NUMERICAL SOLUTION of NONLINEAR ALGEBRAIC SYSTEMS in BUILDING ENERGY MODELING

by
David Michael Lorenzetti

B.S. Electrical and Computer Engineering
University of Cincinnati, 1986

S.M. Building Technology
Massachusetts Institute of Technology, 1992

Submitted to the Department of Architecture
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Building Technology
at the Massachusetts Institute of Technology

February 1997

Signature of Author_____
Department of Architecture
10 January 1997

Certified by_____
Leslie K. Norford
Associate Professor of Building Technology
Thesis Supervisor

Accepted by_____
Leon R. Glicksman
Professor of Building Technology
Chairman, Departmental Committee on Graduate Students

Numerical Solution of Nonlinear Algebraic Systems in Building Energy Modeling

by

David Michael Lorenzetti

Submitted to the Department of Architecture
on January 10, 1997 in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in
Building Technology

ABSTRACT

When solving a system of nonlinear equations by Newton-Raphson's method, a common means of avoiding divergence requires each step to reduce some vector norm of the residual errors, usually the convenient and tractable sum of squares. Unfortunately, the descent requirement subjects the solver to difficulties typically associated with function minimization-- stagnation, and convergence to local minima. The descent requirement also can disrupt a successful Newton-Raphson sequence.

To explore these problems, the thesis reformulates the theory of function minimization in terms of the familiar Jacobian matrix, which linearizes the equations, and a vector which relates first-order changes in the norm to first-order changes in the residuals. The resulting expressions give the norm's gradient, and approximate its Hessian, as functions of the key variables defining the underlying equations. Therefore when Newton-Raphson diverges, the solver can choose a reasonable alternate search strategy directly from the Jacobian model, and subsequently construct an appropriate norm for evaluating the search.

Applying these results, the thesis modifies a standard equation-solving algorithm, the double dogleg method. Replacing the published algorithm's r-square norm with a general family of weighted r-square norms, it develops and tests a variety of rules for choosing the particular weighting factors. Selecting new weights at each iteration avoids local minima; in tests on a standard suite of nonlinear systems, the resulting algorithms prove more robust to stagnation, and often converge faster, than the double dogleg.

In separate investigations, the thesis specializes to equation-solving a double dogleg variation which minimizes the norm model in the plane of its steepest descent and Newton-Raphson directions, and develops a scalar measure of divergence which, unlike a residual norm, does not depend on results from function minimization.

Thesis Supervisor: Leslie K. Norford
Title: Associate Professor of Building Technology

DEDICATION

To A.J. Lorenzetti

who taught me what pancakes are for,
and why waffles are better.

CONTENTS

CHAPTER 1
INTRODUCTION

This thesis modifies a standard method of solving general systems of nonlinear algebraic equations. Ultimately it defines algorithms which can converge in fewer iterations, and better resist stagnation, than some standard methods. Section 1.1 motivates this work by discussing the role of algebraic equations in building energy modeling. Algebraic systems may be of interest in their own right, for instance in steady-state models. They may also arise when defining evolution problems, that is, dynamic problems characterized by time derivatives of at least some of the variables of interest. Finally they arise from the spatial discretization of the partial differential equations used for detailed modeling of building energy processes.

Section 1.2 gives an overview of the thesis, highlighting the practical difficulties associated with solving systems of nonlinear equations.

Sections 1.3 and 1.4 demonstrate some modeling principles and problems by formulating two simple algebraic systems-- a two-dimensional wall convection problem, and a three-dimensional duct flow problem, respectively. Section 1.5 defines other test problems, taken from a standard suite of nonlinear systems.

1.1 Algebraic Equations in Building Energy Modeling

Algebraic equations occur frequently in building energy modeling, from their natural application in steady-state problems to their sometimes unexpected appearance in even small dynamic simulations. Building energy systems combine many complex components, whose physical proximity, plus the presence of feedback control systems, create loops of energy and information flows, relating many of the component variables simultaneously. In part because the components' characteristic response times can vary greatly, and in part because current modeling practice relies on simplified physics and correlations, the resulting models are rich in algebraic systems.

