SAND91-8230 • UC-405
Unlimited Release
Reprinted February 1996

The Twopnt Program for Boundary Value Problems

Version 3.10 of March 1992

J. F. Grcar

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94551
for the United States Department of Energy
under Contract DE-ACO4-94AL85000

SAND91-8230 UC-405
Unlimited Release
Printed April 1992
Reprinted February 1996

The Twopnt Program for Boundary Value Problems

Version 3.10 of March 1992

Joseph F. Grcar

Sandia National Laboratories
Livermore, California USA 94551-0969
(510) 294-2662
na.grcar@na-net.ornl.gov
sepp@snll-arpagw.llnl.gov

Abstract

Twopnt is a computer program that finds steady state solutions for systems of differential equations, usually, in one space dimension. This is a guide to the program's use and a description of its operation.

Acknowledgements

I thank Dr. R. J. Kee for encouraging this work and awaiting its completion. Thanks to him and to Prof. M. D. Smooke for advice in writing the program, to Dr. J. A. Miller and F. M. Rupley for enduring many errors, to Dr. D. S. Dandy and Dr. G. H. Evans for the example in Chapter 7, and to F. M. Rupley for the data in Appendix 1. I thank Dr. W. G. Houf for reading the preliminary manuscript. And thanks to Prof. D. E. Knuth for $\TeX$ and to Blue Sky Research for Macintosh *Textures*.

Contents

1 Introduction

When computers were invented, John von Neumann foresaw computer simulations would have four indelible errors [26]. One is arithmetic precision. Results can't be more accurate than machines allow, and they could be much worse if errors compound perversely. Two is approximation. Mathematical expressions must be made amenable to machine calculation, and unending searches for results must be terminated. Three is the model. It may simplify the laws of science, which simplify the laws of nature. Four is measurement. Physical constants must be ascertained experimentally.

Twopnt (pronounced *two point*) is a computer program that finds steady-state solutions for systems of differential equations, usually, in one space dimension. Those equations are called two point boundary value problems, whence the program's name. Twopnt addresses von Neumann's point two. It works inside simulation programs—like "original equipment" that secondary manufacturers interconnect to build complete systems. Twopnt now supports roughly a dozen reacting flow simulations; some are in daily use throughout the world.

This publication describes how to use a simulation based on twopnt. Twopnt runs automatically, generally, and reports its progress in printed output described here. Simulators running twopnt provide access to controls for twopnt's operation. Those controls are described here too. Often, only the control for "turning off" printing is needed. If a simulation appears to be inefficient however, or if it fails, then investigating twopnt's printed output may suggest better control settings.

This publication also describes how to write a new simulation program using twopnt. Twopnt's advantages are its independence from specific applications, and its ability to solve otherwise intractable problems. Among these are combustion simulations rich in chemical detail [22]. Twopnt owes much to collaboration with R. J. Kee, J. A. Miller and M. D. Smooke; the first and still the most extensive use of twopnt is in the premix simulator for laminar flames [21]. The success of that simulation prompted the strict segregation of twopnt from the rest of the simulator to facilitate twopnt's migration to other simulations.

Twopnt is both an algorithm and a computer program. The algorithm has $\star$ "*controls*" (for simulation users) and the program has $\bullet$ arguments (for simulation writers). All are marked and cross-referenced as follows.

⋆	`ADAPT`	perform grid selection (control)	16, 34
⋆	`PMAX`	maximum points in any grid (control)	18, 34 (● 63, 68)
●	`PMAX`	maximum points in any grid (argument)	63, 68 (⋆ 18, 34)
○	`PIVOT`	pivoting data (argument, but not twopnt's)	

The cross-references make it easy to look things up, but they shouldn't be pursued when reading straight through.

This manual is organized as follows. Chapter 2 describes what twopnt can do and the controls that change what twopnt does. Chapter 3 explains twopnt's printed output. Chapter 4 lists twopnt's controls and offers advice for setting them. Chapter 5 lists twopnt's error messages. Chapters 6 and 7 explain how to write simulation programs using twopnt. Appendix 1 cross-references twopnt's controls with the "keywords" that manipulate the controls in simulators written at Sandia National Laboratories in Livermore, California. Appendix 2 contains some programming notes. Appendix 3 records changes made to twopnt.

This publication has diamond lanes, |◇|, not dangerous bends. They are easier than N. Bourbaki's curves (which were popularized by D. E. Knuth), but read carefully just the same. Correspondence on this manual and on twopnt is welcome. Correspondence on von Neumann's points three and four should be directed to specific simulators' authors.

When a problem in pure or applied mathematics is "solved" by numerical computation, errors, that is, deviations of the numerical "solution" obtained from the true, rigorous one, are unavoidable. Such a "solution" is therefore meaningless, unless there is an estimate of the total error in the above sense.

Such estimates have to be obtained by a combination of several different methods, because the errors that are involved are aggregates of several different kinds of contributory, primary errors. These primary errors are so different from each other in their origin and character, that the methods by which they have to be estimated must differ widely from each other. A discussion of the subject may, therefore, advantageously begin with an analysis of the main kinds of primary errors, or rather, of the sources from which they spring.

This analysis of the sources of errors should be objective and strict inasmuch as completeness is concerned, but when it comes to the defining, classifying, and separating of the sources, a certain subjectiveness and arbitrariness is unavoidable. With these reservations, the following enumeration and classification of sources of errors seems to be adequate and reasonable.

— J. von Neumann and H. H. Goldstine [26]

2 What Twopnt Does

2.0 Introduction

A simulation is built in three steps. Each step incurs some of von Neumann's four simulation errors, see **Figure 2.1** and Chapter 1. The first step is a scientific model. Twopnt leaves that to scientists, and concerns itself with the rest. So far as twopnt is concerned, a simulation is a system of equations to be solved. To this end, twopnt performs three interrelated tasks

- grid selection
- Newton's search
- time evolution

which are introduced here together and then discussed separately in subsections.

Twopnt does not require the simulation have spatial dependence, but since this is often the case, twopnt provides special facilities for two point boundary value problems. Twopnt never sees the differential equations, however, because twopnt expects the simulator it serves to *"discretize"* the equations beforehand. Since twopnt has little to do with this aspect of the simulation, it can solve problems with more than one spatial dimension, or with none.

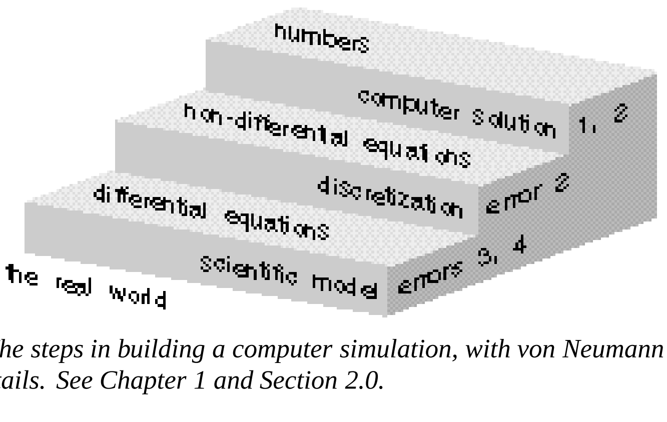

Figure 2.1 *The steps in building a computer simulation, with von Neumann's errors which each step entails. See Chapter 1 and Section 2.0.*

For two point boundary value problems, twopnt expects the simulator to provide a *"grid"* of p numbers

$$x_1 \qquad x_2 \qquad \cdots \qquad x_p$$

and at each of these *"grid points,"* twopnt seeks values for c solution *"components."*

$$\begin{array}{cccc} u_{1,1} & u_{1,2} & \cdots & u_{1,p} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ u_{c,1} & u_{c,2} & \cdots & u_{c,p} \end{array}$$

These values are unknown at the outset and must be determined. There may be many unknowns since $c = 50$ and $p = 100$ are common in reacting flow simulations. For example, $u_{k,n}$ might be the concentration of the k-th chemical species at the distance x_n from a nozzle. $|\diamondsuit|$ Twopnt also may have some unknowns associated with *"boundary conditions"* at the ends of the grid.

For two point boundary value problems, the equations solved by twopnt may arise in the following way. The values at all grid points for each component

$$u_{k,1} \qquad u_{k,2} \qquad \cdots \qquad u_{k,p}$$

constitute a discrete sampling of a quantity, u_k, that varies continuously in space. From this discrete sample, approximations can be built for the differentials of u_k. For example, if x is the spatial variable, then

$$\begin{aligned} \left.\frac{du_k}{dx}\right|_{x=x_n} &\approx \frac{u_{k,n+1} - u_{k,n}}{x_{n+1} - x_n} \\ \text{or } &\approx \frac{u_{k,n} - u_{k,n-1}}{x_n - x_{n-1}} \\ \text{or perhaps } &\approx \frac{\Delta_n^2 u_{k,n+1} + (\Delta_{n+1}^2 - \Delta_n^2) u_{k,n} - \Delta_{n+1}^2 u_{k,n-1}}{\Delta_n \Delta_{n+1} (\Delta_n + \Delta_{n+1})} \\ &\text{with } \Delta_i = x_i - x_{i-1} \end{aligned}$$

depending, of course, on the simulator's choice. *"Discretizing"* the differential equations may amount to replacing the equation's differentials by the approximations above. More elaborate discretizations are possible, for example, reacting flow simulations might have equations representing mass conservation for each species, k, in control volumes around each grid point, n. By these or other means, the unknowns $u_{k,n}$ become variables in nondifferential equations.

For differential equations, the values sought for the unknowns are not exactly correct, in the following sense. The solution values for each component

$$u_{k,1} \qquad u_{k,2} \qquad \cdots \qquad u_{k,p}$$

sample only approximately the continuously varying quantity u_k in the differential equations.

$$\begin{aligned} u_{k,n} &\approx u_k(x_n) \\ \text{not } u_{k,n} &= u_k(x_n) \end{aligned}$$

These errors are the first aspect of John von Neumann's approximation error (mentioned in Chapter 1). They occur because the differentials are only approximated, and moreover, only at a few points. Twopnt can add points to better the approximation. The solution values found for one grid suggest additional points for a next grid, and so on. In combustion simulations, M. D. Smooke [31] demonstrated that successive enlargements, beginning from grids too coarse to afford much resolution, lead to good placement of efficiently few points. Thus, twopnt has facilities to solve nondifferential equations repeatedly, for the unknowns associated with successive grids.

(The chief numerical issues in designing a simulation are the discretization and solution of the differential equations. The best discretization is the prevailing one in the subject at hand, one that has been discovered and improved by trial and error. The mathematical theory of solving two-point boundary value problems also offers guidance, but may include recommendations not tempered by experience. For example, the text [2] reduces equations to "standard first order form" and solves them by "shooting." In reacting flow problems the system is usually too large to reduce the order, and further, the resulting equations are usually too sensitive to solve by shooting.)

Twopnt expects all the equations it solves to have been "brought to one side" so they can be written symbolically as

$$f(v) = 0$$

in which v is the vector of all unknowns and $f(v)$ is the vector of all equations. If arbitrary numbers are chosen for the unknowns, as when $v_\bullet$ is a collection of "good guesses," then the equations likely will not be solved—which is to say the vector of *"residuals"* $f(v_\bullet)$ formed by evaluating the *"residual function"* f likely will not vanish.

$$f(v_\bullet) \neq 0$$

Twopnt seeks values, $v_\star$, with zero residuals, $f(v_\star) = 0$.

(These equations are nonlinear, hence difficult to solve. "Functional iteration," "nonlinear Gauss-Seidel," "operator split," and "uncoupled" methods determine values for some unknowns by assuming values for others, in round-robin fashion. This process must be repeated many times to obtain self-consistent solutions, and the details vary with the subject matter. For example, the "simpler" method and the Gummel method both solve convection-diffusion equations, but for fluids and semiconductors, respectively. Since twopnt is intended for a variety of applications, it employs a solution process for "fully coupled" equations.)

Twopnt solves the equations by successive approximation using a variant of *"Newton's method"* [11] [28]. From a reasonable guess v_0, Sir Issac Newton's namesake constructs an unending sequence of improving approximations

$$v_0 \qquad v_1 \qquad \cdots \qquad v_n \qquad \cdots$$

that converge on a vector $v_\star$ with $f(v_\star) = 0$. After reasonably many steps, twopnt accepts some "approximate solution" v_n in lieu of $v_\star$, and this v_n becomes the "result" of the simulation. Accepting v_n in place of $v_\star$ is the second aspect of John von Neumann's approximation error.

The chief difficulty with Newton's method is that even a well chosen v_0 needn't spawn a convergent sequence of approximate solutions. In this case twopnt tempers the guess by evolving it in time. When Newton's method goes astray, twopnt begins building a different sequence

$$v_\bullet = v_{(0)} \qquad v_{(1)} \qquad v_{(2)} \qquad \cdots \qquad v_{(m)} \qquad \cdots$$

which samples the time evolution of the simulated phenomena at grid points in time.

$$0 = t_0 \qquad t_1 \qquad t_2 \qquad \cdots \qquad t_m \qquad \cdots$$

The time dependent states eventually converge on the steady state too, but twopnt stops far short of that. Early transient states usually are sufficiently "better guesses" that Newton's method, beginning from them, can find the steady state directly. In combustion simulations, J. F. Grcar and company [15] found the systematic use of this shortcut efficient.

The time dependent states are obtained by solving time dependent rather than steady state equations. Simulated phenomena invariably have a temporal aspect, so the simulation's equations naturally include time derivatives. Indeed, the steady state formulation simply assumes those derivatives vanish. Thus, the time dependent residual functions, ${}^{\mathrm{td}}f_{(m)}$, are just more elaborate versions of the steady state residual function, which is henceforth named ${}^{\mathrm{ss}}f$. In particular, the time dependent equations must incorporate values from previous time points to permit building approximations for the time differentials, hence the inclusion of a time point index, m, in the notation ${}^{\mathrm{td}}f_{(m)}$.

(Many pioneering steady state simulations were performed entirely by time evolution. Consult [31] for references to early simulations in combustion chemistry. This practice continues because some time discretizations, "explicit" ones, lend themselves to uncoupled solution methods. Their advantages are ease of programming and economy of computer memory; disadvantages are choosing where to stop and taking too long to get there. Since twopnt already expects to use Newton's fast method for the steady state, when transient states are needed, it uses "implicit" time discretizations.)

In summary, twopnt uses three devices to solve what could be hard problems. For two point boundary value problems, twopnt undertakes calculations on small grids to determine good point locations and good solution guesses for large grids. Moreover, twopnt improves solution guesses by time evolution when necessary. Finally, twopnt solves equations by Newton's rapidly converging method. The following sections explain these three tasks and how to control them.

2.1 What Newton's Search Does

Twopnt solves nonlinear equations

$$ {}^{\mathrm{ss}}f(v) = 0 \quad \text{and} \quad {}^{\mathrm{td}}f_{(m)}(v) = 0 $$

for the steady and time dependent states, respectively. Twopnt has a control that omits searches for the steady state if desired.

⋆ `STEADY`	search for the steady state (yes or no)	37

This control can avoid exhaustive searches whose results might be unnecessarily accurate, for example, as preliminary estimates for subsequent simulations. Other controls can prevent searches for transient states, see Section 2.2.

Whatever the equations' origin, Newton's method [11] [28] solves nonlinear equations $f(v) = 0$ by successive approximations

$$ v_0 \qquad v_1 \qquad \cdots \qquad v_n \qquad \cdots $$

formed by repeated iterations

$$ s_n = -J^{-1} f(v_n) $$

that is, given v_n solve $J s_n = -f(v_n)$ for s_n, and then

$$ v_{n+1} = v_n + s_n $$

in which J is the Jacobian matrix of f with respect to v evaluated at v_n. The i-th column of J contains the derivatives of all the residual functions with respect to the i-th unknown. The linear part of Taylor's series representation for f

$$ f(v) \approx f(v_n) + J(v - v_n) $$

suggests $f(v) \approx 0$ when $v = v_n - J^{-1}(f(v_n))$ which is Newton's choice for v_{n+1}.

The expense of matrix evaluation is usually so great a *"modified"* Newton's method is used which retains the Jacobian matrix through several steps. Twopnt has controls that retire the matrix after a specified number of steps (Newton's *"pure"* method retires the matrix immediately).

⋆ SSAGE	steady state Jacobian's retirement age	37
⋆ TDAGE	time dependent Jacobian's retirement age	37

In the steady state case the first Jacobian matrix is evaluated at v_0, so with SSAGE $= 5$, the others are at v_5, v_{10} and so on. The time dependent case is the same. If successive intervals in the time grid are the same length, however, then twopnt retains the Jacobian matrix from one Newton search to the next. |◇| In the time dependent case, the Jacobian matrix of ${}^{\mathrm{td}}f_{(1)}$ is evaluated at $v_0 := v_{(0)}$ first, and with TDAGE $= 5$, at v_5 next. If v_9 is accepted as the time dependent state $v_{(1)}$, then the search for $v_{(2)}$ begins from the guess $v_0 := v_{(1)}$ with a matrix of age 4. Thus, the Jacobian matrix of ${}^{\mathrm{td}}f_{(2)}$ is evaluated at v_1, v_6 and so on.

The selection of an acceptable approximate solution v_n depends on the size of the correction, s_n. The residual $f(v_n)$ itself is ignored because the relationship between good approximations and small residuals is difficult to quantify, though in fact, the residual for the chosen v_n is much nearer 0 than the residual for the guess v_0. Proofs that Newton's modified method converges suggest the error $v_\star - v_n$ decreases at least geometrically, which implies s_n decreases geometrically too. Presumably, if $s_{i,n}$ is small relative to $v_{i,n}$, then the error $v_{i,\star} - v_{i,n}$ should be of the same relative magnitude (the notation here means $s_{i,n}$ is the i-th entry in the vector s_n, and similarly for v_n and $v_\star$). Twopnt has controls for relative accuracy measured in this indirect way.

⋆ SSREL	steady state relative convergence test's bound	35
⋆ TDREL	time dependent relative convergence test's bound	35

In many simulations some quantities are vanishingly small. In reacting flows for example, reactants and products generally do not coexist. Since approximating zero to high relative accuracy is meaningless, twopnt also has controls for absolute accuracy.

⋆ SSABS	steady state absolute convergence test's bound	35
⋆ TDABS	time dependent absolute convergence test's bound	35

These controls can be interpreted as meaning v_n is chosen as the approximate solution provided each entry in v_n has error either absolutely or relatively small. Twopnt's precise criteria for stopping at v_n is

$$|s_{i,n}| \;\le \begin{cases} \text{either ABS, or} \\ \text{REL} \times |v_{i,n}| \end{cases}$$

in which i indexes all the entries in the vectors, that is, all the unknowns.

|◇| Actually, twopnt applies the convergence criteria to $(v_n + s_n) - v_n$ rather than to s_n itself, for the following reason. If s_n is very small relative to v_n, then arithmetic rounding could make v_{n+1} identical to v_n, and then further searching is pointless. In this case, $(v_n+s_n)-v_n$ probably equals 0 not s_n, so the convergence criteria with $(v_n+s_n)-v_n$ in place of s_n will stop the search.

|◇| |◇| Twopnt shouldn't be asked to find a v_n with an s_n insignificantly small. Usually, rounding makes f, J and s_n difficult to evaluate with precision near $v_\star$, so Newton's search wanders aimlessly near its goal. Thus with ABS $= 0$ and REL $= 0$ two things can happen. (1) Newton's search ends successfully at some v_n the machine can't improve and which

is presumably almost $v_\star$. More likely, (2) the convergence monitors, below, end Newton's search in failure.

All proofs of convergence for Newton's method derive from L. Kantorovich's famous one [28, p. 428] which has stringent assumptions difficult to verify. Implementations of Newton's method therefore must take care all goes well and make adjustments when not. The simplest precaution confines the search to safe or desired territory. That is, the equations $f(v) = 0$ may have multiple solutions with some not wanted, or the residual function f may be impossible to evaluate for some v. For example, chemical species must not have negative densities. Twopnt therefore has controls that limit the acceptable values for each unknown.

⋆ `ABOVE(`i`)`	upper limit for the i-th unknown	33 (● 62, 66)
⋆ `BELOW(`i`)`	lower limit for the i-th unknown	33 (● 62, 66)

Twopnt expects the initial guess, v_0, obeys the bounds

$$\texttt{BELOW(}i\texttt{)} \le v_{i,0} \le \texttt{ABOVE(}i\texttt{)}$$

and it ensures the subsequent solution estimates obey the bounds too, in the following way. Twopnt evaluates

$$\begin{aligned} v_{n+1} &= v_n + \delta_B s_n \\ \text{in place of} \quad v_{n+1} &= v_n + s_n \end{aligned}$$

where $0 < \delta_B \le 1$ is the largest number that keeps v_{n+1} within bounds. |◊| Three things can happen. (1) If v_n lies on a boundary and s_n points outside, then Newton's method is judged a failure unless an old Jacobian has been used and a new one turns s_n around. (2) If Newton fails to find the steady state, then twopnt begins a time evolution—from v_0 where Newton started, not v_n where Newton failed. (3) If Newton fails to find a time dependent state, then refer to Section 2.2.

Twopnt also uses the following check for progress. Given v_n and a likely v_{n+1}, twopnt evaluates what would be the next step, s_{n+1}.* Twopnt accepts v_{n+1} provided the potential next step is shorter than the present step, that is, provided

$$\|s_{n+1}\|_\infty \le \|s_n\|_\infty$$

in which $\|\cdot\|_\infty$ is the maximum-magnitude norm (the max or infinity norm). If not shorter, then twopnt retreats from v_{n+1}, evaluating

$$\begin{aligned} v_{n+1} &= v_n + \delta_D \delta_B s_n \\ \text{in place of} \quad v_{n+1} &= v_n + \delta_B s_n \end{aligned}$$

where the *"damping factor"* δ_D decreases geometrically.

$$\delta_D = 2^{-0.5}, \quad 2^{-1.0}, \quad \ldots, \quad 2^{-2.5}$$

|◊| Three things can happen. (1) If no v_{n+1} passes the test above, then Newton's method is judged a failure unless an old Jacobian has been used and a new one produces better s_n, v_{n+1} and s_{n+1}. (2) If Newton fails to find the steady state, then twopnt begins a time evolution—from v_0 where Newton started, not v_n where Newton failed. (3) If Newton fails to find a time dependent state, then refer to Section 2.2.

* |◊| For economy, the same Jacobian matrix produces both s_n and s_{n+1}. If a Jacobian evaluation is not scheduled at v_{n+1}, then this s_{n+1} would be the next step.

2.2 What Time Evolution Does

Twopnt undertakes a transient evolution with states

$$v_\bullet = v_{(0)} \qquad v_{(1)} \qquad v_{(2)} \qquad \cdots \qquad v_{(m)} \qquad \cdots$$

at successive points in time

$$0 = t_0 \qquad t_1 \qquad t_2 \qquad \cdots \qquad t_m \qquad \cdots$$

solely to replace the starting guess in Newton's search for the steady state by something better, namely $v_{(m)}$ for some m. Time evolution begins at the guess $v_\bullet$ supplied by the simulator or suggested by the previous spatial grid—see Section 2.3. Twopnt obtains the time evolution by solving the time-dependent equations

$$^{\mathrm{td}}f_{(m)}(v_{(m)}) = 0$$

for successive m by Newton's method, beginning the search for $v_{(m)}$ at the "guess" $v_{(m-1)}$. These searches can fail, too.

Twopnt can be directed to pass time even before the search for the steady state begins and whenever Newton's method fails thereafter.

⋆ `STEPS0`	time steps before first steady state search	38
⋆ `STEPS1`	time steps upon failing to find a steady state	38

Thus, with `STEPS0` $= 100$ and `STEPS1` $= 50$, Newton's search for a steady state begins at $v_{(100)}$, and if unsuccessful begins again at $v_{(150)}$, and if still unsuccessful begins again at $v_{(200)}$, and so on without limit.

The initial time evolution, `STEPS0` > 0, has two uses. When the solution guess is poor and time evolution is needed, then `STEPS0` > 0 avoids one fruitless search for the steady state. When there are multiple steady states and Newton's search might find a nonphysical one, then `STEPS0` > 0 moves the solution toward a stable state.

A well developed branch of numerical analysis chooses time points t_0, t_1, t_2, $\ldots$ to follow transient evolutions accurately [4] [18]. Twopnt does not use the methods developed there, for two reasons. First, accurate transient simulations may incur computing expenses incidental to twopnt's goal. Second, the proper choice of time grids varies with the discretization of the time differentials. As a convenience to the simulators it serves, twopnt leaves the choice of time discretizations open.

Twopnt chooses the steps between the time points as follows. Twopnt takes many steps of the same length,

⋆ `STRID0`	initial stride between time points	38

and expecting longer steps to approach the steady state in fewer steps, twopnt periodically lengthens its stride, but only so far.

⋆ `STEPS2`	time steps before increasing the stride	38
⋆ `TINC`	multiplier by which to increase the stride	38
⋆ `TMAX`	maximum stride	38

Conversely, twopnt shortens its stride in the following situations. If Newton's method cannot solve the transient equations for a newly increased stride, then twopnt reverts to the previous stride. If Newton's method cannot solve the transient equations for a not-newly-increased stride, then twopnt shortens its stride, but again only so far.

⋆ `TDEC`	divisor by which to decrease the stride	38
⋆ `TMIN`	minimum stride	38

For example, with

$$\texttt{STEPS2} = 75 \quad \texttt{STRID0} = 10^{-6} \quad \texttt{TINC} = 10 \quad \texttt{TDEC} = 3.16$$

the first seventy-five steps have stride 10^{-6}, and the next have stride $10^{-6} \times 10 = 10^{-5}$. If Newton's method initially fails in the search for $v_{(137)}$, then twopnt shortens the stride to $10^{-5} \div 3.16 = 10^{-5.5}$ and tries again. If Newton's method repeatedly fails despite shortening the stride to `TMIN`, then twopnt abandons the time evolution and begins one last search for the steady state from the last available time dependent state, $v_{(136)}$. If that search fails, then twopnt fails.

2.3 What Grid Selection Does

Twopnt constructs a succession of finer grids with more points for two point boundary value problems, when so directed.

⋆ `ADAPT`	perform grid selection (yes or no)	34

Twopnt must be given an initial grid, and when it finds a solution there, twopnt selects another grid, and so on.

⋆ `X`	the grid	40 (● 58, 63, 71)
⋆ `POINTS`	points (in the initial grid)	35 (● 58, 62, 68)

Each grid's solution is used to suggest the next grid, and moreover, is used to provide a guess for the next solution. As the grids grow, the solutions of the discrete equations better approximate the solution of the differential equations. Since the approximations improve from grid to grid, so the solutions differ less, and thus the guesses improve too. *Solving the equations therefore becomes easier although the grids become larger.* The use of successively chosen grids to improve solution accuracy is well established and difficult to credit, but the use also to improve solution efficiency apparently originated with M. D. Smooke [31] [32].

$|\diamondsuit|$ In multiple simulations such as parameter studies, large grids may result if twopnt adds points to successive simulation grids. Twopnt has no means to remove points, so some simulators may have their own.

Twopnt constructs the next grid by adding points to the present grid

$$\begin{array}{cccc} x_1 & x_2 & \cdots & x_p \end{array}$$

based on examining some components of the solution.

$$\begin{array}{cccc} u_{1,1} & u_{1,2} & \cdots & u_{1,p} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ u_{c,1} & u_{c,2} & \cdots & u_{c,p} \end{array}$$

An examined component

$$\begin{array}{cccc} u_{k,1} & u_{k,2} & \cdots & u_{k,p} \end{array}$$

should approximate a quantity that varies continuously in space.

$$u_{k,n} \approx u_k(x_n)$$

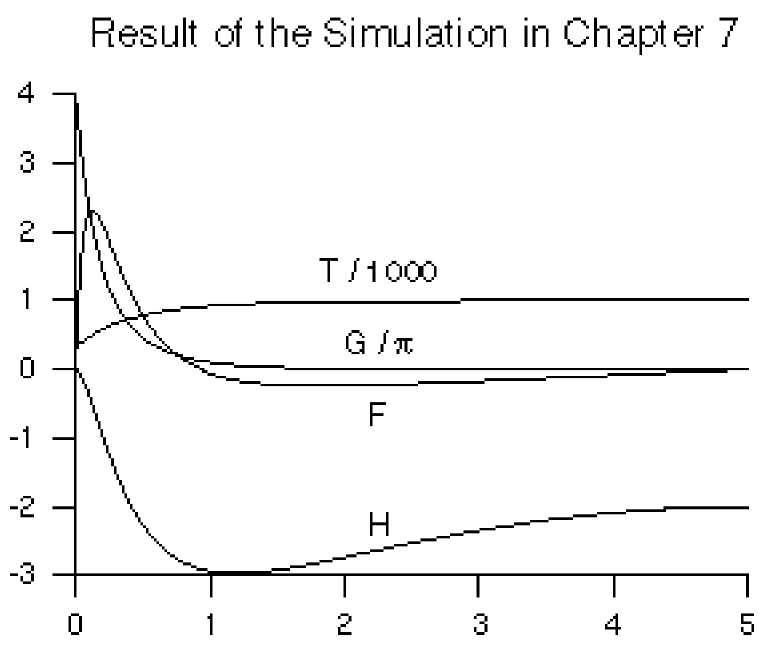

Figure 2.2 *Four solution components of the example in Chapter 7 found by twopnt with* `TOLER1` $= 0.1$ *and* `TOLER2` $= 0.1$. *Figures 2.3.1 and 2.3.2 enlarge a portion of the* F *curve for close viewing. Section 2.3 discusses this graph.*

When graphed, the pairs $(x_n, u_{k,n})$ approximate the graph of $u_k(x)$.

