{find:} \quad x = \left[u_0^T, x_1^T, u_1^T, x_2^T, \ldots, u_{N-1}^T, x_N^T \right]^T \qquad (3a)$$

to minimize:

$$J = \sum_{k=0}^{N-1} L_k(x_k,u_k) + V[x_N] \qquad (3b)$$

subject to:

$$x_0 \text{ given} \qquad (3c)$$

$$x_{k+1} = f_k(x_k,u_k) \quad \text{for } k = 0 \ldots N-1 \qquad (3d)$$

$$a_{e_k}(x_k,u_k) = 0 \quad \text{for } k = 0 \ldots N-1 \qquad (3e)$$

$$a_{i_k}(x_k,u_k) \le 0 \quad \text{for } k = 0 \ldots N-1 \qquad (3f)$$

$$a_{e_N}(x_N) = 0 \qquad (3g)$$

$$a_{i_N}(x_N) \le 0 \qquad (3h)$$

where the correspondence between the discrete-time control time history, $u_0, u_1, u_2, \ldots, u_{N-1}$, and the continuous-time $u(t)$ has been defined in Eq. (2). The discrete-time state time history, $x_0, x_1, x_2, \ldots, x_N$, corresponds to a sampling of the continuous-time state time history, $x(t_0), x(t_1), x(t_2), \ldots, x(t_N)$.

The difference equation in the discrete-time problem, Eq. (3d), models the system's dynamics. The function $f_k(x_k,u_k)$ on the right-hand side is defined by the solution of an initial value problem using the continuous-time system's

† The terms step and stage are used interchangeably to refer to one discrete-time interval.

fferential equation, Eq. (1d). Defining $x^k(t)$ to be the ..ate between sample times t_k and t_{k+1}, it is the solution of the following initial value problem,

$$\dot{x}^k = f[x^k(t),u_k,t] \quad (4a)$$

$$x^k(t_k) = x_k \quad (4b)$$

with the control held fixed at u_k for the interval. Given that Eqs. (4a) and (4b) define $x^k(t)$, the discrete-time dynamics function is defined as

$$f_k(x_k,u_k) \equiv x^k(t_{k+1}) \quad (5)$$

A consistent definition can also be made of the summand in the discrete-time cost, Eq (3b). Given the definition of $x^k(t)$ in Eqs. (4a) and (4b), the summand is

$$L_k(x_k,u_k) \equiv \int_{t_k}^{t_{k+1}} L[x^k(t),u_k,t]\,dt \quad (6)$$

Equations (4a)-(6) ensure that the discrete-time cost in Eq. (3b) and the discrete-time dynamics in Eq. (3d) exactly model their continuous-time counterparts, Eqs. (1b) and (1d). Equations (3b) and (3d) accurately model the effect of the zero-order hold on Eqs. (1b) and (1d). This modeling ·echnique is borrowed from Ref. 13.

Note that the definitions in Eqs. (4a), (4b) and (6) involve integration. The computer subroutines that evaluate the $f_k(x_k,u_k)$ and $L_k(x_k,u_k)$ functions use numerical integration and user-supplied continuous-time functions, $f[x(t),u(t),t]$ and $L[x(t),u(t),t]$. Section 2.5 explains how to transform partial derivatives from continuous-time to discrete-time in a way that is consistent with these definitions.

The auxiliary continuous-time constraints in the original continuous-time problem are sampled to produce the auxiliary constraints of the discrete-time problem:

$$a_{e_k}(x_k,u_k) \equiv a_e(x_k,u_k,t_k) \quad \text{for } k = 0 \ldots N\text{-}1 \quad (7a)$$

$$a_{i_k}(x_k,u_k) \equiv a_i(x_k,u_k,t_k) \quad \text{for } k = 0 \ldots N\text{-}1 \quad (7b)$$

Care must be taken so that the sample interval does not get too long. Otherwise, the continuous-time constraints in Eqs. (1e) and (1f) may get violated by significant amounts in the interval between sample times.

The terminal constraints of the continuous-time problem carry over directly to the discrete-time problem:

$$a_{e_N}(x_N) \equiv a_{e_f}(x_N) \quad (8a)$$

$$a_{i_N}(x_N) \equiv a_{i_f}(x_N) \quad (8b)$$

2.2. Transformation to Static Parameter Optimization Problem Form.

The problem in Eqs. (3a)-(3h) is in a dynamic programming form. It has a special structure that can be used to develop an efficient algorithm. Nevertheless, this section transforms the problem into a general static parameter optimization form. This is done to simplify notation during the subsequent discussion of the overall nonlinear programming solution procedure, sections 2.3 and 2.4. Later, in sections 2.5-2.7, the special structure will be discussed to show how efficiency and parallelism can be achieved.

In static form, the problem becomes

$$\text{find:} \quad \boldsymbol{x} \quad (9a)$$

$$\text{to minimize:} \quad J(\boldsymbol{x}) \quad (9b)$$

$$\text{subject to:} \quad \boldsymbol{c}_e(\boldsymbol{x}) = 0 \quad (9c)$$

$$\boldsymbol{c}_i(\boldsymbol{x}) \le 0 \quad (9d)$$

where the parameter vector $\boldsymbol{x}$ is defined in Eq. 3a. It is a long vector that contains the entire discrete-time control and state time histories. The cost in Eq. 9b is defined by the cost in Eq. 3b.

The equality constraint in Eq. (9c) has many rows. It includes the state difference equations and the auxiliary equality constraints for stages 0 through N-1, Eqs. (3d) and (3e), and the terminal equality constraint, Eq. (3g):

$$\boldsymbol{c}_e(\boldsymbol{x}) \equiv \begin{bmatrix} a_{e_0}(x_0,u_0) \\ f_0(x_0,u_0) - x_1 \\ a_{e_1}(x_1,u_1) \\ \vdots \\ a_{e_{N-1}}(x_{N-1},u_{N-1}) \\ f_{N-1}(x_{N-1},u_{N-1}) - x_N \\ a_{e_N}(x_N) \end{bmatrix} \quad (10)$$

The inequality constraint in Eq. (9d) also has many rows. It includes the auxiliary inequality constraints for stages 0 through N-1, Eq. (3f), and the terminal inequality constraint, Eq. (3h):

$$\mathbf{c}_i(\mathbf{x}) \equiv \begin{bmatrix} a_{i_0}(x_0,u_0) \\ a_{i_1}(x_1,u_1) \\ \vdots \\ a_{i_{N-1}}(x_{N-1},u_{N-1}) \\ a_{i_N}(x_N) \end{bmatrix} \qquad (11)$$

2.3. An Augmented Lagrangian Nonlinear Programming Algorithm.

The augmented Lagrangian algorithm is a standard nonlinear programming algorithm for dealing with nonlinear constraints [14,15]. Normally, the augmented Lagrangian method solves a series of unconstrained minimizations. Each unconstrained minimization works on a penalty function that includes linear and quadratic penalties of the constraint violations. The linear terms arise from guesses of the constraint multipliers, and they allow the algorithm to achieve exact constraint satisfaction with finite penalty weights on the quadratic terms. After each unconstrained minimization, the algorithm corrects its guesses of the constraint multipliers in order to improve the degree of constraint satisfaction.

The augmented Lagrangian algorithm used in this paper formulates the problem in terms of slack variables, which get penalized. The slack-variable augmented Lagrangian algorithm solves a series of sub-problems of the form

find: $\mathbf{x}$, $\underline{\theta}_e$, and $\underline{\theta}_i$ (12a)

to minimize: $J(\mathbf{x}) + \frac{h}{2}\underline{\theta}_e^T\underline{\theta}_e + \frac{h}{2}\left[\underline{\theta}_i\right]^{+T}\left[\underline{\theta}_i\right]^{+}$ (12b)

subject to: $\mathbf{c}_e(\mathbf{x}) - \sqrt{h/\rho}\ \underline{\theta}_e + (1/\rho)\,\underline{\Delta}_e^* = 0$ (12c)

$\mathbf{c}_i(\mathbf{x}) - \sqrt{h/\rho}\ \underline{\theta}_i + (1/\rho)\,\underline{\Delta}_i^* = 0$ (12d)

where the parameter vector $\mathbf{x}$, the cost function $J(\mathbf{x})$, and the constraint functions $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$ have been defined in section 2.2. The vector $\underline{\theta}_e$ contains the slack variables for the equality constraints, and $\underline{\theta}_i$ contains the slack variables for the inequality constraints. The scalar constant h is arbitrary and is included to enhance numerical stability. The scalar ρ is a penalty weight. The vectors $\underline{\Delta}_e^*$ and $\underline{\Delta}_i^*$ constitute the guesses of the multipliers for the equality and the inequality constraints, respectively.

The $[\ ]^+$ symbol in Eq. (12b) is an operator that takes a vector input and produces a vector result in which all of the nonpositive elements have been replaced by zeros. This operator allows the inequality constraints' slack variable penalty terms to penalize only positive constraint violations.

A more traditional augmented Lagrangian algorithm arises when Eqs. (12c) and (12d) are used to eliminate the slack variables from the cost. The resulting cost function is called the augmented Lagrangian function:

$$J_{aug}(\mathbf{x};\underline{\Delta}_e^*,\underline{\Delta}_i^*,\rho) = J(\mathbf{x}) + \frac{\rho}{2}\left(\mathbf{c}_e(\mathbf{x}) + (1/\rho)\underline{\Delta}_e^*\right)^T\left(\mathbf{c}_e(\mathbf{x}) + (1/\rho)\underline{\Delta}_e^*\right) + \frac{\rho}{2}\left[\mathbf{c}_i(\mathbf{x}) + (1/\rho)\underline{\Delta}_i^*\right]^{+T}\left[\mathbf{c}_i(\mathbf{x}) + (1/\rho)\underline{\Delta}_i^*\right]^{+} \qquad (13)$$

The traditional augmented Lagrangian algorithm performs a series of unconstrained minimizations of the this function with respect to $\mathbf{x}$. Our reasons for preferring the formulation in Eqs. (12a)-(12d) to this formulation centers around algorithm numerical stability issues for large ρ, which are discussed in Ref. 12.

The solution to the augmented Lagrangian sub-problem is the solution of the original problem when the multiplier guesses are correct. More specifically, if the multiplier guesses in Eqs. (12c) and (12d) are the Kuhn-Tucker multipliers associated with a local minimum of Eqs. (9a)-(9d), and if ρ is large enough, then a corresponding local minimum occurs in Eqs. (12a)-(12d). One can see this by comparing the first-order necessary conditions for the two problems.

The multiplier guesses act as biases on the constraints in Eqs. (12c) and (12d). They cause the slack variables to be nonzero when the constraint functions are zero. In effect, they bias the penalties associated with the slack variables so that a nonzero penalty can occur when a constraint is exactly satisfied. This feature allows the minimum of the augmented Lagrangian function to occur at a value of $\mathbf{x}$ that yields zero constraint violation. This is true even for a finite penalty weighting, ρ.

The main difficulty of the augmented Lagrangian algorithm is to determine the correct multipliers. The following algorithm has been proved to be globally convergent under appropriate assumptions [15]:

Augmented Lagrangian Outer-Loop Iteration

1. Start with guesses $\underline{\Delta}_e^*$, $\underline{\Delta}_i^*$, and ρ. Set k = 0 and $\|c^{(0)}\|_\infty = \infty$. Be sure $\underline{\Delta}_i^* \geq 0$.

2. Minimize $J_{aug}(\mathbf{x};\underline{\Delta}_e^*,\underline{\Delta}_i^*,\rho)$ with respect to $\mathbf{x}$ to find $\mathbf{x}(\underline{\Delta}_e^*,\underline{\Delta}_i^*,\rho)$.

3. Let $c \equiv \begin{bmatrix} \mathbf{c}_e(\mathbf{x}(\Delta_e^*, \Delta_i^*, \rho)) \\ \left[\mathbf{c}_i(\mathbf{x}(\Delta_e^*, \Delta_i^*, \rho)) + (1/\rho)\Delta_i^*\right]^+ - (1/\rho)\Delta_i^* \end{bmatrix}$

4. If $\|c\|_\infty > \frac{1}{100}\|c^{(k)}\|_\infty$ set $\rho = 100\rho$ and go to step 2.

5. Set $k = k + 1$ and $c^{(k)} = c$.

6. Set $\begin{bmatrix} \Delta_e^* \\ \Delta_i^* \end{bmatrix} = \begin{bmatrix} \Delta_e^* \\ \Delta_i^* \end{bmatrix} + \rho c$ and go to step 2.

The critical steps of this algorithm are steps 2, 3, 5, and 6. They form an iteration that feeds back constraint errors from step 3 to update multiplier estimates in step 6, which, in turn, affect the constraint violations that will result after the next minimization in step 2. This "feedback" has been proved to be stable for large enough ρ, and its rate of convergence is adjustable through ρ, which is why step 4 appears in the algorithm. The algorithm starts with initial multiplier guesses of 0 and successfully estimates the correct multipliers via this technique.

The most expensive part of the algorithm is step 2. The algorithm completely solves the sub-problem in Eqs. (12a)-(12d) for each iteration of step 2. A technique akin to sequential quadratic programming is used to solve this sub-problem. Fortunately, after the first iteration of the main augmented Lagrangian outer-loop, a good first guess to the sub-problem of step 2 is usually available. In this case the algorithm used in step 2 requires very few iterations, sometimes just one.

Another important point about step 2 is that the sub-problem can be ill-posed if ρ is not large enough. The problem may have an infinite minimum or it may have a minimum at $\mathbf{x} = \infty$. Reference 12 describes a method of detecting difficulties and increasing ρ when necessary.

An initial feasibility portion of the algorithm has been included as a further precaution to help ensure convergence. The algorithm performs an initial step-2 minimization with only the norm of the constraint violations for a cost function. The original cost function, $J(\mathbf{x})$, is not considered. This step-2 minimization iterates until the maximum constraint violation is brought down below some user-specified tolerance. The Newton search directions during these iterations may be under-determined because there are fewer constraints than problem variables. The feasibility algorithm uses the minimum-norm search direction to resolve any such ambiguity. After this feasibility portion has been carried out, the algorithm initializes multiplier guesses at 0 and proceeds to step 1.

2.4. Sub-Sub-Problems in the Augmented Lagrangian Algorithm.

The second step of the outer-loop of the augmented Lagrangian algorithm is solved by a method that is like the method of successive quadratic programs (SQP). The basic SQP method guesses a solution and then expands the cost and constraints about the guessed solution in a Taylor series to create a quadratic program (QP) sub-problem. The cost gets expanded out to quadratic terms, and the constraints get expanded out to linear terms. This sub-problem then gets solved via a quadratic programming technique.

In this paper the SQP-like technique is being applied to a sub-problem; therefore, each successive QP is a sub-sub-problem of the overall augmented Lagrangian algorithm. The sub-sub-problem solved here uses quadratic approximations of both the cost and the constraints in Eqs. (12a)-(12d). In addition, it includes box trust region bounds. The sub-sub-problem is

find: $\Delta\mathbf{x}$, $\Delta\underline{\theta}_e$, and $\Delta\underline{\theta}_i$ (14a)

to minimize:

$$J_{sub} = \frac{1}{2}\Delta\mathbf{x}^T\mathbf{H}\Delta\mathbf{x} + \mathbf{g}^T\Delta\mathbf{x} + \frac{h}{2}\left\{\Delta\underline{\theta}_e + \sqrt{\rho/h}\,\tilde{\mathbf{c}}_e\right\}^T\left\{\Delta\underline{\theta}_e + \sqrt{\rho/h}\,\tilde{\mathbf{c}}_e\right\} + \frac{h}{2}\left[\Delta\underline{\theta}_i + \sqrt{\rho/h}\,\tilde{\mathbf{c}}_i\right]^{+T}\left[\Delta\underline{\theta}_i + \sqrt{\rho/h}\,\tilde{\mathbf{c}}_i\right]^+ \quad (14b)$$

subject to:
$$\mathbf{A}_e\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_e\Delta\mathbf{x} - \sqrt{h/\rho}\;\Delta\underline{\theta}_e = 0 \quad (14c)$$
$$\mathbf{A}_i\Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_i\Delta\mathbf{x} - \sqrt{h/\rho}\;\Delta\underline{\theta}_i = 0 \quad (14d)$$
$$-\Delta\mathbf{x}_{box} \le \Delta\mathbf{x} \le \Delta\mathbf{x}_{box} \quad (14e)$$

where $\Delta\mathbf{x}$, $\Delta\underline{\theta}_e$, and $\Delta\underline{\theta}_i$ are increments to the corresponding quantities in Eqs. (12a)-(12d). The matrix $\mathbf{H}$ is the Hessian of the cost function $J(\mathbf{x})$, and $\mathbf{g}$ is the gradient of $J(\mathbf{x})$. The matrices $\mathbf{A}_e$ and $\mathbf{A}_i$ are the Jacobians of $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$, respectively, and the 3-index tensors $\mathbf{B}_e$ and $\mathbf{B}_i$ are the second derivatives of $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$, respectively. The vector $\tilde{\mathbf{c}}_e \equiv \mathbf{c}_e + (1/\rho)\Delta_e^*$, and the vector $\tilde{\mathbf{c}}_i \equiv \mathbf{c}_i + (1/\rho)\Delta_i^*$. The vector $\Delta\mathbf{x}_{box}$ contains the box trust region limits on the increments $\Delta\mathbf{x}$. The guesses of $\underline{\theta}_e$ and $\underline{\theta}_i$ are selected so that Eqs. (12c) and (12d) are satisfied. That is why Eqs. (14c) and (14d) are satisfied for $\Delta\mathbf{x} = \Delta\underline{\theta}_e = \Delta\underline{\theta}_i = 0$.