Mathematical models

A model of a building energy system, or of a system component, represents some behavior of interest, in order to study the effects of outside influences on the component, the interactions among components, or the characteristics of the system as a whole. However the model represents this behavior-- using transfer functions, bond graphs, tabular data, algorithms, equations, etc. [Sahlin §2.1]-- it expresses a mathematical relation between the variables of interest [Clarke].

These mathematical models range in complexity from:

(1) Partial differential equations, for example the differential continuity equation

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho \cdot u)}{\partial x} + \frac{\partial(\rho \cdot v)}{\partial y} + \frac{\partial(\rho \cdot w)}{\partial z} = 0$$

[Gerhart §3.4.1], expressing mass conservation at a point; to

(2) Ordinary differential equations, e.g. the integral continuity equation for one-dimensional flow,

$$\frac{d}{dt}(\int \rho \cdot dV) = \Sigma \dot{m}_{in} - \Sigma \dot{m}_{out}$$

[Gerhart §3.4.2], expressing mass conservation in a lumped control volume; to

(3) Algebraic equations, e.g. the integral continuity equation of steady one-dimensional flow,

$$\Sigma \dot{m}_{in} = \Sigma \dot{m}_{out}$$

[Gerhart §3.4.3], expressing mass conservation in a lumped control volume at steady-state.

Each of these equations in turn simplifies the problem physics, making the model easier to use, but narrowing its intended realm of application. At another level of simplification, correlations and curve fits relate variables, typically for a specific application or situation. For example, the pressure-flow correlation

$$\Delta p = a_0 + a_1 \cdot \dot{m} + a_2 \cdot \dot{m}^2 + a_3 \cdot \dot{m}^3$$

[Walton §3.4] represents the steady-state operation of a fan. With coefficients a_i chosen to match the pressure curve on a particular fan over a limited range of flows, the model does not even represent the steady-state operation of the same fan at a different speed. While they do not express the fundamental physics of a problem-- for instance the expression above matches the derived pressure-flow relationship for a centrifugal fan in only some of its terms [Lorenzetti]-- correlations may capture the behavior of interest well enough to replace the more complete or computationally-demanding equations.

For instance, [Braun87a] develops a steady-state model for a variable-speed centrifugal chiller, using mass, momentum, and energy balances on its compressor, evaporator, condenser, and expansion valve. Then in a companion paper [Braun87b], the same investigators study the performance of a cooling system, using a five-term polynomial in place of their mechanistic chiller model, on grounds the full model requires too much computational overhead for use in a seasonal simulation.

Finally, a model may represent the relations between its variables in tabular or other algorithmic form. For example, a program may represent a component's flow relations by interpolating between discrete points taken from along its measured curve [Feustel].

Algebraic equations may enter building energy models at each level of representational complexity described above. Of course, steady-state descriptions of any physical process give algebraic relations directly. Moreover, the numeric treatment of ordinary and partial differential equations discretizes their variables in order to approximate them algebraically.

Evolution problems

The numeric solution of dynamic systems replaces the time derivatives in an evolution equation such as

$$\frac{dy}{dt} = f\{y, t\} \tag{1.1.1}$$

with a finite-difference approximation such as

$$\left.\frac{dy}{dt}\right|_{t_{[n]}} \approx \frac{y\{t_{[n+1]}\} - y\{t_{[n]}\}}{t_{[n+1]} - t_{[n]}}$$

where $t_{[n+1]} > t_{[n]}$ represents a discrete time separated from $t_{[n]}$ by some finite interval. Note that y may be a vector of state variables, and $f\{y, t\}$ a vector-valued function, so that Equation 1.1.1 defines the evolution of multiple variables over time.