Twopnt has two controls that choose the solution components it examines. The first control is additive: it selects components outright. The second control is subtractive: it eliminates seemingly insignificant components.

⋆	`ACTIVE(`k`)`	use component k for grid selection (yes or no)	33 (● 63, 66)
⋆	`TOLER0`	absolute and relative significance floor	39

The k-th component is judged "insignificant" if its variation (maximum less minimum) is either absolutely small or small relative to its magnitude.

$$\max_n u_{k,n} - \min_n u_{k,n} \ \le \begin{cases} \text{either } \texttt{TOLER0}\text{, or} \\ \texttt{TOLER0} \times \max_n |u_{k,n}| \end{cases}$$

Twopnt seeks two features in each graph it examines. First, the variation over each grid interval should be less than a controlled fraction of the variation over all grid intervals.

$$\texttt{RATIO1}_{k,n} := \frac{|u_{k,n} - u_{k,n-1}|}{\max_n u_{k,n} - \min_n u_{k,n}} \le \texttt{TOLER1}$$

⋆	`TOLER1`	bound on change of value in each interval	39

If this condition fails, then twopnt "marks" for attention the interval between x_{n-1} and x_n. These marks ensure the grid concentrates points where the solution changes abruptly, for example, in fronts. $|\diamondsuit|$ The marking process for each feature in the graph is more or less independent of any other feature because only the maximum variation (not the total variation) occurs in the ratio's denominator. For example, if a graph has many identical ups and downs in succession, then twopnt marks the same intervals for subdivision in each front, no matter how many fronts there may be.

The second feature twopnt seeks is, the variation in angle of slope at each interior grid point should be less than a controlled fraction of the variation over all grid points.

$$\theta_{k,n} = \text{atan}\left(\frac{u_{k,n+1} - u_{k,n}}{x_{n+1} - x_n}\right)$$

$$\texttt{RATIO2}_{k,n} \; := \; \frac{|\,\theta_{k,n} - \theta_{k,n-1}|}{\max_n \theta_{k,n} - \min_n \theta_{k,n}} \; \leq \; \texttt{TOLER2}$$

⋆ `TOLER2`	bound on change of angle at each point	39

If this condition fails, then twopnt marks for attention the intervals on either side of x_n. These marks ensure the grid concentrates points where the solution's slope changes abruptly, for example, at peaks and troughs. Note the angles' sensitivity to scaling. Twopnt omits this portion of its examination if the k-th component's angles are judged insignificant, as follows.

$$\max_n \theta_{k,n} - \min_n \theta_{k,n} \; \leq \begin{cases} \text{either } \texttt{TOLER0}\text{, or} \\ \texttt{TOLER0} \times \max\limits_n |\theta_{k,n}| \end{cases}$$

The complementary effects of `TOLER1` and `TOLER2` are illustrated by some solutions obtained for Chapter 7's example. **Figure 2.2** shows four of the simulation's five solution components that have been obtained with both controls set at 0.1. **Figure 2.3.1** shows part of this solution and part of another that has `TOLER2` $= 1$. Only `TOLER1` governs the other solution's grid, and clearly, `TOLER1` resolves fronts but not peaks or troughs. Conversely, **Figure 2.3.2** shows part of the reference solution and part of another that has `TOLER1` $= 1$. Only `TOLER2` governs this other solution's grid, and clearly, `TOLER2` resolves peaks or troughs but not fronts.

In general, `TOLER1` and `TOLER2` place grid points where the first derivative and the curvature, respectively, have large magnitudes. Usually, `TOLER1` controls accuracy and `TOLER2` controls appearance. In Figures 2.3.1 and 2.3.2 for example, `TOLER1` places a coarse peak at the correct altitude, while `TOLER2` places a smooth peak at the wrong altitude. Since twopnt uses the same grid for all solution components, simulations with overlapping regions of peaks and fronts may muddy the distinction between these controls.

Twopnt constructs its grids, as follows. After examining each component and marking the grid intervals according to `TOLER1` and `TOLER2`, three things can happen. (1) Twopnt constructs a new grid by halving marked intervals. Since the equations for a grid with many new points may be difficult to solve, twopnt has a control to limit the introduction of new points, and of course, a control to limit the total number of points.

⋆ `PADD`	maximum points added to any one grid	34
⋆ `PMAX`	maximum points in the grid	34 (● 63, 68)

When more intervals are marked than points can be added, twopnt halves the most frequently marked intervals. (2) If some intervals have been marked but none can be halved, because either `PADD` $= 0$ or `POINTS` $=$ `PMAX`, then twopnt fails. (3) If no intervals have been marked, then twopnt supposes the boundary value problem has been solved.

$|\Diamond|$ The features that twopnt seeks in the graphs constitute twopnt's measure of discretization accuracy. This measure is necessarily superficial because twopnt does not choose the discretization. Twopnt's grid selection process can be interpreted geometrically as above, or in terms of discretization error as follows. "Anonymous" discretizations of complicated boundary value problems likely have truncation errors dependent on low derivatives of the analytic solution functions, u_k. Twopnt's first criterion for rejecting intervals approximates

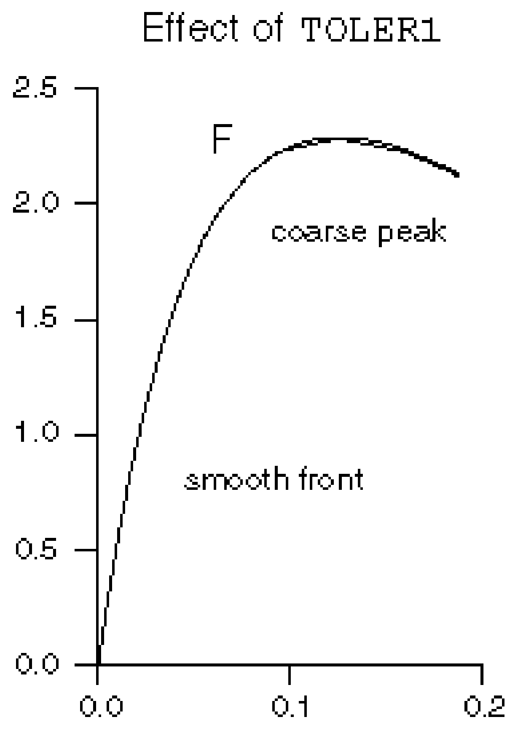

Figure 2.3.1 *Close view of Figure 2.2 (smooth curve) together with a solution found by twopnt for identical controls except* `TOLER2` $= 1$ *(rough curve). Alone,* `TOLER1` *resolves the front, but not the peak. Section 2.3 discusses this graph.*

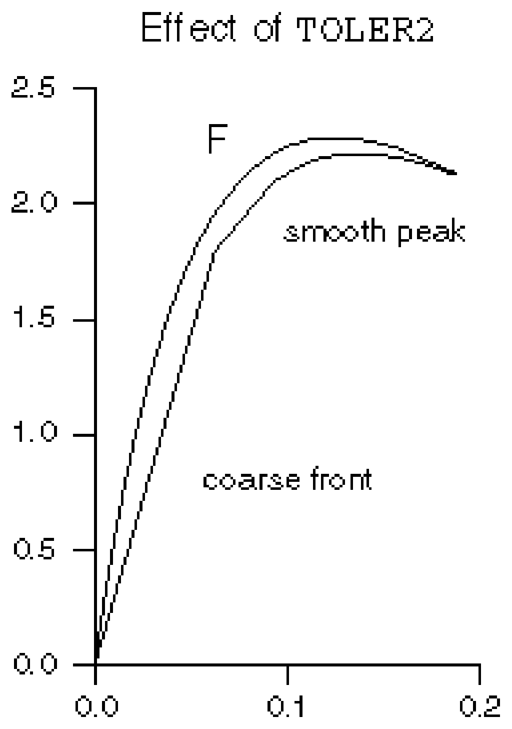

Figure 2.3.2 *Close view of Figure 2.2 (smooth curve) together with a solution found by twopnt for identical controls except* `TOLER1` $= 1$ *(rough curve). Alone,* `TOLER2` *resolves the peak, but not the front. Section 2.3 discusses this graph.*

the constraint that each interval accounts for only a controlled fraction of the maximum variation in u_k.

$$\int_{x_{n-1}}^{x_n} \left| \frac{du_k}{dx} \right| dx \;\le\; \texttt{TOLER1} \times (\max u_k - \min u_k)$$

When the grids have been successively enlarged many times by halving offending intervals, then all the integrals might have roughly the same magnitude, because "the raised nail gets the hammer" (Japanese proverb). V. Pereyra and G. Sewell relate this condition to approximation error by deriving error bounds, in some cases, when the definite integrals do have the same magnitude [2, pp. 365, 554]. Twopnt's second criterion amounts to the constraint

$$\int_{(x_{n-1}+x_n)/2}^{(x_n+x_{n+1})/2} |\kappa_k| \, ds \;\le\; \texttt{TOLER2} \times \left(\max_x \theta_k - \min_x \theta_k \right)$$

in which s is arc length and κ_k is the curvature of the graph of u_k.

$$\theta_k = \frac{d}{ds} \left(\text{atan} \left(\frac{du_k}{dx} \right) \right)$$

The relation between curvature and approximation error is unknown. In general, grid selection remains a research topic [2]. The process used by twopnt is a modified version of one used by M. D. Smooke [31], who apparently originated it.

2.3 What Grid Selection Does

(A) The mathematical formulation that is chosen to represent the underlying problem may represent it only with certain idealizations, simplifications, neglections. This is even conceivable in pure mathematics, when the numerical calculation is effected in order to obtain a preliminary orientation over the underlying problem. It will, however, be the rule and not the exception in applied mathematics, where these things are hardly avoidable in a mathematical representation. This complex is further closely related to the methodological observation that a mathematical formulation necessarily represents only a (more or less explicit) theory of some phase of reality, and not reality itself.

— J. von Neumann and H. H. Goldstine [26]

3 Text Output

3.0 Introduction

Twopnt's written output has selectable levels of detail. The levels correspond to the hierarchy of tasks twopnt performs during the simulation, see **Figure 3.1**. Twopnt's controls choose the level in the hierarchy beyond which output stops. Higher levels provide more detailed output.

⋆	`LEVELM`	level of informative message output	34
⋆	`LEVELD`	level of solution data output, no larger than `LEVELM`	34

For example, `LEVELM` $= 0$ prohibits output except error messages, and $= 1$ enables output from the decision-making level, while $= 2$ adds output from the mid-level tasks in Figure 3.1. If `LEVELM` $= 3$ and if time evolution occurs, then Newton's search for each transient state provides output too—this could be voluminous because there may be hundreds of time points.

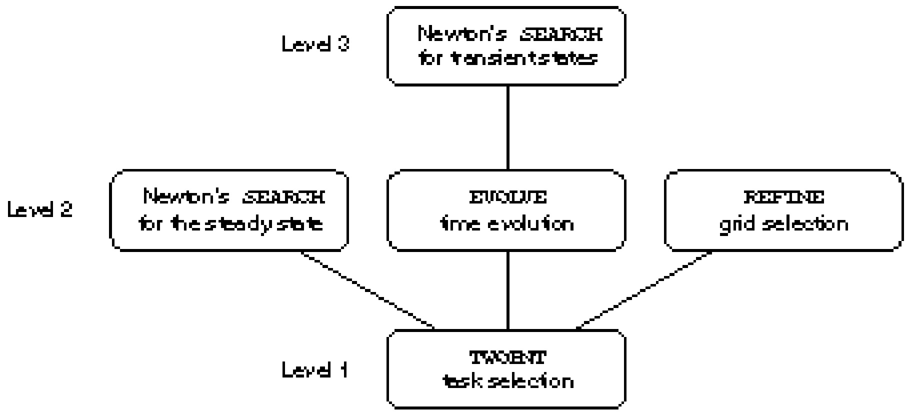

Figure 3.1 *Levels in the hierarchy of twopnt's tasks.*

The informative output controlled by `LEVELM` is explained below in separate sections for the decision-making level and for each task at higher levels. The output is illustrated by examples from the simulation described in Chapter 7. This output chronicles twopnt's internal activities, so the examples should clarify Chapter 2's explanation of what twopnt does. In unsuccessful simulations, the output may suggest how to improve efficiency or how to forestall failure. Successful simulations usually "turn off" this output or reduce it to a minimum, $= 1$.

The control for solution output, `LEVELD`, functions similarly. The latest solution estimate appears after other output for each task at the chosen level. For example, `LEVELD` $=$ 0 writes no solution data, while $= 1$ writes only the initial guess and the final result, and $= 2$ writes the latest solution estimate after each mid-level task in Figure 3.1. `LEVELD` $= 3$ is discouraged.

The solution output controlled by `LEVELD` is simulation specific and is not discussed further in this publication. Simulators usually provide other, separate output for graphics and the like. If the simulator does provide detailed written output via `LEVELD`, then $= 1$ may be useful for a quick look at the final result, while $= 2$ may help diagnose a malfunctioning simulation.

3.1 Briefest Output

The output described here, `LEVELM` $= 1$, is the briefest next to none-at-all. This output has two parts: one chronicles what twopnt does

- grid selection
- Newton's search
- time evolution

and the other explains what doing these things costs. The first part, the chronicle, summarizes data from higher output levels and so only appears when `LEVELM` $= 1$. The second part, the expense account, always appears after everything else, whenever output is requested.

Figure 3.2 shows the chronicle for the simulation in Chapter 7. The first column in the Figure, TASK, names the various tasks twopnt performs; the other columns report how the tasks fare. This particular simulation requires eleven grids which grow from 6 to 76 points. Lines 7 though 14 describe twopnt's efforts to find the solution on the smallest grid. Thrice Newton's search fails (LINES 8, 10 AND 12), and thrice time evolution of 50 time steps improves the guess (LINES 9, 11 AND 13). Thus, after 150 time steps, Newton's search succeeds on the fourth try (LINE 14). The solution found there suggests a new grid of 11 points (LINE 16) whose guess is sufficiently good that Newton's search succeeds immediately (LINE 17). Similarly, no subsequent grid requires time evolution (LINES 19 TO 44). The solution on the grid of 76 points produces no new grid (LINE 46), so it is the result of the simulation.

The second column in Figure 3.2, `NORM F`, indicates twopnt's progress toward the solution for each grid. This column reports the residual for the latest solution estimate, specifically

$$\log_{10} \|{}^{\mathrm{ss}}f(v)\|_\infty$$

in which $\|\cdot\|_\infty$ is the maximum-magnitude norm (the max or infinity norm) and v is the latest proposed solution. Section 2.2 explains that twopnt ignores the residual when deciding accuracy, so column two is for information only. In this example, the residual for the guess on the first grid measures 6.14 (LINE 7), and time evolution makes matters worse, increasing the measure to 6.37 (LINE 11) before dropping it to 6.02 (LINE 13). Newton's search then converges, stopping with a residual measuring -3.08 (LINE 14).

$|\Diamond|$ The third column in Figure 3.2, `COND J`, indicates the potential difficulty of solving the discrete steady state and time dependent equations. This column reports the

```
TWOPNT:  DOUBLE PRECISION (TWO POINT BOUNDARY VALUE PROBLEM) SOLVER,
         VERSION 3.08 OF JANUARY 1992 BY DR. JOSEPH F. GRCAR.

                     LOG10   LOG10
              TASK   NORM F  COND J   REMARK

             START     6.14           6 GRID POINTS
            SEARCH             7.00   DIVERGING
            EVOLVE     6.32    1.85   50 TIME STEPS, 3.2E-03 LAST STRIDE
            SEARCH             6.73   GOING OUT OF BOUNDS
            EVOLVE     6.37    2.69   50 TIME STEPS, 1.0E-02 LAST STRIDE
            SEARCH             6.36   GOING OUT OF BOUNDS
            EVOLVE     6.02    2.72   50 TIME STEPS, 1.0E-02 LAST STRIDE
            SEARCH    -3.08    5.64   14 SEARCH STEPS

            REFINE     6.32           1.00 AND 1.00 RATIOS, 11 GRID POINTS
            SEARCH    -2.16    5.77   13 SEARCH STEPS

            REFINE     6.65           0.94 AND 1.00 RATIOS, 16 GRID POINTS
            SEARCH    -3.10    5.94   7 SEARCH STEPS

            REFINE     6.87           0.92 AND 1.00 RATIOS, 23 GRID POINTS
            SEARCH    -0.75    6.07   10 SEARCH STEPS

            REFINE     7.01           0.91 AND 0.97 RATIOS, 35 GRID POINTS
            SEARCH    -1.07    6.22   6 SEARCH STEPS

            REFINE     7.11           0.76 AND 0.93 RATIOS, 42 GRID POINTS
            SEARCH    -0.69    6.25   4 SEARCH STEPS

            REFINE     7.16           0.52 AND 0.85 RATIOS, 47 GRID POINTS
            SEARCH    -0.97    6.26   3 SEARCH STEPS

            REFINE     7.20           0.31 AND 0.69 RATIOS, 55 GRID POINTS
            SEARCH     0.74    6.28   2 SEARCH STEPS

            REFINE     7.22           0.17 AND 0.47 RATIOS, 62 GRID POINTS
            SEARCH    -0.52    6.29   2 SEARCH STEPS

            REFINE     7.23           0.10 AND 0.27 RATIOS, 69 GRID POINTS
            SEARCH    -1.73    6.30   2 SEARCH STEPS

            REFINE     7.23           0.10 AND 0.16 RATIOS, 76 GRID POINTS
            SEARCH     2.00    6.29   1 SEARCH STEP

            REFINE                    0.10 AND 0.09 RATIOS
```

Figure 3.2. *The portion of* `LEVELM` $= 1$ *output that describes what twopnt does. Section 3.1 interprets this data. Chapter 7 explains the particular simulation.*

worst condition number for any Jacobian matrix encountered in Newton's searches for steady or transient states. Specifically, it reports

$$\max_J \log_{10} (\kappa(J))$$

in which, for this example, $\kappa(J)$ is the linpack condition number estimator [8]. The condition number measures the difficulty of solving accurately the matrix equations at each step of Newton's search. Lower condition numbers indicate easier problems. Usually, as here, the time dependent equations are easier than the steady state equations.

```
TWOPNT:  17.34 SECONDS TOTAL COMPUTER TIME (SEE BREAKDOWN BELOW).                 1
                                                                                  2
         PERCENT OF TOTAL COMPUTER TIME FOR VARIOUS TASKS:                        3
                                                                                  4
                         TASK                  SUBTASK                            5
           GRID    GRID  --------------------  ---------------------------        6
         POINTS  TOTALS  EVOLVE SEARCH REFINE  EVAL F PREP J  SOLVE  OTHER        7
                                                                                  8
              6    51.0    37.3   13.5    0.0    10.6   15.3   12.4   12.8        9
             11     2.6     0.0    2.5    0.1     0.3    1.3    0.5    0.5      10
             16     3.8     0.0    3.7    0.1     0.6    2.1    0.7    0.5      11
             23     3.7     0.0    3.5    0.1     0.7    1.6    0.9    0.5      12
             35     4.4     0.0    4.2    0.1     0.6    2.4    0.8    0.6      13
             42     4.4     0.0    4.1    0.1     0.5    2.7    0.5    0.6      14
             47     4.8     0.0    4.4    0.1     0.6    3.1    0.5    0.6      15
             55     5.4     0.0    5.0    0.2     0.5    3.7    0.5    0.6      16
             62     6.1     0.0    5.5    0.2     0.7    4.3    0.7    0.4      17
             69     6.7     0.0    6.2    0.2     0.7    4.7    0.7    0.6      18
             76     7.1     0.0    6.5    0.4     0.6    5.4    0.5    0.5      19
                                                                                 20
           TASK TOTALS:    37.3   58.9    1.5    16.6   46.6   18.6   18.2      21
                                                                                 22
         AVERAGE COMPUTER TIMES FOR, AND NUMBERS OF, SUBTASKS:                   23
                                                                                 24
                  AVERAGE SECONDS             NUMBER OF SUBTASKS                 25
           GRID   -------------------------   -------------------------         26
         POINTS    EVAL F   PREP J    SOLVE    EVAL F   PREP J    SOLVE         27
                                                                                 28
              6     0.002    0.060    0.003       814       44      809         29
             11     0.004    0.115    0.005        17        2       15         30
             16     0.006    0.180    0.008        17        2       15         31
             23     0.009    0.270    0.013        14        1       12         32
             35     0.012    0.410    0.020         9        1        7         33
             42     0.013    0.470    0.018         7        1        5         34
             47     0.018    0.540    0.022         6        1        4         35
             55     0.018    0.650    0.027         5        1        3         36
             62     0.024    0.740    0.040         5        1        3         37
             69     0.024    0.820    0.040         5        1        3         38
             76     0.028    0.940    0.045         4        1        2         39
                                                                                 40
TWOPNT:  SUCCESS.  PROBLEM SOLVED.                                               41
```

Figure 3.3. *The portion of* LEVELM > 0 *output that describes the cost of what twopnt does. Section 3.1 interprets this data. Chapter 7 explains the particular simulation.*

The fourth column in Figure 3.2, REMARK, contains miscellaneous information about each task. The grid selection task, REFINE, reports Section 2.3's worst ratios for the most recent grid's solution

$$\mathtt{RATIO}\,i := \max_{k,n} \mathtt{RATIO}\,i_{\,k,n} \qquad i = 1 \text{ and } 2$$

and when those ratios exceed TOLER1 and TOLER2, it reports the size of the then newly created grid. Newton's search for the steady state, SEARCH, reports the number of steps taken when successful (LINE 14), or the reason for failure when not successful (LINES 8, 10 AND 12). Time evolution, EVOLVE, reports the number of time steps and either the last step's size (LINES 9, 11 AND 13) or the last step's reason for failure.

The second part of twopnt's output, in **Figure 3.3**, displays the cost of what twopnt does. The unit of measure is computing time, so costs vary by machine and by machine load. In this example, an unstressed vax 8700 needs 17.34 seconds (LINE 1) to complete Chapter 7's simulation. Some machines may provide no timing data, or may provide real

```
TWOPNT:  DOUBLE PRECISION (TWO POINT BOUNDARY VALUE PROBLEM) SOLVER,          1
         VERSION 3.08 OF JANUARY 1992 BY DR. JOSEPH F. GRCAR.                  2
                                                                               3
                    LOG10   LOG10                                              4
            TASK   NORM F  COND J   REMARK                                     5
                                                                               6
           START     6.14                                                      7
          SEARCH             7.00   DIVERGING                                  8
          EVOLVE     6.32    1.85   50 TIME STEPS, 3.2E-03 LAST STRIDE         9
          SEARCH             6.73   GOING OUT OF BOUNDS                       10
          EVOLVE     6.37    2.69   50 TIME STEPS, 1.0E-02 LAST STRIDE        11
          SEARCH             6.36   GOING OUT OF BOUNDS                       12
          EVOLVE     6.02    2.72   50 TIME STEPS, 1.0E-02 LAST STRIDE        13
          SEARCH    -3.08    5.64   14 SEARCH STEPS                           14
                                                                              15
TWOPNT:  8.49 SECONDS TOTAL COMPUTER TIME (SEE BREAKDOWN BELOW).              16
                                                                              17
                       SUBTASK                             TASK               18
                       ---------------------------------   ---------------   19
                       EVAL F   PREP J    SOLVE    OTHER   EVOLVE   SEARCH   20
                                                                              21
          % OF TOTAL    21.3     30.2     23.0     25.6     74.1     25.7    22
         MEAN SECONDS   0.002    0.058    0.002                              23
             QUANTITY     814       44      809                              24
                                                                              25
TWOPNT:  SUCCESS.  PROBLEM SOLVED.                                            26
```

Figure 3.4. *Complete* LEVELM $= 1$ *and* LEVELD $= 0$ *output when grid selection is turned off,* ADAPT $= no$. *Section 3.1 interprets this data. Chapter 7 explains the particular simulation.*

time (that is, elapsed wall-clock time) rather than computing time. Appendix 2 explains how twopnt acquires timing data.

Figure 3.3 analyzes twopnt's costs three ways (LINES 3 TO 21). Analysis one is by grid. The Figure shows the smallest grid needs 51.0% of all time (LINE 9). Subsequent grids need much less time because better solution guesses are available for them. Analysis two is by task. For the only grid to require time evolution, Figure 3 shows evolution needs 37.3% of all computing time (LINE 9), while Newton's search needs only 13.5%. Thus, Newton's search for the steady state cuts short what could be a very expensive time evolution to the steady state. Analysis three is by subtask, that is, by the building blocks of twopnt's larger tasks.

• EVAL F	residual function evaluation
• PREP J	Jacobian matrix preparation
• SOLVE	solution of matrix equations

Figure 3.3 shows function evaluations need 16.6% of all time (LINE 21), and matrix preparations need 46.6%. In this example, matrix preparations include both evaluating and factoring the Jacobian matrices, and moreover, matrix evaluation is by numerical approximation requiring still more function evaluations. Altogether, nearly 63.7% of all time is spent evaluating the residual function—and in this example the function is particularly easy to evaluate! Thus as usual, the greatest performance improvements come from speeding residual function evaluation, see [12] [27].

Figure 3.3 also displays the costs and quantities of subtasks (LINES 23 TO 39). More subtasks are needed for earlier grids because twopnt finds solutions more easily once it has good solution guesses—the first grid needs 814 function evaluations and the last needs 4 (LINES 29 AND 39). Yet the earlier subtasks are less expensive because the cost per subtask

```
SEARCH:  SOLVE NONLINEAR, NONDIFFERENTIAL EQUATIONS.                             1
                                                                                 2
                  LOG10                                                          3
           SLTN   ----------------------------------------------------------     4
         NUMBER   NORM F   COND J   NORM S      ABS AND REL    DELTA B AND D     5
                                                                                 6
              0     6.02     5.03     2.42     2.42    0.37                      7
              1     5.31              1.61     1.61    0.59             -0.30    8
              2     4.93              1.54     1.54    0.58             -0.30    9
              3     4.85              1.51     1.51    0.91                     10
              4     4.90              1.26     1.26    0.95             -0.30   11
              5     4.88     5.50     1.72     1.72    0.35                     12
              6     4.53     5.64     1.10     1.10    1.52                     13
              7     3.65              0.02     0.02   -0.16                     14
              8     2.55             -0.76    -0.76   -1.01                     15
              9     1.65             -2.17    -2.17   -1.87                     16
             10     0.65             -2.80    -2.80   -2.86                     17
             11    -0.26             -3.91    -5.75   -3.86                     18
             12    -1.22             -4.69    -6.64   -4.62                     19
             13    -2.07             -5.75    -8.01   -5.71                     20
             14    -3.08             -6.53     ZERO   -7.29                     21
                                                                                22
SEARCH:  SUCCESS.                                                               23
```

Figure 3.5. *The portion of* LEVELM $= 2$ *output that describes Newton's search for the steady state. Section 3.2 interprets this data. Chapter 7 explains the particular simulation.*

is proportional to grid size—function evaluation time per grid point varies little between the smallest and the largest grids (LINES 29 AND 39).

$$\frac{0.002 \text{ seconds}}{6 \text{ points}} = 0.000333 \qquad \frac{0.028 \text{ seconds}}{76 \text{ points}} = 0.000368$$

Thus, the hardest problems occur where the subtasks are cheapest; the easiest problems occur where the subtasks are dearest. Using a succession of grids is clearly efficient.

When twopnt uses only one grid, ADAPT $=$ *no*, then the briefest output can be very brief. In this case, **Figure 3.4** displays the entire output for Chapter 7's example. The succession of grids is gone from the lists of tasks and expenses.

3.2 Output for Newton's Search

The output explained here describes both Newton's search for the steady state when LEVELM $= 2$, and Newton's searches for the transient states when $= 3$. **Figure 3.5** show output for the fourth and successful search for the steady state on the first grid in Chapter 7's example.

The first three columns in Figure 3.5 report data summarized on line 14 in Figure 3.2. Column 1, SLTN NUMBER, counts the solution estimates. Column 2 measures the steady state residual

$$\|{}^{\text{ss}}f(v_n)\|_\infty$$

in which v_n is the n-th solution estimate. Column 3, COND J, measures the condition of the Jacobian matrices

$$\log_{10}\left(\kappa(J)\right)$$

in which, for this example, $\kappa(J)$ is the linpack condition estimator [8].