In order to solve the sub-problem in step 2 of the outer-loop augmented Lagrangian algorithm, a sequence of sub-sub-problems of the form in Eqs. (14a)-(14e) must be solved until $\Delta\mathbf{x} \to 0$. For a well-posed sub-problem, this sequence of sub-sub-problems can be made globally convergent by adaptive selection of the box trust region size $\Delta\mathbf{x}_{box}$ [15].

The sub-sub-problem in Eqs. (14a) - (14e) is, itself, difficult to solve. If the quadratic terms in the constraints were not included, i.e., if $\mathbf{B}_e = \mathbf{B}_i = \mathbf{0}$, then the problem would be a piecewise quadratic program with continuous first derivatives. It would be solvable in a finite number of operations by a QP-type technique. The addition of the constraint curvature terms makes the problem piecewise quartic, and no algorithm exists that can exactly solve it in a fintie number of steps. It must get solved iteratively.

The complication of constraint curvature has been added to this sub-sub-problem in order to reduce the number of sub-sub-problem solutions [Eqs. (14a)-(14e)] needed for a sub-problem solution [Eqs. (12a)-(12d)]. This happens because the sub-sub-problem provides an accurate model of the sub-problem for a larger range of $\Delta\mathbf{x}$, which speeds the convergence rate remote from the solution. This reduces the number of times that first and second derivatives need to get calculated to set up the sub-sub-problem, which is important because these derivatives are computationally expensive. They involve numerical integration to do transformations from continuous-time to discrete-time, and the number of derivatives is large, on the order of the number of discrete-time steps.

The solution procedure for the sub-sub-problem in Eqs. (14a)-(14e) is, itself, an iterative technique. It guesses a solution $\Delta\mathbf{x}_g$, calculates a search direction, and does a line search. The line searches use a piecewise-quartic merit function with continuous-first derivatives. The discontinuous-changes in the higher derivatives occur when inequality constraints change from active to inactive or vice versa. Constraint activity changes occur when an element of the vector in the expression $[\ ]^+$ changes sign.

The search direction calculations for this sub-sub-problem involve solution of an equality-constrained quadratic program. This is accomplished by some matrix factorizations that are used to implement the null space method of quadratic programming [14]. First, a right QR factorization is performed on a constraint Jacobian matrix of the form

$$\begin{bmatrix} \mathbf{A}_e & -\sqrt{h/\rho}\,\mathbf{I} & \mathbf{0} \\ \mathbf{A}_{i_a} & \mathbf{0} & -\sqrt{h/\rho}\,\mathbf{I} \\ \mathbf{E}_{box} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{15}$$

The matrix $\mathbf{A}_{i_a}$ is the Jacobian of the active inequality constraints, the constraints corresponding to nonnegative values of $\Delta\underline{\theta}_i + \sqrt{\rho/h}\ \tilde{\mathbf{c}}_i$. The matrix $\mathbf{E}_{box}$ corresponds to active box trust region constraints. For a trajectory optimization problem all of these matrices are large, but they are also sparse and structured.

The algorithm must then determine the projected Hessian and factorize it. The orthogonal transformation from the QR factorization is used to transform the original Hessian matrix to determine the projected Hessian. The original Hessian takes the form

$$\begin{bmatrix} \mathbf{H} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & h\mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & h\mathbf{I} \end{bmatrix} \tag{16}$$

Each of the matrices will be large, and $\mathbf{H}$ will be block diagonal for a trajectory optimization problem. After transformation, the last factorization is a Cholesky factorization of the projected Hessian.

These factorizations are used to calculate a search direction, and a line search is performed. The merit function used in the line search is the augmented Lagrangian function associated with the sub-sub-problem in Eqs. (14a)-(14e):

$$\begin{aligned} J_{aug} = {} & \tfrac{1}{2}\Delta\mathbf{x}^T\mathbf{H}\Delta\mathbf{x} + \mathbf{g}^T\Delta\mathbf{x} \\ & + \frac{\rho}{2}\left\{ \tilde{\mathbf{c}}_e + \mathbf{A}_e\Delta\mathbf{x} + \tfrac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_e\Delta\mathbf{x} \right\}^T \\ & \quad \left\{ \tilde{\mathbf{c}}_e + \mathbf{A}_e\Delta\mathbf{x} + \tfrac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_e\Delta\mathbf{x} \right\} \\ & + \frac{\rho}{2}\left[\tilde{\mathbf{c}}_i + \mathbf{A}_i\Delta\mathbf{x} + \tfrac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_i\Delta\mathbf{x} \right]^{+T} \\ & \quad \left[\tilde{\mathbf{c}}_i + \mathbf{A}_i\Delta\mathbf{x} + \tfrac{1}{2}\Delta\mathbf{x}^T\mathbf{B}_i\Delta\mathbf{x} \right]^{+} \end{aligned} \tag{17}$$

The foregoing discussion has been in terms of a general parameter optimization framework. This has been done for the sake of notational convenience, but the actual algorithm has been specialized to the dynamic programming problem structure in Eqs. (3a)-(3h). This is necessary for efficiency and parallelism. The most time-consuming parts of this algorithm are the calculation of derivatives to set up the sub-sub-problem, the factorization of matrices at each search step of the sub-sub-problem, and the calculation of the augmented Lagrangian function during line searches for the sub-sub-problem. The next three sub-sections explain how parallelism and the special dynamic programming problem structure can both be exploited to speed the algorithm steps described in this sub-section.

2.5. Derivative Calculations.

The sub-sub-problem in Eqs. (14a)-(14e) has various vectors and matrices that are first or second partial derivatives of functions from the problem in Eqs. (12a)-(12d): $\mathbf{g}$, $\mathbf{H}$, $\mathbf{A}_e$, $\mathbf{B}_e$, $\mathbf{A}_i$, and $\mathbf{B}_i$. For trajectory optimization, the problem functions in Eqs. (12a)-(12d) are defined in terms of discrete-time problem functions from Eqs. (3a)-(3h). This sub-section shows how the discrete-time problem functions' derivatives can be determined from the continuous-time problem functions' derivatives, and it presents an efficient means of parallelizing these derivative calculations. The parallel derivatives scheme is almost identical to that developed by Betts and Huffman [9], except

that it operates on a different discretization of the ntinuous-time problem. Their idea is extended to include second derivative calculations via finite differencing.

Discrete-Time Derivatives from Continuous-Time Derivatives. The algorithm assumes that first partial derivatives with respect to x and u of the continuous-time problem functions in Eqs. (1a)-(1h) are available. Second partial derivatives may also be available, or they may be estimated by finite differencing of the first derivatives. Given this information, the partial derivatives of most of the discrete-time problem functions in Eqs. (3a)-(3h) are known. The only two functions whose partial derivatives are difficult to calculate are $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$ and $L_k(\mathbf{x}_k,\mathbf{u}_k)$.

Reference 13 explains how to differentiate these two functions with respect to their arguments. First, one differentiates Eqs. (4a) and (4b) with respect to the argument to get an initial value problem for the partial derivative of $x^k(t)$ with respect to the argument. Suppose one wants the derivative with respect to $\mathbf{u}_k$. Then the partial derivatives of Eqs. (4a) and (4b) yield

$$\frac{d}{dt}\left[\frac{\partial x^k(t)}{\partial \mathbf{u}_k}\right] = \frac{\partial f}{\partial x}\Big|_{[x^k(t),\mathbf{u}_k,t]}\left[\frac{\partial x^k(t)}{\partial \mathbf{u}_k}\right] + \frac{\partial f}{\partial u}\Big|_{[x^k(t),\mathbf{u}_k,t]} \tag{18a}$$

$$\frac{\partial x^k(t_k)}{\partial \mathbf{u}_k} = 0 \tag{18b}$$

which is a nonhomogeneous linear time-varying matrix initial value problem for $\partial x^k(t)/\partial \mathbf{u}_k$, the control effectiveness matrix. To finish determining the partial derivative of $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$, one differentiates Eq. (5):

$$\frac{\partial \mathbf{f}_k}{\partial \mathbf{u}_k} \equiv \frac{\partial x^k(t_{k+1})}{\partial \mathbf{u}_k} \tag{19}$$

The partial derivative of Eq. (6) finishes the calculation for $L_k(\mathbf{x}_k,\mathbf{u}_k)$:

$$\frac{\partial L_k}{\partial \mathbf{u}_k} \equiv \int_{t_k}^{t_{k+1}} \left\{\frac{\partial L}{\partial x}\Big|_{[x^k(t),\mathbf{u}_k,t]}\left[\frac{\partial x^k(t)}{\partial \mathbf{u}_k}\right] + \frac{\partial L}{\partial u}\Big|_{[x^k(t),\mathbf{u}_k,t]}\right\} dt \tag{20}$$

First partial derivatives with respect to $\mathbf{x}_k$ are calculated in a similar manner.

If Eqs. (4a), (6), (18a), and (20) are all integrated simultaneously with a Runge-Kutta algorithm, then truncation errors enter the calculations in a consistent way. This translates into the following fact: the derivatives calculated numerically in Eqs. (19) and (20) are the exact derivatives, neglecting round-off error, of the numerical-integration approximations of the functions in Eqs. (5) and (6). In other words, the computed derivatives are consistent with the computed functions. This is important to numerical optimization. It guarantees that, in the computer, the sub-sub-problem in Eqs. (14a)-(14e) is locally an accurate approximation of the sub-problem in Eqs. (12a)-(12d).

Second derivatives are calculated in a similar way [13]. Equations (18a)-(20) (or their $\partial/\partial \mathbf{x}_k$ equivalents) are differentiated once more. This results in a nonhomogeneous linear tensor initial value problem, which can be solved numerically.

Parallel Differentiation. For each iteration in step 2 of the main augmented Lagrangian algorithm, a sub-sub-problem of the form in Eqs. (14a)-(14e) must be set up and solved. Sub-sub-problem set-up requires derivatives of all of the functions at each stage of the discrete-time problem [Eqs. (3a)-(3h)]. The cost gradient $\mathbf{g}$ in Eq. (14b) is composed of the gradients of $L_k(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1 and the gradient of $V(\mathbf{x}_N)$. The cost Hessian $\mathbf{H}$ in Eq. (14b) is composed of the Hessians of $L_k(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1 and the Hessian of $V(\mathbf{x}_N)$. The equality constraints' Jacobian matrix $\mathbf{A}_e$ in Eq. (14c) is composed of identity matrices, the Jacobians of $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$ and $\mathbf{a}_{e_k}(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1, and the Jacobian of $\mathbf{a}_{e_N}(\mathbf{x}_N)$. The equality constraints' second derivative tensor $\mathbf{B}_e$ in Eq. (14c) is composed of the second derivative tensors of $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$ and $\mathbf{a}_{e_k}(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1, and the second derivative tensor of $\mathbf{a}_{e_N}(\mathbf{x}_N)$. The inequality constraints' Jacobian matrix $\mathbf{A}_i$ in Eq. (14d) is composed of the Jacobians of $\mathbf{a}_{i_k}(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1, and the Jacobian of $\mathbf{a}_{i_N}(\mathbf{x}_N)$. The inequality constraints' second derivative tensor $\mathbf{B}_i$ in Eq. (14d) is composed of the second derivative tensors of $\mathbf{a}_{i_k}(\mathbf{x}_k,\mathbf{u}_k)$ for k = 0, ..., N-1, and the second derivative tensor of $\mathbf{a}_{i_N}(\mathbf{x}_N)$.

Thus, first and second partial derivatives of the following functions must be calculated for each stage: $L_k(\mathbf{x}_k,\mathbf{u}_k)$, $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$, $\mathbf{a}_{e_k}(\mathbf{x}_k,\mathbf{u}_k)$, and $\mathbf{a}_{i_k}(\mathbf{x}_k,\mathbf{u}_k)$. This is a time-consuming operation because of the numerical integration required to transform from continuous-time to discrete-time and because of the numerical differentiation required to calculate second derivatives.

The derivative calculations that get carried out at one stage are independent of the derivative calculations that get carried out at any other stage. Therefore, these operations can be done in parallel for different stages. If the number of stages equals the number of processors, then each stage's derivatives can be computed on a separate processor. If the number of processors is fewer than the number of stages, then each processor can compute the derivatives for several

stages. A speed-up factor of p can be obtained on p processors for this part of the algorithm if the number of problem stages, N, is an integer multiple of p and if each stage requires an identical amount of derivative calculation time.

No message passing needs to occur during this process; therefore, no bottlenecks can occur. Nevertheless, one must be careful about which processors calculate derivatives for which stages. The results of the derivative calculations are used by a special parallel matrix factorization algorithm. That algorithm requires that the derivative information be distributed over the nodes of a binary tree in a particular way in order to expedite its message passing. The parallel factorization algorithm is described in Ref. 12.

For a 24 stage problem (N = 23) running on an 8-node processor each processor could calculate the derivatives for 3 stages. Figure 2 shows the 8 processors with a binary tree connection topology. The numbers next to the processor nodes show a distribution of problem stages for parallel derivative calculation. This distribution is consistent with the factorization algorithm of Ref 12.

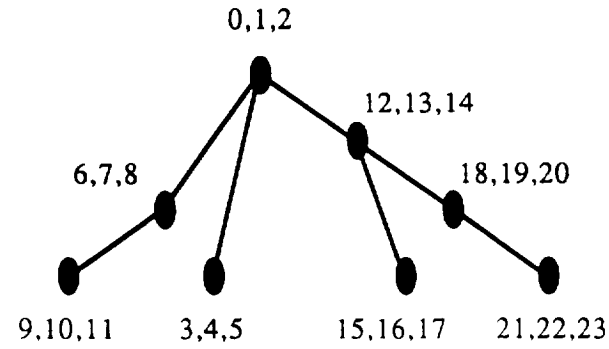


Fig 2. Node Locations of Derivative Calculations for Each Stage of a 24-Stage Problem Run on 8 Processors.

2.6. Parallel Linear Algebra.

The current algorithm uses matrix factorization to determine a search direction. This is equivalent to solving the linearized state/adjoint boundary value problem that arises in calculus-of-variations-based numerical trajectory optimization algorithms [11]. The matrix factorizations/linear algebra associated with Eqs. (15) and (16) can be time consuming because the matrices are large. Furthermore, these factorizations may get carried out one hundred or more times during one trajectory optimization; one complete factorization is performed for each search step in a sub-sub-problem. Therefore, the algorithm makes use of parallelism and the special dynamic programming problem structure to expedite these factorizations.

All of the matrices in Eqs. (15) and (16) are sparse (have many zero entries), and their nonzero elements fall into a block structure. Their structures are

$$H = \begin{bmatrix} X & & & \\ & X & & 0 \\ & & X & \\ & 0 & & \ddots \end{bmatrix} \qquad (21)$$

$$A_e = \begin{bmatrix} X & & & \\ X & X & & 0 \\ & X & & \\ & & X & X \\ & & & X \\ & 0 & & \ddots \end{bmatrix} \qquad (22)$$

$$A_{i_a} = \begin{bmatrix} X & & & \\ & X & & 0 \\ & & X & \\ & 0 & & \ddots \end{bmatrix} \qquad (23)$$

$$E_{box} = \begin{bmatrix} X & & & \\ & X & & 0 \\ & & X & \\ & 0 & & \ddots \end{bmatrix} \qquad (24)$$

where the X's indicate nonzero blocks. The blocks are not necessarily square, except for the blocks of the **H** matrix. Each column of the above block structures corresponds to a different discrete-time stage. The A_e matrix is the only one with coupling between the stages. This is known as the "staircase" structure of dynamic programing, and it arises from the linearizations of the difference equations [Eq. (3d)].