Substituting the finite-difference approximation into Equation 1.1.1,

$$\frac{y\{t_{[n+1]}\} - y\{t_{[n]}\}}{t_{[n+1]} - t_{[n]}} = f\{y\{t_{[n]}\}, t_{[n]}\}$$

or, rearranging with $y_{[n]} = y\{t_{[n]}\}$ and $\Delta t_{[n]} = t_{[n+1]} - t_{[n]}$,

$$y_{[n+1]} = y_{[n]} + \Delta t_{[n]} \cdot f\{y_{[n]}, t_{[n]}\} \tag{1.1.2}$$

Equation 1.1.2 defines Euler's method for solving Equation 1.1.1 [Dahlquist §8.2]. The method generates each new point in the sequence by applying the time rate of change at the last point to the entire discrete time step.

Euler's method requires a starting value

$$y_{[0]} = y\{t_{[0]}\} \tag{1.1.3}$$

Taken together, Equations 1.1.1 and 1.1.3 define an initial-value problem [Dahlquist §8.1.1]. If $f\{y, t\}$ does not include any algebraic dependencies, then Euler's method unambiguously generates a sequence $y_{[0]}, y_{[1]}, y_{[2]}, \ldots$, and no algebraic relations result. However in practice simultaneous equations appear in evolution problems in a variety of ways: (1) through the physics (or simplified physics) of the simulated system; (2) while calculating the initial values at the start of the simulation; or (3) through the use of implicit finite-difference approximations.

In addition, the building community has a tradition of using purely steady-state models in simulation problems characterized by time-varying inputs, e.g. [Bourdouxhe, Braun87b, Sauer]. Typically these simulations involve large systems, i.e. whole buildings or primary plants [Hensen], and typically they assume the time-varying loads which affect the system change slowly compared to its response-- in other words, they assume the system maintains a dynamic steady-state. (Note this practice runs counter to the terminology typically used in numerical analysis, which identifies simulation as the computed solution of time-dependent differential equations [Dahlquist §8.1].) While these treatments do not represent evolution problems, they may define, at each time step, a system of simultaneous algebraic equations, which differs from that at the previous time step because the load has changed.

For these reasons, and because steady-state (i.e. fully algebraic) problems are of interest in their own right, the problem of solving (possibly large) systems of (possibly nonlinear) simultaneous equations represents an important aspect of building energy simulation.

The subsections below treat these topics in greater detail.

Evolution problem definition

An evolution problem may include algebraic equations directly. A completely general differential equation may be implicit in the time derivatives, defining algebraic relations between the derivatives of its variables:

$$0 = F\left\{\frac{dy}{dt}, y, t\right\} \tag{1.1.4}$$

[Sahlin §2.3]. More commonly, the governing equations include both explicit differential equations of the form of Equation 1.1.1, and implicit or explicit algebraic equations. Mathematically the system describes a vector evolution equation plus a vector algebraic equation:

(1.1.5)
$$\frac{dy}{dt} = f\{y, t\}$$
$$0 = g\{y, t\}$$

The algebraic relations in g supplement the evolution problem, describing equilibrium or quasi-equilibrium processes. For example, g might express a psychrometric property of air,

$$p - p_w = \rho \cdot R_{air} \cdot T$$

[ASHRAE §6]; it might approximate a dynamic leakage element with a steady-state relation,

$$\dot{m} = c \cdot \Delta p^n$$

[ASHRAE §32]; or it might define an implicit variable used in a separate differential or algebraic equation, such as the friction factor f in a duct,

$$\frac{1}{\sqrt{f}} = -2.0 \cdot \log_{10}\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{Re_D\sqrt{f}}\right)$$

[Gerhart §7.2.5].

Arguably, any algebraic equation representing a real process in an evolution problem comes from neglecting some dynamic in the physical system. In fact, one method of solving a system of nonlinear equations introduces dynamics to break up the dependencies between its variables, then simulates the resulting system until it reaches steady-state [Sahlin §2.3]. When the time derivatives reach zero, the ordinary differential equations reduce to the original algebraic system, and the variable values $y_{[n]}$ at the end of the simulation solve the system of equations.