The fourth and the last two columns in Figure 3.5 describe the step from one solution estimate to the next.

$$s_n := -J^{-1}\ {}^{\mathrm{ss}}f(v_n)$$
$$v_{n+1} := v_n + \delta_D \delta_B s_n$$

Column 4, `NORM S`, reports $\log_{10} \|s_n\|_\infty$. When δ_B and δ_D differ from 1.0, then their logarithms appear in columns 7 and 8, `DELTA B AND D`.

The data in Figure 3.5 reveals much about the performance of Newton's method. The search would stray in the first few steps if not for the convergence monitors explained in Section 2.1. The monitors use δ_B to keep v_{n+1} in bounds, and they use δ_D to ensure $\|s_{n+1}\|_\infty \leq \|s_n\|_\infty$. Whenever appropriate δ_B and δ_D can't be found for some s_n, then another s_n is obtained from a new Jacobian matrix. For this reason, since the first few steps are difficult, matrix evaluations occur at many early solution estimates (LINES 8 TO 13). Finally, when Newton's modified method can retain the Jacobian matrix over several steps, then columns 2 and 4 show convergence is roughly linear on a logarithmic scale (LINES 14 TO 21). In contrast, Newton's pure method converges quadratically. Although that method would need fewer steps, it would need more computing overall because matrix preparation is expensive, hence the controls `SSAGE` and `TDAGE`.

|◊| The fifth and sixth columns in Figure 3.5, `ABS` and `REL`, indicate convergence of the search. Section 2.1 explains the criteria for stopping at v_n is

$$|s_{i,n}| \ \leq \begin{cases} \text{either ABS, or} \\ \text{REL} \times |v_{i,n}| \end{cases}$$

in which i indexes all the entries in the vectors, and in which the controls `ABS` and `REL` are either `SSABS` and `SSREL` in searches for steady states, or `TDABS` and `TDREL` in searches for transient states. Columns 5 and 6 report the smallest `ABS` and `REL` sufficient to accept the unknowns violating the other's criterion. For example, all the entries in the initial guess, v_0, that do not meet the absolute bound, do meet a relative bound

$$|s_{i,0}| \not\leq \text{ABS} \quad \Longrightarrow \quad |s_{i,0}| \leq 10^{0.37} \times |v_{i,0}|$$

so Figure 3.5 reports 0.37 to measure the relative error in v_0 (LINE 7). Similarly, all the entries in the initial guess, v_0, that do not meet the relative bound, do meet an absolute bound

$$|s_{i,0}| \not\leq \text{REL} \times |v_{i,0}| \quad \Longrightarrow \quad |s_{i,0}| \leq 10^{2.42}$$

so Figure 3.5 reports 2.42 to measure the absolute error in v_0 (LINE 7). These numbers generally decline as the search progresses (LINES 7 TO 21). The search stops when the numbers drop below the respective control values, which in this case are `SSABS` $= 10^{-9}$ and `SSREL` $= 10^{-6}$. For example, at the chosen solution v_{14}, column 6 reports -7.29 for `REL`, which suggests any unknowns that do not meet the absolute bound have acceptable relative errors below $10^{-7.29}$. |◊||◊| This example reports `ZERO` for `ABS`, meaning all the unknowns do meet the relative bound.

3.3 Output for Time Evolution

The output explained here describes time evolution when `LEVELM` $= 2$. At higher levels this output gives way to that of Newton's search for each transient state interspersed by messages counting time points. For Chapter 7's example, **Figure 3.6** shows the output from the time evolution following the initial failure of Newton's method. This task is summarized on line 9 in Figure 3.2.

```
EVOLVE:  BEGIN TIME EVOLUTION.

            TIME   LOG10                     NEWTON SEARCH
           POINT   ------------------------  ----------------------------
          NUMBER   NORM F   CHANGE   STRIDE  STEPS   J'S   COND J   REMARK
               0     6.14
               1     5.83     1.10    -3.00      3     1     1.25
               2     5.81    -0.65    -3.00      3
               3     5.80    -0.67    -3.00      3
               4     5.79    -0.68    -3.00      3
               5     5.78    -0.69    -3.00      3
               6     5.76    -0.71    -3.00      3
               7     5.75    -0.72    -3.00      3     1     1.25
               8     5.73    -0.73    -3.00      2
               9     5.72    -0.75    -3.00      2
              10     5.71    -0.76    -3.00      2
              11     5.69    -0.78    -3.00      3
              12     5.67    -0.79    -3.00      2
              13     5.66    -0.81    -3.00      2
              14     5.64    -0.82    -3.00      2
              15     5.63    -0.84    -3.00      2
              16     5.61    -0.86    -3.00      2
              17     5.59    -0.87    -3.00      2     1     1.25
              18     5.60    -0.89    -3.00      2
              19     5.64    -0.91    -3.00      2
              20     5.68    -0.93    -3.00      2
              21     5.71    -0.93    -3.00      2
              22     5.74    -0.90    -3.00      2
              23     5.77    -0.87    -3.00      2
              24     5.80    -0.84    -3.00      2
              25     5.82    -0.82    -3.00      2
              26     5.89    -0.25    -2.50      2     1     1.85
              27     5.94    -0.20    -2.50      3
              28     5.99    -0.15    -2.50      3
              29     6.03    -0.11    -2.50      3
              30     6.07    -0.07    -2.50      3
              31     6.10    -0.04    -2.50      3
              32     6.13    -0.01    -2.50      3
              33     6.15     0.02    -2.50      3     1     1.85
              34     6.18     0.05    -2.50      3
              35     6.20     0.07    -2.50      3
              36     6.21     0.09    -2.50      3
              37     6.23     0.11    -2.50      3
              38     6.24     0.12    -2.50      3
              39     6.26     0.14    -2.50      3     1     1.85
              40     6.27     0.15    -2.50      2
              41     6.28     0.16    -2.50      2
              42     6.29     0.17    -2.50      3
              43     6.29     0.18    -2.50      3
              44     6.30     0.19    -2.50      3
              45     6.30     0.19    -2.50      3
              46     6.31     0.20    -2.50      3
              47     6.31     0.20    -2.50      2     1     1.84
              48     6.32     0.21    -2.50      2
              49     6.32     0.21    -2.50      3
              50     6.32     0.21    -2.50      3

EVOLVE:  SUCCESS.  TIME EVOLUTION COMPLETED.
```

Figure 3.6. *The portion of* LEVELM $= 2$ *output that describes time evolution. Section 3.3 interprets this data. Chapter 7 explains the particular simulation.*

The cumulative count of time points, m, appears in the Figure's first column, `TIME POINT NUMBER`. Twopnt's control, `STEPS1`, specifies 50 time points per time evolution, so the second time evolution continues from point 50. Output from that evolution is not pictured here.

As in other output tables, the second column in Figure 3.6, `NORM F`, measures the steady state residual, specifically

$$\log_{10} \| {}^{\mathrm{ss}}f(v_{(m)}) \|_\infty$$

in which $v_{(m)}$ is the m-th transient state. The residual quickly decreases over the first step to 5.83 (LINE 7), then decreases more slowly to 5.59 (LINE 23), and finally increases to 6.32 (LINE 54). Newton's search for the steady state usually fails when begun from transient states with large or increasing residuals (the residuals oscillate in time when the underlying simulated phenomena has oscillatory transient behavior). In this simulation too, after the residual peaks at 6.32 for $v_{(48)}$, and again at 6.62 for $v_{(85)}$ (not shown), then Newton's search for the steady state succeeds from $v_{(150)}$ where the residual is 6.02.

The third column in Figure 3.6, `CHANGE`, measures the change to the solution from one time point to the next.

$$\log_{10} \| \, v_{(m)} - v_{(m-1)} \|_\infty$$

$|\Diamond|$ If the transient residual is explicit in time

$${}^{\mathrm{td}}f(v) = \frac{dv}{dt} - {}^{\mathrm{ss}}f(v)$$

then the change to the solution equals the product of the steady state residual and the time stride. In the Figure's logarithmic scale, column 3 then equals the sum of columns 2 and 4, which report the residual and the stride, respectively. Even when the transient residual is not explicit in time, as in Chapter 7's simulation, column 3 still moves jointly with columns 2 and 4. In this example, column 3 jumps from -0.82 to -0.25 when the stride increases one-half order of magnitude (LINES 31 AND 32).

The fourth column in Figure 3.6, `STRIDE`, displays the time stride chosen by the controls discussed in Section 2.2. With

$$\texttt{STRID0} = 10^{-3} \quad \texttt{STEPS2} = 25 \quad \texttt{TINC} = \sqrt{10} \quad \texttt{TMAX} = 10^{-2}$$

the stride is 10^{-3} for the first 25 steps, then $10^{-2.5}$ for the next 25, and finally 10^{-2} thereafter.

The remaining columns in Figure 3.6 describe Newton's search for each transient state. Column 5, `STEPS`, counts search steps; column 6, `J'S`, counts Jacobian matrices; column 7, `COND J`, reports the largest condition number measured in the manner of Section 3.1; column 8, `REMARK`, explains why Newton's search fails when it does. The count of steps, column 5, indicates the work of solving the transient equations. Shorter, easier time strides need few search steps. Longer, harder strides need more searching, particularly as the Jacobian matrix grows older and yields poorer search directions. In this example `TDAGE` $=$ 20, so after the matrix evaluation at one of the 3 search steps for $v_{(39)}$ (LINE 45), at most twenty Newton search steps pass before the next matrix evaluation at one of the 2 steach steps for $v_{(47)}$ (LINE 53).

3.4 Output for Grid Selection

The output explained here describes grid selection when `LEVELM` ≥ 2. For each grid, this output analyzes the latest solution and shows the resultant new grid, if any. **Figure 3.7** shows

```
REFINE:  SELECT A GRID.                                                       1
                                                                              2
                  RATIO 1   RATIO 2                                           3
                  -------   -------                                           4
         ACTUAL     0.939     1.000                                           5
        DESIRED     0.100     0.100                                           6
                                                                              7
         THE NEW GRID (* MARKS NEW POINTS):                                   8
                                                                              9
                                     LARGEST RATIOS AND                      10
          INDEX         GRID POINT     NUMBER TOO LARGE                      11
         ------   ----------------   -------------------                     12
                                      RATIO 1    RATIO 2                     13
              1    0.000000000E+00                                           14
              2*   2.500000037E-01   0.94   4                                15
              3    5.000000075E-01             1.00   3                      16
              4*   7.500000112E-01   0.70   4                                17
              5    1.000000015E+00             0.42   3                      18
              6*   1.250000019E+00   0.23   3                                19
              7    1.500000022E+00             0.45   4                      20
              8*   1.750000026E+00   0.06                                    21
              9    2.000000030E+00             0.23   3                      22
             10*   2.250000052E+00   0.02                                    23
             11    2.500000075E+00             0.08                          24
                                     0.02                                    25
             12    3.000000119E+00             0.04                          26
                                     0.02                                    27
             13    3.500000089E+00             0.03                          28
                                     0.02                                    29
             14    4.000000060E+00             0.04                          30
                                     0.01                                    31
             15    4.500000030E+00             0.04                          32
                                     0.01                                    33
             16    5.000000000E+00                                           34
```

Figure 3.7. *The portion of* `LEVELM` $= 2$ *output that describes grid selection. Section 3.4 interprets this data. Chapter 7 explains the particular simulation.*

this output for the transition from the second to the third grids in Chapter 7's simulation. This task is summarized on line 19 of Figure 3.2 when `LEVELM` $= 1$.

The first portion of Figure 3.7 (LINES 3 TO 6) reports Section 2.3's largest ratios for the most recent grid's solution. In this example these ratios exceed their desired values, `TOLER1` and `TOLER2`, so twopnt constructs a new grid.

The second portion of Figure 3.7 (LINES 10 TO 34) lists the new and old grid points. Twopnt's grid construction process retains all old points and then adds new points to halve selected intervals. Column 1 in the Figure, `INDEX`, counts points in the new grid and marks new points with asterisks. Column 2, `GRID POINT`, shows point locations. The old points lie on every other line; the new points, if any, lie between. Grid regions with no new points therefore have many blank lines (LINES 24 TO 34).

Columns 3 through 6 in Figure 3.7, `RATIO 1` and `RATIO 2`, reveal the workings of Section 2.3's grid acceptance criteria.

$$\max_{k,n} \texttt{RATIO}\,i_{\,k,n} \overset{?}{\leq} \texttt{TOLER}\,i \qquad i = 1 \text{ and } 2$$

`RATIO1` monitors the greatest change in value from one grid point to the next, so entries in column 3 appear on lines between old points. Column 3 lists the largest $\texttt{RATIO1}_{k,n}$ for any component, k, in a given interval. Column 4 counts the components whose ratios exceed `TOLER1` in that interval. Intervals with entries in column 4 have been marked for halving. For example, in the interval between the second and third old points (LINE 17), the largest ratio is 0.70, and altogether, four solution components have ratios exceeding TOLER1 $= 0.1$.

Similarly, RATIO2 monitors the greatest change in angle at grid points, so the entries in column 4 appear on lines with the old grid's points. Column 5 lists the largest RATIO2$_{k,n}$ for any component, k, at a given point. Column 6 counts the components whose ratios exceed TOLER2 at that point. Intervals on either side of old points with entries in column 6 have been marked for halving.

Twopnt halves marked intervals. If too many intervals need halving, then those with more marks win priority. For example, in Figure 3.7 the interval between the second and third old points (LINES 16 TO 18) has $4 + (3 + 3) = 10$ marks, while the interval between the fifth and sizth old points (LINES 22 TO 24) has $0 + (3 + 0) = 3$ marks. Twopnt halves all marked intervals in this example because, with PADD $=$ PMAX, twopnt has no limit on new points.

(B) Even if the mathematical formulation is not questioned, that is, if the theoretical description which it represents and the idealizations, simplifications, and neglections which it involves are accepted as final (and not viewed as sources of errors), this further point remains: The description may involve parameters, the values of which have to be derived directly or indirectly (that is, through other theories or calculations) from observations. These parameters will be affected with errors, and these underlying errors will cause errors in the result of our calculation.

— J. von Neumann and H. H. Goldstine [26]

4 Using Controls

4.0 Introduction

Simulators using twopnt must provide the means to access twopnt's controls. Usually, values for twopnt's controls are intermingled with other data in text files read by the simulators. In this case, the simulators may use their own names, or "*keywords,*" to indicate twopnt's control data. Those simulators' reference manuals should be consulted for explanation. Appendix 1 cross-references twopnt's controls with the keywords used by some common simulators.

In extreme cases, the simulator programs themselves may have to be modified to adjust twopnt's behavior. Some controls may be inaccessible this way because they are inapplicable, or because simulation writers know best how to set them. Other controls may be accessible but seldom in need of changing. Twopnt provides default values for its controls in case simulators do not.

4.1 Short List of Controls

This short list of controls collects in one place everything introduced elsewhere. If a value appears after an $=$ sign, then that value is twopnt's default for the control. Simulators generally replace twopnt's defaults by their own. Page numbers refer to the explanations in Chapter 2 or 3, and to the caveats in Section 4.2.

⋆ `ABOVE(`i`)`	upper limit for the i-th unknown	14, 33 (● 62, 66)
⋆ `ACTIVE(`k`)`	use the k-th component for grid selection	17, 33 (● 63, 66)
⋆ `ADAPT` $=$ *no*	perform grid selection	16, 34
⋆ `BELOW(`i`)`	lower limit for the i-th unknown	14, 33 (● 62, 66)
⋆ `LEVELD` $= 1$	level of solution data output	21, 34
⋆ `LEVELM` $= 1$	level of informative message output	21, 34
⋆ `PADD` $=$ `PMAX`	maximum points added to any one grid	18, 34
⋆ `PMAX`	maximum points in any grid	18, 34 (● 63, 68)
⋆ `POINTS`	points (in the initial grid)	16, 35 (● 58, 62, 68)
⋆ `SSABS` $= 10^{-9}$	steady state absolute convergence test's bound	13, 35
⋆ `SSAGE` $= 10$	steady state Jacobian's retirement age	13, 37

⋆ SSREL $= 10^{-6}$	steady state relative convergence test's bound	13, 35
⋆ STEADY $=$ *yes*	search for the steady state	12, 37
⋆ STEPS0 $= 0$	time steps before searching for the steady state	15, 38
⋆ STEPS1 $= 200$	time steps after failing to find a steady state	15, 38
⋆ STEPS2 $= 100$	time steps before increasing the stride	15, 38
⋆ STRID0 $= 10^{-4}$	initial stride between time points	15, 38
⋆ TDABS $= 10^{-9}$	time dependent absolute convergence test's bound	13, 35
⋆ TDAGE $= 20$	time dependent Jacobian's retirement age	13, 37
⋆ TDEC $= \sqrt{10}$	divisor by which to decrease the stride	15, 38
⋆ TDREL $= 10^{-6}$	time dependent relative convergence test's bound	13, 35
⋆ TINC $= 10$	multiplier by which to increase the stride	15, 38
⋆ TMAX $= 10^{-2}$	maximum stride	15, 38
⋆ TMIN $= 10^{-20}$	minimum stride	15, 38
⋆ TOLER0 $= 10^{-9}$	absolute and relative significance floor	17, 39
⋆ TOLER1 $= 0.2$	bound on change of value in each interval	17, 39
⋆ TOLER2 $= 0.2$	bound on change of angle at each point	18, 39
⋆ U	initial guess	40 (● 61, 71)
⋆ X	initial grid	16, 40 (● 58, 63, 71)

4.2 Long List of Controls

This list matches Section 4.1's short list but has long explanations. Details are explained here, so familiarity with Chapter 2 and sometimes Chapter 6 is assumed.

⋆ ABOVE(i) $=$	upper limit for the i-th unknown	14 (● 62, 66)
⋆ BELOW(i) $=$	lower limit for the i-th unknown	14 (● 62, 66)

Twopnt most commonly fails when Newton's search "goes out of bounds." Writers and users of simulators then offer differing explanations: the simulation is being abused, some say; the choice of limits is too conservative, others say. Simulators usually choose appropriate limits but provide controls to override their choices. The bounds can be "turned off" only by choosing very large limits. Simulations "going out of bounds" might be repeated with looser limits, but simulations consistently going out of bounds may be improperly formulated. The bounds are an independent check on the veracity of the scientific model.

The controls provided by the simulator may not have full functionality. For example, chemistry simulations usually have one control for the lower limit of all species' concentrations simultaneously. $|\Diamond|$ Twopnt's own controls have some coarseness too: unknowns belonging to the same component must have the same limits. See Chapter 6.

Whatever limits are chosen, they should be relaxed from the precisely correct physical limits. For example, chemistry simulations should not limit species concentrations to ≥ 0. The equations underlying the simulation usually allow non-physical values, < 0, and it may be expedient to permit searches for physically correct values to pass through non-physical regimes. Moreover, computer imprecision enables quantities to lie vanishingly near boundaries but on either side without harm.

⋆ ACTIVE(k) $=$	use the k-th component for grid selection	17 (● 63, 66)

These controls apply only to two point boundary value problems with automatic grid selection. Simulators usually set them without provision for change. They know which components are physically real and smoothly varying. These controls are meant to exclude other components which might exist only for computational convenience. For example, simulators may find it convenient to treat finite elements' coefficients as components, or to replicate eigenvalues throughout the grid (though this last computational device is unnecessary, see [16]).

$\star$ ADAPT $=$ *no* perform grid selection 16

Twopnt selects grids automatically only for two point boundary value problems, and then only when permitted. When ADAPT $=$ *yes*, then twopnt checks whether each grid is adequate even though PADD $= 0$ or POINTS $=$ PMAX might prevent an inadequate grid from improving. Some simulators reserve ADAPT for use behind the scenes. For example, they may perform a sequence of increasingly difficult simulations to obtain solution guesses, and while doing so, they may set ADAPT $=$ *no* to prevent grid selection.

$\star$ LEVELD $= 1$ level of solution data output 21
$\star$ LEVELM $= 1$ level of informative message output 21

These controls affect only twopnt's output and not the graphical output or the solution files some simulators may provide. Twopnt acts as though the controls satisfy the following inequalities.

$$\text{too much output} \;=\; 3 \;\geq\; \texttt{LEVELM} \;\geq\; \texttt{LEVELD} \;\geq\; 0 \;=\; \text{no output}$$

Some simulators repackage these controls as one number: 00, 10, 11 and so on. Level 00 permits only error messages.

$\star$ PADD $=$ PMAX maximum points added to any one grid 18

This control applies only to two point boundary value problems with automatic grid selection. Twopnt's default, PADD $=$ PMAX, limits each grid's growth only by the available memory. Since twopnt enlarges grids by halving intervals, a grid of p points acquires at most $p - 1$ new points. If PADD > 0 then ADAPT $=$ *no* still can forbid grid selection.

Many simulations restrict PADD to a few points for reasons of efficiency, as follows. Early in the simulation when no grid is adequate, much different grids can have much different solutions. For example, flame fronts migrate as grid points permit. Thus, if an early grid acquires many new points, then its solution may poorly guess the next. Twopnt then must solve, with some difficulty, a still inaccurately resolved problem. Restricting the grids' growth restricts the solutions' change, and so eases progress to grids that do resolve the simulation.

Very small values for PADD may lead to unnecessarily large grids. If several regions need improvement in locations dependent on one another, then improving only some regions may add points ultimately not needed. Thus, PADD $= 1$ is not recommended.

$\star$ PMAX $=$ maximum points in any grid 18 (● 63, 68)

This control applies only to two point boundary value problems with automatic grid selection. A simulation whose grids grow large needs more computing time and more memory space. A limit on grid size protects against overzealous grid enlargement, and so might be chosen smaller than the simulator would otherwise allow.

Simulation programs that adhere to the fortran standard [1] must fix memory size when they are written, and thus must anticipate the largest grids they may encounter. Since this is difficult to do, and since computer systems penalize programs using much memory, simulation programs sometimes need changing to adjust memory size.

Many things besides the grid affect memory size, and for limited memory, may force grids smaller. For example, Jacobian matrices usually are formed explicitly. Their memory needs then dominate all others and grow as $\texttt{COMPS}^2 \times \texttt{PMAX}$. In chemistry simulations for example, `COMPS` counts chemical species. If a simulator allows this number to change, then its limit on `PMAX` necessarily changes too, in the opposite direction.

⋆ `POINTS` =	points (in the initial grid)	16 (● 58, 62, 68)

Simulators have "points" if they employ the "components at points" convention for grouping their unknowns, see Section 6.2. Usually, simulators provide controls to specify grid size either directly, or else indirectly through other attributes of the grid.

⋆ `SSABS` $= 10^{-9}$	absolute bound for steady state convergence	13
⋆ `TDABS` $= 10^{-9}$	absolute bound for time dependent convergence	13
⋆ `SSREL` $= 10^{-6}$	relative bound for steady state convergence	13
⋆ `TDREL` $= 10^{-6}$	relative bound for time dependent convergence	13

Newton's search halts when the change to each unknown is small ($|\diamondsuit|$ actually, Newton's search halts when the change would be small). This means the estimated solution has approximately the desired accuracy. The conflicting demands for low $\Downarrow$ and high $\Uparrow$ bounds in the convergence tests are the same for every simulation, but of course, the values may change with each simulation.

(a) $\Downarrow$ consistency within the solution
(b) $\Downarrow$ confidence in simulation results
(c) $\Uparrow$ compromises in the simulation
(d) $\Uparrow$ economy of computer use
(e) $\Uparrow$ imprecision in machine arithmetic

(a) $\Downarrow$ High bounds, meaning weak convergence tests, are not recommended. The simulation must be performed with reasonable numerical accuracy to assure consistency within the simulation model.

(b) $\Downarrow$ Any really important simulation should be repeated with more stringent convergent tests, say with bounds one or two orders of magnitude smaller, to gain confidence in the simulation.

(c) $\Uparrow$ Low solution error may be unnecessarily stringent owing to both scientific and mathematical compromises in the simulation. For example, secondary phenomena may be "left out" for lack of data, or coarse grids may be chosen for lack of computing power. Perhaps for many simulations, only the order of magnitude of the results is meaningful.

(d) $\Uparrow$ Newton's modified method converges geometrically or faster, so computing time is inversely proportional to the logarithm of the bounds. That is, increasing all controls from 10^{-8} to 10^{-4} may decrease computing time by fifty percent. Of course, when Newton's searches are few or short, then the computing time may be difficult to control precisely.

(e) $\Uparrow$ Because machines necessarily make (very small) arithmetic errors, there is a level of error impossible to breach. If the convergence tests are set below this floor, then Newton's searches may wander indefinitely among the best solutions the machine can find.

Beware that these bounds do not control all simulation errors, and they do not control with certainty what few they try to. See John von Neumann's four simulation errors in Chapter 1, and the steps for building a simulation in Chapter 2. Simulation accuracy depends ultimately on the scientific model. Twopnt's controls affect other simulation errors.

simulation step	von Neumann's errors	twopnt's controls
3, solution	approximation, precision	`SSABS` and `SSREL`
2, discretization	approximation	`TOLER1` and `TOLER2`
1, modeling	measurement, model	

The bounds of interest here pertain only to the errors of solving nondifferential equations. Errors even in this restricted sense are not controlled precisely. The mathematical analysis required to do so varies with each simulation, and is seldom performed. In general, quantitative relationships between accuracy and simulation controls are unknown.

Twopnt's convergence tests for Newton's searches control absolute and relative accuracy, and they vary for steady and transient searches.

	absolute	relative
steady	`SSABS`	`SSREL`
transient	`TDABS`	`TDREL`

The transient convergence tests can be less stringent because twopnt uses transient states only as guesses for steady state searches.

The absolute and relative tests apply separately to each unknown. That is, the solution estimate for one unknown may be absolutely accurate, while the estimate for another may be relatively accurate. Each unknown is required to pass only one test. If one convergence bound is too stringent or is zero, intending "no error allowed," then the other test still allows some slack. Both should not be set zero because (e, above) imprecise machine arithmetic cannot find exactly accurate solutions. Usually, low relative error is more valuable because relative accuracy means "some digits are significant." Thus, the absolute convergence test is "just in case" some values are too small to have meaningfully low relative error. In chemistry simulations for example, if mass fractions are unknowns, then fractions below 10^{-9} probably mean the corresponding molecules aren't present, so in this case 10^{-9} is reasonable for `SSABS` and `TDABS`.

The relative convergence bounds can range from 0 to 1. Relative accuracy near 1 means no precision at all. The practical upper limit (a, above) may be 10^{-3} because some precision is needed to assure consistency within the simulation. The machine *"roundoff level"* relative to 1.0 establishes a floor beyond which precision cannot improve. Precision at the roundoff level is unattainable because errors compound in long calculations. The practical lower limit (e, above) for the relative bounds may be three orders of magnitude above the roundoff level.

$|\diamondsuit|$ Computing machines provide thirty-two and sixty-four bit binary arithmetic. The roundoff level exceeds $2^{-32} = 10^{-9.6}$ and $2^{-64} = 10^{-19.3}$ because some bits are lost to exponents and signs. For sixty-four bits, the roundoff level is roughly $2^{-48} = 10^{-14.5}$ (cray single precision) or $2^{-52} = 10^{-15.7}$ (vax double precision), and then the attainable floor for relative errors might be (e) $10^{-11.5}$ (cray) or $10^{-12.7}$ (vax). For thirty-two bits, the roundoff level is roughly $2^{-23} = 10^{-6.9}$, and then the attainable floor is $10^{-3.9}$ (vax single precision). Thirty-two bit arithmetic is not recommended because the floor is too near the ceiling (a, above) 10^{-3}.

The absolute convergence bounds can range from 0 to anything. Their attainable lower limit depends both on the roundoff level of the machine and on the anticipated magnitudes of the unknowns. If an unknown is expected to have magnitude 10^4, and if the roundoff level relative to 1 is $10^{-14.5}$, then by the earlier reasoning, the attainable absolute accuracy is three orders of magnitude above $10^4 \times 10^{-14.5}$. This $10^{-7.5}$ is for unknowns with magnitude 10^4.

Twopnt applies the same absolute convergence test to all unknowns, however, so difficulties may result when unknowns have much different magnitudes.

Some simulators perform transformations of variables that complicate choosing these bounds. These simulators might alter their data for printing and plotting. For example, chemistry simulations typically employ mass fractions for unknowns, but they might report mole fractions, or bulk quantities instead. The data reported by the simulator then are not the data to which twopnt applies its convergence tests, so the relationship between twopnt's controls and the accuracy of reported results is unclear.

$\star$ `SSAGE` $= 10$	steady state Jacobian's retirement age	13
$\star$ `TDAGE` $= 20$	time dependent Jacobian's retirement age	13

The retirement age limits the steps in Newton's search between matrix evaluations. The limits may range from 1 to anything, though Section 2.1's convergence monitors may retire some matrices early.