The special matrix factorizations used by this algorithm are described in Ref. 12, and related factorizations are described in Ref. 11. The factorizations amount to a special domain-decomposition† of the null space quadratic programming technique. They can factorize the matrices in wall clock time that is order $O\{n^3[(N/p) + Log_2 p]\}$ for an N-stage problem running on p processors with a block size n. The factorizations have additional beneficial properties. They can check the 2nd-order sufficiency conditions for optimality, they can detect directions of negative curvature, and they can ensure that the search direction is a descent direction for the augmented Lagrangian function.

2.7. Parallel Constraint and Merit Function Evaluations.

The functions $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$ must be evaluated to set up the sub-sub-problem in Eqs. (14a)-(14e). Furthermore,

† Domain decomposition is a parallelization technique that divides the problem into smaller problems, solves them, and then aggregates the problems.

augmented Lagrangian functions must be evaluated at two points in the algorithm. The augmented Lagrangian function in Eq. (17) is used as a merit function during the line searches that seek a solution to the sub-sub-problem in Eqs. (14a)-(14e). The augmented Lagrangian function in Eq. (13) is used as a merit function to evaluate and adapt the L_1 trust region size. Some of these function evaluations involve numerical integration. All involve a number of operations that goes as the number of discrete-time problem stages.

These operations can be expedited by a stage-wise splitting of the evaluation of components. The calculation of $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$ parallelizes in the same manner as the derivative calculations parallelize (see section 2.5). Evaluation of $\mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k)$, $\mathbf{a}_{e_k}(\mathbf{x}_k,\mathbf{u}_k)$, and $\mathbf{a}_{i_k}(\mathbf{x}_k,\mathbf{u}_k)$ for all of the stages constitutes evaluation of $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$ [see Eqs. (10) and (11)], and these functions can be evaluated for different stages simultaneously on different processors.

The augmented Lagrangian function in Eq. (13) can be rewritten as a sum over all of the discrete-time problem stages of a stage-wise augmented Lagrangian function.

$$J_{aug}(\mathbf{x};\Delta_e^*,\Delta_i^*,\rho) = \sum_{k=0}^{N} J_{aug_k} \qquad (25a)$$

where

$$\begin{aligned} J_{aug_k} &= L_k(x_k,u_k) \\ &+ \frac{\rho}{2}\left(f_k(x_k,u_k) - x_{k+1} + (1/\rho)\lambda_k^*\right)^T \\ &\quad \left(f_k(x_k,u_k) - x_{k+1} + (1/\rho)\lambda_k^*\right) \\ &+ \frac{\rho}{2}\left(a_{e_k}(x_k,u_k) + (1/\rho)\mu_{e_k}^*\right)^T \\ &\quad \left(a_{e_k}(x_k,u_k) + (1/\rho)\mu_{e_k}^*\right) \\ &+ \frac{\rho}{2}\left[a_{i_k}(x_k,u_k) + (1/\rho)\mu_{i_k}^*\right]^{+T} \\ &\quad \left[a_{i_k}(x_k,u_k) + (1/\rho)\mu_{i_k}^*\right]^{+} \qquad \text{for } k = 0, \ldots, N-1 \end{aligned} \qquad (25b)$$

$$\begin{aligned} J_{aug_N} &= V(x_N) \\ &+ \frac{\rho}{2}\left(a_{e_N}(x_N) + (1/\rho)\mu_{e_N}^*\right)^T \\ &\quad \left(a_{e_N}(x_N) + (1/\rho)\mu_{e_N}^*\right) \\ &+ \frac{\rho}{2}\left[a_{i_N}(x_N) + (1/\rho)\mu_{i_N}^*\right]^{+T} \\ &\quad \left[a_{i_N}(x_N) + (1/\rho)\mu_{i_N}^*\right]^{+} \end{aligned} \qquad (25c)$$

Equations (10) and (11) have been used to replace $\mathbf{c}_e(\mathbf{x})$ and $\mathbf{c}_i(\mathbf{x})$ by their discrete-time problem component functions. The multiplier guess vectors λ_0^*, λ_1^*, λ_2^*, ... and $\mu_{e_0}^*$, $\mu_{e_1}^*$, $\mu_{e_2}^*$, ... are the components of the large equality-constraint multiplier guess vector Δ_e^*, and the multiplier guess vectors $\mu_{i_0}^*$, $\mu_{i_1}^*$, $\mu_{i_2}^*$, ... are the components of the large inequality-constraint multiplier guess vector Δ_i^*. A similar breakdown can be made for the discrete-time form of Eq. (17).

The parallel computation of the augmented Lagrangian function in Eq. (25a) first computes the component functions, J_{aug_k}, in parallel. Each component function can be computed on a separate node, or if there are fewer nodes than problem stages, each node can compute the augmented Lagrangian components for several stages. Figure 2 shows a sensible distribution of component function calculations when solving a 24-stage problem in 8 processors. The only message passing that will be required during this first part of the calculation will be to send the value of $\mathbf{x}_{k+1}$ to the node that is calculating J_{aug_k} for those values of k where this is necessary (for k = 2, 5, 8, 11, 14, 17, and 20 on Fig. 2). The computation of the component functions is the most time-consuming part of the calculation, and it is completely parallelizable.

The augmented Lagrangian calculation finishes by summing the component functions in what is called a "fan-in" on the binary tree. This can be illustrated in Figure 2 for the 24-stage problem. First, each node sums the component functions for its three stages. Next, the 4 nodes at the bottom of the figure send their sums to the node above them in the tree. The four receiving nodes then sum their 3-stage result with the received 3-stage result to produce a 6-stage result. The process then repeats itself; the 2 nodes that are one row up from the bottom of the figure send their sums to the node above them in the tree. This process leaves some nodes idle for a fraction of the time, but it constitutes a very small fraction of the time required to calculate the augmented Lagrangian function. Therefore, the augmented Lagrangian calculation is almost 100% parallelizable.

During solution of the sub-sub-problem in Eqs. (14a)-(14e) the algorithm does line searches to a univariate minimum of the function in Eq. (17). This function is piecewise quartic, and the univariate minimization procedure needs to know the exact quartic function for a given line search direction and interval of search length. This

calculation can be performed in a manner similar to the calculation of the augmented Lagrangian function. In this case, there is a binary fan-in to calculate each of the 5 terms in the 4th-order polynomial.

The solution procedure for the sub-sub-problem in Eqs. (14a)-(14e) also calculates quadratic approximations of $f_k(x_k,u_k)$, $a_{e_k}(x_k,u_k)$, and $a_{i_k}(x_k,u_k)$. Parallel evaluation of the quadratic approximations is performed in the same way as parallel evaluation of the functions themselves (see above). The solution procedure also needs to know when the quadratic approximation of $a_{i_k}(x_k,u_k) + (1/\rho)\mu_{i_k}^*$ changes sign during a line search. This signals a discontinuity in the second derivative of the piecewise quartic merit function and triggers calculation of new coefficients of a 4th-order polynomial. The solutions of a quadratic equation give the search step lengths at which the inequality constraints change activity. These quadratic equation solutions can be distributed over different processors in the same stage-wise manner as depicted in Fig. 2.

2.8. Review and Unification of Algorithm Parallelism.

Sections 2.5-2.7 explain how to parallelize portions of the trajectory optimization algorithm presented in sections 2.3 and 2.4. The parallel calculations have a recurrent theme: split the calculations by stages. Figure 2 shows how a 24-stage problem can be mapped onto 8 processors for parallelization of the derivative calculations, the linear algebra, the function evaluations, and the merit function evaluation.

Information for a given stage resides on a single node, as in Fig. 2, in the over-all algorithm. That node stores the current guesses of x_k, u_k, λ_k^*, $\mu_{e_k}^*$, and $\mu_{i_k}^*$. It evaluates the functions $f_k(x_k,u_k)$, $L_k(x_k,u_k)$, $a_{e_k}(x_k,u_k)$, $a_{i_k}(x_k,u_k)$, and J_{aug_k}. It stores the search directions Δx_k and Δu_k, and it updates the solution guesses of x_k, u_k, λ_k^*, $\mu_{e_k}^*$, and $\mu_{i_k}^*$. Each node communicates with other nodes only during certain operations: matrix factorizations to determine the search step, merit function calculations to check the box trust region size, and line search calculations in the sub-sub-problem of Eqs. (14a)-(14e). The algorithm uses one "master" node. The master node does things like determine the final line search length, and it synchronizes the other processors so that each major and minor iteration of the algorithm happens simultaneously on all processors. The master node communicates with the other nodes in a "fan-out" along the binary tree. In the example of Fig. 2, the master node is the node at the top of the figure.

3. Performance of Algorithm Components.

The algorithm described in section 2 is still under development. Two key components have been developed and tested. One component is the augmented Lagrangian nonlinear programming algorithm described in sections 2.3 and 2.4. The other component is the algorithm that does parallel matrix factorizations to solve a QP for the search direction, which is briefly described in section 2.6 and fully described in Ref. 12. The performance of these components is described in this section.

3.1. Augmented Lagrangian Algorithm Performance in Test Problems.

A serial parameter optimization form of the augmented Lagrangian algorithm has been encoded and tested. It has been encoded in matlab and tested on an IBM PC-AT. This section compares its performance to that of NPSOL version 4.02. Two questions are to be answered: is the algorithm more likely to converge than NPSOL, and does the algorithm converge in fewer iterations than NPSOL?

Each NPSOL iteration is one in which it calculates a gradient, does a BFGS quasi-Newton update to the projected Hessian, solves a QP, and performs a line search. Each augmented Lagrangian iteration is one in which it calculates derivatives and solves the sub-sub-problem in Eqs. (14a)-(14e). For dense parameter optimization on a serial processor, an augmented Lagrangian iteration is more expensive than an NPSOL iteration. The augmented Lagrangian requires more gradients per iteration (to do a discrete-Newton Hessian evaluation), it does multiple matrix factorizations per iteration, and it does multiple line searches per iteration. Nevertheless, iterations of the two algorithms are compared on a one-to-one basis because the augmented Lagrangian algorithm's iterations will be greatly sped up when put into the trajectory optimization/parallel processor framework.

The performance of the two algorithms is discussed on a problem-by-problem basis for 5 different test problems.

Test problem 1:

$$\text{find:} \quad x_1, x_2 \tag{26a}$$

$$\text{to minimize:} \quad J = -x_1 - x_2 \tag{26b}$$

$$\text{subject to:} \quad x_1^2 + x_2^2 - 1 \le 0 \tag{26c}$$

$$(x_1 + 1)^2 + x_2^2 - 4 \le 0 \tag{26d}$$

$$-2x_1 + x_2 - 2 \le 0 \tag{26e}$$

Both algorithms have been started from the same first guess: $(x_1, x_2) = (2,0)$. This is an infeasible first guess, and the Jacobians of constraints (26c) and (26d) are degenerate at this point: they have linearly dependent rows. The optimum is (.7071,.7071). NPSOL reaches it in 6 iterations, and the present algorithm reaches it in 1 feasibility and 5 optimality iterations. The two algorithms are about equally fast on this problem.

Test problem 2:

find: x_1, x_2 (27a)

to minimize: $J = -x_2$ (27b)

subject to:

$$(x_1 - 1)^2 + x_2^2 + 10000\,(x_1^2 + x_2^2 - 1)^2 - .0625 \le 0 \quad (27c)$$

Both algorithms have been started from the same first guess: $(x_1, x_2) = (0,.99)$. This is an infeasible first guess. The constraint function has a deep curved valley whose bottom also has a gentle slope. Only a narrow section at the very bottom of a part of the valley is feasible. The first guess is slightly up the wall of the valley and remote from the feasible part. The optimum is (.9688,-.2480). NPSOL quits without reaching the solution. It is not able to achieve feasibility in 34 iterations, and it stops making progress. The present algorithm reaches the solution after 52 feasibility and 25 optimality iterations. This test problem was designed to have particularly difficult constraint curvature. The curvature was enough to cause NPSOL to fail. The present algorithm was slowed down, but it eventually reached the optimum.

Test problem 3: find the minimum-area right triangle that circumscribes a given circle [15].

find: x_1, x_2, x_3, x_4 (28a)

to minimize: $J = x_1 x_2$ (28b)

subject to:

$$\frac{(x_1 x_3 + x_2 x_4)^2}{x_1^2 + x_2^2} - x_3^2 - x_4^2 + 1 = 0 \quad (28c)$$

$$-x_1 + x_3 + 1 \le 0 \quad (28d)$$

$$-x_2 + x_4 + 1 \le 0 \quad (28e)$$

$$-x_3 + x_4 \le 0 \quad (28f)$$

$$-x_4 + 1 \le 0 \quad (28g)$$

Both algorithms have been started from the same first guess: $(x_1, x_2, x_3, x_4) = (1,1,1,1)$. This is an infeasible first guess. The optimum is (3.41,3.41,2.41,1). NPSOL quits without reaching the solution. The present algorithm reaches it after 3 feasibility and 16 optimality iterations.

Test problem 4: find the minimum-area right triangle that circumscribes a given circle. Problem modeling differs from test problem 3.

find: $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ (29a)

to minimize: $J = \frac{1}{2}(x_4 - x_2)(x_3 - x_1)$ (29b)

subject to: $(x_4 - x_6)(x_3 - x_1) - (x_5 - x_1)(x_2 - x_4) = 0$ (29c)

$$x_5 - x_7(x_4 - x_2) = 0 \quad (29d)$$

$$x_6 - x_7(x_3 - x_1) = 0 \quad (29e)$$

$$-x_5^2 - x_6^2 + 1 \le 0 \quad (29f)$$

$$x_1 + 1 \le 0 \quad (29g)$$

$$x_2 + 1 \le 0 \quad (29h)$$

$$-x_j \le 0 \quad \text{for } j = 3,4,5,6,7 \quad (29i)$$

Both algorithms have been started from the same first guess: $(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = (-2, -3, 6, 6, 6, 6, 6)$. This is an infeasible first guess. The optimum is (-1, -1, 2.41, 2.41, .71, .71, .21). NPSOL quits without reaching the solution. The present algorithm reaches it after 4 feasibility and 9 optimality iterations.

Test problem 5: find the minimum-fuel 2-burn impulsive orbit transfer to go from equatorial geosynchronous Earth orbit (GEO) to low Earth orbit with a 28° inclination (LEO). This could be used by a space taxi to plan a rendezvous with the National Aerospace Plane in LEO after having ferried an astronaut out to GEO to fix a communications satellite.

find: $m_0, \Delta V_1, \Delta v, \Delta V_2$ (30a)

to minimize: $J = m_0$ (30b)

subject to: Newton's laws for a spherical Earth (30c)

Finite specific impulse of fuel (30d)

$$r_{LEO} - \varepsilon_r \le r_f \le r_{LEO} + \varepsilon_r \quad (30e)$$

$$V_{circ.} - \varepsilon_v \le V_f \le V_{circ.} + \varepsilon_v \quad (30f)$$

$$-\varepsilon_\gamma \le \gamma_f \le +\varepsilon_\gamma \quad (30g)$$

$$28^\circ - \varepsilon_i \le i_f \le 28^\circ + \varepsilon_i \quad (30h)$$

$$m_{empty} \le m_f \quad (30i)$$

where the usual terminal equality constraints on the final orbital elements have been replaced by the upper and lower bounds on the terminal quantities r_f (geocentric radius), V_f (inertial speed), γ_f (flight path angle), and i_f (inclination) in Eqs. (30e)-(30h). This has been done to make the problem harder. The quantities m_0 and m_f are the initial and terminal masses; their difference is the mass of the fuel burned during the two impulses. The quantities ΔV_1 and ΔV_2 are the vector velocity changes during the two impulses. The angle Δv is the coast arc length between the burns.