However, introducing dynamics to solve equilibrium problems is not suitable for automatic computation. Component-based dynamic building energy modeling programs such as HVACSIM+ [Clark], TRNSYS [Klein], and IDA [Sahlin] allow a user to interconnect predefined component models, at run time, into systems of arbitrary structure and complexity. Clearly such a program can neither introduce dynamics unmodeled by the user, nor expect the user to introduce unwanted dynamics to the component models, merely in order to avoid potential algebraic loops in the resulting systems.

Moreover, experience indicates that modelers seek to remove dynamics, rather than to introduce them. Typically these models approximate dynamic processes whose characteristic times are supposed much faster than those of interest. The modeler replaces the fast process with some equilibrium equivalent. Thus the polynomial fan correlation might appear as part of an evolution problem describing the thermal response of a building, for which the dynamics associated with the fan are assumed negligible.

Such quasi-dynamic models, requiring the solution of algebraic systems within an evolution problem, usually result from imbedding steady-state flow problems involving air and water distribution systems [Sahlin §2.3], or the buoyancy-induced flow of air [Walton §C.4], into a thermal problem. Like the dynamic steady-state described above for simulations using steady-state models, these quasi-dynamic models supress some dynamics for tractability when formulating or solving the problem [Rodríquez].

For instance, to couple a flow problem to a thermal problem, [Axley] expresses the relations governing airflow as an algebraic system in the zone pressures, and formulates the thermal response by first-order time derivatives of the node temperatures. The

equations couple through the temperature-dependent pressures and the flow-dependent thermal excitations. Knowing the special nature of the coupling allows several plausible solution schemes; while these may or may not account for nonlinearities in the thermal system due to the temperature-dependent flow field [Axley], all treat the flow system as an instantaneously-equilibrating process, with no transient response to changes in the zone pressures. Removing the dynamics from the flow system avoids the problem of significantly different time scales in the evolution equations describing the overall system. These "stiff" problems slow a simulation, forcing it to take small time steps $\Delta t_{[n]}$, because they restrict the difference scheme by the steps required to capture the fastest process [Dahlquist §8.3.5].

Initial-value calculation

To solve the evolution problem of Equation 1.1.1 by any means, analytic or numeric, requires an initial value as in Equation 1.1.3.

With explicit time derivatives and no algebraic equations, the problem allows direct specification of the initial values $y_{[0]}$. Alternately, a modeler may wish to start the problem from steady-state, in order to study the effects of perturbing the system. This requires finding the $y_{[0]}$ such that $y'\{t_{[0]}\} = f\{y_{[0]}, t_{[0]}\} = 0$ in Equation 1.1.1. Although this defines an algebraic problem for $y_{[0]}$, in practice the solver may start with arbitrary values of y, then simulate the system until it reaches steady-state (though evaluating f at $t_{[0]}$ in each time step).

For a system with an algebraic part, on the other hand, initialization constitutes the most difficult task facing the solver [Sahlin §2.3], because of the relatively poor quality of the initial guess. At subsequent time steps, the solver begins with a relatively good guess-- the values from the previous time step.

In one exception, the system requires a new initial-value calculation at each jump discontinuity [Sahlin §2.4]. Discontinuities occur when a system includes discrete states, such as on-off control signals from a thermostat [Sahlin §4.1], or laminar and turbulent flow regimes in a duct [Gerhart §7.1.1]. These states create hysteresis-- once a component enters a state, it tends to remain in that state. State-dependent components have regions of operation wherein two or more states are possible. Thus a thermostat has a dead zone of temperatures over which it may or may not call for heat, and a duct has a range of Reynolds numbers over which flow may be laminar or turbulent. [Sahlin §4] discusses the problem of discontinuities in the defining equations of an algebraic or differential system.