Smaller retirement ages improve the chances for successful searches, because more current matrices produce better search directions. If the retirement age is 1, then the search reverts to the pure, but still damped, Newton's method. This may succeed where the modified method does not. For example, `SSAGE=1` makes time integration unnecessary on the first grid in Chapter 7's simulation. Transient searches should be easy, so `TDAGE=1` should be unnecessary for them.

Larger retirement ages generally reduce computing time. Although older matrices produce slightly misdirected searches and so entail more search steps, fewer matrices are needed, usually, when the retirement age increases. Since matrix preparation likely is the most expensive subtask twopnt performs, on balance, any reduction in the matrix population also reduces the overall computing time.

$|\diamondsuit|$ The preceding discussion can be quantified as follows. When as usual, Gauss' elimination method solves the matrix equations in Newton's method (we stand on the shoulders of giants), then the computing time for Newton's search is roughly

$$n_m \, \tau_m + n_s \, \tau_s$$

in which "$_m$" notes matrices and "$_s$" notes search steps, and in which n is the count and τ is the computing time of such things. If matrices are computed on schedule then $n_m \approx n_s \, / \, \texttt{AGE}$ in which `AGE` is `SSAGE` or `TDAGE`. The time of one step, τ_s, is mostly the time of one function evaluation, τ_f. Matrix preparation involves forming and factoring the matrices, but τ_m is mostly the time of forming. Owing to the complexity of residual functions, simulators usually form Jacobian matrices by numerical rather than analytic differentiation. In two point boundary value problems for example, matrix evaluation requires at least $3 \times \texttt{COMPS}$ function evaluations, so $\tau_m \approx 3 \times \texttt{COMPS} \times \tau_f$. In chemistry simulations moreover, `COMPS` may be 50. With these substitutions, total computing time might be as follows.

$$n_m \, \tau_m + n_s \, \tau_s \;\approx\; \left(\frac{150}{\mathsf{AGE}} + 1 \right) n_s \, \tau_f$$

Thus, increasing the retirement age will reduce the total computing time so long as n_s does not change proportionally with `AGE`.

$\star$ `STEADY` $=$ *yes*	search for the steady state (yes or no)	12

Twopnt undertakes the search for the steady state, twopnt's primary task, only when permitted. When not, twopnt performs the initial time evolution controlled by `STEPS0`, but no grid selection follows. Time evolution alone sometimes reaches the steady state with reasonable efficiency. See the discussion of `STRID0` and related controls.

The search for the steady state might be omitted to diagnose a malfunctioning simulation. A simulation with agitated time evolution, and seemingly without a steady state, might be better understood by inspecting the transient states. With `STEADY` $= no$, twopnt finishes after the initial time evolution, and lets the simulator draw pictures and so on.

The search for the steady state might be omitted to avoid unnecessary work in a sequence of simulations. Many simulators improve their scientific model piecemeal, and then the result of one simulation is only the guess for the next. Short transient evolutions might find sufficient guesses, while exhaustive searches might be wasted finding preliminary results too accurately.

Finally, the search for the steady state might be omitted to avoid convergence difficulties in a sequence of simulations. Many simulators undertake sequences of simulations to perform parameter studies, and then the result of one simulation is a very good guess for the next. Unfortunately, steady states usually occur where strong opposing trends balance, so with imprecise machine arithmetic, residual functions have large relative errors near steady states. The search directions in Newton's method then have large relative errors too, and thus Newton's search may have difficulty continuing or recognizing it need not. A time evolution might approach the steady state more reliably, or at least, might have less trouble recognizing the steady state. Beware if Newton's search consistently fails to find the steady state while time evolution appears to, then there is reason to doubt the veracity of the simulator.

$\star$	`STEPS0` $= 0$	time steps before searching for the steady state	15
$\star$	`STEPS1` $= 200$	time steps after failing to find a steady state	15

Twopnt evolves in time the guess for the steady state either when asked, by `STEPS0` > 0, or when Newton's search is unable find the steady state, provided `STEPS1` > 0. The requested quantities of steps may range from 0 to anything. If automatic grid selection has been requested, then the initial time evolution specified by `STEPS0` occurs only for the first grid.

It is best to perform many transient steps, `STEPS1` $\gg 0$, when any are needed. Much computing time can be wasted by frequently reverting to unsuccessful searches for the steady state. Searches for transient states usually need comparatively little computing time.

$\star$	`STRID0` $= 10^{-4}$	initial stride between time points	15
$\star$	`STEPS2` $= 100$	time steps before increasing the stride	15
$\star$	`TINC` $= 10$	multiplier by which to increase the stride	15
$\star$	`TMAX` $= 10^{-2}$	maximum stride	15
$\star$	`TDEC` $= \sqrt{10}$	divisor by which to decrease the stride	15
$\star$	`TMIN` $= 10^{-20}$	minimum stride	15

These controls choose and change the time stride. The initial stride and the stride limits must be positive, the other controls must range upward from 1. The stride cannot lengthen if `TMAX` $=$ `STRID0` or `TINC` $= 1$ or `STEPS2` is very large. The stride cannot shorten if `TMIN` $=$ `STRID0` or `TDEC` $= 1$.

Twopnt's use of time evolution supposes the simulated phenomena relax to a steady state over time. The time stride should be chosen to reach, in as little computing time as possible, either (1) the steady state, or (2) a state from which Newton's search for the steady state can be successful. The conflicting demands for short $\Downarrow$ and long $\Uparrow$ strides are the same for every simulation, but again, the values may change for each simulation.

(a) $\Downarrow$ follow the transient evolution to the steady state
(b) $\Downarrow$ ease searches for transient states
(c) $\Uparrow$ reduce searches for transient states
(d) $\Uparrow$ forgo detailed resolution of transient response

The transient evolution is efficient if searches for transient states are easy and few. (b) $\Downarrow$ Shorter strides make the searches easier because the states change less from one time point to the next, but (c) $\Uparrow$ longer strides need fewer steps to reach an acceptable state. For very long strides, searches for transient states degenerate to seeking the steady state directly.

The time evolution is accurate if the transient state, $v_{(m)}$, computed at time point t_m closely approximates the “true” state, $v(t_m)$, determined by the time dependent differential equations. It is not possible to have $v_{(m)} = v(t_m)$ due to the errors explained in Chapter 1, but it is possible to have $v_{(m)} \approx v(t_m)$ provided the time strides, $t_m - t_{m-1}$, are short. Thus, on the way to an acceptable $v_{(m)}$, short strides keep $v_{(m)}$ near $v(t_m)$, but long strides allow $v_{(m)}$ to stray. (a) $\Downarrow$ Short strides preserve the physical meaning of the transient simulation, but (d) $\Uparrow$ long strides avoid the expense of tracking minor variations in the transient response.

Usually, longer strides are easier and safer as the evolution proceeds, hence twopnt’s controls to gradually lengthen stride and to shorten it if trouble occurs. Frequent adjustments to the stride are not recommended because each change requires a new Jacobian matrix.

The aggressive strategy for choosing the time stride (1, above) ignores other advice given here regarding efficiency. It favors long strides to reach the steady state “quickly.” This strategy begins with `STRID0` large, relies on search failures to prompt stride reductions if needed, and allows large increases after few steps, perhaps `STEPS2` $= 20$. An aggressive strategy may reach the steady state through time evolution alone, and thus its expense may be partly compensated by eliminating the search for the steady state.

Simulators favor success over speed, and so err conservatively: they try to follow the natural relaxation with reasonable haste. The conservative strategy (2, above) uses short strides to keep transient searches easy and to avoid straying from the true, transient evolution. This strategy begins with `STRID0` small and allows small increases, perhaps `TINC` $= 2$, after many steps. These control settings might be changed after some experience with them. For example, twopnt’s own choices are more aggressive. The initial stride is long, and large increases are permitted, but only after many steps.

$\star$ `TOLER0` $= 10^{-9}$	absolute and relative significance floor	17

This control applies only to two point boundary value problems with automatic grid selection. Many simulators have no provision for changing the floor because it seldom matters when the `ACTIVE(`k`)` controls are properly set. Twopnt inspects only components whose absolute and relative variations exceed the floor. Twopnt’s default value suits sixty-four bit arithmetics and chemistry simulations. A higher absolute floor might exclude trace quantities of intermediate chemical species. For thirty-two bits, twopnt’s floor is below the roundoff level, so the relative floor passes every component and only the absolute floor discriminates.

$\star$ `TOLER1` $= 0.2$	bound on change of value in each interval	17
$\star$ `TOLER2` $= 0.2$	bound on change of angle at each point	18

These controls apply only to two point boundary value problems with automatic grid selection. They limit the change to the solution occurring at any one grid point by concentrating points in regions where large changes occur. The bounds may range from 0 to 1, that is, from insatiable to no effect. Simulation accuracy and expense increase as the bounds decrease.

Beware that these bounds do not control all simulation errors, and they do not control with certainty what few they try to. See John von Neumann’s four simulation errors in

Chapter 1, and the steps for building a simulation in Chapter 2. Simulation accuracy depends ultimately on the scientific model. Twopnt's controls affect other simulation errors.

simulation step	von Neumann's errors	twopnt's controls
3, solution	approximation, precision	`SSABS` and `SSREL`
2, discretization	approximation	`TOLER1` and `TOLER2`
1, modeling	measurement, model	

Errors even in twopnt's restricted sense are not controlled precisely. The mathematical analysis required to do so varies with each simulation, and is seldom performed. In general, quantitative relationships between accuracy and simulation controls are unknown. Section 2.3 explains the apparent effects on accuracy of the grid selection process controlled by `TOLER1` and `TOLER2`.

Some aspects of a simulation may be particularly sensitive to the grid. In reacting flows for example, these include the locations of flame fronts and the thicknesses of boundary layers. Simulations should be repeated with different tolerances to assess the accuracies of such things. The effects of `TOLER2` seem to be cosmetic, so `TOLER2` might be made less stringent when computing resources are limited.

`TOLER1` and `TOLER2` affect memory space more strongly than computing time. Lower tolerances require larger grids, but these are grids whose problems are solved more easily because good guesses are available. In Chapter 7's simulation for example, Figures 3.2 and 3.3 show continuing from 35 to 76 points requires only 34.5% of all computing time. Memory space and computing time thus increase disproportionately, by 88% and 52% respectively in this example.

`TOLER1` and `TOLER2` affect grid size differently. For example, Figure 3.2 shows `RATIO1` decreases more rapidly with grid size than does `RATIO2`. A simulation stops when it finds grids for which `RATIO1` $\leq$ `TOLER1` and `RATIO2` $\leq$ `TOLER2`. Output like Figures 3.2 and 3.7 indicate how `TOLER1` and `TOLER2` affect grid size, and output like Figure 3.3 indicates how grid size affects computing time.

$\star$ `U` initial guess (● 61, 71)

There is no substitute for a good guess. Simulators either (1) choose guesses themselves based on problem data, or (2) provide controls that specify guesses directly, or (3) provide controls that specify guesses indirectly via expected properties of the solution, or (4) perform sequences of increasingly complex simulations with each supplying the guess for the next, or (5) borrow guesses from other simulations. There is no substitute for a good guess. There is no substitute for a good guess . . .

$\star$ `X` initial grid 16 (● 58, 63, 71)

This control applies only to two point boundary value problems with automatic grid selection. Other simulations may have grids too, and though twopnt has nothing to do with theirs, well chosen grids are no less important for them. Proper grids ease twopnt's work. For example, adding one point to the initial grid of Chapter 7's simulation (in the boundary layer at the disk) halves the computing expense by eliminating the need for time evolution. Simulators may provide controls to specify the initial grid either directly or indirectly through various attributes of the grid such as spacing and the like.

5 Messages

5.0 Introduction

A simulator is an ensemble of many fortran subroutines only some of which write messages. Twopnt's subroutines identify themselves when they write, and their names are mnemonic, so the context of twopnt's messages is clear.

• `TWOPNT`	task selection
• `SEARCH`	Newton's search
• `EVOLVE`	time evolution
• `REFINE`	grid selection

Long, informative messages from these subroutines appear in Sections 3.1 through 3.4, respectively. The error and outcome messages discussed here are shorter.

Outcome messages are needed because "success" and "failure" are not mutually exclusive. For example, twopnt neither succeeds nor fails when it exhausts memory space before finding a satisfactory grid. Twopnt writes messages to explain such outcomes, and it warns simulators of inconclusive results, see Section 6.2. Simulators then must decide what to do. They may continue past unsuccessful preliminary phases of complex simulations, for example, or they may report inconclusive final results in the expectation these are "good enough."

Twopnt fails unconditionally when it encounters programming errors and nonsensical controls. Twopnt checks the consistency of its data as a precaution against such errors, and writes messages like those in **Figure 5.1** when errors are found. Error messages usually occur in groups. The subroutine that discovers the error writes a detailed message first, and then each subroutine in the calling sequence reports successive failure. In this Figure, `SEARCH` discovers a negative bound for a convergence test. `EVOLVE` is the next subroutine to complain, so the offending control is the one for transient state searches, `TDABS`.

All twopnt's messages have the same format. (1) The first line and the first column are blank. The examples in this manual omit these. (2) The name of the writing subroutine begins line two and ends at column eight. (3) The text begins in column eleven and extends no further than column eighty. Simulators usually ignore twopnt's format conventions when they write their own messages, and particularly when they write much data. They, not twopnt, write solution data.

```
SEARCH:  ERROR.  THE BOUNDS FOR THE ABSOLUTE AND RELATIVE
         CONVERGENCE TESTS MUST BE ZERO OR POSITIVE.

          -1.00E+00  SSABS OR TDABS, ABSOLUTE ERROR
           1.00E-06  SSREL OR TDREL, RELATIVE ERROR

EVOLVE:  ERROR.  SEARCH FAILS.

TWOPNT:  ERROR.  EVOLVE FAILS.
```

Figure 5.1. *A sequence of fatal error messages spawned by one error. Section 5.0 explains this Figure.*

5.1 Short List of Messages

This alphabetical, short list of messages displays the first line of every message twopnt writes. It also includes the few messages written by the subroutines below, that are supplied with twopnt but are not part of twopnt, see Section 6.4.

- `TWPREP` matrix preparer
- `TWSHOW` solution data writer
- `TWSOLV` equation solver

Page numbers in the list below refer to the complete texts and detailed explanations of messages found either in Chapter 3 for informative messages, or in Section 5.2 for outcome and error messages. Some messages have identical first lines.

EVOLVE:	BEGIN TIME EVOLUTION.	28
EVOLVE:	BEGIN TIME EVOLUTION.	44
EVOLVE:	CONTINUE TIME EVOLUTION.	44
EVOLVE:	CONTINUE TIME EVOLUTION.	44
EVOLVE:	CONTINUE TIME EVOLUTION WITH INCREASED STRIDE.	44
EVOLVE:	ERROR. NUMBERS OF COMPONENTS AND POINTS MUST BE	44
EVOLVE:	ERROR. SEARCH FAILS.	45
EVOLVE:	ERROR. THE BOUNDS ON THE TIME STRIDE ARE OUT OF	45
EVOLVE:	ERROR. THE COUNT OF TIME STEPS MUST BE ZERO OR	45
EVOLVE:	ERROR. THE FACTORS FOR CHANGING THE TIME STRIDE	45
EVOLVE:	ERROR. THE INITIAL TIME STRIDE MUST LIE BETWEEN	45
EVOLVE:	ERROR. THE NUMBER OF TIME STEPS MUST BE POSITIVE.	45
EVOLVE:	ERROR. THE TIME STEPS BEFORE STRIDE INCREASES	45
EVOLVE:	FAILURE. NO TIME EVOLUTION.	45
EVOLVE:	PARTIAL SUCCESS. TIME EVOLUTION INCOMPLETE.	45
EVOLVE:	PARTIAL SUCCESS. TIME EVOLUTION INCOMPLETE.	45
EVOLVE:	RETRY THE STEP WITH A DECREASED TIME STRIDE.	46
EVOLVE:	SUCCESS. TIME EVOLUTION COMPLETED.	46
EVOLVE:	SUCCESS. TIME EVOLUTION COMPLETED.	46
EVOLVE:	THE LATEST SOLUTION:	46
EVOLVE:	THE SOLUTION DID NOT CHANGE. RETRYING THE STEP	46
REFINE:	ERROR. POINTS IS OUT OF RANGE.	46
REFINE:	ERROR. SOME INTERVALS IN THE GRID ARE TOO SHORT.	46
REFINE:	ERROR. THE BOUNDS ON MAGNITUDE AND RELATIVE CHANGE	46
REFINE:	ERROR. THE BOUNDS ON RELATIVE CHANGES IN MAGNITUDE	46
REFINE:	ERROR. THE GRID IS NOT ORDERED.	46
REFINE:	ERROR. THE LIMIT ON POINTS ADDED TO A GRID MUST BE	46
REFINE:	ERROR. THERE MUST BE AT LEAST ONE COMPONENT AND AT	47
REFINE:	ERROR. THERE ARE NO ACTIVE COMPONENTS.	47
REFINE:	FAILURE. MORE POINTS ARE NEEDED BUT NONE CAN BE	47

```
TWPREP:  ERROR.  THE MATRIX SPACE IS TOO SMALL.                  54
TWSHOW:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE . . .  54
TWSOLV:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE        54
TWSOLV:  ERROR.  THE MATRIX SPACE IS TOO SMALL. . . . . . .      55
```

5.2 Long List of Messages

This list matches Section 5.1's short list but displays the complete text of, and provides explanations for, outcome and error messages. Long, informative messages are cut short with vertical ellipsis, ⋮. Their full texts appear in Chapter 3 on the pages found in Section 5.1. When printing levels increase, these messages break into many short messages in the list below. No explanations are provided for those.

Also, no explanations are provided for error messages that trap bugs. These unlikely messages are precautions against programming mistakes in twopnt. For example, all twopnt's subroutines check to see the number of unknowns is positive. Assuming twopnt contains no mistakes, the checks after the first one are redundant and necessarily find nothing wrong unless the computer itself is faulty, $2 + 2 = 5$.

```
EVOLVE:  BEGIN TIME EVOLUTION.
         ⋮
```

```
EVOLVE:  BEGIN TIME EVOLUTION.

                 0  LATEST TIME POINT
             -1.23  LOG10 STEADY STATE RESIDUAL HERE
             -4.00  LOG10 STRIDE TO NEXT TIME POINT

         SEARCHING FOR THE NEXT TRANSIENT STATE.
```

```
EVOLVE:  CONTINUE TIME EVOLUTION.
         ⋮
```

```
EVOLVE:  CONTINUE TIME EVOLUTION.

               100  LATEST TIME POINT
             -1.23  LOG10 STEADY STATE RESIDUAL HERE
             -3.00  LOG10 STRIDE TO NEXT TIME POINT

         SEARCHING FOR THE NEXT TRANSIENT STATE.
```

```
EVOLVE:  CONTINUE TIME EVOLUTION WITH INCREASED STRIDE.

               100  LATEST TIME POINT
             -1.23  LOG10 STEADY STATE RESIDUAL HERE
             -4.00  LOG10 INCREASED STRIDE TO NEXT TIME POINT

         SEARCHING FOR THE NEXT TRANSIENT STATE.
```

```
EVOLVE:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE
         EITHER BOTH ZERO OR BOTH POSITIVE, NUMBERS OF ALL TYPES
         OF UNKNOWNS MUST BE AT LEAST ZERO, AND TOTAL UNKNOWNS
         MUST BE POSITIVE.

                 0  COMPS, COMPONENTS
                 0  POINTS
                 0  GROUPA, GROUP A UNKNOWNS
                 0  GROUPB, GROUP B UNKNOWNS
                 0  TOTAL UNKNOWNS
```

```
EVOLVE:  ERROR.  SEARCH FAILS.
```

Newton’s search encountered an error when trying to find a transient state. A preceding message from `SEARCH` explains what went wrong.

```
EVOLVE:  ERROR.  THE BOUNDS ON THE TIME STRIDE ARE OUT OF
         ORDER.

          -1.00E-20  TMIN, SHORTEST STRIDE
           1.00E-02  TMAX, LONGEST STRIDE
```

The values given to the controls `TMAX` and `TMIN` are nonsense.

```
EVOLVE:  ERROR.  THE COUNT OF TIME STEPS MUST BE ZERO OR
         POSITIVE.

               -123  STEP
```

```
EVOLVE:  ERROR.  THE FACTORS FOR CHANGING THE TIME STRIDE
         MUST BE NO SMALLER THAN 1.

           3.16E+00  TDEC, DECREASE FACTOR
           0.00E+00  TINC, INCREASE FACTOR
```

The value given to the control `TDEC` or `TINC` is nonsense.

```
EVOLVE:  ERROR.  THE INITIAL TIME STRIDE MUST LIE BETWEEN
         THE LOWER AND UPPER BOUNDS.

           1.00E-20  TMIN, SHORTEST STRIDE
           1.00E-01  STRID0, INITIAL STRIDE
           1.00E-02  TMAX, LONGEST STRIDE
```

The values given to the controls `STRID0`, `TMAX` and `TMIN` are inconsistent.

```
EVOLVE:  ERROR.  THE NUMBER OF TIME STEPS MUST BE POSITIVE.

               -100  STEPS0 OR STEPS1, DESIRED NUMBER OF STEPS
```

The value given to the control `STEPS0` or `STEPS1` is nonsense.

```
EVOLVE:  ERROR.  THE TIME STEPS BEFORE STRIDE INCREASES
         MUST BE POSITIVE.

               -100  STEPS2, TIME STEPS BEFORE STRIDE INCREASES
```

The value given to the control `STEPS2` is nonsense.

```
EVOLVE:  FAILURE.  NO TIME EVOLUTION.
```

Newton’s search was unable to find the first transient state in the current sequence. Depending on the printing level, `LEVELM`, a preceding message from `EVOLVE` or `SEARCH` explains what went wrong.

```
EVOLVE:  PARTIAL SUCCESS.  TIME EVOLUTION INCOMPLETE.
```

Newton’s search was unable to complete the current sequence of transient states. A preceding message from `EVOLVE` explains what went wrong.

```
EVOLVE:  PARTIAL SUCCESS.  TIME EVOLUTION INCOMPLETE.

                 13  LAST TIME POINT
              -0.67  LOG10 STEADY STATE RESIDUAL HERE
```

Newton’s search was unable to complete the current sequence of transient states. A preceding message from `SEARCH` explains what went wrong.

```
EVOLVE:  RETRY THE STEP WITH A DECREASED TIME STRIDE.

                 13  LATEST TIME POINT
              -1.23  LOG10 STEADY STATE RESIDUAL HERE
              -4.00  LOG10 DECREASED STRIDE TO NEXT TIME POINT

         SEARCHING FOR THE NEXT TRANSIENT STATE, AGAIN.
```

```
EVOLVE:  SUCCESS.  TIME EVOLUTION COMPLETED.
```

```
EVOLVE:  SUCCESS.  TIME EVOLUTION COMPLETED.

                200  LAST TIME POINT
              -2.35  LOG10 STEADY STATE RESIDUAL HERE
```

```
EVOLVE:  THE LATEST SOLUTION:
```

```
EVOLVE:  THE SOLUTION DID NOT CHANGE.  RETRYING THE STEP
         WITH AN INCREASED TIME STRIDE.

                 13  LATEST TIME POINT
              -0.67  LOG10 STEADY STATE RESIDUAL HERE
              -4.00  LOG10 INCREASED STRIDE TO NEXT TIME POINT

         SEARCHING FOR THE NEXT TRANSIENT STATE, AGAIN.
```

```
REFINE:  ERROR.  POINTS IS OUT OF RANGE.

                  8  POINTS
               -123  PMAX, LIMIT ON POINTS
```

```
REFINE:  ERROR.  SOME INTERVALS IN THE GRID ARE TOO SHORT.
         THE NEW GRID WOULD NOT BE ORDERED.
```

The grid is so fine that, if the interval between two points is halved, then imprecise machine arithmetic places the midpoint at an endpoint. This outcome might be considered merely unsuccessful, but twopnt views it as an error because it is so rare.

```
REFINE:  ERROR.  THE BOUNDS ON MAGNITUDE AND RELATIVE CHANGE
         OF MAGNITUDE FOR INSIGNIFICANT COMPONENTS MUST BE
         POSITIVE.

           0.00E-09  TOLER0, SIGNIFICANCE LEVEL
```

The value given to the control `TOLER0` is nonsense.

```
REFINE:  ERROR.  THE BOUNDS ON RELATIVE CHANGES IN MAGNITUDE
         AND ANGLE MUST LIE BETWEEN 0 AND 1.

          -1.00E-01  TOLER1
           1.00E-01  TOLER2
```

The value given to the control `TOLER1` or `TOLER2` is nonsense.

```
REFINE:  ERROR.  THE GRID IS NOT ORDERED.
```

The initial grid, `X`, provided by the simulator should be a decreasing or an increasing sequence of numbers without duplications.

```
REFINE:  ERROR.  THE LIMIT ON POINTS ADDED TO A GRID MUST BE
         ZERO OR POSITIVE.

               -123  PADD, LIMIT ON ADDED POINTS
```

The value given to the control `PADD` is nonsense.

```
REFINE:  ERROR.  THERE MUST BE AT LEAST ONE COMPONENT AND AT
         LEAST TWO POINTS.

                  0  COMPS, COMPONENTS
                  0  POINTS
```

The values given to the controls `ADAPT`, `COMPS` and `POINTS` are inconsistent. `ADAPT` = *yes* requests automatic grid selection, but the other two controls indicate the unknowns are not arranged to permit grid selection.

```
REFINE:  ERROR.  THERE ARE NO ACTIVE COMPONENTS.
```

The values given to the controls `ADAPT` and `ACTIVE` are inconsistent. `ADAPT` = *yes* requests automatic grid selection, but `ACTIVE` excludes all solution components from consideration for grid selection.

```
REFINE:  FAILURE.  MORE POINTS ARE NEEDED BUT NONE CAN BE
         ADDED.
```

The solution doesn't meet the requirements established by `TOLER1` and `TOLER2`, but a new grid cannot be formed because either `PADD=0` or `POINTS=PMAX`.

```
REFINE:  SELECT A GRID.
```

```
REFINE:  SUCCESS.  THE GRID IS ADEQUATE.
```

```
REFINE:  SUCCESS.  THE GRID IS ADEQUATE BECAUSE ALL ACTIVE
         COMPONENTS ARE INSIGNIFICANT.
```

According to the criteria established by `TOLER0`, all the `ACTIVE` solution components are insignificant—either very small or very "flat" across the grid—so there are no features to guide the selection of a new grid.

```
REFINE:  THE SOLUTION GUESS FOR THE NEW GRID:
```

```
SEARCH:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE
         EITHER BOTH ZERO OR BOTH POSITIVE, NUMBERS OF ALL TYPES
         OF UNKNOWNS MUST BE AT LEAST ZERO, AND TOTAL UNKNOWNS
         MUST BE POSITIVE.

                  5  COMPS, COMPONENTS
                  8  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                -10  GROUPB, GROUP B UNKNOWNS
                 30  TOTAL UNKNOWNS
```

```
SEARCH:  ERROR.  THE BOUNDS FOR THE ABSOLUTE AND RELATIVE
         CONVERGENCE TESTS MUST BE ZERO OR POSITIVE.

          -1.00E-09  SSABS OR TDABS, ABSOLUTE ERROR
           1.00E-06  SSREL OR TDREL, RELATIVE ERROR
```

The values given to the controls `SSABS` and `SSREL`, or to `TDABS` and `TDREL`, are nonsense. If `TWOPNT` writes the next error message then `SSABS` and `SSREL` are wrong, but if `EVOLVE` writes next then `TDABS` and `TDREL` are at fault.

```
SEARCH:  ERROR.  THE DAMPING COEFFICIENT FOR STAYING
         IN BOUNDS IS NEGATIVE.

          -1.00E+00  DELTA B
```

```
SEARCH:  ERROR.  THE GUESSES FOR SOME UNKNOWNS ARE OUT OF
         BOUNDS.

                  0  GROUP A UNKNOWNS (A)
                  0  GROUP B UNKNOWNS (B)
                  5  COMPONENTS AT POINTS (C)
                  6  POINTS (P)
                 30  TOTAL UNKNOWNS
                  2  NUMBER OUT OF BOUNDS

              LOWER                  UPPER
              BOUND        VALUE     BOUND   UNKNOWN

           1.50E+02    3.00E+03   2.00E+03   T  (C 5 P 1)
          -1.00E+04   -1.00E+03   1.00E+04   G  (C 2 P 3)
```

This error suggests either a programming mistake or faulty input to a simulator. Some guesses violate their bounds, `ABOVE` or `BELOW`. Either the bounds or the guesses must change. Simulators often assign the same values to many bounds and guesses, so many may be wrong if any are. Twopnt lists only the first twenty. A simulator may provide names for the unknowns, and twopnt adds this identification: `(A 6)` is the sixth unknown in group A, and `(C 2 P 3)` is the second component at the third point, and so on.

```
SEARCH:  ERROR.  THE LOWER AND UPPER BOUNDS ON SOME UNKNOWNS
         ARE OUT OF ORDER.

                  0  GROUP A UNKNOWNS (A)
                  0  GROUP B UNKNOWNS (B)
                  5  COMPONENTS AT POINTS (C)
                  5  TOTAL TYPES OF UNKNOWNS
                  2  NUMBER OF BOUNDS OUT OF ORDER

              LOWER       UPPER
              BOUND       BOUND   UNKNOWN

          -1.00E+04   -1.00E+04   LAMBDA  (C 4)
           1.50E+02    0.00E+00   T  (C 5)
```

```
SEARCH:  ERROR.  THE NUMBER OF NAMES IS WRONG.

                  5  NAMES

                  5  COMPS, COMPONENTS
                  3  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                  8  TOTAL NUMBER
```

```
SEARCH:  ERROR.  THE RETIREMENT AGE OF THE JACOBIAN MATRIX
         MUST BE POSITIVE.

                -10  SSAGE OR TDAGE, MATRIX RETIREMENT AGE
```

The value given to the control `SSAGE` or `TDAGE` is nonsense. If `TWOPNT` writes the next error message then `SSAGE` is wrong, but if `EVOLVE` writes next then `TDAGE` is at fault.

```
SEARCH:  FAILURE.  THE SEARCH DIVERGES.
```

```
SEARCH:  FAILURE.  THE SEARCH FOR THE FOLLOWING UNKNOWNS GOES
         OUT OF BOUNDS.

         BOUND        VALUE   UNKNOWN

         LOWER   -2.00E+00   H  (C 3 P 2)
```

Newton's search fails because the first convergence monitor discussed in Section 2.1 cannot be met. Twopnt stops values headed out of bounds at the boundary. Usually one or two unknowns arrive there before the others, so twopnt's list of those going out is short. A simulator may provide names to identify the unknowns, to which twopnt adds this annotation: `(A 6)` indicates the sixth unknown in group A, and `(C 3 P 2)` indicates the third component at the second point, and so on.