The effects of the decision quantities in Eq. 30a on the terminal quantities in Eqs. (30e)-(30h) are calculated by numerical integration of 6 coupled scalar equations of motion of the system. These equations can be found in Ref. 16. The formula for the mass change due to impulsive thrusting is

$$m_f = m_0 \exp\left[\frac{-\|\Delta V_1\| - \|\Delta V_2\|}{V_f}\right] \quad (31)$$

where V_f is the nozzle exit velocity of the fuel. Note that $V_f = gI$, where g is the acceleration of gravity at the Earth's surface and I is the specific impulse of the fuel.

Both algorithms have been started from the same first guess, which satisfies all of the constraints except Eq.

(30i). NPSOL reaches the solution in 9 iterations. The present algorithm solves the problem in 6 feasibility iterations followed by 22 optimality iterations. It is about 3 times slower than NPSOL. It gets near the solution fairly quickly, in about 6 feasibility iterations and 9 optimality iterations, but it takes time to converge to the correct multipliers and to determine a high enough value of the penalty parameter, ρ.

NPSOL is about 3 times faster on this problem for the given first guess, but it has trouble for a different first guess. If the first guess value of m_0 is increased to make the first guess entirely feasible, then NPSOL fails to solve the problem, which seems counter-intuitive. Failure occurs because the cost has no curvature of its own [Eq. (30b)]. When none of the constraints are active, the Lagrangian function also has no curvature; its Hessian is zero. NPSOL attempts to do a BFGS quasi-Newton update of a Hessian approximation. This results in a divide-by-zero, and the algorithm fails. Thus, NPSOL cannot handle linear programs. The algorithm presented in this paper has no such problems.

The forgoing results indicate that NPSOL is more likely to have convergence problems than the algorithm of this paper. We believe that the main reason for this is because our algorithm first achieves feasibility. NPSOL could be modified to do the same. It would probably be slower, but its convergence would probably be more robust. Another problem with NPSOL could be its merit function. NPSOL is an SQP procedure. It is difficult to design a merit function to ensure global convergence for an SQP procedure when applying it to problems with nonlinear constraints. This may explain NPSOL's failure on test problem 2.

The present algorithm can also fail to converge. This will happen if, during the feasibility phase, it reaches a nonzero local minimum of the norm of the constraint violations. Most algorithms would have difficulty in such a situation. The way to avoid such situations is to make a more reasonable first guess. Of course, a better first guess would help NPSOL as well, but NPSOL's ability to converge seems to be more sensitive to the choice of first guess.

3.2. Performance of Parallel Algorithm for Matrix Factorizations and QP Solution.

The algorithm that performs parallel calculation of search directions has been described in Ref. 12, where it is tested on a problem that is like test problem 5 in section 3.1 above. The test problem in Ref. 12 performs the same orbit transfer, but it uses aeromaneuvering and three impulsive burns. A dynamic programming problem form is set up similar to the form described in Eqs. (3a)-(3h). The problem has 6 state vector elements and 3 or 4 control vector elements at each stage. The search direction algorithm has been tested on a linear-quadratic expansion of the nonlinear problem about a guessed solution. Figure 3, borrowed from Ref. 12, shows how the wall-clock time of the algorithm varies with the number of processors and the number of problem stages. It also shows results for a backwards-sweep serial algorithm that solves the same problem. The parallel algorithm executes 10 time faster than the serial algorithm when solving a 128-stage problem on 32 nodes of an INTEL iPSC/2 hypercube.

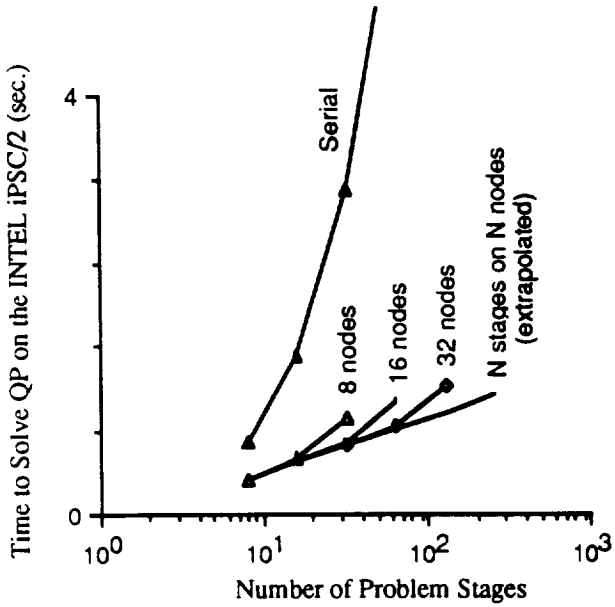


Fig. 3. Wall Clock Time to Solve for the Search Direction as it Varies with Problem Size and with Number of Processors.

4. Planned Developments (the LORD willing!).

The full algorithm is being developed to run on the INTEL iPSC/2 hypercube. Currently, the serial code for augmented Lagrangian parameter optimization is being translated from matlab code into FORTRAN code. We plan first to implement the algorithm serially on one processor of the hypercube. Afterwards we plan to replace the time-consuming serial parts with the parallel algorithms described in sections 2.5-2.7. Code that is borrowed from Ref. 13 will be used, with minor modifications, to do the parallel derivative calculations. The parallel linear search solver is already running. The only part of the parallel code that needs significant development is the parallel augmented Lagrangian calculation portion, described in section 2.7.

The current plan is to develop the code to work with generic problem function forms. This would allow a user to rapidly change system models, cost functions, and constraints. The specification of these functions would be through user-written subroutines. The hope is that this code will be accessible to a wide variety of applications from different disciplines.

5. Conclusions

A nonlinear trajectory optimization algorithm has been developed. It has been formulated to run on a message-passing parallel processor. Special care has been taken to ensure convergence robustness. The algorithm solves a discrete-time trajectory optimization problem, which results from a continuous-time problem after a zero-order-hold discretization of the control time history. Numerical integration is used to complete the continuous-time to discrete-time transformation. It has been designed to handle both state and control constraints. The entire algorithm has not, as yet, been fully encoded, but the most complex portions have been encoded and tested.

The algorithm uses the nonlinear programming technique known as the augmented Lagrangian algorithm. The discrete-time problem form can be directly solved by the augmented Lagrangian algorithm. The particular augmented Lagrangian algorithm of this paper has been specially tailored to increase the terminal convergence speed and to handle extreme constraint curvature.

A serial parameter optimization form of the algorithm compares favorably with NPSOL version 4.02. It has a higher likelihood of converging from a given initial guess than NPSOL: NPSOL failed to work on more than half of the test problems, but the present algorithm worked on all of them. The present algorithm converges as quickly as NPSOL on one test problem, but has run as much as 3 times slower on another.

The algorithm has been designed to parallelize its most time-consuming parts. Function and derivative calculations, search direction calculations, and merit function calculations can all be parallelized on a stage-wise basis. Information about a particular problem stage always resides on a single node. The solutions for stages on different nodes affect each other only in two ways: through a special parallel routine that calculates a global search direction and through the limit to the search length as determined by a global merit function. The global search direction routine operating on 32 processors has achieved a speed-up factor as large as 10 over the best known serial algorithm. The other parallel portions should have speed-up factors approaching the number of processors.

Acknowledgement

This research was supported in part by the National Aeronautics and Space Administration under grant no. NAG-1-1009.

References

1. Hardtla, J.W., "Gamma Guidance for the Inertial Upper Stage", **Proceedings of the AIAA Guidance and Control Conf.**, Aug. 7-9, 1978, Palo Alto, CA, pp. 357-362.
2. Breakwell, J.V., "The Optimization of Trajectories," **J. Soc. Indust. Appl. Math.**, Vol. 7, 1959, pp. 215-247.
3. Bryson, A.E. and Ho, Y.C., **Applied Optimal Control**, Hemisphere Publishing, (Washington, D.C., 1975).
4. Kelley, H.J., "Method of Gradients," **Optimization Techniques**, G. Leitmann, ed., Academic Press, (New York, 1962), Chap. 6, pp. 206-254.
5. Miele, A., Pritchard, R.E., and Damoulakis, J.N., "Sequential Gradient-Restoration Algorithm for Optimal Control Problems," **Journal of Optimization Theory and Applications**, Vol. 5, 1970, pp. 235-282.
6. Yakowitz, S.J., "The Stagewise Kuhn-Tucker Condition and Differential Dynamic Programming," **IEEE Trans. Auto. Cont.**, Vol. AC-31, 1986, pp. 25-30.
7. 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, July-Aug. 1987, pp. 338-342.
8. Feeley, T. and Speyer, J., "Techniques for Developing Approximate Optimal Advanced Launch System Guidance", **Proceedings of the 1991 American Control Conf.**, June 26-28, 1991, Boston, pp. 2238-2243.
9. Betts, J.T. and Huffman, W.P., "Trajectory Optimization on a Parallel Processor," **Journal of Guidance, Control, and Dynamics**, Vol. 14, No. 2, March-April 1991, pp. 431-439.
10. Wright, S.J., "Solution of Discrete-Time Optimal Control Problems on Parallel Computers," Report No. MCS-P89-0789, Argonne National Lab., Chicago, Illinois, July, 1989. (To appear in **Parallel Computing**.)
11. Psiaki, M.L. and Park, K., "A Parallel Solver for Trajectory Optimization Search Directions", Nov. 1990. (to appear in the **Journal of Optimization Theory and Applications**).
12. Psiaki, M.L. and Park, K., "Trajectory Optimization on a Hypercube via Step-Wise Parallelism", Presented and distributed at the 1991 American Control Conf., June 26-28, 1991, Boston, MA.
13. Psiaki, M.L., "Control of Flight Through Microburst Wind Shear Using Deterministic Trajectory Optimization," Ph.D. Thesis, Princeton University, October, 1987.
14. Gill, P.E., Murray, W., and Wright, M.H., **Practical Optimization**, Academic Press, (New York, 1981).
15. Fletcher, R., **Practical Methods of Optimization, 2nd Ed.**, J. Wiley and Sons, (New York, 1987).
16. Miele, A., and Lee, W.Y., "Optimal Trajectories for Hypervelocity Flight", **Proceedings of the 1989 American Control Conference**, June 21-23, 1989, Pittsburgh, pp. 2017-2023.

Appendix D

Specifications for problem model encoding

Example code for the minimum-time proglem.

User-Defined Input Arguments to the Host Machine Multi-Processor Trajectory Optimization Program (MTOP)

Note: All arguments must be dimensioned and typed in the FORTRAN 77 calling program exactly as dimensioned and typed in the definitions below.

1. Problem size/dimension specifications:

N	The INTEGER*4 SCALAR stage number of the terminal stage. The initial stage is number 0; so, there are N+1 stages in total.
NUVEC(0:N)	The INTEGER*4 ARRAY that specifies the number of controls at each stage (e.g., NUVEC(K) is the number of controls at stage K). It is permissible for this number to be zero for some stages.
NXVEC(0:N)	The INTEGER*4 ARRAY that specifies the number of states at each stage (e.g., NXVEC(K) is the number of states at stage K). It is not permissible for this number to be zero for any stage except stage 0. Such a problem would better be solved as two seperate trajectory optimization problems. If NXVEC(K) ≠ NXVEC(K+1), then the dynamic transition from stage K to stage K+1 must be defined by an algebraic discrete-time function (see the discussion of the FLFNCT function in section 5). An algebraic discrete-time state transition function will be signalled by setting IDT(K)=1, as discussed in section 3.

NECVEC(0:N) The INTEGER*4 ARRAY that specifies the number of auxiliary equality constraints of the form $c_k(\mathbf{x}_k,\mathbf{u}_k,t_k) = 0$ at each stage. It is permissible for this number to be zero for some stages.

NICVEC(0:N) The INTEGER*4 ARRAY that specifies the number of auxiliary inequality constraints of the form $c_k(\mathbf{x}_k,\mathbf{u}_k,t_k) \leq 0$ at each stage. It is permissible for this number to be zero for some stages.

NUMAX An INTEGER*4 SCALAR that is used in dimensioning arrays that store control vectors. It must be true that NUMAX ≥ NUVEC(K) for all K = 0, ..., N.

NXMAX An INTEGER*4 SCALAR that is used in dimensioning arrays that store state vectors. It must be true that NXMAX ≥ NXVEC(K) for all K = 0, ..., N.

NCMAX An INTEGER*4 SCALAR that is used in dimensioning arrays that store auxiliary equality and inequality constraint vectors. It must be true that NCMAX ≥ NECVEC(K)+NICVEC(K) for all K = 0, ..., N.

2. Initial guess of solution:

U0(1:NUMAX,0:N) The REAL*8 ARRAY that contains the initial guess of the optimal control time history. Note that U0(J,K) will be ignored for J > NUVEC(K).

X0(1:NXMAX,0:N) The REAL*8 ARRAY that contains the initial guess of the optimal state time history. Note that X0(J,K) will be ignored for J > NXVEC(K). The initial state guess need not satisfy the dynamic equations defined in the subroutine FLFNCT of section 5 of this document, nor must the initial guess satisfy the auxiliary constraints defined in the subroutine CFNCT of section 6 of this document. The optimization process will force the final solution trajectory to obey the dynamic equations and the auxiliary constraints.

Note that X0(1:NXVEC(0),0) constitutes the state vector initial condition of the trajectory optimization problem. It will not be altered by the trajectory optimization algorithm.

3. Problem modelling:

IDT(0:N) This INTEGER*4 ARRAY tells the algorithm whether the user-supplied FLFNCT function, defined below and in section 5, returns continuous-time functions that define the right-hand-side of the state differential equation and the cost integrand for stage K, or discrete-time state transition and cost functions for stage K.

IDT(K) = 0 implies continuous-time stage-K dynamics and cost of the form:

Dynamics: $\dot{\mathbf{x}} = \mathbf{f}_k(\mathbf{x},\mathbf{u},t)$

Stage cost: $\int_{t_k}^{t_{k+1}} L_k(\mathbf{x},\mathbf{u},t)\, dt$

IDT(K) = 1 (or any nonzero integer) implies discrete-time stage-K dynamics of the form

Dynamics: $\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k,t_k)$

Stage cost: $L_k(\mathbf{x}_k,\mathbf{u}_k,t_k)$

The algorithm ignores IDT(N), the stage N value, and assumes that a value of 1 applies. This lets it call FLFNCT to determine the terminal cost. No state transition function or differential equation function is needed at the terminal stage.

T(0:N)

This REAL*8 ARRAY gives the values of the problem's independent variable (normally called time) at the different stages, i.e., X0(:,K) is the state vector initial guess for time T(K). For free end time problems, one must model the real time as a function of an artificial problem time whose end value is fixed. The vector T(0:N) stores the artificial problem time values. In this case, the rate of change of real time with respect to problem time will usually be a function of the controls and, perhaps, the states as well. The real time may need to get modelled as an extra state. The elements T(0:N) must not decrease as K increases. If a stage is modelled as a discrete-time process, if IDT(K) = 1 (or anything other than 0), then T(K+1) = T(K) is permissible. Otherwise, if the stage is modelled by a continuous-time process [IDT(K) = 0], then T(K+1) > T(K) is required. In

this latter case, T(K+1)-T(K) is the time interval used for numerical integration.

FLFNCT

The name of a user-supplied SUBROUTINE that calculates the functions $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ and $L_k(\mathbf{x},\mathbf{u},t)$, which define the dynamics and cost at stage K. This subroutine also must calculate the first partial derivatives of these functions with respect to the state vector, $\mathbf{x}$, and the control vector, $\mathbf{u}$. Section 5 gives detailed specifications for the correct design of this subroutine.

CFNCT

The name of a user-supplied SUBROUTINE that calculates the function $\mathbf{c}_k(\mathbf{x},\mathbf{u},t)$, which defines the auxiliary equality and inequality constraints at stage K. Regardless of whether the stage is a continuous-time stage or a discrete-time stage, these constraints are enforced only at the beginning of the stage

$$c_{k_i}(\mathbf{x}_k,\mathbf{u}_k,t_k) = 0 \quad \text{for } i = 1, \ldots, \text{NECVEC(K)}$$

and

$$c_{k_i}(\mathbf{x}_k,\mathbf{u}_k,t_k) \le 0 \quad \text{for } i = [\text{NECVEC(K)}+1], \ldots, [\text{NECVEC(K)}+\text{NICVEC(K)}]$$

The first NECVEC(K) elements of the $\mathbf{c}_k$ constraint vector are equality constraints, and the last NICVEC(K) elements are inequality constraints.

The CFNCT subroutine also must calculate the first partial derivatives of the $\mathbf{c_k(x,u,t)}$ function with respect to the state vector, **x**, and the control vector, **u**. Section 6 gives detailed specifications for the correct design of this subroutine.