Implicit difference equations

Euler's method, Equation 1.1.2, defines an explicit time-stepping scheme because the next values $y_{[n+1]}$ depend only on previous values, that is, on the estimated values of y at previous time steps. Other difference approximations, for example the general formula

$$\frac{y\{t_{[n+1]}\} - y\{t_{[n]}\}}{\Delta t_{[n]}} = (1 - \alpha)\cdot f\{y_{[n]}, t_{[n]}\} + \alpha\cdot f\{y_{[n+1]}, t_{[n+1]}\}$$

[Axley], define time steps implicitly, by making the unknown value $y_{[n+1]}$ depend on the value of $f\{y, t\}$ evaluated at $y_{[n+1]}$. Euler's equation sets $\alpha = 0$, removing this dependency.

A solver might use implicit methods: (1) to overcome stability problems with explicit schemes [Press §16.6]; (2) to speed the solution of stiff problems [Dahlquist §8.3.5, Press §16.6, §16.7]; or (3) if the evolution problem itself defines the time derivatives implicitly, as in Equation 1.1.4 [Sahlin §2.5].

Although the implicit time-differencing schemes define algebraic equations, they may not require solution by a general method for nonlinear algebraic systems (such as those examined in this thesis). Not only does the solver begin the search for the $y_{[n+1]}$ with good starting values in the $y_{[n]}$, it also may use an explicit approximation, such as Euler's method, to develop a first guess, $y_{[n+1]}'$. Then the implicit equation acts as a formula to give $y_{[n+1]}$ on the left as a function of $y_{[n+1]}'$ on the right [Dahlquist §8.3.4]. However a predictor-corrector method with a fixed iteration scheme defines an explicit method, even if the corrector step uses an implicit equation [Press §16.7]. Therefore the scheme must iterate until the values $y_{[n+1]}'$ and $y_{[n+1]}$ converge. Often a nonlinear equation-solving technique such as Newton-Raphson's method uses the implicit time-stepping rule more effectively than a predictor-corrector iteration [Press §16.7].

Spatial discretization

As with evolution problems, the numeric solution of partial differential equations uses finite-difference approximations to discretize space, expressing the spatial derivatives as algebraic relations [Dahlquist §8.6]. For specific problems the governing equations may even be expressed directly as difference equations [Incropera §4.4]. Unlike the time derivative in Equation 1.1.1, a spatial discretization makes the finite-difference approximation of the partial derivative at a point depend on the values of points nearby, and the method generates simultaneous equations [Incropera §4.5], typically in large, sparse systems [Clarke]. However these systems usually are not solved by general nonlinear equation techniques, both because their size and sparsity makes general matrix factorization impractical, and because the special nature of the defining equations makes them amenable to specialized solution methods (e.g. multi-grid methods [Press §19.6]).

Applicability of thesis work

Systems of nonlinear algebraic equations arise in many fields, and a general algebraic equation solver has no particular relation to the physical processes or problems the equations define (of course, equations describing a particular type of problem may be especially susceptible to solution by a specialized technique). The numeric methods presented throughout the thesis, intended as general nonlinear equation solvers, are not optimized for any particular building energy problem. Nevertheless, the building energy simulation community needs robust and reliable general-purpose solvers.

Some simulation programs, by treating a particular equational form, such as $\dot{m} = \dot{m}\{\Delta p\}$ in AIRNET [Walton §5.2], allow the specialization of the solver to speed convergence by a variety of means, e.g. [Walton §2.2-2.6, §5.1]. However the trend towards modular, general-purpose simulation environments, intended for use by nonexpert practitioners, and which allow user-defined component models, increases the role of general automatic computation. Often starting from a relatively poor initial trial solution, a general-purpose solver may encounter piecewise-continuous component behavior (for instance the laminar-to-turbulent transition in duct flow) and mode dependencies (for instance in a thermostat), as well as the already-difficult nonlinearities governing most flow processes. (In fact, this thesis work grew out of an attempt to simulate, by computer, a feedback controller in a ducted ventilation system. The controller and flow component models created an algebraic loop in the defining equations, which prevented an initial-value calculation by the dynamic solvers available.)