Newton's search reaches bounds from bad luck, or more likely, from bad guesses. Both the bounds and the guesses embody expectations: when a usually successful simulation repeatedly fails at bounds, it may be operating under conditions not anticipated. Twopnt's controls `ABOVE` and `BELOW` can relax the bounds. Better guesses can be obtained from performing time evolution (which twopnt performs automatically) or from performing easier simulations (which some simulators might provide automatically).

```
SEARCH:  SOLVE NONLINEAR, NONDIFFERENTIAL EQUATIONS.
```

```
SEARCH:  SUCCESS.
```

```
SEARCH:  SUCCESS.  THE SOLUTION:
```

```
TWOPNT:  17.34 SECONDS TOTAL COMPUTER TIME (SEE BREAKDOWN BELOW).
```

```
TWOPNT:  CALLING EVOLVE TO PERFORM TIME EVOLUTION.
```

```
TWOPNT:  CALLING REFINE TO PRODUCE A NEW GRID.
```

```
TWOPNT:  CALLING SEARCH TO SOLVE THE STEADY STATE PROBLEM.
```

```
TWOPNT:  DOUBLE PRECISION (TWO POINT BOUNDARY VALUE PROBLEM) SOLVER,
         VERSION 3.07 OF JANUARY 1992 BY DR. JOSEPH F. GRCAR.
```

```
TWOPNT:  ERROR.  A CONTROL NAME IS NOT RECOGNIZED.

                  3  POSITION IN THE CONTROL LIST
                  6  CNTRLS, LENGTH OF THE CONTROL LIST

              NAME:  ADEPT
```

Simulators pass lists of control names and values to twopnt, see Section 6.2. The simulator either spelled a name wrong or expects some version of twopnt with different control names.

```
TWOPNT:  ERROR.  EVOLVE FAILS.
```

```
TWOPNT:  ERROR.  NEITHER THE INITIAL TIME EVOLUTION NOR THE
         SEARCH FOR THE STEADY STATE IS ALLOWED.
```

The values given to the controls `STEADY` and `STEPS0` are inconsistent. If `STEADY` $=$ *no* and `STEPS0` $= 0$, then twopnt has nothing to do.

```
TWOPNT:  ERROR.  NUMBERS OF ALL TYPES OF UNKNOWNS MUST BE AT
         LEAST ZERO.

                  5  COMPS, COMPONENTS
                  8  POINTS
                 -1  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
```

The values given to the controls `COMPS`, `POINTS`, `GROUPA` or `GROUPB` are nonsense. The simulator may have a programming mistake, or may have been given incorrect data.

```
TWOPNT:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE
         EITHER BOTH ZERO OR BOTH POSITIVE.

                  5  COMPS, COMPONENTS
                  0  POINTS
```

The values given to the controls `COMPS` and `POINTS` are inconsistent. The simulator may have a programming mistake, or it may have been given incorrect data.

```
TWOPNT:  ERROR.  ONE OR BOTH WORK SPACES ARE TOO SMALL.

                             INTEGER        REAL

          PRESENT SIZE          5000      100000
         REQUIRED SIZE          7395       63921
```

The simulator has been given a problem that requires too much memory. Either the program must be changed to include more space, or the problem must be made smaller. Many simulators check memory size themselves, based on twopnt's announced memory needs in Chapter 6. Thus, a simulator reaching this error might have a programming mistake.

```
TWOPNT:  ERROR.  REFINE FAILS.
```

```
TWOPNT:  ERROR.  SEARCH FAILS.
```

```
TWOPNT:  ERROR.  THE CALLING PROGRAM EXPECTS A VERSION OF
         TWOPNT NOT COMPATIBLE WITH THIS VERSION.

              EXPECTS:  DOUBLE PRECISION VERSION 3.08

         THIS VERSION:  SINGLE PRECISION VERISON 3.02
          CAN REPLACE:  SINGLE PRECISION VERISON 3.01
          CAN REPLACE:  SINGLE PRECISION VERISON 3.00
```

The simulator expects a different version of twopnt. Either the arithmetic precision or the version number is wrong. If the precision, then Appendix 2 explains how to change that. If the version, then a different twopnt or a different simulator must be obtained. The message lists all older twopnts compatible with the present one, but some newer twopnts might be compatible too. At some risk of catastrophe, the simulator program might be changed to identify the present twopnt as the one expected.

```
TWOPNT:  ERROR.  THE LOWER AND UPPER BOUNDS ON SOME UNKNOWNS
         ARE OUT OF ORDER.

                  0  GROUP A UNKNOWNS (A)
                  0  GROUP B UNKNOWNS (B)
                  5  COMPONENTS AT POINTS (C)
                  5  TOTAL TYPES OF UNKNOWNS
                  2  NUMBER OF BOUNDS OUT OF ORDER

              LOWER       UPPER
              BOUND       BOUND   UNKNOWN

          -1.00E+04   -1.00E+04   LAMBDA  (C 4)
           1.50E+02    0.00E+00   T  (C 5)
```

This error suggests either a programming mistake or faulty input to a simulator. Some values given to the controls `ABOVE` and `BELOW` are inconsistent. Simulators often assign the same values to many bounds, so many may be wrong if any are. Twopnt lists only the first twenty. A simulator may provide names to identify the unknowns, but in any case, twopnt includes the following annotation: `(A 6)` indicates the sixth unknown in group A, and `(C 4)` indicates the fourth component at the points (a component has the same bounds at all points), and so on.

```
TWOPNT:  ERROR.  THE NUMBER OF CONTROLS MUST BE POSITIVE.

                  0  CNTRLS, LENGTH OF THE CONTROL LIST
```

Simulators pass lists of control names and values to twopnt. This error occurs when a simulator passes none, and may indicate a programming mistake.

```
TWOPNT:  ERROR.  THE NUMBER OF NAMES IS WRONG.

                  3  NAMES

                  3  COMPS, COMPONENTS
                  7  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                 10  TOTAL NUMBER
```

This error indicates a programming mistake. Simulators may pass lists of names for unknowns to twopnt. If so, the number of names must be `COMPS` + `GROUPA` + `GROUPB`.

```
TWOPNT:  ERROR.  THE PRINTING LEVELS ARE OUT OF ORDER.
         LEVELD CANNOT EXCEED LEVELM.

                  2  LEVELD, FOR SOLUTIONS
                  1  LEVELM, FOR MESSAGES
```

The values given to the controls `LEVELM` and `LEVELD` are inconsistent.

```
TWOPNT:  ERROR.  THERE ARE TOO MANY POINTS.

                591  POINTS
                100  PMAX, LIMIT ON POINTS
```

The values given to the controls `POINTS` and `PMAX` are inconsistent. This error may indicate a programming mistake.

```
TWOPNT:  ERROR.  TOTAL UNKNOWNS MUST BE POSITIVE.

                  0  COMPS, COMPONENTS
                  0  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                  0  TOTAL NUMBER
```

The values given to the controls `COMPS`, `POINTS`, `GROUPA` or `GROUPB` are nonsense. Since these controls determine numbers of unknowns, this error indicates a programming mistake.

```
TWOPNT:  ERROR.  TWGRAB FAILS.
```

```
TWOPNT:  ERROR.  UNKNOWN TASK.
```

```
TWOPNT:  ERROR.  UNKNOWN REPORT CODE.
```

```
TWOPNT:  EVOLVE DID NOT PERFORM A TIME EVOLUTION.
```

Newton's search was unable to find the first transient state in the current sequence. Depending on the printing level, `LEVELM`, a preceding message from `EVOLVE` or `SEARCH` explains what went wrong.

```
TWOPNT:  EVOLVE PERFORMED A TIME EVOLUTION.
```

```
TWOPNT:  FAILURE.  A SOLUTION WAS FOUND FOR A GRID WITH 67              1
         POINTS, BUT ONE OR BOTH RATIOS ARE TOO LARGE.                  2
                                                                        3
                        RATIO 1     RATIO 2                             4
                                                                        5
              FOUND        0.31        0.78                             6
            DESIRED        0.50        0.50                             7
                                                                        8
         A LARGER GRID COULD NOT BE FORMED.                             9
```

Twopnt did not find a sufficiently good solution. A larger grid cannot be formed because either `PADD=0` or `POINTS=PMAX`. The need for a larger grid can be eliminated by increasing `TOLER1` or `TOLER2` to the values shown (LINE 6).

```
TWOPNT:  FAILURE.  A SOLUTION WAS FOUND FOR A GRID WITH 67              1
         POINTS, BUT ONE OR BOTH RATIOS ARE TOO LARGE.                  2
                                                                        3
                        RATIO 1     RATIO 2                             4
                                                                        5
              FOUND        0.31        0.78                             6
            DESIRED        0.50        0.50                             7
                                                                        8
         A SOLUTION COULD NOT BE FOUND FOR A LARGER GRID.               9
```

Twopnt did not find a sufficiently good solution. Solutions were found for smaller grids, but not for the largest, indicating the smaller grids do not resolve some critical solution feature. Twopnt's other messages may suggest how to adjust the controls to find the solution for the largest grid, or the sequence of grids can be stopped short of the largest by increasing `TOLER1` or `TOLER2` to the values shown (LINE 6).

```
TWOPNT:  FAILURE.  NO SOLUTION WAS FOUND.                               1
```

Twopnt did not find a solution. Twopnt's other messages may suggest how to adjust the controls for Newton's search and time evolution so twopnt can succeed.

```
TWOPNT:  FINAL SOLUTION:                                                1
```

```
TWOPNT:  INITIAL GUESS:                                                 1
```

```
TWOPNT:  REFINE DID NOT SELECT A NEW GRID.                              1
```

```
TWOPNT:  REFINE SELECTED A NEW GRID.                                    1
```

```
TWOPNT:  SEARCH DID NOT FIND THE STEADY STATE.                          1
```

```
TWOPNT:  SEARCH FOUND THE STEADY STATE.                                 1
```

```
TWOPNT:  SINGLE PRECISION (TWO POINT BOUNDARY VALUE PROBLEM) SOLVER,    1
         VERSION 3.07 OF JANUARY 1992 BY DR. JOSEPH F. GRCAR.           2
```

```
TWOPNT:  SOLVE THE PROBLEM.                                             1
```

```
TWOPNT:  SUCCESS.  PROBLEM SOLVED.                                      1
```

```
TWPREP:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE               1
         EITHER BOTH ZERO OR BOTH POSITIVE, NUMBERS OF ALL TYPES        2
         OF UNKNOWNS MUST BE AT LEAST ZERO, AND TOTAL UNKNOWNS          3
         MUST BE POSITIVE.                                              4
                                                                        5
                 0  COMPS, COMPONENTS                                   6
                 8  POINTS                                              7
                31  GROUPA, GROUP A UNKNOWNS                            8
                 0  GROUPB, GROUP B UNKNOWNS                            9
                31  TOTAL UNKNOWNS                                     10
```

The values given to the controls COMPS, POINTS, GROUPA or GROUPB are nonsense. Since TWOPNT should have checked these values already, there may be a programming mistake.

```
TWPREP:  ERROR.  SOME COLUMNS ARE ZERO.

                  5  COMPS, COMPONENTS
                  8  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                 40  TOTAL COLUMNS
                  4  ZERO COLUMNS

         UNKNOWNS WITH ZERO COLUMNS:

         COMPONENT 2 AT POINT 1
         COMPONENT 2 AT POINT 2
         COMPONENT 2 AT POINT 3
         COMPONENT 2 AT POINT 4
```

The Jacobian matrix has a zero column. There are two causes. (1) The residual function is independent of some unknown. Usually, this is a programming mistake. The residual function should be examined to see every unknown is used. In rare cases, special values for some unknowns might make the residual function insensitive to other unknowns, and then refer to the second cause. (2) The numerical perturbations used to approximate the differentials might be too small to perturb the residual function. Either the residual function should be reformulated, or the perturbations used by TWPREP should be increased.

```
TWPREP:  ERROR.  SOME ROWS ARE ZERO.

                  5  COMPS, COMPONENTS
                  8  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                 40  TOTAL ROWS
                  2  ZERO ROWS

         ZERO ROWS:

         COMPONENT 5 AT POINT 1
         COMPONENT 5 AT POINT 8
```

The Jacobian matrix has a zero row. There are two causes. (1) Some residual value is independent of all unknowns. Usually, this error indicates a programming mistake. The residual function should be examined to see every equation is evaluated. In rare cases, special values for the unknowns might make some equations insensitive to the unknowns, and then refer to the second cause. (2) The numerical perturbations used to approximate the differentials might be too small to perturb some equation. Either the residual function should be reformulated, or the perturbations used by TWPREP should be increased.

```
TWPREP:  ERROR.  THE JACOBIAN MATRIX IS SINGULAR.
```

The linpack subroutine DGBCO or SGBCO [8] finds a singular Jacobian matrix. The matrix may be very badly conditioned rather than exactly singular, but in either case, the matrix equations for the search directions in Newton's method cannot be solved. If this error occurs at the start of a simulation, then usually either (1) the initial guess is bad or (2) the simulation is formulated incorrectly.

When this error occurs late in the simulation, the cause is usually one of two others. (3) The search has strayed too far. Twopnt's controls might be changed to make the search more conservative. (4) The simulation is near a turning point or a bifurcation point. This means the simulation has two or more valid outcomes for the same data. Twopnt's controls might be changed to coax twopnt to one outcome, but such simulations are inherently difficult to perform. Either the simulation should be reformulated, or twopnt should be replaced by continuation and path following software which can find all the multiple outcomes, see [24].

```
TWPREP:  ERROR.  THE MATRIX SPACE IS TOO SMALL.

                  5  COMPS, COMPONENTS
                  8  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                 40  MATRIX ORDER
                  9  STRICT HALF BANDWIDTH

               1160  SPACE REQUIRED
               1000  ASIZE, PROVIDED
```

The matrix requires more memory than the simulator allows. Either the simulator must be changed to include more space, or the problem must be changed to make the matrix smaller. Many simulators check memory size themselves, based on TWPREP's announced memory needs in Section 6.4. Thus, a simulator reaching this error might have a programming mistake.

```
TWSHOW:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE
         EITHER BOTH ZERO OR BOTH POSITIVE, NUMBERS OF ALL TYPES
         OF UNKNOWNS MUST BE AT LEAST ZERO, AND TOTAL UNKNOWNS
         MUST BE POSITIVE.

                  5  COMPS, COMPONENTS
                  0  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                  0  TOTAL UNKNOWNS
```

The values given to the controls COMPS, POINTS, GROUPA or GROUPB are nonsense. The simulator may have a programming mistake.

```
TWSOLV:  ERROR.  NUMBERS OF COMPONENTS AND POINTS MUST BE
         EITHER BOTH ZERO OR BOTH POSITIVE, NUMBERS OF ALL TYPES
         OF UNKNOWNS MUST BE AT LEAST ZERO, AND TOTAL UNKNOWNS
         MUST BE POSITIVE.

                  5  COMPS, COMPONENTS
                  0  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                 10  GROUPB, GROUP B UNKNOWNS
                 10  TOTAL UNKNOWNS
```

The values given to the controls COMPS, POINTS, GROUPA or GROUPB are nonsense. Since TWOPNT and TWPREP should have checked these values already, there may be a programming mistake.

```
TWSOLV:  ERROR.  THE MATRIX SPACE IS TOO SMALL.

                  5  COMPS, COMPONENTS
                  8  POINTS
                  0  GROUPA, GROUP A UNKNOWNS
                  0  GROUPB, GROUP B UNKNOWNS
                 40  MATRIX ORDER
                  9  STRICT HALF BANDWIDTH

               1160  SPACE EXPECTED
               1000  ASIZE, PROVIDED
```

The equation solving subroutine has been given a matrix that seems to require more memory than the simulator allows. Since `TWPREP` should have checked this space when it prepared the matrix, there may be a programming mistake.

(C) Now let mathematical formulation and observational data go unquestioned. The next stumbling block is this: The mathematical formulation will in general involve transcendental operations (for example, functions like sin or log, operations like integration or differentiation, and so on) and implicit definitions (for example, solutions of algebraical or transcendental equations, proper value [eigenvalue] problems of various kinds, and so on). In order to be approached by numerical calculation, these have to be replaced by elementary processes (involving only those elementary arithmetical operations which the computer can handle directly) and explicit definitions, which correspond to a finite, constructive procedure that resolves itself into a linear sequence of steps.

Similarly, every convergent, limiting process, which in its strict mathematical form is infinite, must in a numerical computation be broken off at some final stage, where the approximation to the limiting value is known to have reached a level that is considered to be satisfactory. It would be easy to give further examples.

— J. von Neumann and H. H. Goldstine [26]

6 Writing a Simulator

6.0 Introduction

A simulator based on twopnt has three major parts. One gathers and reports data. Two evaluates a residual. Three calls twopnt. An entirely new simulator might be written step by step, by adding the software needed to support each part. Alternatively, a simulator might be written by modifying someone else's. To that end, twopnt's software distribution includes the rudimentary simulator whose output appears throughout this manual, see Appendix 2. That simulation may have independent interest, so Chapter 7 discusses it separately. Section 6.1 here, and Section 7.3 there, illustrate the programming needed to use twopnt.

Twopnt is a collection of fortran subroutines [1]. Appendix 2 lists them all. The intended entry point is the subroutine with the mnemonic name and the twenty-nine arguments shown in **Figure 6.1**. Some arguments have been introduced elsewhere as controls, but in general, controls are not arguments. Sections 6.1 and 6.2 discuss groups of arguments with similar uses, while Section 6.3 explains each argument alone.

6.1 Calling Twopnt

A simulation is a scientific model and an evaluation algorithm. Its software consists of a simulator which poses numerical problems, and a solver which solves them. The simulator formulates the problem and interprets the solution, by doing such things as gathering parameters and drawing graphs. Some simulations can make do with "black box" solvers, and then the software has simple control and communication paths: the simulator gives problems to the solver, and the solver gives solutions back.

Software for complex simulations is rarely tidy. (1) Parameters may be numerous and not easily communicated. (2) Problem formulation and solution may be special and inseparable. (3) Solutions may be obtained by expensive methods tuned to each problem. In these situations, the distinction between simulator and solver can be lost, and then the experience gained from successful simulations cannot be transferred easily.

Twopnt is a solver for complex simulations that deals with the difficulties above by delegating many tasks back to the simulator. In some sense, the simulator serves twopnt because it performs much of the numerical work under twopnt's direction. Twopnt is

```
     SUBROUTINE TWOPNT
    +  (ERROR, TEXT, VERSIO,
    +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
    +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
    +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
    +   STRIDE, TIME, U, X)
```

Figure 6.1. *Twopnt's arguments with those used for reverse communication highlighted. Section 6.1 explains reverse communication.*

intended to serve many simulators without modification to itself, so it omits features that would limit its use. For example, other software for differential equations choose both the discretization and, when the discretization is implicit, the linear algebra. Twopnt chooses neither because no choice is appropriate for all simulations. This means simulators using twopnt must provide their own discretization and, under twopnt's direction, must perform their own linear algebra. Since the simulator has temporal precedence, it is necessarily the calling program, but the control and communication paths between twopnt and the simulator are elaborate.

Twopnt passes commands back to the simulator by "*reverse communication.*" That is, twopnt's arguments include one that signals the simulator to perform some task and then to call twopnt again. Figure 6.1 highlights this and related arguments in twopnt's calling sequence. Briefly, twopnt begins a new problem when called with `SIGNAL` *blank*, and if twopnt returns with `SIGNAL` not *blank*, then some task must be performed.

• `SIGNAL`	reverse communication signal	69
• `BUFFER`	array for reverse communication data	61, 66

The tasks operate on data twopnt places in `BUFFER`, and if twopnt expects some result, then twopnt generally receives it there too.

Twopnt asks three major tasks of the simulator. One is to evaluate the residual function.

• `SIGNAL` = `` `RESIDUAL' ``	signal to evaluate the residual

Evaluation should occur at the approximate solution found in `BUFFER`, and twopnt wants the residual vector there when it resumes. This task needs more information than `BUFFER` can supply because there are two residual functions.

• `TIME`	signal to use the transient residual	71
• `STRIDE`	time stride for the transient residual	71

The transient residual, moreover, needs earlier transient states to approximate time differentials, so a supporting task identifies those.

• `SIGNAL` = `` `RETAIN' ``	signal to retain a transient state

When the simulator receives this signal, it must store the transient state found in `BUFFER` for later use in evaluating the transient residual.

The other two major tasks twopnt asks of the simulator involve the matrix equations in Newton's method.

• `SIGNAL` = `` `PREPARE' ``	signal to prepare the Jacobian matrix	
• `CONDIT`	matrix condition number	67
• `SIGNAL` = `` `SOLVE' ``	signal to solve matrix equations	

Simulation designers prefer direct matrix operations, so simulators usually form and factor the Jacobian matrix when signaled to prepare. Alternatively, they might build preconditioners for iterative equation solving algorithms. The Jacobian matrix should be evaluated at the solution estimate in `BUFFER`, and should be for either the steady state or the transient residual as indicated by `TIME`. Twopnt needs no direct knowledge of the matrix, so the simulator can store what it prepares where and how it likes. If a condition estimate is available for the matrix, then that should be placed in `CONDIT` when twopnt resumes. The `SOLVE` signal indicates the most recently prepared matrix should be used to solve linear equations with the "right side," or constant terms, found in `BUFFER`. When twopnt resumes, it expects to find the solution in `BUFFER` too.

The matrix tasks are the most onerous twopnt asks of the simulator. Many solvers undertake such tasks internally, but then unlike twopnt, they need nontrivial reprogramming when matrix operations need revising. Twopnt includes subroutines, `TWPREP` and `TWSOLV`, that the simulator can use for these tasks if desired. These routines are discussed later in this Section and again in Section 6.4.

The remaining three tasks requested by reverse communication are less strenuous. (1) Simulators may want to prepare some things ahead when twopnt advances to a new grid.

• `SIGNAL` = `` `UPDATE' ``	signal to update to a new grid	
• `POINTS`	number of points	62, 68 (⋆ 16, 35)
• `MARK`	array of markers for new points	63, 68
• `X`	array of grid points	63, 71 (⋆ 16, 40)

For example, twopnt constructs new grids by halving intervals, and it guesses new solution values by averaging old ones. If more than linear interpolation is wanted, then simulators must provide it. (2) Simulators like to save work in progress in case machines fail during long computations.

• `SIGNAL` = `` `SAVE' ``	signal to save the solution

The latest, best solution estimate in `BUFFER` should be saved; it can serve as a guess to begin anew. If twopnt is choosing grids, then `POINTS` and `X` should be saved too. (3) Finally, twopnt has controls that govern writing solutions, but the simulator does the actual writing.

• `SIGNAL` = `` `SHOW' ``	signal to show the solution

The solution data in `BUFFER` should be written to the fortran unit number, `TEXT`, where twopnt writes it own messages. This output is what appears when `LEVELD` > 0. Twopnt includes a subroutine, `TWSHOW`, that can do the simulator's writing, if desired.

Figure 6.2 shows what is needed to support reverse communication. This program fragment might be used as a template for calling twopnt. It sets the reverse communication signal *blank* to begin (LINE 1), and when twopnt returns, the signal explains what to do (LINES 11, 15, . . .). The program performs one of six chores—`RESIDUAL`, `PREPARE`, `SOLVE`, `SHOW`, `RETAIN` and `UPDATE` (it ignores `SAVE`)—and then it calls twopnt again with `SIGNAL` not *blank* (LINE 43).

The program fragment in Figure 6.2 delegates the `RESIDUAL` task to a subroutine (LINE 12). This is the heart of the simulation because it embodies the scientific model. **Figure 6.3** shows the arguments for such a subroutine, `TWFUNC`, which Chapter 7 describes fully. The subroutine's arguments include parameters for the model and parameters for the discretization. The most important is `BUFFER`, which is where twopnt places the solution estimate, and where the subroutine places the residual.

The program fragment in Figure 6.2 leaves the matrix tasks to the `TWPREP` and `TWSOLV` subroutines supplied with twopnt (LINES 18 AND 28). **Figure 6.4** depicts the

```
      SIGNAL = ' '

0100  CALL TWOPNT
     +  (ERROR, TEXT, 'DOUBLE PRECISION VERSION 3.08',
     +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
     +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
     +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
     +   STRIDE, TIME, U, X)
      IF (ERROR) GO TO 9001

      IF (SIGNAL .EQ. 'RESIDUAL') THEN
         CALL TWFUNC (ERROR, TEXT, BUFFER, ...)
         IF (ERROR) GO TO 9002

      ELSE IF (SIGNAL .EQ. 'PREPARE') THEN
         RETURN = .FALSE.

0200     CALL TWPREP (ERROR, TEXT, A, ... BUFFER, ... RETURN, ...)
         IF (ERROR) GO TO 9003

         IF (RETURN) THEN
            CALL TWFUNC (ERROR, TEXT, BUFFER, ...)
            IF (ERROR) GO TO 9002
            GO TO 0200
         END IF

      ELSE IF (SIGNAL .EQ. 'SOLVE') THEN
         CALL TWSOLV (ERROR, TEXT, A, ... BUFFER, ...)
         IF (ERROR) GO TO 9005

      ELSE IF (SIGNAL .EQ. 'SHOW') THEN
         CALL TWSHOW (ERROR, TEXT, BUFFER, ...)
         IF (ERROR) GO TO 9004

      ELSE IF (SIGNAL .EQ. 'RETAIN') THEN
         CALL TWCOPY (N, BUFFER, U0)

      ELSE IF (SIGNAL .EQ. 'UPDATE') THEN
         N = GROUPA + COMPS * POINTS + GROUPB

      END IF

      IF (SIGNAL .NE. ' ') GO TO 0100
```

Figure 6.2. *Segment of a fortran program calling twopnt and illustrating the use of reverse communication. Section 6.1 explains the program.*

```
      SUBROUTINE TWFUNC
     +  (ERROR, TEXT,
     +   BUFFER, F, F0, G, G0, H, K, LAMBDA, MU, OMEGA, POINTS, RHO,
     +   STRIDE, T, T0, TIME, TMAX, TZERO, U0, WMAX, X)
```

Figure 6.3. *Typically idiosyncratic arguments for a subroutine evaluating a simulation's residual function. Figure 6.2 shows the use of such a subroutine in calling twopnt. Section 7.3 explains this particular subroutine, which is supplied with twopnt.*

Figure 6.4. *The* TWPREP *and* TWSOLV *subroutines that are supplied with twopnt support a tridiagonal, blocked arrangement of nonzeroes in the Jacobian matrices. All blocks must have the same size except, optionally, the extremes. Section 6.1 describes these subroutines briefly. Section 6.4 describes them in detail.*

type of Jacobian matrices these subroutines expect. This tridiagonal, blocked structure predominates in two point boundary value problems. If a simulator has different looking matrices, then it must respond to the PREPARE and SOLVE requests by doing its own thing. Section 6.3 explains what that might be.