4. Algorithm control:

KRK(1:2,0:(N-1)) This INTEGER*4 ARRAY contains quantities that control the algorithm and number of steps used for numerical integration of the continuous-time segments, the segments for which IDT(K) = 0. If KRK(1,K) = 1, then the standard 4th-order Runge-Kutta algorithm is used on step K to do numerical integration. If KRK(1,K) = 2 (or any integer other than 1), then a 7-step, 6th-order Runge-Kutta method is used. In either case, KRK(2,K) gives the number of Runge-Kutta steps used to integrate from time T(K) to time T(K+1).

ISECGR(0:N) This INTEGER*4 ARRAY specifies whether the FLFNCT and CFNCT subroutines have been programmed to calculate analytic second derivatives. If ISECGR(K) = 0, then all of the required analytic second derivatives at stage K are calculated by the FLFNCT and CFNCT subroutines. If ISECGR(K) = 1, then the optimization algorithm will approximate these second derivatives via one-sided finite-differencing of the first derivatives provided by the FLFNCT and CFNCT subroutines. For example, an element of the

second derivative of $\mathbf{f}(\mathbf{x},\mathbf{u},t)$ with respect to $\mathbf{x}$ is approximated by

$$\frac{\partial^2 \mathbf{f}}{\partial x_i \partial x_j} \approx \frac{\left.\frac{\partial \mathbf{f}}{\partial x_i}\right|_{(\mathbf{x}+\mathbf{e}_j \Delta x_j)} - \left.\frac{\partial \mathbf{f}}{\partial x_i}\right|_{(\mathbf{x})}}{\Delta x_j}$$

where $\mathbf{e}_j$ is a unit vector with a 1 on row j. If ISECGR(K) = 2 (or any larger integer), then the optimization algorithm will approximate the second derivatives via central-differencing of FLFNCT's and CFNCT's first derivatives. For example, an element of the second derivative of $\mathbf{f}(\mathbf{x},\mathbf{u},t)$ with respect to $\mathbf{x}$ is approximated by

$$\frac{\partial^2 \mathbf{f}}{\partial x_i \partial x_j} \approx \frac{\left.\frac{\partial \mathbf{f}}{\partial x_i}\right|_{(\mathbf{x}+\mathbf{e}_j \Delta x_j)} - \left.\frac{\partial \mathbf{f}}{\partial x_i}\right|_{(\mathbf{x}-\mathbf{e}_j \Delta x_j)}}{2\Delta x_j}$$

Additionally, symmetrization of the finite difference second derivatives is performed by averaging each Hessian (of a scalar) with its transpose. This symmetrization is performed in both the one-sided and central difference cases.

Note that ISECGR(K) = 2 will execute more slowly than ISECRG(K) = 1, but the second derivatives will be more accurate. The numerical differentiation steps used are contained in the user-supplied arrays DELTU and DELTX.

DELTU(1:NUMAX,0:N) This REAL*8 ARRAY contains the finite-difference values used for calculating numerical second derivatives when analytic second derivatives have not been supplied. If ISECGR(K) $\neq$ 0, then the values in

DELTU(1:NUVEC(K),K) are used as described above in the section on the ISECGR flag array.

DELTX(1:NXMAX,0:N) This REAL*8 ARRAY contains the finite-difference values used for calculating numerical second derivatives when analytic second derivatives have not been supplied. When ISECGR(K) $\neq$ 0, then the values in DELTX(1:NXVEC(K),K) are used as described above in the section on the ISECGR flag array.

5. Detailed specifications for the FLFNCT SUBROUTINE

This is the dummy name of the subroutine that determines the dynamics and the cost function at each stage. The user uses this subroutine to model the problem. The user has the option of modelling any given stage as a continuous-time process or as a discrete-time process.

If stage k is modelled as a continuous-time process, if IDT(K) = 0, then a zero-order-hold will be assumed for the control inputs, and FLFNCT's output functions $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ and $L_k(\mathbf{x},\mathbf{u},t)$ model a state differential equation and a cost integrand, respectively, for stage k. They are used as follows. Given

$$x(t) = \mathbf{x}_k + \int_{t_k}^{t} \mathbf{f}_k[x(\tau),\mathbf{u}_k,\tau]\, d\tau$$

the state at the next stage and the cost for the stage are

$$\mathbf{x}_{k+1} = x(t_{k+1})$$

$$(\text{Cost})_k = \int_{t_k}^{t_{k+1}} L_k[x(\tau),\mathbf{u}_k,\tau]\, d\tau$$

where $\mathbf{u}_k$ and $\mathbf{x}_k$ are, respectively, the control and state vectors at stage k.

If, on the other hand, stage k is modelled as a discrete-time process, if IDT(K) = 1 or some other nonzero integer, then FLFNCT's output functions $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ and $L_k(\mathbf{x},\mathbf{u},t)$ define, respectively, the discrete-time state difference equation transition function and the discrete-time cost:

$$\mathbf{x}_{k+1} = \mathbf{f}_k(\mathbf{x}_k,\mathbf{u}_k,t_k)$$

$$(\text{Cost})_k = L_k(\mathbf{x}_k,\mathbf{u}_k,t_k)$$

In this case, it is permissible that $\dim(\mathbf{x}_{k+1}) \neq \dim(\mathbf{x}_k)$ (i.e., NXVEC(K+1) ≠ NXVEC(K)).

Regardless of whether stage k is a continuous-time stage or a discrete-time stage, the FLFNCT subroutine must also calculate the first partial derivatives of $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ and $L_k(\mathbf{x},\mathbf{u},t)$ with respect to **x** and **u** when an input flag calls for these to be calculated. Optionally, the user can program the function to calculate the second partial derivatives of $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ and $L_k(\mathbf{x},\mathbf{u},t)$ with respect to **x** and **u** upon request as signalled by an input flag.

If FLFNCT has not been programmed to calculate second derivatives for stage k, then the user must set ISECGR(K) to some other value than 0 so that the optimization algorithm will estimate the second derivatives via finite differencing. Also, values must specified for DELTU(1:NUVEC(K),K) and DELTX(1:NXVEC(K),K) if FLFNCT does not calculate second derivatives for stage k.

The user is responsible for programming FLFNCT correctly so that it accurately models each stage k. The current stage number is provided as an input argument so that FLFNCT can perform the necessary branching to change models for stages that require

such changes. Note that, at stage N, FLFNCT must not return a value for the $\mathbf{f}_N(\mathbf{x},\mathbf{u},t)$ function because there is no next stage to which a transition must occur. Any $\mathbf{f}_N$ value calculated for stage N will be ignored by the optimization algorithm.

The FLFNCT program must have the following argument list and dimensioning and data typing information:

```
      SUBROUTINE  FLFNCT(X,U,T,K,IFLAG,NX,NU,NF,F,L,DFDX,DLDX,
     1     DFDU,DLDU,D2FDX2,D2LDX2,D2FDXU,D2LDXU,D2FDU2,
     2     D2LDU2)
      INTEGER*4  K,IFLAG,NX,NU,NF
      REAL*8 X(NX),U(NU),T,F(NF),L,DFDX(NF,NX),DLDX(NX),
     1     DFDU(NF,NU),DLDU(NU), D2FDX2(NF,NX,NX),
     2     D2LDX2(NX,NX),D2FDXU(NF,NX,NU),D2LDXU(NX,NU),
     3     D2FDU2(NF,NU,NU),D2LDU2(NU,NU)
```

Note that all of the above arrays must be adjustable-sized arrays in order for the program to work properly.

The input arguments for the subroutine are:

X — REAL*8 ARRAY containing the state vector where X(I) = x_i, the ith element of **x**.

U — REAL*8 ARRAY containing the control vector where U(I) = u_i, the ith element of **u**.

T — REAL*8 SCALAR containing the problem time, t. It is expressed in the same units and with the same origin as the T(0:N) input array defined in section 3 above.

K — INTEGER*4 SCALAR specifying the current stage number.

IFLAG — INTEGER*4 SCALAR flag specifying which outputs are needed:

IFLAG = 0: Only the functions, outputs F(:) and L, need be calculated on this call.

IFLAG = 1: Only the functions and their first partial derivatives, outputs F(:), L, DFDX(:,:), DLDX(:), DFDU(:,:), and DLDU(:), need be calculated on this call.

IFLAG = 2: The functions, their first partial derivatives, and their second partial derivatives, F(:), L, DFDX(:,:), DLDX(:), DFDU(:,:), DLDU(:), D2FDX2(:,:,:), D2LDX2(:,:), D2FDXU(:,:,:), D2LDXU(:,:), D2FDU2(:,:,:), and D2LDU2(:,:), need to be calculated on this call. Note that, if ISECGR(K) $\neq$ 0, then the subroutine FLFNCT will always be called with IFLAG = 0 or IFLAG = 1.

NX INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays.

NU INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays.

NF INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays. This must be allowed to be different than NX because of the possibility of having explicit discrete-time steps in which the dimension of the state vector changes during a state transition.

The output arguments for the subroutine are:

F REAL*8 ARRAY containing the $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$ function where F(I) = f_{k_i}, the ith element of $\mathbf{f}_k(\mathbf{x},\mathbf{u},t)$.

L REAL*8 SCALAR containing $L_k(\mathbf{x},\mathbf{u},t)$.

DFDX REAL*8 ARRAY containing $\partial \mathbf{f}_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}$ where DFDX(I,J) = $\partial f_{k_i}/\partial x_j$. This output needs to be computed only when IFLAG $\neq 0$.

DLDX REAL*8 ARRAY containing $\partial L_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}$ where DLDX(J) = $\partial L/\partial x_j$. This output needs to be computed only when IFLAG $\neq 0$.

DFDU REAL*8 ARRAY containing $\partial \mathbf{f}_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}$ where DFDU(I,J) = $\partial f_{k_i}/\partial u_j$. This output needs to be computed only when IFLAG $\neq 0$.

DLDU REAL*8 ARRAY containing $\partial L_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}$ where DLDU(J) = $\partial L/\partial u_j$. This output needs to be computed only when IFLAG $\neq 0$.

The following 6 optional output arguments need not be calculated if the user specifies the use of finite differencing to compute second gradients -- by setting ISECGR(K) $\neq 0$. In all cases, however, the names of these optional arguments must appear in the argument list.

D2FDX2 REAL*8 ARRAY containing $\partial^2 \mathbf{f}_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}^2$ where D2FDX2(I,J,M) = $\partial^2 f_{k_i}/\partial x_j \partial x_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2LDX2 REAL*8 ARRAY containing $\partial^2 L_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}^2$ where D2LDX2(J,M) = $\partial^2 L_k/\partial x_j \partial x_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2FDXU REAL*8 ARRAY containing $\partial^2 \mathbf{f}_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x} \partial \mathbf{u}$ where D2FDXU(I,J,M) = $\partial^2 f_{k_i}/\partial x_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2LDXU REAL*8 ARRAY containing $\partial^2 L_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x} \partial \mathbf{u}$ where D2LDXU(J,M) = $\partial^2 L_k/\partial x_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2FDU2 REAL*8 ARRAY containing $\partial^2 \mathbf{f}_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}^2$ where D2FDU2(I,J,M) = $\partial^2 f_{k_i}/\partial u_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2LDU2 REAL*8 ARRAY containing $\partial^2 L_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}^2$ where DL2DU2(J,M) = $\partial^2 L_k/\partial u_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

WARNINGS:

a) Never write to one of the input arguments. The optimization program that calls this subroutine assumes that FLFNCT does not change the value of any of the input arguments.

b) Never write to an adjustable-sized output array if it has any zero dimension, NU = 0 or NF = 0. The quantity NU will be equal to the value that the user has input to the main optimization routine as NUVEC(K). The quantity NF will be equal to the value that the user has input to the main optimization routine as NXVEC(K+1) except for stage N, in which case NF will be zero. As an example, if NU is zero, then the output arrays DFDU, DLDU, D2FDU2, D2LDU2, D2FDXU and D2LDXU will have no elements because one of their adjustable dimensions is zero. The main optimization program will not work in this case if FLFNCT tries to write to any of these arrays.

c) Never write to any of the partial derivative output arrays if IFLAG has not been set to call for that quantity to be output. That is, do not write to DFDX, DLDX, DFDU, or DLDU unless IFLAG $\geq$ 1, and do not write to D2FDX2, D2LDX2, D2FDXU, D2LDXU, D2FDU2, or D2LDU2 unless IFLAG $\geq$ 2.

d) The output value of L must be defined for every problem stage K. If stage K does not contribute to the cost, then the assignment statement L = 0 must be included in FLFNCT. Likewise, for the terminal stage where NF = 0, a value must be

assigned to the L output argument despite the fact that no values will be assigned to the output array F.

6. Specifications for the CFNCT SUBROUTINE

This is the dummy name of the subroutine that determines the auxiliary constraints at each stage. The user uses this subroutine to further model the problem. The constraints defined in this subroutine are always enforced at the initial time of the interval. The constraints are specified in terms of the elements of the $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ vector function:

$$c_{k_i}(\mathbf{x_k},\mathbf{u_k},t_k) = 0 \quad \text{for i = 1, ..., NECVEC(K)}$$

$$c_{k_i}(\mathbf{x_k},\mathbf{u_k},t_k) \le 0 \quad \text{for i = [NECVEC(K)+1], ..., [NECVEC(K)+NICVEC(K)]}$$

Note that CFNCT must be correctly programmed so that the first NECVEC(K) elements of $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ define the equality constraint functions for stage k, and the last NICVEC(K) elements of $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ define the inequality constraint functions for stage k.

The CFNCT subroutine must also calculate the first partial derivatives of $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ with respect to **x** and **u** when an input flag calls for these to be calculated. Optionally, the user can program the function to calculate the second partial derivatives of $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ with respect to **x** and **u** upon request as signalled by an input flag.

If CFNCT has not been programmed to calculate second derivatives for stage k, then the user must set ISECGR(K) to some other value than 0 so that the optimization algorithm will estimate the second derivatives via finite differencing. Also, values must specified for DELTU(1:NUVEC(K),K) and DELTX(1:NXVEC(K),K) if CFNCT does not calculate second derivatives for stage k.

The user is responsible for programming CFNCT correctly so that it correctly models each stage k. There may be stages where the length of the equality part of the constraint

vector is zero or where the length of the inequality part of the constraint vector is zero. The current stage number is provided as an input argument so that CFNCT can perform the necessary branching to change constraints if the constraints are different at different stages.

The CFNCT program must have the following argument list and dimensioning and data typing information:

```
      SUBROUTINE  CFNCT(X,U,T,K,IFLAG,NX,NU,NC,C,DCDX,DCDU,
     1     D2CDX2,D2CDXU,D2CDU2)
      INTEGER*4  K,IFLAG,NX,NU,NC
      REAL*8 X(NX),U(NU),T,C(NC),DCDX(NC,NX),DCDU(NC,NU),
     1     D2CDX2(NC,NX,NX),D2CDXU(NC,NX,NU),D2CDU2(NC,NU,NU)
```

Note, all argument arrays must be adjustable-sized arrays with the dimensions as defined above in order for the calling program to function properly.

The input arguments for the subroutine are:

X	REAL*8 ARRAY containing the state vector where X(I) = x_i, the ith element of **x**.
U	REAL*8 ARRAY containing the control vector where U(I) = u_i, the ith element of **u**.
T	REAL*8 SCALAR containing the problem time, t. It is expressed in the same units and with the same origin as the T(0:N) input array defined in section 3 above.
K	INTEGER*4 SCALAR specifying the current stage number.
IFLAG	INTEGER*4 SCALAR flag specifying which outputs are needed:
	IFLAG = 0: Only the function C(:) needs to be calculated on this call.

IFLAG = 1: Only the function and its first partial derivatives, outputs C(:), DCDX(:,:), and DCDU(:,:), need be calculated on this call.

IFLAG = 2: The function, its first partial derivatives, and its second partial derivatives, outputs C(:), DCDX(:,:), DCDU(:,:), D2CDX2(:,:,:), D2CDXU(:,:,:), and D2CDU2(:,:,:), need to be calculated on this call. Note that, if ISECGR(K) $\neq$ 0, then the subroutine CFNCT will always be called with IFLAG = 0 or IFLAG = 1.

NX INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays.

NU INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays.

NC INTEGER*4 SCALAR used to define the size of adjustable-sized dummy argument arrays.