1.2 Thesis Overview

This section previews the thesis, its discussion of the difficulties associated with solving systems of nonlinear equations, its criticisms of the standard descent-based solution methods, and its proposed responses.

Newton-Raphson's method

Most techniques for solving a general system of nonlinear equations begin by linearizing the equations at the current best estimate of the solution, then solving that linear system to produce a new estimated solution. Chapter 2 discusses Newton-Raphson's method in detail. Probably this method is the most direct model-based equation-solving strategy. It forms the simplest possible models of the equations-- linear models, from a first-order Taylor series expansion of the residual form of the equations-- and it uses these models in the simplest possible way-- treating them as a single linear system.

When Newton-Raphson's method fails, then, a solver can either improve the models, or change the way it uses them. Higher-order models increase both the computational costs and the storage requirements, so all variations of Newton-Raphson's method at least seek to improve on its use of the information contained in the model.

Newton-Raphson's method, with no modifications, fails almost exclusively by diverging. Unless it happens to find a singular system when it linearizes the residual equations, it defines a step, and unless that step carries the method to a point where the linear equations are not defined (say by taking the square root of a negative number), it can always take another step. In numeric tests, Newton-Raphson's method tends either to solve a problem directly with a very few number of iterations, or, if left unchecked, to step around the variable space until it happens to land close to the root (after which it converges very quickly). In a very few exceptions, the method diverges unrecoverably.

Descent-based equation solvers

A standard means of preventing divergence requires that each step reduce a residual norm. If a trial step increases the norm, it clearly exceeds the predictive capability of the linearized models (since the Newton-Raphson step zeros each residual model). In response, the algorithm cuts the step length. Requiring a step to decrease a residual norm, rather than, say, that it reduce the magnitude of every residual, allows some tradeoff between the errors the step induces in the linear residual models.

To enforce a descent requirement on a residual norm, an equation-solving algorithm must adopt results from function minimization. Chapter 3 discusses minimization methods, algorithms designed to seek out local minima in a scalar cost function. The methods of interest choose trial points in two fundamental search directions: (1) the steepest descent direction, which follows the gradient of the cost function; and (2) the Newton direction, which steps towards the minimizer of a quadratic model of the cost function. A hybrid method such as the double dogleg algorithm combines the search directions, according to the anticipated length of a successful step.

Since the gradient of a cost function goes to zero at a local minimum, the Newton step zeros the linear model of the gradient. Its strong parallels with the Newton-Raphson step, which zeros a linear model of the vector of residual errors, simplifies the task of adapting a function minimization algorithm to equation solving, but makes the solver resemble a minimization routine with only computational ties to the underlying equational system.

These descent-based algorithms particularize a minimization technique to equation solving by: (1) choosing a residual norm as the cost function; (2) modeling the norm using the linearized residuals; (3) calculating the minimizer of the model by Newton-Raphson's, rather than Newton's, method; and (4) adding a termination test for a zero residual vector. For instance, the double dogleg method of equation solving minimizes the residual norm r-square, using a search path defined by the steepest descent direction on r-square for short steps, and the Newton-Raphson direction for long steps.

Unfortunately, adopting a descent requirement subjects the solver to difficulties typically associated with function minimization-- stagnation, and convergence to local minima in the residual norm (which do not correspond to roots of the nonlinear system). In function minimization, generally the global minimum remains unknown, and the methods converge to any local minimum in the cost function. Too, numeric effects in the computed sequence can slow an algorithm, stagnating it in flat regions of the cost function. Formulating an equation solver as a function minimization routine introduces the same effects to the solution of nonlinear algebraic systems. To compound the problem, the linearized system becomes singular at such local minima, making the Newton-Raphson sequence unstable in its vicinity. Thus the Newton-Raphson steps will diverge, forcing the algorithm to fall back on its minimization-based search techniques, guaranteeing convergence to the local minimum.