The TWPREP subroutine uses reverse communication too. A small portion of Figure 6.2 (LINES 16 TO 25) thus embeds one reverse communication system within another. Section 6.4 explains TWPREP's arguments in detail. Briefly, RETURN is its reverse communication signal, and function evaluation is its only request, hence the second use of TWFUNC (LINE 22). TWPREP stores the matrix in the large array A that likely accounts for most of the simulator's storage.

```
TWOPNT:  FINAL SOLUTION:


           GRID POINT     COMP 1     COMP 2     COMP 3     COMP 4     COMP 5
      1>    0.000000  0.000E+00  1.257E+01  0.000E+00  8.562E-05  3.000E+02
      2>    0.000977  5.783E-02  1.250E+01 -5.665E-05  8.562E-05  3.018E+02
      3>    0.001953  1.143E-01  1.243E+01 -2.256E-04  8.562E-05  3.037E+02
      4>    0.002930  1.696E-01  1.237E+01 -5.051E-04  8.562E-05  3.055E+02
      5>    0.003906  2.236E-01  1.230E+01 -8.932E-04  8.562E-05  3.073E+02
     70>    2.250000 -2.287E-01  1.404E-02 -2.629E+00  8.562E-05  9.904E+02
     71>    2.500000 -2.128E-01  7.534E-03 -2.527E+00  8.562E-05  9.935E+02
     72>    3.000000 -1.733E-01  2.655E-03 -2.342E+00  8.562E-05  9.969E+02
     73>    3.500000 -1.309E-01  9.656E-04 -2.194E+00  8.562E-05  9.985E+02
     74>    4.000000 -8.759E-02  3.376E-04 -2.086E+00  8.562E-05  9.993E+02
     75>    4.500000 -4.386E-02  9.496E-05 -2.021E+00  8.562E-05  9.998E+02
     76>    5.000000  0.000E+00  0.000E+00 -2.000E+00  8.562E-05  1.000E+03
```

Figure 6.5. *The portion of* LEVELD > 0 *output prepared by* TWSHOW *that describes the final result of the simulation in Chapter 7. Figure 2.2 plots some of this data. Components 1, 2, 3 and 5 here are F, G, H and T there. Section 6.1 explains* TWSHOW*'s use, while Section 6.4 explains the subroutine in detail.*

The program fragment in Figure 6.2 leaves the SHOW task to the TWSHOW subroutine (LINE 32) also supplied with twopnt and also explained in Section 6.4. **Figure 6.5** shows what is shown, if LEVELD > 0, after the simulation discussed in Chapter 7 finishes. TWSHOW does not name solution data, but it can be changed easily to do so.

The program fragment in Figure 6.2 performs the RETAIN task by copying the proffered transient state into an array, U0 (LINE 36). TWFUNC needs this array when it evaluates transient residuals. Finally, the program fragment performs the UPDATE task by recounting unknowns (LINE 40). Since twopnt changes POINTS when it selects a new grid, BUFFER and other arrays must be able to accommodate the largest possible grid, see Section 6.2.

6.2 What the Arguments Do

Numerical software isn't the focus of research in programming languages, so as a result, many subroutine arguments are needed to implement (with some clumsiness) programming devices (such as reverse communication) revered only in the numerical folklore. Twopnt's own arguments thus divide into functional groups.

- reverse communication
- grids and unknowns
- work space arrays
- status reporting
- control lists

A few arguments serve both reverse communication and some other purpose. Reverse communication is discussed in Section 6.1; the other uses are discussed here.

```
      SUBROUTINE TWOPNT
     +  (ERROR, TEXT, VERSIO,
     +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
     +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
     +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
     +   STRIDE, TIME, U, X)
```

Figure 6.6. *Twopnt's arguments with those used for grids and unknowns highlighted. Section 6.2 explains them.*

Figure 6.6 highlights twopnt's arguments for communicating grids and unknowns. Two arrays communicate values for the unknowns.

• BUFFER	array of values and residuals in reverse communication	66
• U	array of guesses on input, solution values on output	71 ($\star$ 40)

The data in these arrays must have a particular arrangement only when twopnt selects grids for two point boundary value problems. Otherwise, the arrangement is no concern to twopnt, but even then, it should not be entirely arbitrary.

The arrangement of unknowns and residuals establishes the nonzero structure of the Jacobian matrices. If the i-th equation depends on the j-th unknown, then the matrix reflects that through its entry in row i and column j. If there is no dependence, then the entry there is zero. Usually, matrices encountered in simulations have many zeroes. Unknowns and equations may have a spatial distribution, for example, with only those at nearby locations dependent. The pattern of nonzeroes determines the ease of solving matrix equations, and this often determines the feasibility of the simulation.

Software for matrix equations usually assumes some pattern of nonzeroes. The TWPREP and TWSOLV subroutines supplied with twopnt assume a tridiagonal pattern of blocks, see Figure 6.4. This pattern is natural in two point boundary value problems, and some aspects of it appear in other simulations, as follows. Usually, unknowns are associated with equations, and then values and residuals might appear in the same order. Moreover, unknowns and equations often have a spatial distribution, and then coincident values are conveniently made contiguous. In chemistry simulations for example, species are associated with their conservation equations, and both are associated with points in space. (The tridiagonal matrix structure occurs when, additionally, the points line up, and the equations at each point need only the values there or at neighboring points.)

The *"components at points"* arrangement of unknowns is convenient whenever a simulation involves replication. This convention supposes every one of p points has exactly c solution components.

$$\begin{matrix} u_{1,1} & u_{1,2} & \cdots & u_{1,p} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ u_{c,1} & u_{c,2} & \cdots & u_{c,p} \end{matrix}$$

Simulators employing this convention should order values for the unknowns in `BUFFER` and `U` point by point—or in terms of the picture above, column by column—so all the values at the same point are contiguous.

$$u_{1,1} \quad u_{2,1} \quad \ldots \quad u_{c,1} \qquad u_{1,2} \quad u_{2,2} \quad \ldots \quad u_{c,2} \qquad u_{1,3} \quad u_{2,3} \quad \ldots \quad u_{c,3} \qquad \ldots$$

This is the memory arrangement prescribed for fortran arrays dimensioned `(`c`,`p`)`. Twopnt does not use fortran dimensioning, however, because twopnt allows two groups of extra unknowns to precede and follow the others.

$$\acute{u}_1 \quad \acute{u}_2 \quad \ldots \quad \acute{u}_a \quad \ldots \quad \text{the others} \quad \ldots \quad \grave{u}_1 \quad \grave{u}_2 \quad \ldots \quad \grave{u}_b$$

Altogether there are $a + cp + b$ unknowns. The extra unknowns are intended for boundary conditions in two point boundary value problems.

Twopnt counts unknowns using four arguments with names suggested by the components at points convention.

• `COMPS`	number of components	67
• `POINTS`	number of points	58, 68 (⋆ 16, 35)
• `GROUPA`	number of initial, group A unknowns	67
• `GROUPB`	number of final, group B unknowns	67

Simulators that don't use automatic grid selection may choose these arguments arbitrarily to count their unknowns so long as `GROUPA` $+$ `COMPS` $\times$ `POINTS` $+$ `GROUPB` is the proper number. For example, a simulator with n unknowns and no spatial dimensions might choose `GROUPA` $= n$. A simulator with three quantities distributed throughout a two dimensional, $m \times n$, grid might choose `COMPS` $= 3$ and `POINTS` $= mn$.

The grouping of unknowns imposed by the arguments above extends to bounds and names for the unknowns, as follows.

• `ABOVE`	array of upper bounds	66 (⋆ 14, 33)
• `BELOW`	array of lower bounds	66 (⋆ 14, 33)
• `NAME`	array of names	68
• `NAMES`	dimension of the `NAME` array	68

These arrays hold bounds and names for group A unknowns first, then those for components, and finally bounds and names for group B. Each unknown in groups A and B has its own bounds and name, but unknowns associated with a component have the same bounds and name at every point. The arrays thus have a dimension, `GROUPA` + `COMPS` + `GROUPB`, which is independent of grid size. If names for unknowns are not available, then `NAMES` should be 1, and then twopnt ignores `NAME`, which may be a blank character string.

Twopnt uses some arrays only with automatic grid selection. Memory space needn't be wasted when grid selection isn't permitted because twopnt also dimensions these `(*)`.

• `ACTIVE`	array marking components for examination	66
• `MARK`	array marking new grid points	58, 68
• `X`	array of grid points	58, 71 ($\star$ 16, 40)

Section 6.3 provides the expected dimensions for these and other arrays. The `ACTIVE` array communicates the control values of the same name: `ACTIVE(`k`)` *true* tells twopnt to examine the k-th component. The `MARK` array marks new grid points when twopnt makes the `UPDATE` reverse communication request: `MARK(`n`)` *true* means `X(`n`)` is new.

With automatic grid selection, the extent of meaningful data in arrays `BUFFER`, `MARK`, `U` and `X` grows with each new grid. If the simulator conforms to the fortran standard [1], then array space must be fixed when the program is written. These arrays therefore must accommodate the maximum points allowed.

• `PMAX`	maximum points in any grid	68 ($\star$ 18, 34)

`MARK` and `X` must have size `PMAX` not `POINTS`, while `BUFFER` and `U` must have size `GROUPA + COMPS` $\times$ `PMAX + GROUPB`. Simulators conforming to the fortran standard thus cannot leave `PMAX` entirely to choice. Usually, they impose an upper limit that can be adjusted downward, if desired, when the simulation is performed.

```
      SUBROUTINE TWOPNT
    +  (ERROR, TEXT, VERSIO,
    +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
    +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
    +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
    +   STRIDE, TIME, U, X)
```

Figure 6.7. *Twopnt's arguments with those used for work space highlighted. Section 6.2 explains them.*

Figure 6.7 highlights twopnt's arguments for work space. Twopnt's work space isn't scratch space, so the simulator can't use it for other things.

• `ISIZE`	size of the integer work array	67
• `IWORK`	integer work array	68
• `RSIZE`	size of the real work array	69
• `RWORK`	real work array	69

The integer size must be at least $3p$ and the (single or double precision) real size must be at least $3p + 9n$, in which $p =$ `PMAX` and $n =$ `GROUPA` + `COMPS` $\times$ `PMAX` + `GROUPB`. Twopnt allows `PMAX` $= 0$ when there are no "points," but then the work array sizes must be 3 and $3 + 9n$, respectively.

```
      SUBROUTINE TWOPNT
     +  (ERROR, TEXT, VERSIO,
     +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
     +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
     +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
     +   STRIDE, TIME, U, X)
```

Figure 6.8. *Twopnt's arguments with those used for status reporting highlighted. Section 6.2 explains them.*

Figure 6.8 highlights twopnt's arguments for status reporting. The three most important arguments occur first and outside the otherwise alphabetical ordering. Twopnt sets ERROR *true* when catastrophes occur, and then error messages explain what went wrong.

• ERROR	error signal	66
• TEXT	fortran unit number for the message file	66

All messages, including those controlled by LEVELD and LEVELM, go to the unit number designated by TEXT, unless TEXT ≤ 0, in which case twopnt writes no messages of any kind anywhere.

Reporting is bidirectional: most is from but some is to twopnt. The simulator announces the version of twopnt it expects through VERSIO.

• VERSIO	expected version of twopnt	66

This argument's value should be a character string like the following.

```
SINGLE PRECISION VERSION 3.00
```

The precision can change to DOUBLE and the version number can change to something of the same form, 3.08, but otherwise no variation is allowed. Twopnt's version number should be the one on this publication's cover; other versions might have different arguments or controls. Twopnt's precision can change easily, see Appendix 2. The simulator's precision should change as easily, to facilitate portability between machines.

Twopnt announces the outcome of the simulation through REPORT.

• REPORT	outcome status	69

This character string is significant when twopnt returns with SIGNAL *blank* and ERROR *false*. The string then is one of these.

- *blank*
- NO SPACE
- SOME FOUND
- NONE FOUND

(1) *Blank* means all is well. (2) NO SPACE means the solution doesn't satisfy the grid selection criteria, and twopnt hasn't space for larger grids. (3) SOME FOUND means the solution doesn't satisfy the grid selection criteria, and twopnt couldn't find a solution for larger grids. (4) NONE FOUND means no solution was found. In all cases, U is the result of the simulation, and if automatic grid selection was requested then POINTS and X contain output too. For NONE FOUND twopnt returns the initial guess.

```
      SUBROUTINE TWOPNT
     +  (ERROR, TEXT, VERSIO,
     +   ABOVE, ACTIVE, BELOW, BUFFER, CNTRL, CNTRLS, COMPS, CONDIT,
     +   GROUPA, GROUPB, ISIZE, IVALUE, IWORK, LVALUE, MARK, NAME,
     +   NAMES, PMAX, POINTS, REPORT, RSIZE, RVALUE, RWORK, SIGNAL,
     +   STRIDE, TIME, U, X)
```

Figure 6.9. *Twopnt's arguments with those used for control lists highlighted. Section 6.2 explains them.*

Figure 6.9 highlights twopnt's arguments for communicating control values. When twopnt begins a simulation, it searches the array CNTRL for control names, and to each control it finds, it gives the value in IVALUE, LVALUE or RVALUE. The value for the i-th control should be in the i-th position of the appropriate value array.

• CNTRL	array of control names	67
• CNTRLS	dimension of the control arrays	67
• IVALUE	array of integer values	68
• LVALUE	array of logical values	68
• RVALUE	array of real values	69

For example, if CNTRL(3) is SSABS, then RVALUE(3) becomes twopnt's value for SSABS. The third entries in IVALUE and LVALUE are ignored.

i	CNTRL	IVALUE	LVALUE	RVALUE
1	ADAPT	*ignored*	.TRUE.	*ignored*
2	LEVELM	2	*ignored*	*ignored*
3	SSABS	*ignored*	*ignored*	1.0E-7
⋮				

The controls that receive values this way, and the data types of their values, are as follows.

integer IVALUE:	LEVELD LEVELM PADD STEPS0 STEPS1 STEPS2 TDAGE
logical LVALUE:	ADAPT STEADY
real RVALUE:	SSABS SSAGE SSREL STRID0 TDABS TDEC TDREL TINC TMAX TMIN TOLER0 TOLER1 TOLER2

A few controls listed in Section 4.1 are not listed here. They receive values directly through twopnt's arguments with the same names.

Twopnt's search for names in CNTRL is straightforward. Unrecognized names are errors, but since arrays must have at least one entry, *blank* names are accepted. If a name appears more than once, then the last value supersedes. If a name does not appear, then twopnt's default does not change. Beware the default value is always twopnt's: a second simulation does not inherit the first's control values.

6.3 Long List of Arguments

This long list of twopnt's arguments provides some details omitted from earlier discussions. The arguments appear in Figure 6.9 and in several preceding Figures.

Note four things. (1) Arguments occur alphabetically after the first three. This is their order both in subroutine calls and in the list below. (2) The arguments ignore fortran naming conventions. `TEXT` is an integer variable, for example. (3) Floating point arguments are all single or all double precision. Thirty-two bit computers generally use double precision, while sixty-four bit machines use single precision. (4) Twopnt's precision changes easily by altering a few fortran statements, see Appendix 2, but twopnt's subroutine names do not change, so confusion might result when moving twopnt among machines that need different precisions.

- `ERROR` use: error signal 64
  declaration: output logical

  If *true*, then a numerical or programming error occurs. Error messages appear in the message file.

- `TEXT` use: fortran unit number for the message file 64
  declaration: input integer

  Twopnt writes all error and informative messages to this unit. Twopnt neither opens nor closes the file, and if the unit number isn't positive, then twopnt writes nothing.

- `VERSIO` use: expected version of twopnt 64
  input character length `(*)`

  This string identifies the precision and version expected of twopnt. It has the form `` `SINGLE PRECISION VERSION 3.00' ``, in which `SINGLE` can change to `DOUBLE` and `3.00` can change to something similar.

- `ABOVE` use: upper bounds 62 (⋆ 14, 33)
  declaration: input floating dimensioned `(GROUPA + COMPS + GROUPB)`

  Values for unknowns never exceed their bounds. Bounds should be arranged with those for group A unknowns first, then bounds for components, and finally bounds for group B unknowns.

- `ACTIVE` use: markers for components to be examined during grid selection 63 (⋆ 17, 33)
  declaration: input logical dimensioned `(*)`, or dimensioned `(COMPS)` for grid selection

  Twopnt uses this augument only with automatic grid selection. The selection process examines component k provided `ACTIVE(`k`)` is *true*. An error occurs if no component is active.

- `BELOW` use: lower bounds 62 (⋆ 14, 33)
  see: `ABOVE`

- `BUFFER` use: reverse communication data consisting of residual values or search directions on input, and values for unknowns on output 57, 61
  declaration: input and output floating dimensioned `(GROUPA + COMPS * PMAX + GROUPB)`

  When making reverse communication requests, `TWOPNT` always places an approximate solution in `BUFFER`, and sometimes twopnt expects to find something there when it resumes. If twopnt expects nothing, then `BUFFER` can be used for scratch space. For automatic grid selection, values should be ordered in `BUFFER` by the components at points convention.

- `CNTRL` use: control names 65
  declaration: input character length `(*)` dimensioned `(CNTRLS)`

This array identifies control values in `IVALUE`, `LVALUE` and `RVALUE`. The control named by `CNTRL(`i`)` has value `IVALUE(`i`)`, `LVALUE(`i`)`, or `RVALUE(`i`)`. Whether an `I`, `L` or `RVALUE` depends on the control name.

integer `IVALUE:`	`LEVELD LEVELM PADD STEPS0 STEPS1 STEPS2 TDAGE`
logical `LVALUE:`	`ADAPT STEADY`
real `RVALUE:`	`SSABS SSAGE SSREL STRID0 TDABS TDEC TDREL TINC TMAX TMIN TOLER0 TOLER1 TOLER2`

Names may appear in any order in `CNTRL`. They must be the control names above, or *blank*. For repeated names, the last value supersedes. For absent control names, twopnt assigns default values, see Section 4.1.

- `CNTRLS` use: number of control values 65
  declaration: input integer

The number must be at least 1 because it dimensions arrays `CNTRL`, `IVALUE`, `LVALUE` and `RVALUE`.

- `COMPS` use: number of components 62
  declaration: input integer

On input, this number must be at least zero, and it must be positive if `POINTS` is positive. It contributes to the dimensions of arrays `ABOVE`, `ACTIVE`, `BELOW`, `BUFFER` and `U`, and it contributes to the total number of unknowns, `GROUPA` $+$ `COMPS` $\times$ `POINTS` $+$ `GROUPB`.

- `CONDIT` use: matrix condition number 57
  declaration: input floating

When twopnt requests matrix preparation, `SIGNAL` $=$ `` `PREPARE' ``, then it also accepts the matrix condition number for writing in informative messages. Zero and negative values mean no number is available.

- `GROUPA` use: number of initial, group A unknowns 62
  declaration: input integer

The number must be at least zero. It contributes to the dimensions of arrays `ABOVE`, `BELOW`, `BUFFER` and `U`, and it contributes to the total number of unknowns, `GROUPA` $+$ `COMPS` $\times$ `POINTS` $+$ `GROUPB`.

- `GROUPB` use: number of final, group B unknowns 62
  see: `GROUPA`

- `ISIZE` use: dimension of the integer work array, `IWORK` 63
  declaration: input integer

The size must be at least $3 \times$ PMAX, or at least 3 if PMAX $= 0$.

- IVALUE use: integer control values 65
  declaration: input integer dimensioned (CNTRLS)

If CNTRL(i) is either LEVELD, LEVELM, PADD, STEPS0, STEPS1, STEPS2 or TDAGE, then twopnt expects a value for that control in IVALUE(i).

- IWORK use: integer work array 63
  declaration: input integer dimensioned (ISIZE)

The work space should not be changed while twopnt returns for reverse communication.

- LVALUE use: logical control values 65
  declaration: input logical dimensioned (CNTRLS)

If CNTRL(i) is ADAPT or STEADY, then twopnt expects to find a value for that control in LVALUE(i).

- MARK use: array marking new grid points for the UPDATE reverse communication request 58, 63
  declaration: output logical dimensioned (*), or dimensioned (PMAX) for grid selection

Twopnt uses this augument only with automatic grid selection. When twopnt signals it has a new grid, SIGNAL $=$ `UPDATE', then the new grid is in X, and if MARK(i) is *true*, then X(i) is a new grid point.

- NAME use: array of names for the unknowns 62
  declaration: input character length (*) dimensioned (NAMES)

Twopnt writes these names in messages and may truncate long names. Names should be arranged with those for group A unknowns first, then names for components, and finally names for group B unknowns. The array should be replaced by a single blank character string if no names are available.

- NAMES use: number of names for the unknowns, either 1 or GROUPA + COMPS + GROUPB 62
  declaration: input integer

NAMES tells twopnt the number of names in NAME. The value should be 1 if no names are available.

- PMAX use: maximum points in any grid 63 (⋆ 18, 34)
  declaration: input integer

The number must be at least POINTS. It contributes to the dimensions of arrays BUFFER, U and X.

- POINTS use: number of points 58, 62 (⋆ 16, 35)
  declaration: input and output integer

On input, this number must be at least zero, and positive if COMPS is. It contributes to the total number of unknowns, GROUPA $+$ COMPS $\times$ POINTS $+$ GROUPB. For automatic grid selection, `POINTS` is the size of the current grid (it should be the size of the initial grid initially, and twopnt increases it as the grids grow).

- `REPORT` use: outcome status 64
  declaration: output character length `(*)`

If `ERROR` is *false* and `SIGNAL` is *blank*, then twopnt has finished and `REPORT` explains one of four outcomes. (1) *Blank* means all is well. (2) `NO SPACE` means the solution doesn't satisfy the grid selection criteria, and twopnt hasn't space for larger grids. (3) `SOME FOUND` means the solution doesn't satisfy the grid selection criteria, and twopnt couldn't find a solution for larger grids. (4) `NONE FOUND` means no solution was found. When twopnt finishes, `U` is the result of the simulation, and if automatic grid selection was requested then `POINTS` and `X` contain output too. For `NONE FOUND` twopnt returns the initial guess.

- `RSIZE` use: dimension of the floating work array, `RWORK` 63
  declaration: input integer

The size must be at least $3 \times \texttt{PMAX} + 9 \times n$ in which $n = \texttt{GROUPA} + \texttt{COMPS} \times \texttt{PMAX} + \texttt{GROUPB}$, or if $\texttt{PMAX} = 0$ then the size must be at least $3 + 9 \times n$.

- `RVALUE` use: floating control values 65
  declaration: input integer dimensioned `(CNTRLS)`

When `CNTRL(`i`)` is `SSABS`, `SSAGE`, `SSREL`, `STRID0`, `TDABS`, `TDEC`, `TDREL`, `TINC`, `TMAX`, `TMIN`, `TOLER0`, `TOLER1` or `TOLER2`, then twopnt expects to find a value for that control in `RVALUE(`i`)`.

- `RWORK` use: floating work array 63
  declaration: input floating dimensioned `(RSIZE)`

The work space should not be changed while twopnt returns for reverse communication.

- `SIGNAL` use: reverse communication signal 57
  declaration: input and output character length `(*)`

Blank on input marks the start of a simulation; *blank* on output marks the finish. Not *blank* on output signals a reverse communication request. In this case, the following tasks must be performed.

`SIGNAL` = `` `PREPARE' `` signal to prepare the Jacobian matrix

This signal means subsequent requests to solve matrix equations should use the Jacobian matrix (of the residual function) evaluated at the approximate solution now in `BUFFER`. The (i, j) matrix entry is the partial derivative of the i-th equation's residual with respect to the j-th unknown. Twopnt expects equations to be solved with this matrix when the `` `SOLVE' `` signal supplies right sides. Twopnt doesn't want the matrix and doesn't care how the equations are solved, so simulator writers must decide what to do here. The simplest option is to use the `TWPREP` and `TWSOLV` subroutines supplied with twopnt.

Simulators usually solve matrix equations by factorization—that is, by Gaussian elimination—so they form and factor the Jacobian matrix when signaled to `` `PREPARE' ``. In this case, (1) evaluate the matrix at the approximate solution in `BUFFER`. When `TIME` is *true*, the matrix should be the one for the time dependent residual. The matrix entries can be obtained either by evaluating formulas for partial derivatives or by approximating those derivatives with divided differences. Usually, computing approximations is easier than deriving formulas. (2) Factor the matrix and store the factors somewhere for later use. Mathematical software (mathware) libraries usually contain programs for factoring matrices and then solving equations [8] [9] [14]. (3) If a condition estimate for the matrix is available, place that in `CONDIT`. If not, set `CONDIT` zero. (4) Call twopnt. The `TWPREP` subroutine supplied with twopnt will perform these steps with reasonable efficiency for matrices like the one pictured in Figure 6.4. Sections 6.1 and 6.4 discuss `TWPREP`.

Some simulators solve matrix equations iteratively—that is, by successive approximation—because Gaussian elimination might need too much computer memory. Iterative algorithms use the matrix only for matrix-vector products. For Jacobian matrices, matrix-vector products are directional derivatives which can be evaluated without the matrix. Unfortunately, each of the many iterative algorithms succeeds only in special situations [14]. If this is one of them, then prepare to use an iterative algorithm for solving equations when `SIGNAL` = `` `SOLVE' ``, as follows. (1) Either evaluate and store the matrix (for use in forming matrix-vector products later), or store the approximate solution found now in `BUFFER` (in case matrix-vector products are to be evaluated as directional derivatives). If some preconditioning matrix is needed, then prepare it now too. (2) If a condition estimate for the matrix is available, place that in `CONDIT`. If not, set `CONDIT` zero. (3) Call twopnt.

`SIGNAL` = `` `RESIDUAL' `` signal to evaluate the residual

(1) Evaluate the residual at the approximate solution in `BUFFER`. When `TIME` is *true*, the residual should be the time dependent one. (2) Place the residual in `BUFFER`. (3) Call twopnt.

`SIGNAL` = `` `RETAIN' `` signal to retain a transient state

`BUFFER` contains a transient state that will be needed later when evaluating transient residuals. (1) Store the contents of `BUFFER`. (2) Call twopnt.

`SIGNAL` = `` `SAVE' `` signal to save the solution

The latest approximate solution can be used to restart the simulation should the computer crash or the money run out. (1) If desired, store the solution found in `BUFFER` with whatever else describes the simulation, such as `POINTS` and `X` when grid selection is used. (2) Call twopnt.

`SIGNAL` = `` `SHOW' `` signal to show the solution

The control `LEVELD` stipulates that solution data be displayed occasionally. (1) Write to unit `TEXT` the approximate solution found in `BUFFER`. (2) Call twopnt. The `TWSHOW` subroutine supplied with twopnt will perform this task with the result pictured in Figure 6.5. Sections 6.1 and 6.4 discuss `TWSHOW`.

`SIGNAL` = `` `SOLVE' `` signal to solve matrix equations

The matrix equations to be solved are the ones in Newton's method, but twopnt chooses the matrix and supplies the right side, see Section 2.1, so the simulator must only do the following. (1) Solve matrix equations using the most recently prepared matrix and the "right side," or constant terms, found in `BUFFER`. (2) Place the solution in `BUFFER`. (3) Call twopnt. The `TWSOLV` subroutine supplied with twopnt can perform these tasks for matrices prepared by `TWPREP`. Figure 6.4 and Sections 6.1 and 6.4 discuss `TWSOLV`.

`SIGNAL` = ``` `UPDATE' ``` signal to update the problem to a new grid

When twopnt chooses a new grid it must guess the solution there. The residual function also changes at this time because the grid is larger. (1) If desired, replace twopnt's guess in `BUFFER` by a better guess. (2) If desired, prepare to evaluate the new residual function, for example, by computing some things ahead. (3) Call twopnt.

- `STRIDE` use: time stride for the transient residual 57
  declaration: output floating

Use this stride when approximating time differentials for the transient residual or its Jacobian matrix.

- `TIME` use: signal to use the transient residual 57
  declaration: output logical

If *true* with reverse communication requests ``` `PREPARE' ``` and ``` `SOLVE' ```, then use the transient residual.

- `U` use: the guess and the solution 61 (⋆ 40)
  declaration: input and output floating dimensioned
  `(GROUPA + COMPS * PMAX + GROUPB)`

`U` should contain the initial guess at the start of the simulation. It contains approximate solutions from Newton's search and time evolution during the simulation. `U` contains a solution at the finish of the simulation, unless `REPORT` is `NONE FOUND`, then twopnt restores the initial guess. For automatic grid selection, values should be ordered in `U` by the components at points convention.

- `X` use: grid points 58, 63 (⋆ 16, 40)
  declaration: input and output floating dimensioned
  `(*)`, or dimensioned `(PMAX)` for grid selection

Twopnt uses this augument only with automatic grid selection. `X` should contain the initial grid at the start of the simulation. It contains the current grid during the simulation. `X` contains the last grid for which a solution was found at the finish of the simulation, unless `REPORT` is `NONE FOUND`, then twopnt restores the initial grid. The initial grid must be ordered

$$\texttt{X(1)} < \texttt{X(2)} < \cdots \quad \text{or} \quad \texttt{X(1)} > \texttt{X(2)} > \cdots$$

and twopnt retains the ordering when it selects grids, so twopnt may move old points to new array positions.