The output arguments for the subroutine are:

C REAL*8 ARRAY containing the $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$ function where C(I) = c_{k_i}, the ith element of $\mathbf{c_k}(\mathbf{x},\mathbf{u},t)$.

DCDX REAL*8 ARRAY containing $\partial \mathbf{c_k}(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}$ where DCDX(I,J) = $\partial c_{k_i}/\partial x_j$. This output needs to be computed only when IFLAG $\neq$ 0.

DCDU REAL*8 ARRAY containing $\partial \mathbf{c_k}(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}$ where DCDU(I,J) = $\partial c_{k_i}/\partial u_j$. This output needs to be computed only when IFLAG $\neq$ 0.

The following 3 optional output arguments need not be calculated if the user specifies the use of finite differencing to compute second gradients -- by setting ISECGR(K) $\neq$ 0. In

all cases, however, the names of these optional arguments must appear in the argument list.

D2CDX2 REAL*8 ARRAY containing $\partial^2 c_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x}^2$ where D2CDX2(I,J,M) = $\partial^2 c_{k_i}/\partial x_j \partial x_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2CDXU REAL*8 ARRAY containing $\partial^2 c_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{x} \partial \mathbf{u}$ where D2CDXU(I,J,M) = $\partial^2 c_{k_i}/\partial x_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

D2CDU2 REAL*8 ARRAY containing $\partial^2 c_k(\mathbf{x},\mathbf{u},t)/\partial \mathbf{u}^2$ where D2CDU2(I,J,M) = $\partial^2 c_{k_i}/\partial u_j \partial u_m$. This output needs to be computed only when IFLAG is neither 0 nor 1.

WARNINGS:

a) Never write to one of the input arguments. The optimization program that calls this subroutine assumes that CFNCT does not change the value of any of the input arguments.

b) Never write to an adjustable-sized output array if any of its dimensions is NU and NU = 0 for the current call of CFNCT. (CFNCT will not be called if NC is zero.) The quantity NU will be equal to the value that the user has input to the main optimization routine as NUVEC(K). The quantity NC will be equal to the sum of two values that the user has input to the main optimization routine: NECVEC(K)+NICVEC(K). If NU is zero, then the output arrays DCDU, D2CDXU and D2CDU2 will have no elements because one of their adjustable dimensions is zero. The main optimization program will not work in this case if CFNCT tries to write to any of these arrays.

c) Never write to any of the partial derivative output arrays if IFLAG has not been set to call for that quantity to be output. That is, do not write to DCDX or DCDU

unless IFLAG ≥ 1, and do not write to D2CDX2, D2CDXU, or D2CDU2 unless IFLAG ≥ 2.

```
C ************************** MINTM32 SOURCE FILE **************************
C
C
C  THESE SUBROUTINES DEFINE THE DYNAMICS, COST, AND
C  CONSTRAINTS OF A 32-STAGE, ACCELERATION-LIMITED
C  MINIMUM-TIME-TO-THE-ORIGIN-IN-THE-PLANE
C  PROBLEM, WHICH IS A STANDARD PROBLEM IN THE OPTIMAL CONTROL
C  LITARATURE.  THE SOLUTION IS THE FAMEOUS LINEAR-TANGENT STEERING
C  CONTROL LAW.  THIS VERSION OF THE PROBLEM USES CONTROL RATES AS THE
C  CONTROL IN ORDER TO GET A PIECE-WISE LINEAR CONTROL.  THE 0TH STAGE IS
C  A DISCRETE-TIME STAGE THAT IS NECESSARY IN ORDER TO SET UP THE
C  THE CONTROL INITIAL CONDITION.  AN EXTRA CONTROL VARIABLE IS ADDED
C  TO STAGES 1 TO (N-1  IN ORDER TO ALLOW PROBLEM TIME TO BE DIFFERENT
C  FROM REAL TIME.  THE ACCELERATION MAGNITUDE IS LIMITED TO 1.
C
C
C
C  THIS CODE HAS BEEN WRITTEN FOR USE WITH THE PARALLEL TRAJECTORY OPTIMIZATION
C  PROGRAM THAT C. PARK HAS DEVELOPED FOR HIS PH.D. RESEARCH.
C
C  THESE SUBROUTINES WERE WRITTEN BY M.L. PSIAKI IN MARCH 1993.
C
      SUBROUTINE F_MNTM(X,U,T,A,IFLAG,NX,NU,NF,F,L,DFDX,DLDX,
     1                  DFDU,DLDU,D2FD2,D2LD2,D2FDXU,D2LDXU,D2FDU2,
     2                  D2LDU2)
      IMPLICIT REAL*8(A-H,O-Z)
      IMPLICIT INTEGER*4(I-N)
      PARAMETER (N=31)
      REAL*8 L
      DIMENSION X(NX),U(NU),F(NF),
     1          DFDX(NF,NX),DLDX(NX),DFDU(NF,NU),DLDU(NU),
     2          D2FD2(NF,NX,NX),D2LD2(NX,NX),D2FDXU(NF,NX,NU),
     3          D2LDXU(NX,NU),D2FDU2(NF,NU,NU),D2LDU2(NU,NU)
      SAVE
C
C  NOTE THE FOLLOWING DEFINITIONS APPLY TO STAGES 1 THROUGH (N-1)
C
C           X(1) = X1 POSITION
C           X(2) = X2 POSITION
C           X(3) = X1 VELOCITY
C           X(4) = X2 VELOCITY
C           X(5) = X1 ACCELERATION (USUALLY CALLED U1)
C           X(6) = X2 ACCELERATION (USUALLY CALLED U2)
C           X(7) = RATE OF PASSAGE OF REAL TIME (SEC) WITH RESPECT
C                  TO PROBLEM TIME (STAGES 1:N).
C
C           U(1) = RATE OF CHANGE OF X1 ACCELERATION
C           U(2) = RATE OF CHANGE OF X2 ACCELERATION
C           U(3) = RATE OF PASSAGE OF REAL TIME (SEC) WITH RESPECT
C                  TO PROBLEM TIME (STAGE 0 ONLY).
C
C  NOTE THAT THE PROBLEM TIME HAS A TERMINAL VALUE OF 1.
C
C
C  DO STAGE 1. ASSUME NF = 7, NX = 7, AND NU = 3.  THIS DISCRETE-TIME STAGE
C  SETS UP THE FIXED POSITION AND VELOCITY INITIAL CONDITIONS, AND IT
C  ALLOWS THE OPTIMIZATION ROUTINE TO SET UP OPTIMAL INITIAL ACCELERATIONS
C  AND THE TERMINAL PROBLEM TIME.
C
      IF (T.EQ.0) THEN
```

```
            L = 0.D+00
            F(1) = -73.605382551487390D+00
            F(2) = -35.125486154217420D+00
            F(3) =   2.949018340281470D+00
            F(4) =    .430989313478260D+00
            F(5) = U(1)
            F(6) = U(2)
            F(7) = U(3)
            IF (IFLAG.GE.1) THEN
               DO 20 II = 1,3
                  DLDU(II) = 0.D+00
                  DO 10 JJ = 1,7
                     DFDU(JJ,II) = 0.D+00
   10             CONTINUE
   20          CONTINUE
               DFDU(5,1) = 1.D+00
               DFDU(6,2) = 1.D+00
               DFDU(7,3) = 1.D+00
               IF (IFLAG.EQ.2) THEN
                  DO 50 II = 1,3
                     DO 40 JJ = 1,3
                        D2LDU2(JJ,II) = 0.D+00
                        DO 30 MM = 1,7
                           D2FDU2(MM,II,JJ) = 0.D+00
   30                   CONTINUE
   40                CONTINUE
   50             CONTINUE
               END IF
            END IF
C
C  DO STAGES 1 THROUGH (N-1). ASSUME NF = NX = 7 AND NU = 3.
C
         ELSE IF (K.LE.(N-1)) THEN
            L = X(7)
            F(1) = X(3)**X(7)
            F(2) = X(4)*X(7)
            F(3) = X(5)**X(7)
            F(4) = X(6)**X(7)
            F(5) = U(1)*X(7)
            F(6) = U(2)*X(7)
            F(7) = 1.D+00
            IF (IFLAG.GE.1) THEN
               DO 120 II = 1,7
                  DLDX(II) = 0.D+00
                  DO 100 JJ = 1,7
                     DFDX(II,JJ) = 0.D+00
  100             CONTINUE
                  DO 110 JJ = 1,3
                     DFDU(II,JJ) = 0.D+00
  110             CONTINUE
  120          CONTINUE
               DO 140 II = 1,3
                  DLDU(II) = 0.D+00
  140          CONTINUE
               DLDX(7) = 1.D+00
               DFDX(1,3) = X(7)
               DFDX(1,7) = X(3)
               DFDX(2,4) = X(7)
               DFDX(2,7) = X(4)
               DFDX(3,5) = X(7)
               DFDX(3,7) = X(5)
               DFDX(4,6) = X(7)
               DFDX(4,7) = X(6)
               DFDU(5,1) = X(7)
               DFDX(5,7) = U(1)
               DFDU(6,2) = X(7)
```

```
            DFDX(6,7) = U(2)
            IF (IFLAG.EQ.2) THEN
               DO 250 II = 1,7
                  DO 240 JJ = 1,7
                     D2LDX2(II,JJ) = 0.D+00
                     DO 230 MM = 1,7
                        D2FDX2(II,JJ,MM) = 0.D+00
230                  CONTINUE
                     DO 235 MM = 1,2
                        D2FDXU(II,JJ,MM) = 0.D+00
235                  CONTINUE
240               CONTINUE
                  DO 245 JJ = 1,2
                     D2LDXU(II,JJ) = 0.D+00
                     DO 243 MM = 1,2
                        D2FDU2(II,JJ,MM) = 0.D+00
243                  CONTINUE
245               CONTINUE
250            CONTINUE
               DO 270 II = 1,2
                  DO 260 JJ = 1,2
                     D2LDU2(II,JJ) = 0.D+00
260               CONTINUE
270            CONTINUE
               D2FDX2(1,3,7) = 1.D+00
               D2FDX2(1,7,3) = 1.D+00
               D2FDX2(2,4,7) = 1.D+00
               D2FDX2(2,7,4) = 1.D+00
               D2FDX2(3,5,7) = 1.D+00
               D2FDX2(3,7,5) = 1.D+00
               D2FDX2(4,6,7) = 1.D+00
               D2FDX2(4,7,6) = 1.D+00
               D2FDXU(5,7,1) = 1.D+00
               D2FDXU(6,7,2) = 1.D+00
            END IF
         END IF
C
C  DO STAGE N. ASSUME NF = 0, NX = 7, AND NU = 0.
C
      ELSE IF (I.EQ.N) THEN
         L = 0.D+00
         IF (IFLAG.GE.1) THEN
            DO 320 II = 1,7
               DLDX(II) = 0.D+00
320         CONTINUE
            IF (IFLAG.EQ.2) THEN
               DO 450 II = 1,7
                  DO 440 JJ = 1,7
                     D2LDX2(II,JJ) = 0.D+00
440               CONTINUE
450            CONTINUE
            END IF
         END IF
      END IF
      RETURN
      END
C
C
C  THIS SUBROUTINE DEFINES THE AUXILIARY CONSTRAINTS FOR THE 32-STAGE
C  MINIMUM-TIME-TO-THE-ORIGIN PROBLEM.
C
      SUBROUTINE CMNTN(X,U,T,L,IFLAG,NX,NU,NC,C,DCDX,
     1                 DCDU,D2CDX2,D2CDXU,D2CDU2)
      IMPLICIT REAL*8(A-H,O-Z)
      IMPLICIT INTEGER*4(I-N)
      PARAMETER  N=31
```

```
      DIMENSION X(NX),U(NU),C(NC),
     1          DCDX(NC,NX),DCDU(NC,NU),D2CDX2(NC,NX,NX),
     2          D2CDXU(NC,NX,NU),D2CDU2(NC,NU,NU)
      SAVE
C
C  FOR THE ZEROTH STAGE, WHICH ONLY SETS UP THE INITIAL ACCELERATION,
C  THERE IS ONLY AN AUXILIARY INEQUALITY CONSTRAINT ON THE ACCEL.
C  MAGNITUDE.  ASSUME NC = 2, NX = 0, AND NU = 3
C
      IF (I.EQ.0) THEN
         C(1) = U(1)**2 + U(2)**2 - 1.D+00
         C(2) = - U(3) + .01D+00
         IF (IFLAG.GE.1) THEN
            DCDU(1,1) = 2.D+00*U(1)
            DCDU(1,2) = 2.D+00*U(2)
            DCDU(1,3) = 0.D+00
            DCDU(2,1) = 0.D+00
            DCDU(2,2) = 0.D+00
            DCDU(2,3) = -1.D+00
            IF  IFLAG.EQ.2) THEN
               DO 30 II = 1,3
                  DO 20 JJ = 1,3
                     DO 10 KK = 1,3
                        D2CDU2(II,JJ,KK) = 0.D+00
   10                CONTINUE
   20             CONTINUE
   30          CONTINUE
               D2CDU2(1,1,1) = 2.D+00
               D2CDU2(1,2,2) = 2.D+00
            END IF
         END IF
C
C  DO STAGES 1 THROUGH (N-1). ASSUME NC = 2, NX = 7, AND NU = 2.
C  IN ADDITION TO LIMITING THE ACCELERATION MAGNITUDE, LIMIT THE
C  AMOUNT OF REAL TIME IN THE ENTIRE TRAJECTORY TO NO LESS THAN .01 SEC.
C
      ELSE IF (I.LE.(N-1)) THEN
         C(1) = X(5)**2 + X(6)**2 - 1.D+00
         C(2) = - X(7) + .01D+00
         IF (IFLAG.GE.1) THEN
            DO 120 II = 1,2
               DO 100 JJ = 1,7
                  DCDX(II,JJ) = 0.D+00
  100          CONTINUE
               DO 110 JJ = 1,2
                  DCDU(II,JJ) = 0.D+00
  110          CONTINUE
  120       CONTINUE
            DCDX(1,5) = 2.D+00*X(5)
            DCDX(1,6) = 2.D+00*X(6)
            DCDX(2,7) = -1.D+00
            IF (IFLAG.EQ.2) THEN
               DO 250 II = 1,2
                  DO 240 JJ = 1,7
                     DO 230 MM = 1,7
                        D2CDX2(II,JJ,MM) = 0.D+00
  230                CONTINUE
                     DO 235 MM = 1,2
                        D2CDXU(II,JJ,MM) = 0.D+00
  235                CONTINUE
  240             CONTINUE
                  DO 245 JJ = 1,2
                     DO 243 MM = 1,2
                        D2CDU2(II,JJ,MM) = 0.D+00
  243                CONTINUE
  245             CONTINUE
```

```
  250             CONTINUE
                  D2CDX2(1,5,5) = 2.D+00
                  D2CDX2(1,6,6) = 2.D+00
               END IF
            END IF
C
C  DO STAGE 1.  ASSUME NC = 5, NX = 7, AND NU = 0.
C  NOTE THAT THIS IS THE ONLY STAGE WITH EQUALITY CONSTRAINTS.
C  THE FIRST 4 CONSTRAINTS ARE EQUALITY CONSTRAINTS.
C
         ELSE IF (I.EQ.1) THEN
            C(1) = X(1)
            C(2) = X(2)
            C(3) = X(3)
            C(4) = X(4)
            C(5) = X(5)**2 + X(6)**2 - 1.D+00
            IF (IFLAG.GE.1) THEN
               DO 320 II = 1,5
                  DO 300 JJ = 1,7
                     DCDX(II,JJ) = 0.D+00
  300             CONTINUE
  320          CONTINUE
               DCDX(1,1) = 1.D+00
               DCDX(2,2) = 1.D+00
               DCDX(3,3) = 1.D+00
               DCDX(4,4) = 1.D+00
               DCDX(5,5) = 2.D+00*X(5)
               DCDX(5,6) = 2.D+00*X(6)
               IF (IFLAG.EQ.2) THEN
                  DO 350 II = 1,5
                     DO 340 JJ = 1,7
                        DO 330 MM = 1,7
                           D2CDX2(II,JJ,MM) = 0.D+00
  330                   CONTINUE
  340                CONTINUE
  350             CONTINUE
                  D2CDX2(5,5,5) = 2.D+00
                  D2CDX2(5,6,6) = 2.D+00
               END IF
            END IF
         END IF
         RETURN
         END
C
C  THIS SUBROUTINE INITIALIZES VECTORS NEEDED FOR PARIS MAIN OPTIMIZATION
C  ALGORITHM.  NOTE THAT THIS IS FOR A 32-STAGE PROBLEM.  STAGES 0 THROUGH 31.
C  IN OTHER WORDS N = 31 IS ASSUMED.
C
C  ONE CAN EASILY SWITCH THESE THREE SUBROUTINES TO USE A DIFFERENT NUMBER
C  OF PARAMETERS BY SIMPLY CHANGING N IN THE THREE PARAMETER STATEMENTS
C  THAT OCCUR NEAR THE BEGINNING OF EACH OF THE THREE SUBROUTINES.
C
C  THIS SUBROUTINE SHOULD GET CALLED BY THE MAIN PROGRAM BEFORE IT
C  CALLS THE OPTIMIZATION ALGORITHM.  THE MAIN PROGRAM MUST BE
C  SURE TO MATCH DIMENSIONS WITH THE FIXED-DIMENSIONING INFORMATION
C  OF THIS SUBROUTINE'S ARRAY ARGUMENTS.  ALL OF THE ARGUMENTS OF
C  THIS SUBROUTINE ARE OUTPUT ARGUMENTS.
C
C  NOTE THAT THE MAIN PROGRAM WILL HAVE TO INCLUDE THE STATEMENT
C
C        EXTERNAL FLMNTM,CMNTM
C
C  IN ORDER TO REFER TO THE ABOVE TWO PROBLEM MODELLING FUNCTIONS.
C  THESE TWO FUNCTIONS REPLACE THE DUMMY FUNCTIONS FLFNCT AND CFNCT
C  AS INPUTS FROM THE MAIN PROGRAM TO THE MAIN OPTIMIZATION SUBROUTINE.
C
```

```
      SUBROUTINE MNTMIN(NUVEC,NXVEC,NECVEC,NICVEC,NUMAX,NXMAX,NCMAX,
     1                  U0,X0,IDT,T,KRK,ISECGR,DELTU,DELTX)
      IMPLICIT REAL*8(A-H,O-Z)
      IMPLICIT INTEGER*4(I-N)
      PARAMETER  N=3,
      DIMENSION NUVEC(0:N),NXVEC(0:N),NECVEC(0:N),NICVEC(0:N),
     1          U0(1:3,0:N),X0(1:7,0:N),IDT(0:N),T(0:N),
     2          KRK(1:2,0:(N-1)),ISECGR(0:N ,DELTU(1:3,0:N),
     3          DELT(1:7,0:N)
      SAVE
C
C  INITIALIZE DIMENSIONING INFORMATION AND OTHER PROBLEM MODELLING
C  AND CONTROL INFORMATION.
C
      NUVEC(0) = 3
      NXVEC(0) = 0
      NECVEC(0) = 0
      NICVEC(0) = 2
      IDT(0) = 1
      T(0) = 0.D+00
      DELT = 1.D+00/DBLE(FLOAT(N-1))
      TDUM = 0.D+00
      KRK(1,0) = 0
      KRK(2,0) = 0
      ISECGR(0) = 0
      DO 10 I = 1,(N-1)
         NUVEC(I) = 3
         NXVEC(I) = 7
         NECVEC(I) = 0
         NICVEC(I) = 2
         IDT(I) = 0
         T(I) = TDUM
         TDUM = TDUM + DELT
         KRK(1,I) = 1
         KRK(2,I) = 1
         ISECGR(I) = 0
   10 CONTINUE
      NUVEC(N) = 0
      NXVEC(N) = 7
      NECVEC(N) = 4
      NICVEC(N) = 1
      IDT(N) = 1
      T(N) = TDUM
      ISECGR(N) = 1
      NUMAX = 3
      NXMAX = 7
      NCMAX = 5
C
C  NOTE, THERE IS REALLY NO NEED TO SPECIFY THE STATE'S INITIAL CONDITION:
C  NXVEC(0) = 0.  THE SUBROUTINE FLMNTM ESSENTIALLY SPECIFIES THE STATE'S
C  INITIAL CONDITION.
C
C
C  INITIALIZE GUESS OF THE SOLUTION.  THIS IS A POOR
C  FIRST GUESS.   IT ONLY BRINGS THE VELOCITY TO ZERO.
C  IT DOES NOT BRING THE POSITION TO ZERO.
C
      X0(1,1) = -72.6053825514873D+00
      X0(2,1) = -35.1254861542174D+00
      X0(3,1) =   2.94901834028147D+00
      X0(4,1) =    .430989313473826D+00
      U3GUESS = DSQRT(X0(3,1)**2 + X0(4,1)**2)
      U1GUESS = - X0(3,1)/U3GUESS
      U2GUESS = - X0(4,1)/U3GUESS
      X0(5,1) =   U1GUESS
      X0(6,1) =   U2GUESS
```

```
      X0(7,1) =   U3GUESS
      U0(1,0) = U1GUESS
      U0(2,0) = U2GUESS
      U0(3,0) = U3GUESS
      U0(1,1) = 0.D+00
      U0(2,1) = 0.D+00
      DTREAL = DELT*U3GUESS
      TREAL = 0.D+00
      DO 30 KK = 2,N
         IF (KK.LT.N) THEN
            U0(1,KK) = 0.D+00
            U0(2,KK) = 0.D+00
         END IF
         TREAL = TREAL + DTREAL
         X0(1,KK) = X0(1,1) + TREAL*X0(3,1  + .5D+0 *(TREAL**2)*
     1                        X0(5,1)
         X0(2,KK) = X0(2,1) + TREAL*X0(4,1) + .5D+0 *(TREAL**2)*
     1                        X0(6,1)
         X0(3,KK) = X0(3,1) + TREAL*X0(5,1)
         X0(4,KK) = X0(4,1) + TREAL*X0(6,1)
         X0(5,KK) = X0(5,1)
         X0(6,KK) = X0(6,1)
         X0(7,KK) = X0(7,1)
   30 CONTINUE
C
C  INITIALIZE DELTX AND DELTU FOR THE SAKE OF CHECKING DERIVATIVES
C
      DO 60 KK = 0,N
         DO 40 JJ = 1,7
            DELTX(JJ,KK) = .0001D+00
   40    CONTINUE
         DO 50 JJ = 1,3
            DELTU(JJ,KK) = .0001D+00
   50    CONTINUE
   60 CONTINUE
C
C  NORMAL POINT OF RETURN TO THE CALLING MAIN PROGRAM.
C
      RETURN
      END
```