Numeric tests show that in addition to causing stagnation and convergence to unwanted minima, the descent requirement can disrupt a potentially-successful Newton-Raphson sequence, causing the solution to evaluate the defining equations and their derivatives more often than required. While this does not constitute a failure per se, it may compromise the method in a practical sense.

Reformulating descent-based solvers

Stagnation, convergence to false roots, and slowed convergence to real roots all can result from a too-literal application of the function minimization results to the problem of solving systems of nonlinear equations. Yet the bodies of theory exist independently: Chapters 2 and 3 make little reference to one another.

Chapter 4 explores the relation between the subjects by reformulating the function minimization results for a general residual norm. Minimization theory requires the derivatives of the norm with respect to the independent variables; since the Jacobian matrix already gives the derivatives of the residual equations, Chapter 4 defines a new vector, p, which gives the derivatives of the norm with respect to the residuals. This vector, and a matrix P of its derivatives, describe how the norm combines the residuals algebraically. Combined with the Jacobian, they can express function minimization theory for a residual norm directly in terms of the linear residual models already used in equation solving.

The resulting expressions relate the steepest descent direction of any norm to the Jacobian model, showing that any norm can have local minima where the linearized residuals define a singular system. They also show how modeling a norm using the linearized residuals captures the constant part of the norm's curvature, but misses the part of its second-order behavior which affects the ability of the model to predict likely search steps.

Chapter 4 also demonstrates that, modeling a norm using the linearized residuals, a Newton step on the norm equals the Newton-Raphson step only for a class of weighted r-square norms. Deriving expressions to implement a double dogleg algorithm using this

family of norms, it gives weighting selections which match the resulting steepest descent direction to that of the one-norm.

Algorithms

The thesis defines equation-solving algorithms beginning with Chapter 5.

Chapter 5 implements the standard double dogleg method, as defined in the literature (although it uses the results of Chapter 4 to make slight improvements in the computational efficiency of the actual code). Chapter 6 demonstrates the algorithm, along with some other important aspects of the preceding chapters, using an extended example, on a two-dimensional trigonometric problem.

Chapters 7 and 8 define new algorithms, based on the mathematical expressions derived in Chapter 4, and on criticisms of the standard methods of equation solving.

These criticisms center on re-establishing the conceptual distinction between equation solving and function minimization. Again, the strong correspondence between Newton's method and Newton-Raphson's method can obscure the different problems each represents, so the discussion focuses on identifying how assumptions from function minimization affect an equation solver.

For example, controlling divergence by minimizing a residual norm introduces the double dogleg search path from function minimization. But for compuational efficiency in minimization problems, the double dogleg path compromises the theoretically optimal search path. The computational problem changes for equation solving, so Chapter 7 defines a new search path, the planar hook, which chooses trial points more effectively by using the structure imposed by the underlying algebraic problem. Imbedded in the same descent-based algorithm, the planar hook consistently outperforms the double dogleg path, requiring fewer evaluations of the residuals and their derivatives to achieve solutions, and sometimes converging where the double dogleg method stagnates.

Similarly, minimizing a residual norm introduces convergence to unwanted minima, and hence stagnation problems. Chapter 8 rejects tying the algebraic problem to a single norm, raising the possibility of changing norms when the solution in one norm appears close to stagnation. Since Chapter 4 shows that r-square belongs to a broader class of norms, possessing the same convenient properties for use in a minimization-based algorithm, Chapter 8 adapts the double dogleg method of function minimization to use this family of weighted r-square norms. Furthermore the results of Chapter 4, combined with geometric interpretations of the Newton-Raphson and steepest descent methods, suggest rules for choosing a specific family member at each iteration. The resulting weighted double dogleg algorithm proves much more robust to stagnation than the double dogleg, and often converges faster when the standard method can find a solution.

Other work

To avoid problems with descent-based algorithms altogether, Chapter 9 begins to investigate alternate divergence measures. Returning to the original notion that in a diverging Newton-Raphson iteration the step length exceeds the predictive capability of at least one residual model, Chapter 9 develops both individual and lumped measures of divergence (applying to single equations, or to the system as a whole). These measures account not only for the length error in the residual models (as does the descent requirement), but for the angle error as well. Though Chapter 9 defines no algorithms, the measures show promise by interpreting differences between the double dogleg, planar hook, and weighted double dogleg step selections.