6.4 Ancillary Subroutines

The `TWPREP`, `TWSOLV` and `TWSHOW` subroutines come with twopnt but aren't part of twopnt. They can perform the more difficult reverse communication requests: `PREPARE`, `SOLVE` and `SHOW`, respectively.

Figure 6.10 shows the arguments for all three ancillary routines. Note four things. (1) Arguments with twopnt's names should be given twopnt's actual arguments. (2) This Section explains only arguments unique to the ancillary routines. (3) For automatic grid selection, arrays sized by `POINTS` should be dimensioned using `PMAX` instead, but the `POINTS`

```
      SUBROUTINE TWSHOW
     +  (ERROR, TEXT,
     +   BUFFER, COMPS, GRID, GROUPA, GROUPB, POINTS, X)

      SUBROUTINE TWPREP
     +  (ERROR, TEXT,
     +   A, ASIZE, BUFFER, COMPS, CONDIT, GROUPA, GROUPB, PIVOT, POINTS,
     +   RETURN)

      SUBROUTINE TWSOLV
     +  (ERROR, TEXT,
     +   A, ASIZE, BUFFER, COMPS, GROUPA, GROUPB, PIVOT, POINTS)
```

Figure 6.10. *`TWSHOW`'s, `TWPREP`'s and `TWSOLV`'s arguments. These subroutines show solution data, prepare matrices and solve matrix equations, respectively. Section 6.1 explains their use with twopnt's reverse communication. Section 6.4 explains their arguments.*

argument should not be given the value `PMAX`. (4) Like twopnt, `TWPREP` evaluates the residual function by reverse communication.

`TWSHOW` can perform the `SHOW` reverse communication request. Figure 6.5 shows what is shown for components at points. `GROUPA` and `GROUPB` unknowns would be shown similarly. `TWSHOW` has one argument different from twopnt's.

○ `GRID` use: grid signal
declaration: input logical

This argument is needed in case twopnt does not perform automatic grid selection. `GRID` *false* signals `TWSHOW` to omit writing grid points.

The matrix subroutines, `TWPREP` and `TWSOLV`, only apply to matrices that resemble the one in Figure 6.4. In this case, `TWPREP` (1) forms a Jacobian matrix by numerical approximation, (2) scales the matrix so all rows have ∞-norm of 1, (3) uses the banded subroutines in the linpack package [8] to factor the matrix, (4) uses those subroutines also to estimate the condition number of the scaled matrix, and (5) stores data in arrays `A` and `PIVOT`. `TWSOLV` uses this data (6) to scale equations to match the matrix scaling, and then (7) to solve equations.

`TWPREP` approximates the i-th column in the Jacobian matrix of a function f evaluated at the vector w, as follows.

$$\left.\frac{\partial f}{\partial v_i}\right|_{v=w} \approx \frac{f(\widetilde{w}) - f(w)}{\epsilon_i} \qquad\qquad \epsilon_i = (w_i \pm 1)\sqrt{\epsilon}$$

In this expression, (a) v_i is the i-th variable in f, (b) $\widetilde{w}$ matches w except ϵ_i increments the i-th entry (this perturbation ignores the bounds on twopnt's unknowns), (c) w_i is the i-th entry in w, (d) ϵ is the unit roundoff for the computer arithmetic in use, and (e) the $\pm$ sign matches the sign of w_i. The tridiagonal, blocked pattern expected for the nonzeroes in the matrix allows `TWPREP` to build several columns at once, so approximately only $3 \times$ `COMPS` function evaluations are needed.

`TWPREP` and `TWSOLV` share many arguments with twopnt. The arrays have dimensions that vary with `POINTS`—which twopnt might increase to `PMAX` during grid selection—so actual dimensions should be chosen using `PMAX` instead. One new argument for matrix storage likely accounts for most of a simulator's memory needs.

○ `A` use: matrix and scale factor storage space
declaration: output (for `TWPREP`) and input (for `TWSOLV`) floating
dimensioned `(ASIZE)`

This space should not be changed because it communicates matrix factors and scale factors from `TWPREP` to `TWSOLV`.

- ○ `ASIZE` use: dimension of the storage space, `A`
  declaration: input integer

The size must be at least $(3b - 1)n$.

$$b = \texttt{COMPS} + \max\{\texttt{COMPS}, \texttt{GROUPA}, \texttt{GROUPB}\}$$

$$\begin{aligned} n &= \texttt{GROUPA} + \texttt{COMP} \times \texttt{POINTS} + \texttt{GROUPB} \\ &\leq \texttt{GROUPA} + \texttt{COMP} \times \texttt{PMAX} + \texttt{GROUPB} \end{aligned}$$

`POINTS` can grow to `PMAX` during automatic grid selection, so the value of twopnt's argument `PMAX` should be used for `POINTS` when choosing a value for `ASIZE`.

- ○ `BUFFER` use: reverse communication data for `TWPREP`, identical to twopnt's argument of the same name 57, 61, 66
  declaration: input and output floating dimensioned (`GROUPA + COMPS * POINTS + GROUPB`)

When twopnt requests matrix preparation, it places in `BUFFER` the approximate solution at which the matrix should be evaluated. This is `TWPREP`'s initial input. When `TWPREP` requests residual evaluation, `TWPREP` places the approximate solution in `BUFFER`, and it expects to find the residual there when it resumes.

- ○ `PIVOT` use: pivoting data from the matrix factorization
  declaration: input and output integer dimensioned (`GROUPA + COMPS * POINTS + GROUPB`)

This space should not be changed because it communicates pivoting data from `TWPREP` to `TWSOLV`. `POINTS` can grow to `PMAX` during automatic grid selection, so `PMAX` should replace `POINTS` when choosing a dimension for `PIVOT`.

- ○ `RETURN` use: reverse communication signal for `TWPREP`
  declaration: input and output logical

False on input marks the start of matrix preparation; *false* on output marks the finish. *True* on output signals a reverse communication request. In this case, (1) evaluate the residual at the approximate solution in `BUFFER`. When twopnt's argument `TIME` is *true*, the residual should be the time dependent one. (2) Place the residual in `BUFFER`. (3) Call `TWPREP`.

(D) Finally, let not only mathematical formulation and observational data, but even the approximation process pass unchallenged. There still remains this limitation: No computing procedure or device can perform the operations which are its "elementary" operations (or at least all of them) rigorously and faultlessly. This point is most important . . .

— J. von Neumann and H. H. Goldstine [26]

7 Swirling Flows

7.0 Introduction

A “solution” of the Navier-Stokes equations often involves a dimension reduction and often must be obtained numerically. The simulation appearing throughout this manual is such a solution for flows between infinite, parallel, rotating and stationary planes. This flow problem illustrates the three steps in building a computer simulation, see Figure 2.1.

7.1 Step 1, Model

Figure 7.1 plots some streamlines for the whirlpool-like flow modelled here. Theodor von Kármán [19] originated the dimension reduction used to model these flows in 1921—he considered flows directed against an infinite, rotating plane—and similar flows have been studied continuously since then, see [3] and [30]. The flows have practical use in modeling chemical reactors [6], so to this end, G. H. Evans and R. Grief [10] recently extended the model to variable fluid properties. The following discussion is based on their work, and on notes of D. S. Dandy [7].

The coordinate system is cylindrical about the planes’ axis of rotation with the usual notation. The fluid has velocity, pressure and temperature which are functions of position and which must be determined.

r	radial position	cm
θ	angular position	radians
z	axial position	cm
u	radial velocity	cm / sec
v	circumferential velocity	cm / sec
w	axial velocity	cm / sec
p	pressure	dynes / cm^2
T	temperature	K

The flow cavity lies between the planes at $z = 0$ and $z = z_{\text{max}}$. The lower plane at $z = 0$ is a *solid disk*: an impermeable surface which is reacting in some simulations [6], but not this one. The upper plane at $z = z_{\text{max}}$ is a *porous disk*: a permeable surface passing

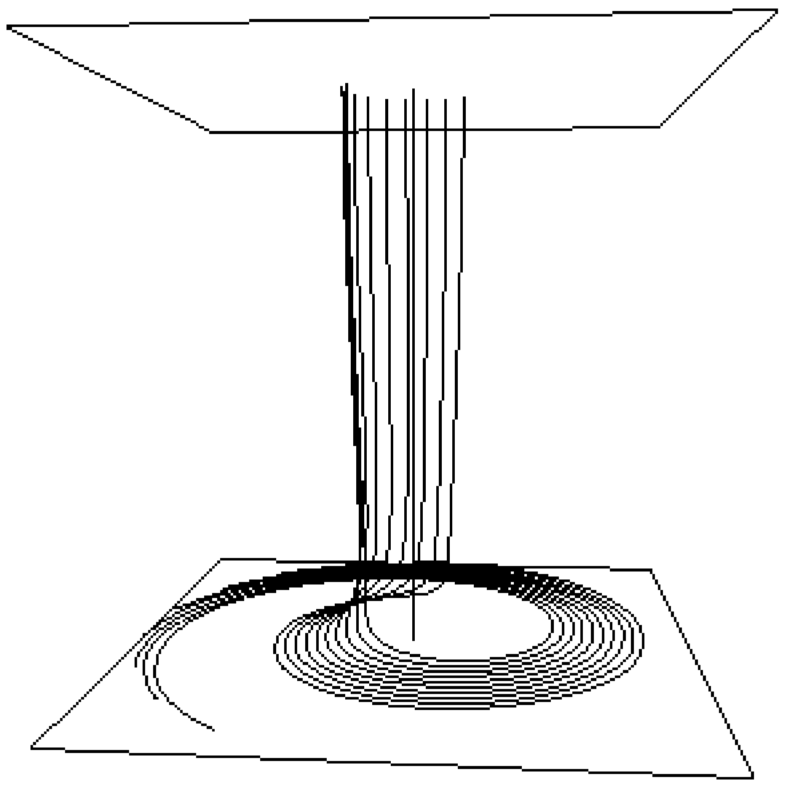

Figure 7.1 *Streamlines computed from the results of Chapter 7's simulation. The flow spirals downward from the porous, stationary plane to the impermeable, rotating plane. Fluid travels the length of the streamlines shown here in* 2.75 *seconds. Section 7.1 describes the flow model.*

fluid with constant normal velocity, w_{max}, independent of location. Since the picture in mind has $0 < z_{\text{max}}$, flow into the cavity has negatively signed axial velocity, $w_{\text{max}} < 0$. Both disks are heated and the solid one is spinning, imparting temperatures and a rotation rate T_0, T_{max} and ω_0, respectively.

T_0	solid disk temperature	300 K
T_{max}	porous disk temperature	1000 K
ω_0	solid disk rotation rate	4π radians / sec
w_{max}	axial inflow velocity	–2 cm / sec
z_{max}	porous disk height	5 cm

The dimension reduction for this flow assumes (1) rigid rotational symmetry—radial and circumferential velocities linear in r, and axial velocity and temperature independent of r, (2) density independent of pressure variation (the low Mach number approximation), and (3) pressure quadratic in r and independent of z. From assumption (1), velocities can be represented in terms of functions F, G and H of z alone.

$$u = rF \qquad v = rG \qquad w = H$$

These functions and T constitute a two point boundary value problem. The boundary conditions are

$$G(0) = \omega_0 \qquad T(0) = T_0 \qquad H(z_{\text{max}}) = w_{\text{max}} \qquad T(z_{\text{max}}) = T_{\text{max}}$$

and other values are zero. The customary equations for radial momentum, angular momentum, continuity, and energy reduce to equations in these new variables.

$$\begin{aligned}
\lambda + \rho\left(\frac{\partial F}{\partial t} + F^2 - G^2 + \frac{\partial F}{\partial z} H\right) - \frac{\partial}{\partial z}\left(\mu \frac{\partial F}{\partial z}\right) &= 0 \\
\rho\left(\frac{\partial G}{\partial t} + 2FG + \frac{\partial G}{\partial z} H\right) - \frac{\partial}{\partial z}\left(\mu \frac{\partial G}{\partial z}\right) &= 0 \\
2F + \frac{\partial H}{\partial z} + H \frac{\partial \ln \rho}{\partial z} &= 0 \\
c_p \rho \left(\frac{\partial T}{\partial t} + \frac{\partial T}{\partial z} H\right) - \frac{\partial}{\partial x}\left(k \frac{\partial T}{\partial z}\right) &= 0
\end{aligned}$$

These equations contain an expression

$$\lambda = \frac{1}{r}\frac{\partial p}{\partial r}$$

which is independent of r and z by assumption (3) and which therefore becomes an "eigenvalue" for the boundary value problem.

Fluid properties are courtesy of the chemkin libraries [20, 23].

c_p	specific heat at constant pressure	ergs / (g K)
k	thermal conductivity	ergs / (cm K s)
μ	dynamic viscosity	g / (cm s)
ρ	density	g / cm^3

The fluid in this example is argon gas. Argon's specific heat is independent of temperature

c_p	specific heat	2.5 R / W
R	universal gas constant	8.314 $\cdot 10^7$ ergs / (mole K)
W	molecular weight	39.948 g / mole

but its conductivity and viscosity do vary with temperature

$$\begin{aligned}
k(T) &= \exp(((0.0121673\,\alpha - 0.284023)\,\alpha + 2.85205)\,\alpha - 1.78377) \\
\mu(T) &= \exp(((0.0121673\,\alpha - 0.284023)\,\alpha + 2.85205)\,\alpha - 17.6539)
\end{aligned}$$

in which $\alpha = \ln T$. Density varies with both pressure and temperature, but this model assumes (2), insensitivity to small pressure changes. The model therefore modifies the ideal gas law as follows.

$$\rho(T) = \frac{p_{\text{ref}}}{T}\,\frac{W}{R}$$

p_{ref}	thermodynamic pressure	1 atmosphere (1013250.0 dynes / cm^2)

Figure 2.2 plots F, G, H and T for the equations, boundary conditions, and fluid properties given above.

7.2 Step 2, Discretization

A straightforward discretization of the model's differential equations uses finite difference methods with p grid points and $4p$ unknowns

$$\begin{array}{rcccccl} 0 = & x_1 & x_2 & x_3 & \dots & x_p & = z_{\mathsf{max}} \\ 0 = & F_1 & F_2 & F_3 & \dots & F_p & = 0 \\ \omega_0 = & G_1 & G_2 & G_3 & \dots & G_p & = 0 \\ 0 = & H_1 & H_2 & H_3 & \dots & H_p & = w_{\mathsf{max}} \\ T_0 = & T_1 & T_2 & T_3 & \dots & T_p & = T_{\mathsf{max}} \end{array}$$

in which F_n, G_n, H_n and T_n sample F, G, H and T at x_n. In general, if f_n and g_n sample functions f and g at x_n, then

$$\begin{aligned} \left.\frac{df}{dz}\right|_{x_n} &\approx \alpha_n^- f_{n-1} + \alpha_n f_n + \alpha_{n+1}^+ f_{n+1} \\ \left.\frac{d}{dz}\left(g\frac{df}{dz}\right)\right|_{x_n} &\approx \beta_n^- f_{n-1} + \beta_n f_n + \beta_{n+1}^+ f_{n+1} \end{aligned}$$

in which the coefficients are messy.

$$\begin{aligned} \alpha_n^- &= \frac{x_n - x_{n+1}}{(x_{n-1}-x_n)(x_{n-1}-x_{n+1})} & \beta_n^- &= \frac{g_n + g_{n-1}}{(x_{n-1}-x_n)(x_{n-1}-x_{n+1})} \\ \alpha_n^+ &= \frac{x_n - x_{n-1}}{(x_n-x_{n+1})(x_{n-1}-x_{n+1})} & \beta_n^+ &= \frac{g_n + g_{n+1}}{(x_n-x_{n+1})(x_{n-1}-x_{n+1})} \\ \alpha_n &= -(\alpha_n^- + \alpha_n^+) & \beta_n &= -(\beta_n^- + \beta_n^+) \end{aligned}$$

The momentum and energy equations discretize by replacing their differentials by these approximations. The resulting equations for $1 < n < p$, and the trivial equations for the boundary values at $n = 1$ and $n = p$, produce $3p$ equations for $3p$ unknowns: F_n, G_n and T_n with $1 \le n \le p$.

Since the energy equation has only first order differentials, it most conveniently discretizes using approximations formed midway between grid points. At $x_{n+1/2} = (x_n + x_{n+1})/2$ there results the equation

$$2\,\frac{F_n + F_{n+1}}{2} + \frac{H_{n+1} - H_n}{x_{n+1} - x_n} + \frac{H_n + H_{n+1}}{2}\,\frac{\ln\rho_{n+1} - \ln\rho_n}{x_{n+1} - x_n} = 0$$

in which ρ_n is the density at x_n.

$$\rho_n = \frac{p_{\mathsf{ref}}}{T_n}\,\frac{W}{R}$$

These equations for $1 < n + \frac{1}{2} < p$, and the trivial equations for the boundary values at $n = 1$ and $n = p$, produce $p + 1$ equations for p unknowns: H_n with $1 \le n \le p$.

Equations and unknowns must balance and do. There are $4p+1$ equations and $4p+1$ unknowns: $4p$ components at points and 1 eigenvalue, λ. If the continuity equation were discretized elaborately like the others however, then only $4p$ equations would result, and the unknowns would be underdetermined.

Figure 6.5 lists some values for the F_n, G_n, H_n, λ_n and T_n solving the discrete equations given above.

7.3 Step 3, Solution

The simulation model ultimately distills to a subroutine that evaluates residual errors given approximate solutions. Section 6.1, and the program fragment in Figure 6.2, show how twopnt uses such subroutines. The present Section shows what goes in them.

The first consideration in writing any computer program is the organization of data. The natural choice here is twopnt's components at points convention

$$\texttt{GROUPA} = 1 \quad \texttt{COMPS} = 4 \quad \texttt{POINTS} = p \quad \texttt{GROUPB} = 0$$

in which F, G, H and T are the components, and the eigenvalue is the single unknown in group A. Unfortunately, the eigenvalue doesn't suit the matrix routines `TWPREP` and `TWSOLV` because it appears in equations at all points. This gives the Jacobian matrices one column outside Figure 6.4's tridiagonal, blocked structure. The resulting matrix equations can be solved by *"stretching"* them [16], but it is traditional to alter the model's nonlinear equations instead. Thus, the eigenvalue is replicated at every point, and extra equations are added to make all the copies the same.

$$\begin{array}{rccccccccccl}
0 = & x_1 & & x_2 & & x_3 & & \dots & & x_p & & = z_{\text{max}} \\
0 = & F_1 & & F_2 & & F_3 & & \dots & & F_p & & = 0 \\
\omega_0 = & G_1 & & G_2 & & G_3 & & \dots & & G_p & & = 0 \\
0 = & H_1 & & H_2 & & H_3 & & \dots & & H_p & & = w_{\text{max}} \\
 & \lambda_1 & = & \lambda_2 & = & \lambda_3 & = & \dots & = & \lambda_p & & \\
T_0 = & T_1 & & T_2 & & T_3 & & \dots & & T_p & & = T_{\text{max}}
\end{array}$$

This device increases equations and unknowns to $5p$ each.

$$\texttt{GROUPA} = 0 \quad \texttt{COMPS} = 5 \quad \texttt{POINTS} = p \quad \texttt{GROUPB} = 0$$

Both the unknowns and the residuals must be arranged carefully, because as explained in Section 6.2, the arrangement determines the matrix structure. For F, G and T (each with $p - 2$ discrete and 2 boundary equations), the equations can pair with the unknowns. However, there is no natural pairing for H (with $p - 1$ discrete and 2 boundary equations), nor for λ (with $p - 1$ equations). An arrangement acceptable to `TWPREP` and `TWSOLV` pairs the boundary equation for H_p with the unknown λ_p, and pairs the energy equation at $x_{n+1/2}$ with the unknown H_n.

```
      SUBROUTINE TWFUNC                                                      1
     +  (ERROR, TEXT,                                                        2
     +   BUFFER, F, F0, G, G0, H, K, LAMBDA, MU, OMEGA, POINTS, RHO,         3
     +   STRIDE, T, T0, TIME, TMAX, TZERO, U0, WMAX, X)                      4
                                                                             5
C///////////////////////////////////////////////////////////////////////     6
C                                                                            7
C     TWFUNC                                                                 8
C                                                                            9
C     SAMPLE RESIDUAL FUNCTION FOR SIMULATING SWIRLING FLOWS.               10
C                                                                           11
C///////////////////////////////////////////////////////////////////////    12
                                                                            13
      IMPLICIT COMPLEX (A - P, R - Z), INTEGER (Q)                          14
      CHARACTER ID*9                                                        15
      INTEGER COMPS, J, POINTS, TEXT                                        16
      LOGICAL ERROR, TIME                                                   17
C*****PRECISION > SINGLE                                                    18
C      REAL                                                                 19
C*****END PRECISION > SINGLE                                                20
C*****PRECISION > DOUBLE                                                    21
      DOUBLE PRECISION                                                      22
C*****END PRECISION > DOUBLE                                                23
     +   A, A0, A1, A2, B0, B1, B2, BUFFER, C0, C1, C2, CP, F, F0, G,       24
     +   G0, H, K, LAMBDA, MU, OMEGA, P, R, RHO, STRIDE, T, T0, TEMP,       25
     +   TMAX, TZERO, U0, W, WMAX, X                                        26
```

The `TWFUNC` subroutine evaluates residuals using the data arrangement described above. The subroutine communicates through the array `BUFFER` by replacing solution data with residual data in the proper arrangement. Some of the other subroutine arguments are parameters for the model—such as `OMEGA`, which is ω_0 (LINES 1 TO 4).

```
                                                                          27
      PARAMETER (ID = 'TWFUNC:  ')                                        28
      PARAMETER (COMPS = 5)                                               29
                                                                          30
C     PRESSURE AT 1 STANDARD ATMOSPHERE                                   31
      PARAMETER (P = 1013250.0)                                           32
                                                                          33
C     MOLECULAR WEIGHT OF ARGON                                           34
      PARAMETER (W = 39.948)                                              35
                                                                          36
C     UNIVERSAL GAS CONSTANT                                              37
      PARAMETER (R = 83140000.0)                                          38
                                                                          39
C     SPECIFIC HEAT OF ARGON AT CONSTANT PRESSURE (ERGS / (GM * K))       40
      PARAMETER (CP = R * 2.5 / W)                                        41
                                                                          42
      DIMENSION                                                           43
     +   BUFFER(COMPS, POINTS), F(POINTS), F0(POINTS), G(POINTS),         44
     +   G0(POINTS), H(POINTS), K(POINTS), LAMBDA(POINTS), MU(POINTS),    45
     +   RHO(POINTS), T(POINTS), T0(POINTS), U0(COMPS, POINTS),           46
     +   X(POINTS)                                                        47
                                                                          48
C/////////////////////////////////////////////////////////////////////// 49
C                                                                         50
C     PROLOGUE.                                                           51
C                                                                         52
C/////////////////////////////////////////////////////////////////////// 53
                                                                          54
C///  CHECK THE ARGUMENTS.                                                55
                                                                          56
      ERROR = .NOT. (2 .LE. POINTS)                                       57
      IF (ERROR) GO TO 9001                                               58
                                                                          59
C///  UNPACK THE VARIABLES.                                               60
                                                                          61
      DO 1010 J = 1, POINTS                                               62
         F(J) = BUFFER(1, J)                                              63
         G(J) = BUFFER(2, J)                                              64
         H(J) = BUFFER(3, J)                                              65
         LAMBDA(J) = BUFFER(4, J)                                         66
         T(J) = BUFFER(5, J)                                              67
         F0(J) = U0(1, J)                                                 68
         G0(J) = U0(2, J)                                                 69
         T0(J) = U0(5, J)                                                 70
1010  CONTINUE                                                            71
```

Most arguments are arrays that ease reference and movement of data within the subroutine (LINES 43 TO 47). The subroutine begins by copying values into these mnemonically named arrays (LINES 62 TO 71). It copies an approximate solution from `BUFFER`, and it copies a transient state from `U0` (where it was placed at the last `SIGNAL` = `` `RETAIN' ``).

```
                                                                          72
C///  CALCULATE DENSITIES, VISCOSITIES AND THERMAL CONDUCTIVITIES.        73
                                                                          74
      DO 1020 J = 1, POINTS                                               75
         RHO(J) = (P * W) / (R * T(J))                                    76
                                                                          77
         A = LOG (T(J))                                                   78
         MU(J) = EXP                                                      79
     +      (((0.0121673 * A - 0.284023) * A + 2.85205) * A - 17.6539)    80
         K(J) = EXP                                                       81
     +      (((0.0121673 * A - 0.284023) * A + 2.85205) * A - 1.78377)    82
1020  CONTINUE                                                            83
                                                                          84
```

The subroutine next performs some preliminary calculations. It stores fluid properties such as density (LINES 75 TO 83) both for convenience and to avoid repeated evaluation.

```
C/////////////////////////////////////////////////////////////////////// 85
C                                                                         86
C     F, G AND T EQUATIONS                                                87
C                                                                         88
C     RHO * (dF/dt + F ** 2 - G ** 2 + F' H) - (MU F')' + P = 0           89
C                                                                         90
C     RHO * (dG/dt + 2 F G + G' H) - (MU G')' = 0                         91
C                                                                         92
C     RHO * CP * (dT/dt + H T') - (K T')' = 0                             93
C                                                                         94
C/////////////////////////////////////////////////////////////////////// 95
                                                                          96
C///  BOUNDARY CONDITIONS.                                                97
                                                                          98
      BUFFER(1, 1) = F(1) - 0.0                                           99
      BUFFER(1, POINTS) = F(POINTS) - 0.0                                100
                                                                         101
      BUFFER(2, 1) = G(1) - OMEGA                                        102
      BUFFER(2, POINTS) = G(POINTS) - 0.0                                103
                                                                         104
      BUFFER(5, 1) = T(1) - TZERO                                        105
      BUFFER(5, POINTS) = T(POINTS) - TMAX                               106
                                                                         107
C///  EQUATIONS.                                                         108
                                                                         109
      DO 2010 J = 2, POINTS - 1                                          110
         TEMP = ((X(J - 1) - X(J)) * (X(J - 1) - X(J + 1)))              111
```

The subroutine finally begins building the residual vector. Since values for the unknowns have been copied out of `BUFFER`, values for the residuals can be stored there directly. Equations associated with F, G, and T depend on values at three points, so their residuals are evaluated together. Residual formulas for boundary equations are simple (LINES 99 TO 106), but formulas for discretized equations are complicated. Those formulas are written more easily if some coefficients are evaluated separately (LINES 111 TO 123). The subroutine evaluates the steady state residual first (LINES 125 TO 139), and then adds time dependent terms if requested (LINES 141 TO 148).

```
         A0 = (X(J) - X(J + 1)) / TEMP                                  112
         B0 = (MU(J) + MU(J - 1)) / TEMP                                113
         C0 = (K(J) + K(J - 1)) / TEMP                                  114
                                                                        115
         TEMP = ((X(J) - X(J + 1)) * (X(J - 1) - X(J + 1)))             116
         A2 = (X(J) - X(J - 1)) / TEMP                                  117
         B2 = (MU(J + 1) + MU(J)) / TEMP                                118
         C2 = (K(J + 1) + K(J)) / TEMP                                  119
                                                                        120
         A1 = - (A0 + A2)                                               121
         B1 = - (B0 + B2)                                               122
         C1 = - (C0 + C2)                                               123
                                                                        124
         BUFFER(1, J)                                                   125
     +      = RHO(J) * (F(J) ** 2 - G(J) ** 2 + H(J)                    126
     +      * (A0 * F(J - 1) + A1 * F(J) + A2 * F(J + 1)))              127
     +      - (B0 * F(J - 1) + B1 * F(J) + B2 * F(J + 1))               128
     +      + LAMBDA(J)                                                 129
                                                                        130
         BUFFER(2, J)                                                   131
     +      = RHO(J) * (2.0 * F(J) * G(J) + H(J)                        132
     +      * (A0 * G(J - 1) + A1 * G(J) + A2 * G(J + 1)))              133
     +      - (B0 * G(J - 1) + B1 * G(J) + B2 * G(J + 1))               134
                                                                        135
         BUFFER(5, J)                                                   136
     +      = RHO(J) * CP * H(J)                                        137
     +      * (A0 * T(J - 1) + A1 * T(J) + A2 * T(J + 1))               138
     +      - (C0 * T(J - 1) + C1 * T(J) + C2 * T(J + 1))               139
                                                                        140
         IF (TIME) THEN                                                 141
            BUFFER(1, J)                                                142
     +         = BUFFER(1, J) + RHO(J) * (F(J) - F0(J)) / STRIDE        143
            BUFFER(2, J)                                                144
     +         = BUFFER(2, J) + RHO(J) * (G(J) - G0(J)) / STRIDE        145
            BUFFER(5, J)                                                146
     +         = BUFFER(5, J) + RHO(J) * CP * (T(J) - T0(J)) / STRIDE   147
         END IF                                                         148
2010  CONTINUE                                                          149
                                                                        150
C///////////////////////////////////////////////////////////////////////151
C                                                                       152
C     H AND P EQUATIONS                                                 153
C                                                                       154
C     2 F + H' + H * (LOG RHO)' = 0                                     155
C                                                                       156
C     P CONSTANT                                                        157
C                                                                       158
C///////////////////////////////////////////////////////////////////////159
                                                                        160
C///  BOUNDARY CONDITIONS.                                              161
                                                                        162
      BUFFER(1, 1) = F(1) - 0.0                                         163
                                                                        164
      BUFFER(3, 1) = H(1) - 0.0                                         165
      BUFFER(4, POINTS) = H(POINTS) - WMAX                              166
                                                                        167
C///  EQUATIONS.                                                        168
                                                                        169
      DO 3010 J = 2, POINTS                                             170
         BUFFER(3, J)                                                   171
     +      = F(J) + F(J - 1)                                           172
     +      + (H(J) - H(J - 1)) / (X(J) - X(J - 1))                     173
     +      + 0.5 * (H(J) + H(J - 1))                                   174
     +      * (LOG (RHO(J)) - LOG (RHO(J - 1))) / (X(J) - X(J - 1))     175
                                                                        176
         BUFFER(4, J - 1) = LAMBDA(J) - LAMBDA(J - 1)                   177
3010  CONTINUE                                                          178
                                                                        179
C///////////////////////////////////////////////////////////////////////180
C                                                                       181
C     ERROR MESSAGES.                                                   182
C                                                                       183
C///////////////////////////////////////////////////////////////////////184
                                                                        185
      GO TO 99999                                                       186
                                                                        187
9001  IF (0 .LT. TEXT) WRITE (TEXT, 99001) ID, POINTS                   188
      GO TO 99999                                                       189
                                                                        190
99001 FORMAT                                                            191
     +   (/1X, A9, 'ERROR.  THERE MUST BE AT LEAST TWO POINTS.'         192
     +   //10X, I10, '  POINTS')                                        193
                                                                        194
C///  EXIT.                                                             195
                                                                        196
```

Equations associated with H and λ depend on values at two points, so their residuals are evaluated apart from the others (LINES 163 TO 176). These equations also have no time dependent terms.

```
99999 CONTINUE
      RETURN
      END
```

Section 6.1 and Figure 6.2 explain how a simulator program might use twopnt and TWFUNC to perform a simulation. The program mostly orchestrates reverse communication and gives values to twopnt's controls. The present simulation begins with a grid of six points spaced uniformly from 0 to $z_{\textsf{max}}$. It guesses solution values for F, G, H and T by linear interpolation of boundary values, and it guesses 0 for λ. It chooses

component	ACTIVE	BELOW	ABOVE
F	*yes*	-4	4
G	*yes*	-10^4	10^4
H	*yes*	-10^4	10^4
λ	*no*	-10^4	10^4
T	*yes*	$T_0/2$	$2\,T_{\textsf{max}}$

with the large bounds, $\pm 10^4$, intended to disable checking of bounds for many unknowns. The simulation accepts twopnt's defaults for many controls, except

ADAPT $=$ *true*	TINC $= \sqrt{10}$
LEVELD $= 0$	STEPS2 $= 25$
STEPS1 $= 50$	TOLER1 $= 0.1$
STRID0 $= 10^{-3}$	TOLER2 $= 0.1$

so CNTRLS $= 8$ in this example. Twopnt's default values would suffice, but different values have been chosen to make the example interesting. Both the residual subroutine discussed here and a fortran main program that uses it are supplied with twopnt, see Appendix 2.