Appendix E

A multi-dimensional cubic spline formlation.

A Multi-Dimensional Cubic Spline Formulation

This appendix describes a relatively simple way to generate a cubic spline interpolation of a single dependent variable as a function of multiple independent variables. It works when the data is laid out on a rectangular gird in the space of the independent variables. The grid spacing for any one variable need not be uniform. This is useful to trajectory optimization because cubic splines are continuous with continuous first and second partial derivatives. Continuity of a function and its first and second partial derivatives is a must to ensure convergence of most second-order optimization algorithms. The present spline procedure is intended to offer rapid on-line computation of the spline function and its partial derivatives given that some off-line calculations of spline coefficients have been performed.

The mathematical problem at hand is to generate a function $y(x_1,x_2,x_3, ..., x_n)$ given the values of y at gird points. Suppose that the rectangular gird points are $(x1_{j_1},x2_{j_2},x3_{j_3}, ...,xn_{j_n})$ where $j_i = 1, ..., k_i$, for all $i = 1, ..., n$. Suppose, also, that $x_{i_j} < x_{i_{j+1}}$ for all $i = 1, ..., n$ and $j = 1, ..., (k_i-1)$, and define $\Delta x_{i_j} \equiv x_{i_{j+1}} - x_{i_j}$. Note that the values of Δx_{i_j} may vary with the index j for fixed i, but they will all be positive. Let the data for the function values at the grid points be denoted as

$$y_{j1,j2,j3,...,jn} = y(x1_{j_1},x2_{j_2},x3_{j_3}, ...,xn_{j_n}) \qquad \text{(E-1)}$$

A cubic interpolation function can be developed that requires only the following data at the grid points: y, $\partial y/\partial x_i$ for $i = 1, ..., n$, $\partial^2 y/\partial x_i \partial x_j$ for $i,j = 1,..., n$, $i \neq j$, $\partial^3 y/\partial x_i \partial x_j \partial x_k$ for $i,j,k = 1,..., n$, $i \neq j \neq k$, ..., $\partial^n y/\partial x_1 \partial x_2 \partial x_3 ... \partial x_n$. In other words, the functions value is needed along with all of the function's partial derivatives that differentiate at most one time with respect to any given independent variable. For $n = 1$ independent variable, the required data is y and $\partial y/\partial x_1$ at each grid point. For $n = 2$ independent variables, the required data is y, $\partial y/\partial x_1$, $\partial y/\partial x_2$, and $\partial^2 y/\partial x_1 \partial x_2$ at each grid point. For $n = 3$ independent variables, the required data is y, $\partial y/\partial x_1$, $\partial y/\partial x_2$, $\partial y/\partial x_3$, $\partial^2 y/\partial x_1 \partial x_2$, $\partial^2 y/\partial x_1 \partial x_3$, $\partial^2 y/\partial x_2 \partial x_3$, and $\partial^3 y/\partial x_1 \partial x_2 \partial x_3$ at each grid point, etc.

The cubic interpolation function uses 4 special basis functions

$$g_1(x) = (x\text{-}1)^2(2x + 1) \tag{E-2a}$$

$$g_2(x) = x^2(-2x + 3) \tag{E-2b}$$

$$g_3(x) = (x\text{-}1)^2x \tag{E-2c}$$

$$g_4(x) = (x\text{-}1)x^2 \tag{E-2d}$$

These functions have the special properties: $g_1(0) = 1$ and $g_1(1) = dg_1/dx|_0 = dg_1/dx|_1 = 0$; $g_2(1) = 1$ and $g_2(0) = dg_2/dx|_0 = dg_2/dx|_1 = 0$; $dg_3/dx|_0 = 1$ and $g_3(0) = g_3(1) = dg_3/dx|_1 = 0$; $dg_4/dx|_1 = 1$ and $g_4(0) = g_4(1) = dg_4/dx|_0 = 0$. These special properties permit the writing of a cubic interpolation directly as a weighted sum of products of these functions, and the weighting factors are simply the function and its partial derivatives at the node points.

To see how the function $y(x_1,x_2,x_3, ..., x_n)$ is evaluated it is best to give the explicit formula for the 2-dimensional cases because a general formula applicable to all dimensions gets very complicated. After presenting the two-dimensional formula, the generalization to higher dimensions can be discussed.

For the 2-dimensional case, suppose that point (x_1,x_2) falls in the gird box (j_1,j_2). That is, suppose that $x_{i_{j_i}} \le x_i \le x_{i_{j_i+1}}$ for i = 1,2. Also, define $x_i \equiv (x_i - x_{i_{j_i}})/\Delta x_{i_{j_i}}$ for i = 1,2. Then the cubic interpolation formula is

$$\begin{aligned}
y(x_1,x_2) = {} & y_{j_1,j_2}\, g_1(x_1)g_1(x_2) + y_{j_1+1,j_2}\, g_2(x_1)g_1(x_2) \\
& + y_{j_1,j_2+1}\, g_1(x_1)g_2(x_2) + y_{j_1+1,j_2+1}\, g_2(x_1)g_2(x_2) \\
& + \left.\frac{\partial y}{\partial x_1}\right|_{j_1,j_2} \Delta x_{1_{j_1}}\, g_3(x_1)g_1(x_2) + \left.\frac{\partial y}{\partial x_1}\right|_{j_1+1,j_2} \Delta x_{1_{j_1}}\, g_4(x_1)g_1(x_2) \\
& + \left.\frac{\partial y}{\partial x_1}\right|_{j_1,j_2+1} \Delta x_{1_{j_1}}\, g_3(x_1)g_2(x_2) + \left.\frac{\partial y}{\partial x_1}\right|_{j_1+1,j_2+1} \Delta x_{1_{j_1}}\, g_4(x_1)g_2(x_2) \\
& + \left.\frac{\partial y}{\partial x_2}\right|_{j_1,j_2} \Delta x_{2_{j_2}}\, g_1(x_1)g_3(x_2) + \left.\frac{\partial y}{\partial x_2}\right|_{j_1+1,j_2} \Delta x_{2_{j_2}}\, g_2(x_1)g_3(x_2) \\
& + \left.\frac{\partial y}{\partial x_2}\right|_{j_1,j_2+1} \Delta x_{2_{j_2}}\, g_1(x_1)g_4(x_2) + \left.\frac{\partial y}{\partial x_2}\right|_{j_1+1,j_2+1} \Delta x_{2_{j_2}}\, g_2(x_1)g_4(x_2) \\
& + \left.\frac{\partial^2 y}{\partial x_1 \partial x_2}\right|_{j_1,j_2} \Delta x_{1_{j_1}} \Delta x_{2_{j_2}}\, g_3(x_1)g_3(x_2) \\
& + \left.\frac{\partial^2 y}{\partial x_1 \partial x_2}\right|_{j_1+1,j_2} \Delta x_{1_{j_1}} \Delta x_{2_{j_2}}\, g_4(x_1)g_3(x_2)
\end{aligned}$$

$$+ \left.\frac{\partial^2 y}{\partial x_1 \partial x_2}\right|_{j1,j2+1} \Delta x1_{j1} \Delta x2_{j2}\, g3(x_1) g4(x_2)$$

$$+ \left.\frac{\partial^2 y}{\partial x_1 \partial x_2}\right|_{j1+1,j2+1} \Delta x1_{j1} \Delta x2_{j2}\, g4(x_1) g4(x_2) \qquad \text{(E-3)}$$

Based on the properties of the $g_i(x)$ functions and the definition of x_i for i = 1,2, it is straight-forward to confirm that $y(x_1,x_2)$ and its partial derivatives take on the values assigned to these quantities at the grid points. It is also straight-forward to prove that the $y(x_1,x_2)$ cubic interpolation function and its first partial derivatives are continuous everywhere, even at the boundaries between grid boxes.

To understand how the formula would generalize to n independent variables, consider the individual terms in eq. (E-3). First notice that there are 16 terms. There are 4^n terms in the general expression for an n-dimensional spline. Each term is of the form $C \cdot g_{i_1}(x_1) \cdot g_{i_2}(x_2) \cdot g_{i_3}(x_3) \cdot \ldots \cdot g_{i_n}(x_n)$ for $i_1, i_2, i_3, \ldots, i_n = 1,2,3,4$. The constant C is a function of the indices $i_1, i_2, i_3, \ldots, i_n$. In general, if $i_k = 1$ or 2, then the expression for C will not include partial differentiation with respect to x_k, but if $i_k = 3$ or 4, then the expression for C will include the operator $(\Delta x_{k_{jk}} \frac{\partial}{\partial x_k})$ operating on y. Additionally, if $i_k = 1$ or 3, then C will be a value associated with a node with the index j_k, but if $i_k = 2$ or 4, then C will be a value associated with a node with the index j_k+1. These rules completely determine the function.

The partial derivative data values at the grid points can be chosen arbitrarily and the function and its first derivatives will be continuous, but the second derivatives will normally be discontinuous at the gird box boundaries.

With care, the grid-point partial derivative values can be chosen to yield continuity of the second derivatives also. A 2-dimensional example is convenient for understanding how to do this. Suppose that $\partial y/\partial x_1$ is assigned arbitrarily only at the extreme grid points in the x_1 direction, at grid points (x_{1_1},x_{2_1}), (x_{1_1},x_{2_2}), (x_{1_1},x_{2_3}), ..., $(x_{1_1},x_{2_{k_2}})$ and at $(x_{1_{k_1}},x_{2_1})$, $(x_{1_{k_1}},x_{2_2})$, $(x_{1_{k_1}},x_{2_3})$, ..., $(x_{1_{k_1}},x_{2_{k_2}})$. Suppose also that $\partial y/\partial x_2$ is assigned arbitrarily only at the extreme grid points in the x_2 direction, at grid points (x_{1_1},x_{2_1}), (x_{1_2},x_{2_1}), (x_{1_3},x_{2_1}), ..., $(x_{1_{k_1}},x_{2_1})$ and at $(x_{1_1},x_{2_{k_2}})$, $(x_{1_2},x_{2_{k_2}})$, $(x_{1_3},x_{2_{k_2}})$, ..., $(x_{1_{k_1}},x_{2_{k_2}})$. Also assume that $\partial^2 y/\partial x_1 \partial x_2$ is arbitrarily

defined only at the 4 extreme corner points of the 2-dimensional grid, $(x1_1,x2_1)$, $(x1_{k_1},x2_1)$, $(x1_1,x2_{k_2})$, and $(x1_{k_1},x2_{k_2})$. This situation is illustrated in Fig. E-1.

These boundary partial derivatives can be used to define the interior point partial derivatives through a sequence of 1-dimensional splines. Suppose that z(w) is a function of just one independent variable, w. Suppose also that z is specified on a 1-dimensional w grid that consists of k points: $z_1 = z(w_1)$, $z_2 = z(w_2)$, $z_3 = z(w_3)$, ..., $z_k = z(w_k)$. Then, knowledge of $\partial z/\partial w|_1$ and $\partial z/\partial w|_k$ is sufficient to uniquely determine a cubic spline that passes through all of the data points and whose second derivatives are everywhere continuous. This is a standard result of 1-dimensional spline theory [6]. This standard cubic spline has well-defined first derivatives at the interior grid points, $\partial z/\partial w|_2$, $\partial z/\partial w|_3$, $\partial z/\partial w|_4$, ..., $\partial z/\partial w|_{k-1}$. These are exactly the remaining data that would be needed to implement the 1-dimensional counterpart to eq. (E-3).

For the 2-dimensional case this process is generalized to determine the required interior-point partial derivatives. First, the values of $\partial y/\partial x_1$ at interior grid points can be determined by applying the 1-dimensional technique for computing interior derivatives to each set of grid points with a fixed value of x_2. This is done k_2 times, once for each for $x_2 \in \{x2_1, x2_2, x2_3, ..., x2_{k_2}\}$. In a similar fashion, the values of $\partial y/\partial x_2$ at interior grid points can be determined by applying the 1-dimensional technique k_1 times, once for each $x_1 \in \{x1_1, x1_2, x1_3, ..., x1_{k_1}\}$. At the end of these two sets of 1-dimensional spline operations, $\partial y/\partial x_1$ and $\partial y/\partial x_2$ are available at every grid point, but $\partial^2 y/\partial x_1 \partial x_2$ is still available only at the 4 corner grid points.

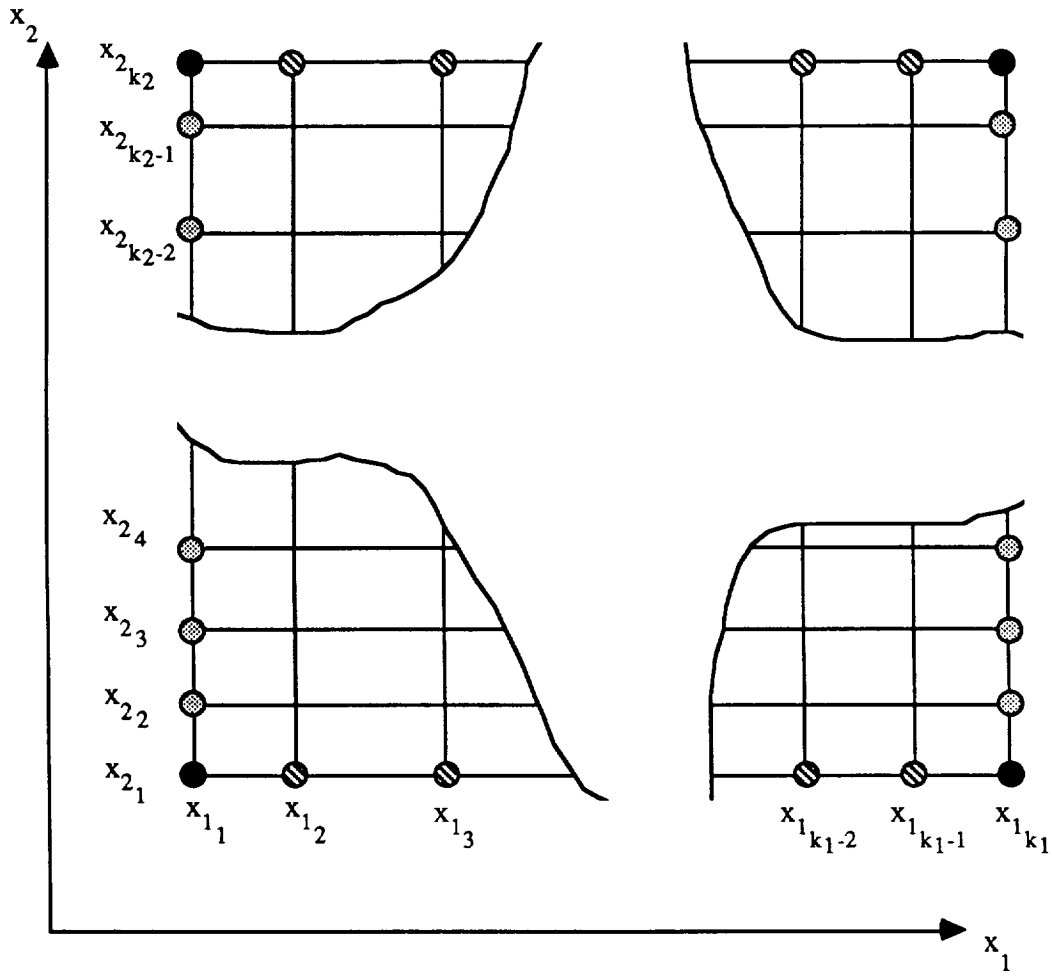
The first step in determining $\partial^2 y/\partial x_1 \partial x_2$ at the interior points is to determine it at the $x1_1$ and $x1_{k_1}$ extremities of the grid. This can be done by using $z = \partial y/\partial x_1$ and $w = x_2$ in a 1-dimensional spline operation. For the two cases, $x_1 = x1_1$ and $x_1 = x1_{k_1}$, z is available at all of the w grid points ($w = x2_1, x2_2, x2_3, ..., x2_{k_2}$) and $\partial z/\partial w$ is available at the w endpoints ($w = x2_1$ and $x2_{k_2}$). Therefore, the spline operation can be used to determine $\partial z/\partial w$ at the interior w points ($w = x2_2$, $x2_3$, ..., $x2_{k_2-1}$, but $\partial z/\partial w = \partial^2 y/\partial x_1 \partial x_2$ according to the current definition of z; so, at the end of this operation $\partial^2 y/\partial x_1 \partial x_2$ will be available at the points marked on Fig. E-2.

The final step in determining $\partial^2 y/\partial x_1 \partial x_2$ at the interior points is to perform a series of 1-dimensional splines of $z = \partial y/\partial x_2$ in the $w = x_1$ direction, one for each grid value of x_2. This can be done because in each case, z is available at all of the interior points and $\partial z/\partial w$ is available at the w end points. The resulting values of $\partial z/\partial w$ at the interior points are just the interior values of $\partial^2 y/\partial x_1 \partial x_2$ that are needed in eq. (E-3) to compute $y(x_1,x_2)$. Note that the roles of x_1 and x_2 can be reversed in the process of determining the interior-point values of $\partial^2 y/\partial x_1 \partial x_2$ without affecting the final results.

All of the 1-dimensional spline operations that are needed to get the interior partial derivatives can be computed off-line, and the resulting grid-point partial derivatives can be stored for rapid on-line evaluation of $y(x_1,x_2)$ via eq (E-3). The total number of off-line 1-dimensional splines required to compute all of the interior-point partial derivatives is $2 + k_1 + 2k_2$, which is relatively inexpensive.

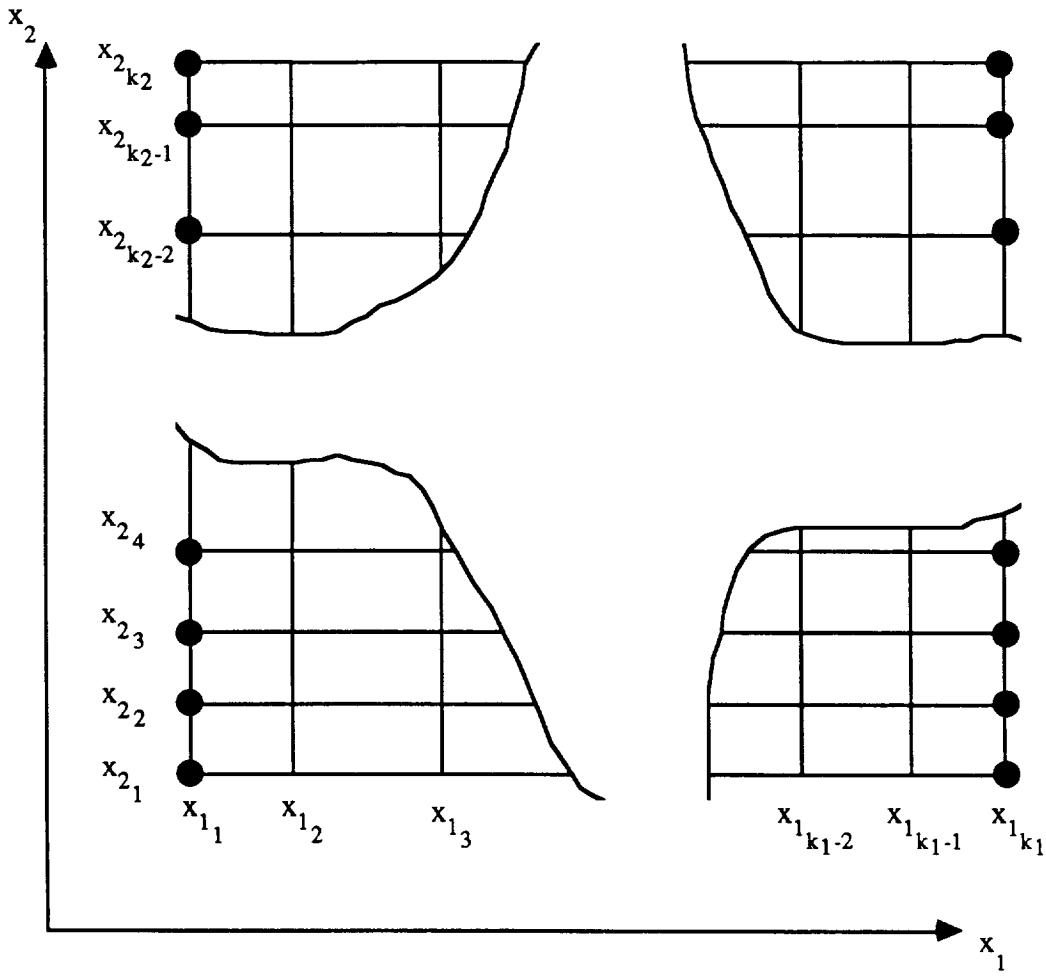
The boundary-point partial derivatives defined on Fig. E-1 are still arbitrarily selectable. In practice, the function $y(x_1,x_2)$ is insensitive to the selected boundary partial derivatives at points far in the interior of the grid. In grid boxes near the boundary the function becomes sensitive to these arbitrary quantities. A useful approach to selecting these quantities is to compute finite difference approximations of these quantities near the boundary and extrapolate the required quantities to the boundary. The required finite-difference approximations can be computed from the y data values at the grid points. Figure E-3 illustrates the interior points at which various finite-difference partial derivative approximations get computed in order to extrapolate a particular partial derivative to a particular boundary point. This method of determining the arbitrary boundary partial derivatives has been used in all the calculations of this report.

There is a straight-forward, but tedious generalization of this method to the problem of computing interior-point partial derivatives for higher-dimensional splines. In the interests of brevity, higher-dimensional generalizations have been omitted.



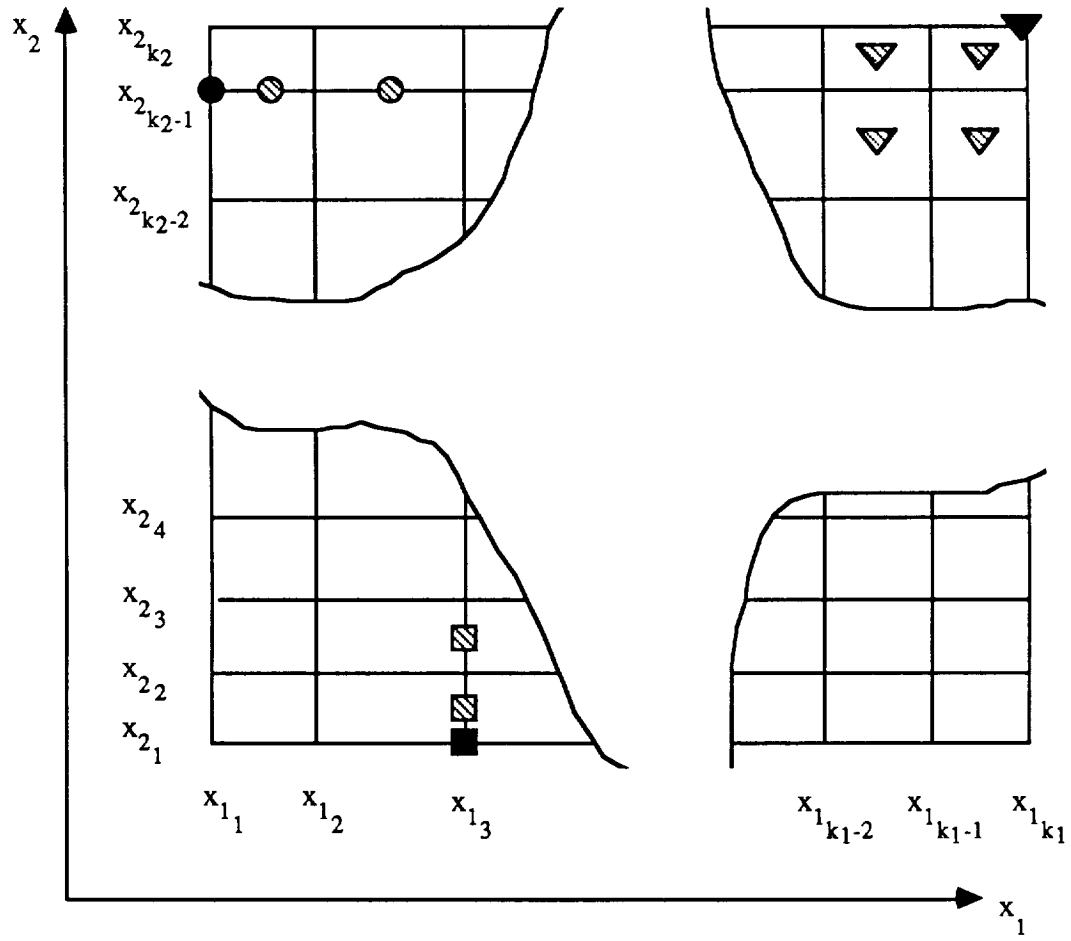
- Points at which $\frac{\partial y}{\partial x_1}$ is arbitrarily defined
- Points at which $\frac{\partial y}{\partial x_2}$ is arbitrarily defined
- Points at which $\frac{\partial y}{\partial x_1}$, $\frac{\partial y}{\partial x_2}$, and $\frac{\partial^2 y}{\partial x_1 \partial x_2}$ are arbitrarily defined

Fig. E-1. Boundary points on a 2-dimensional spline's grid at which partial derivatives may be assigned arbitrarily.



● Points at which $\frac{\partial^2 y}{\partial x_1 \partial x_2}$ is available after splining of $\frac{\partial y}{\partial x_1}$ along $x_1 = x_{1_1}$ and along $x_1 = x_{1_{k1}}$

Fig. E-2. Propagation of the second cross partial derivatives along two edges of the grid from the four corners.



● Example boundary point where $\partial y/\partial x_1$ is calculated by linear extrapolation from two interior points

⊘ Corresponding interior points where finite-difference values of $\partial y/\partial x_1$ are available

■ Example boundary point where $\partial y/\partial x_2$ is calculated by linear extrapolation from two interior points

▨ Corresponding interior points where finite-difference values of $\partial y/\partial x_2$ are available

▼ Example boundary point where $\partial^2 y/\partial x_1 \partial x_2$ is calculated by bi-linear extrapolation from four interior points

▽ Corresponding interior points where finite-difference values of $\partial^2 y/\partial x_1 \partial x_2$ are available

Fig. E-3. Examples of how partial derivatives on the boundary are extrapolated from finite-difference interior values.