7.4 Errors

An assessment of von Neumann's errors for this simulation would proceed as follows. The errors of the model lie in the dimension reduction of the Navier-Stokes and related equations, and ultimately, in the continuum assumptions underlying these equations.

(A)
$$\lambda + \rho \left(\frac{\partial F}{\partial t} + F^2 - G^2 + \frac{\partial F}{\partial z} H \right) - \frac{\partial}{\partial z} \left(\mu \, \frac{\partial F}{\partial z} \right) = 0$$

The errors of measurement occur in the physical "constants," and in the data from which formulas for such things as viscosity have been derived.

(B)

R universal gas constant $8.314 \cdot 10^7$ ergs / (mole K)

$$\mu(T) = \exp(((0.0121673\alpha - 0.284023)\alpha + 2.85205)\alpha - 17.6539)$$

The errors of approximation stem both from the formulas replacing the differential equations' derivatives, and also from the settings of twopnt's controls for grid selection and for halting Newton's search.

(C)
$$\left. \frac{df}{dz} \right|_{x_n} \approx \alpha_n^- f_{n-1} + \alpha_n f_n + \alpha_{n+1}^+ f_{n+1}$$

SSREL $= 10^{-6}$ TOLER1 $= 0.1$

7.4 Errors

The errors of precision lie in the choice of computer, and the manner of evaluating all the various formulas of the calculation.

```
DOUBLE PRECISION VERSION 3.08
```

(D)

$$\alpha_n^- = \frac{x_n - x_{n+1}}{(x_{n-1} - x_n)(x_{n-1} - x_{n+1})}$$

When a problem in pure or applied mathematics is “solved” by numerical computation, errors, that is, deviations of the numerical “solution” obtained from the true, rigorous one, are unavoidable. Such a “solution” is therefore meaningless, unless there is an estimate of the total error . . .

The errors described in (A) are the *errors due to the theory.* While their role is clearly the most important, their analysis and estimation should not be considered part of the mathematical or of the computational phase of the problem, but of the underlying subject, in which the problem originates.

The errors described in (B) are the *errors due to observation.* To this extent they are, strictly construed, again no concern of the mathematician. However, their influence on the result is the thing that really matters. In this way, their analysis requires an analysis of the question: What are the limits of the change of the result, caused by changes of the parameters (data) of the problem within given limits?

The errors described in (C) are those which are most conspicuous as *errors of approximation* or *truncation.* Most discussions in “approximation mathematics” are devoted to analysis and estimation of these errors: Numerical methods to obtain approximate solutions of algebraical equations by iterative and interpolation processes . . . and so on.

We now come to the errors described in (D). As we saw, they are due to the inner “noise level” of the numerical computing procedure or device—in the digital case this means: to the round off errors.

— J. von Neumann and H. H. Goldstine [26]

Appendix 1 Common Keywords

This Appendix cross-references twopnt's controls with "keywords" that manipulate the controls for some simulators written at Sandia National Laboratories in Livermore, California. The cross-reference table is possible thanks to F. M. Rupley [29]. The simulators, their current versions, and what they simulate, are as follows.

simulator	version	simulation
`CRESLAF`		chemical vapor deposition reactors [5]
`PREMIX`		laminar premixed flames [21]
`PSR`		perfectly stirred chemical reactors [13]
`SPIN`	`3.83`	rotating disk and stagnation flow chemical vapor deposition reactors [6]
`SURPSR`		perfectly stirred chemical reactors with both gas and surface reactions [25]

These simulators are used in the Laboratories' own research, which is supported for the most part by the United States Department of Energy, and they are available for others' research too.

The simulators' reference manuals should be consulted before using the keywords. The page numbers in the table below refer to the descriptions of the controls in this manual. Some controls have identical keywords because those keywords accept multiple data. Other controls have multiple keywords, because (in the case of `STEPS1` and `STRID0`) the simulators perform multiple simulations, or because (in the case of `U` and `X`) initial guesses and grids have complicated descriptions.

twopnt	simulator					manual
control	CRESLAF	PREMIX	PSR	SPIN	SURPSR	pages
ABOVE						14, 33
ACTIVE						17, 33
ADAPT						16, 34
BELOW		SFLR	SFLR	SFLR	SFLR	14, 33
LEVELD	TWPR	PRNT	PRNT	PRNT	PRNT	21, 34
LEVELM	TWPR	PRNT	PRNT	PRNT	PRNT	21, 34
PADD		NADP		NADP		18, 34
PMAX		NTOT		NMAX		18, 34
POINTS	NPTS	NPTS		NPTS		35
SSABS	TWAB	ATOL	ATOL	ATOL	ATOL	13, 35
SSAGE	NJAC	NJAC	NJAC	NJAC	NJAC	13, 37
SSREL	TWRE	RTOL	RTOL	RTOL	RTOL	13, 35
STEADY				ISTP		12, 37
STEPS0	ISTP	ISTP	ISTP	ISTP	ISTP	15, 38
STEPS1	TWST	TIME TIM2	TIME TIM2	TIME TIM2	TIME TIM2	15, 38
STEPS2	IRET	IRET	IRET	IRET	IRET	15, 38
STRID0	STP0	TIME TIM2	TIME TIM2	TIME TIM2	TIME TIM2	15, 38
TDABS	TWTA	ATIM	ATIM	ATIM	ATIM	13, 35
TDAGE	TJAC	TJAC	TJAC	TJAC	TJAC	13, 37
TDEC		DFAC	DFAC	DFAC	DFAC	15, 38
TDREL	TWTR	RTIM	RTIM	RTIM	RTIM	13, 35
TINC		UFAC	UFAC	UFAC	UFAC	15, 38
TMAX	DTMX	DTMX	DTMX	DTMX	DTMX	15, 38
TMIN	DTMN	DTMN	DTMN	DTMN	DTMN	15, 38
TOLER0						17, 39
TOLER1		GRAD		GRAD		17, 39
TOLER2		CURV		CURV		18, 39
U	REAC SURF	INTM PROD REAC TEMP WCEN WMIX	PROD REAC TEMP	INTM PROD REAC SURF WCEN WMIX	PROD REAC SURF TEMP	40
X	NPTS XEND	GRID NPTS XEND		GRID NPTS XEND		40

Appendix 2 Software Notes

Twopnt is distributed in two files. The first, `TWOPNT.FOR` or `twopnt.f`, contains the twelve fortran subroutines which constitute twopnt itself and the three ancillary routines discussed in Section 6.4.

`EVOLVE`	perform time evolution
`REFINE`	perform automatic grid selection
`SEARCH`	perform the damped, modified Newton's search
`TWCOPY`	copy one vector to another
`TWGRAB`	reserve space in an array
`TWLAPS`	obtain elapsed computing time in seconds
`TWLAST`	find the last nonblank character in a string
`TWLOGR`	write a common logarithm to a character string
`TWNORM`	compute the infinity-norm of a vector
`TWOPNT`	twopnt main entry and task manager
`TWPREP`	ancillary routine for preparing Jacobian matrices
`TWSHOW`	ancillary routine for writing solution data
`TWSOLV`	ancillary routine for solving matrix equations
`TWSQEZ`	squeeze unnecessary blanks from a character string
`TWTIME`	obtain computing time in seconds

The second file, `TWMAIN.FOR` or `twmain.f`, contains Chapter 7's simulator whose output appears throughout this manual.

`TWMAIN`	main program
`TWFUNC`	evaluate the residual function

All subroutines except `TWTIME` are believed to conform to the fortran standard [1]. In observance of that standard, all subroutines are written in `BIG LETTERS`.

The `TWTIME` subroutine does not conform to the fortran standard because the standard does not specify how to obtain elapsed computing time. Compilers or linkers often give up on the first try when they reach this subroutine, and then `TWTIME` must be adapted to the computing system in use. The subroutine has provisions for a variety of computing machines, and may need merely some editing to select one of those—this can be done painlessly, see below. In some cases however, the subroutine may need rewriting to obtain the correct time. If all else fails, `TWTIME` can return *zero* time. Twopnt does nothing with the time but write it in messages.

All twopnt's subroutines contain "*change blocks*" that facilitate basic reprogramming, when necessary. A change block is a small group of fortran statements, as follows.

```
C*****name of the change block
      ⋮
      text of the change block
      ⋮
C*****END  name of the change block
```

The change block above is *active*; the block below is *inactive* because has been "commented out."

```
C*****name of the change block
C       ⋮
C       text of the inactive block
C       ⋮
C*****END  name of the change block
```

A utility program exists to alter the status of change blocks [17], or the status can be changed by hand, using a text editor.

Twopnt uses change blocks to select arithmetic precision and to select the version of `TWTIME` compatible with the computing machine in use. The blocks have names that explain what they do.

```
COMPUTING TIME > CRI (CRAY) CTSS (LIVERMORE)
COMPUTING TIME > CRI (CRAY) CTSS (LOS ALAMOS)
COMPUTING TIME > CRI (CRAY) UNICOS
COMPUTING TIME > DEC (VAX) VMS
COMPUTING TIME > IBM (RISC System/6000) AIX
COMPUTING TIME > SUN (SPARCstation) SunOS
COMPUTING TIME > generic unix etime
COMPUTING TIME > none

LIST MESSAGES > NO
LIST MESSAGES > YES

PRECISION > DOUBLE
PRECISION > SINGLE
```

Only one block of each kind should be active. When twopnt is distributed, usually, the blocks are set to anticipate the recipient's needs. The `COMPUTING TIME >` ... blocks appear only in the subroutine `TWTIME` which, as noted, may be edited by hand to select the proper block. `LIST MESSAGES > YES` prints messages for preparing this manual; this option should not be selected.

Twopnt is complete and needs no other subroutines beyond those in the `twopnt.f` or `TWOPNT.FOR` file. Twopnt calls, or links, all the subroutines there except the three ancillary ones. If those are used, then the simulator that calls twopnt calls them.

Two ancillary routines are incomplete. `TWPREP` and `TWSOLV` need some subroutines from the linpack library [8]. Specifically, they need the double precision routines `DGBCO` and `DGBSL`, or the single precision routines `SGBCO` and `SGBSL`, and those subroutines in turn need a few more from that library. Linpack isn't needed if the simulator doesn't use it, either directly for matrix chores or indirectly through `TWPREP` and `TWSOLV` (the `TWMAIN` example does use `TWPREP` and `TWSOLV`). Many computing centers supply the linpack library, and a variety of sources distribute it gratis. The simulators discussed in Appendix 1 include the required linpack routines in their distribution.

Appendix 3
Change History

This Appendix chronicles the changes made to twopnt by version number and by date. Version 1 was used in house at Sandia National Laboratories in Livermore, California while twopnt and the first few simulators based on twopnt were developed. Version 2 has been widely distributed with these simulators. Version 3 has a more extensible programming interface and is the first with separate documentation (this manual).

`1.00` November 1984.

`1.01` September 1985. Remove unused subroutine simple. Shorten names to six characters.

`1.02` September 1985 (succeeds 1.00). Remove unused subroutine simple. Rewrite newton and timstp.

`1.03` October 1985 (succeeds 1.01). Rewrite newton and timstp.

`1.04` October 1985. In timstp: add the xsave argument; alter an error message. In refine: accommodate negative points in the grid display.

`1.05` October 1985. Add an absolute/relative stopping test to newton via reverse communication.

`1.06` November 1985. Apply the relative stopping test to each vector entry rather than to the vector norm.

`1.07` November 1985. In twopnt: change the introductory message; go to the main loop rather than to newton after refine.

`1.08` November 1985. Force newton to step to the boundary.

`1.09` November 1985. Add intwor to write integers to strings because some llnl compilers have trouble with internal files. Eliminate a redundant write statement in timstp. Order the integer type statement in refine. In timwor: correct the formula for seconds; allow ten or more hours. In twopnt: add a decision tree, replace computed branches by assigned branches; reorder the arguments; correct the dimension of array column.

`1.10` November 1985. In twopnt: initialize condit to zero; rename and reorder the arguments. In refine: alter the significance criteria; expand above and below with the grid; interpolate inactive components at new points; prevent refinement when toler1 and toler2 equal 1. Correct the calculation of minutes in timwor. Move level2 printing decisions to newton and timstp.

1.11 December 1985. In refine: add a completion status flag; move level2 printing decisions and the update reverse communication request here; increase the dimension of vary from (pmax - 1) to (pmax). In twopnt: store the completion status in a local variable during reverse communication; remove the initialization of condit; alter statistics gathering by removing statement functions and reducing the arguments in elapse.

2.00 December 1985. Correct newton to print condition estimates when a new matrix produces a step that satisfies the termination criteria.

2.01 January 1986. Add the ability to write residuals for graphing. In newton, replace computed branches by assigned branches.

2.02 January 1986. Add change blocks. Replace blas to minimize precision dependencies. In intwor, make character string lengths 8 because cray/ctss requires word boundaries for internal files.

2.03 January 1986. Write residual data in single precision.

2.04 February 1986. In timwor, replace nint because cray/ctss lacks the double precision intrinsic function.

2.05 October 1986. In twopnt, initialize xreent. In timstp, recognize that no change to the solution over a time step is an error.

2.06 August 1987. In timstp, recognize that no change to the solution over a time step after the first is not an error. Add report to explain unsuccessful completions. Modify twopnt and timstp to print the actual number of time steps performed.

2.07 August 1987. Correct timstp to exit on success rather than entering the reverse communication blocks. Correct logical expressions by removing the eq relation.

2.08 September 1987. Correct twopnt to avoid changing completion status flags following reverse communication.

2.09 December 1987. Add plimit to bound grid growth. Add pass and passes to allow several simulations per grid. Make cosmetic changes to twopnt's output. Added report to newton.

2.10 December 1987. Add time stride selection to timstp.

2.11 December 1987. In timstp: correct saving and restoring of solution values; alter printing.

2.12 February 1988. Change the appearance of many informative messages. Remove intwor; replace logwor by logstr; add grab. In timstp: replace the stride change factor by separate factors for increase and decrease; add a minimum stride. In refine, replace integer vary by logical mark. In twopnt: replace several arrays by work space arrays; bring printing and work space arguments to the front.

2.12R February 1988. Correct a bug in 2.12, see 2.15.

2.13 April 1989 (succeeds 2.12). Add the Los Alamos cray/ctss environment to cputim. Correct the counting of matrices in twopnt.

2.14 October 1989. In cputim: add the cray/unicos environment; correct the vax/vms environment to make a system service call. In newton: correct the alignment of column labels. In twopnt: note the precision in the banner.

2.15 November 1989. In twopnt: correct an out of range subscript during initialization; correct printing of the time step count.

`2.16` February 1990. In twopnt, remove the dead branch before the error messages. In newton: remove the record reverse communication request; re-evaluate the function after matrix preparation in case preparation changes the function.

`2.17` February 1990. Force newton to take one step. If newton makes no change, then since linear interpolation preserves the kinks and slopes that prompt grid refinement, refine can add points forever.

`2.18` February 1990. Add sun unix environment to cputim.

`2.19` April 1990. In newton, add printing of inconsistent bounds to the error message. In refine, correct the limits of loops for gathering statistics.

`2.20` August 1990. Simplify stride selection in timstp. Rename limit to steps0; add steps1, ssage, tdage, and tmax. Limit newton to 100 steps per time step.

`2.21` September 1990. Correct timstp logic.

`2.22` September 1990. Correct timstp logic, again.

`2.22B` March 1991. Omit steady state search when steps0 is negative.

`2.22C` March 1991. Relax monitoring of newton step sizes.

`2.23` January 1991 (succeeds 2.22). Rewrite newton. Add norm2.

`3.00` June 1991. Reduce twopnt's arguments, adding a control list for many. Rename all routines, using the tw prefix for most. Rewrite many routines, conforming to the manual. Change the appearance and content of many messages.

`3.01` June 1991. Remove explicit dimensioning of arrays `ACTIVE`, `MARK` and `X`, and remove initialization of array `MARK`, so array space isn't needed when grid selection is disallowed.

`3.02` August 1991. Correct `SEARCH` to place solution in `BUFFER` when `SIGNAL` = `'SHOW'`.

`3.03` August 1991. Change `TWNORM` from the 2-norm to the ∞-norm, thus reverting to pre-version 3.00 practice.

`3.04` August 1991. Correct choice of `DELTAB`.

`3.05` October 1991. Correct subscripting in `TWPREP`.

`3.06` January 1992. (1) Increase default `TDEC` from just below $\sqrt{10}$ to just above, thus ensuring reduced time strides overtake `TMIN`. (2) Correct `TWOPNT` to forgo time evolution when `SEARCH` fails and `EVOLVE` has already failed.

`3.07` January 1992. (1) Remove unnecessary restoration of initial value to `POINTS`, thus allowing a constant value for the actual argument when grid selection is disallowed. (2) Include change blocks for cpu time on more computing systems.

`3.08` January 1992. Add arguments `NAME` and `NAMES`.

`3.09` February 1992. Correct dimension of `NAME`.

`3.10` March 1992. Correct error messages in `SEARCH` and `TWOPNT`.

References

[1] Anonymous, *ANSI X3.9-1978 American National Standard Programming Language FORTRAN*, American National Standards Institute, New York, 1978.

[2] U. M. Ascher, R. M. M. Mattheij and R. D. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[3] G. K. Batchelor, "Note on a Class of Solutions of the Navier-Stokes Equations representing Steady Rotationally-symmetric Flow," *The Quarterly Journal of Mechanics and Applied Mathematics,* volume 4, 1951, pages 29–41.

[4] K. E. Brenan, S. L. Campbell and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations,* North-Holland Elsevier, New York, 1989.

[5] M. E. Coltrin and R. J. Kee, *CRESLAF: A Fortran Program for Modeling Chemically-Reacting Boundary Layer Flow in a Chemical Vapor Deposition Reactor,* Sandia National Laboratories Report SAND9x-xxxx, Livermore, California, 199x. In preparation.

[6] M. E. Coltrin, R. J. Kee, G. H. Evans, E. Meeks, F. M. Rupley and J. F. Grcar, *SPIN (Version 3.83): A Fortran Program for Modeling One-Dimensional Rotating-Disk / Stagnation-Flow Chemical Vapor Deposition Reactors,* Sandia National Laboratories Report SAND91-8003, Livermore, California, May 1991.

[7] D. S. Dandy, *private communication,* Sandia National Laboratories, Livermore, California, 1991.

[8] J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users Guide,* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1978.

[9] I. S. Duff, A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford University Press, Oxford, 1986.

[10] G. H. Evans and R. Greif, "Forced Flow Near a Heated Rotating Disk: A Similarity Solution," *Numerical Heat Transfer,* volume 14, 1988, pages 363–387.

[11] J. H. Ferziger, *Numerical Methods for Engineering Applications,* Wiley, 1981.

[12] V. Giovangigli and N. Darabiha, “Vector computers and complex chemistry combustion,” in *Mathematical Modeling in Combustion and Related Topics,* edited by C.-M. Brauner and C. Schmidt-Lainé, Martinus Nijhoff Publishers, Dordrecht, 1988, pages 491–503.

[13] P. Glarborg, R. J. Kee, J. F. Grcar and J. A. Miller, *PSR: A Fortran Program for Modeling Well-Stirred Reactors,* Sandia National Laboratories Report SAND86-8209, Livermore, California, February 1991.

[14] G. H. Golub and C. F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, Baltimore, 1983.

[15] J. F. Grcar, R. J. Kee, M. D. Smooke and J. A. Miller, “A hybrid Newton / time-integration procedure for the solution of steady, laminar, one-dimensional, premixed flames,” in *Twenty-first Symposium (International) on Combustion,* The Combustion Institute, Pittsburgh, Pennsylvania, 1986, pages 1773–1782.

[16] J. F. Grcar, *Matrix Stretching for Linear Equations,* Sandia National Laboratories Report SAND90-8723, Livermore, California, 1990. Accepted by *SIAM Review*.

[17] J. F. Grcar, *The Change Utility for Customizing Fortran Programs,* Sandia National Laboratories Report SAND9x-xxxx, Livermore, California, 199x. In preparation.

[18] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations, Volume II, Stiff and Differential-Algebraic Problems,* Springer Verlag, 1991.

[19] T. von Kármán, “Über laminare und turbulente Reibung,” *Z. Angew. Math. Mech.,* volume 1, number 4, 1921, pages 233–253.

[20] R. J. Kee, G. Dixon-Lewis, J. Warnatz, M. E. Coltrin and J. A. Miller, *A Fortran Computer Code Package for the Evaluation of Gas-Phase Multicomponent Transport Properties,* Sandia National Laboratories Report SAND86-8246, Livermore, California, 1986. Reprinted November, 1988.

[21] R. J. Kee, J. F. Grcar, M. D. Smooke and J. A. Miller, *A Fortran Program for Modeling Steady Laminar One-Dimensional Premixed Flames,* Sandia National Laboratories Report SAND85-8240, Livermore, California, December 1985.

[22] R. J. Kee, L. R. Petzold, M. D. Smooke and J. F. Grcar, “Implicit Methods in Combustion and Chemical Kinetics Modeling,” in *Multiple Time Scales,* edited by J. U. Brackbill and B. I. Cohen, Academic Press, Orlando, Florida, 1985, pages 113–144.

[23] R. J. Kee, F. M. Rupley and J. A. Miller, *Chemkin-II: A Fortran Chemical Kinetics Package for the Analysis of Gas-Phase Chemical Kinetics*, Sandia National Laboratories Report SAND89-8009, Livermore, California, September 1989.

[24] H. B. Keller, “Numerical Solution of Bifurcation and Nonlinear Eigenvalue Problems,” in *Applications of Bifurcation Theory,* edited by P. H. Rabinowitz, Academic Press, 1977, pages 359–384.

[25] H. K. Moffat, P. Glarborg, R. J. Kee, J. F. Grcar and J. A. Miller, *SURFACE PSR: A Fortran Program for Modeling Well-Stirred Reactors with Gas and Surface Reactions,* Sandia National Laboratories Report SAND91-8001, Livermore, California, May 1991.

[26] J. von Neumann and H. H. Goldstine, “Numerical inverting of matrices of high order,” *Bulletin of the American Mathematical Society*, volume 53, 1947, pages 1021–1099.

[27] J. O. Olsson, O. Lindgren and O. Andersson, “Efficient Formation of Numerical Jacobian Used in Flame Codes,” *Combustion Science and Technology,* volume 77, 1991, pages 319–327.

[28] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables,* Academic Press, New York, 1970.

[29] F. M. Rupley, *Keywords for Reacting Flow Simulators,* Sandia National Laboratories, Livermore, California, 1991.

[30] H. Schlichting, *Boundary-Layer Theory,* seventh edition, McGraw-Hill, New York, reissued 1987.

[31] M. D. Smooke, “Solution of Burner-Stabilized Pre-mixed Laminar Flames by Boundary Value Methods,” *Journal of Computational Physics,* volume 48, 1982, pages 72–105.

[32] M. D. Smooke and R. M. M. Mattheij, “On the solution of nonlinear two point boundary value problems on successively refined grids,” *Applied Numerical Mathematics,* volume 1, 1985, pages 463–487.

UNLIMITED RELEASE

INITIAL DISTRIBUTION

O. Andersson
Gothenburg Universities' Computing Center
S-400 12 Gothenburg
Sweden

J. Y. Chen
University of California, Berkeley
Dept. of Mechanical Engineering
Berkeley, CA 94720

V. Giovangigli
Ecole Polytechnique
Centre de Mathématique Appliquées
U. A. CNRS 756
91128 Palaiseau cedex
France

P. Glarborg
Technical University of Denmark
Laboratory of Heating and Air Conditioning
2800 Lyngby
Denmark

O. Lindgren
Dept. of Physical Chemistry
Chalmers University of Technology
S-412 96 Gothenburg
Sweden

J. O. Olsson
Dept. of Physical Chemistry
Chalmers University of Technology
S-412 96 Gothenburg
Sweden

M. D. Smooke
Yale University
Dept. of Mechanical Engineering
P. O. Box 2159 Yale Station
New Haven, CT 06520

1126	W. G. Breiland
1126	M. E. Coltrin
1126	P. Ho
1126	H. K. Moffat
1512	A. S. Geller
1553	W. L. Hermina
8000	J. C. Crawford Attn: D. L. Crawford, 1900 R. J. Detry, 8200 P. L. Mattern, 8300 R. C. Wayne, 8400
8240	C. W. Robinson
8244	R. S. Larson
8245	D. S. Dandy
8245	H. A. Dwyer
8245	G. H. Evans
8245	J. F. Grcar (10)
8245	W. G. Houf
8245	R. J. Kee
8245	A. E. Lutz
8245	E. Meeks
8245	F. M. Rupley (150)
8245	P. A. Spence
8245	W. S. Winters
8353	J. A. Miller
8361	M. D. Allendorf
8535	Publications for OSTI (10)
8535	Publications/Technical Library Processes, 3141
3141	Technical Library Processes Division (3)
8524-2	Central Technical Files (3)

SUBSEQUENT DISTRIBUTION

MS 9042	J. F. Grcar, 8345
MS 9042	F. M. Rupley, 8345 (100)
MS 0899	Technical Library, 7141 (4)
MS 9018	Central Technical Files, 8950-2 (3)