Chapter 10 concludes by summarizing the thesis in greater technical detail than presented here. In addition, it suggests further work for extending and refining the methods and ideas developed in the thesis.

1.3 Sample Problem: Convection on a Wall

This two-dimensional algebraic system models steady-state heat flow through a wall, with the governing equations simplified to avoid: (1) nonlinearities associated with the temperature variation of the thermal properties of air; and (2) the possible transition between laminar and turbulent flow regimes.

Problem description

The wall, shown in Figure 1.3.1, has an R-value of 10 hr·ft^2·R/Btu. Fourier's law gives the heat flux through the wall at steady state as

$$q_{wall} = \frac{k_{wall}}{W_{wall}} (T_{win} - T_{wout})$$

[Mills §1.3.1]. The total heat flow is $q_{wall} \cdot A$ where $A = L \cdot H = 12.5$ m^2. The thermal conductivity and width of the wall are not given but the thermal resistance gives their ratio

$$\frac{k_{wall}}{W_{wall}} = \frac{1}{10} \frac{Btu}{hr \cdot ft^2 \cdot R} = 0.5678 \frac{W}{m^2 K}$$

so that the heat flux

$$q_{wall} = 0.5678(T_{win} - T_{wout}) \quad \frac{W}{m^2} \tag{1.3.1}$$

from the inner to the outer surface of the wall.

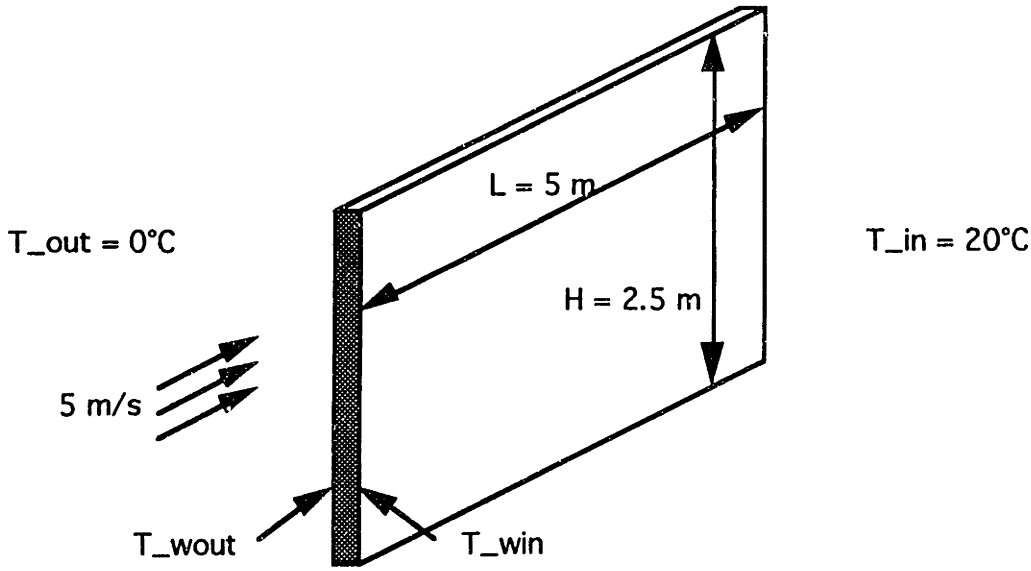

Figure 1.3.1. Exterior wall of a building. Heat transfer is by forced convection on the exterior surface, and by natural convection on the interior surface.

Both wall surfaces interact with the adjacent air by convective heat transfer. The relation

(1.3.2) $$q = h_c \cdot \Delta T$$

defines the convective heat transfer coefficient, h_c [Mills §1.3.3], with SI units W/m^2K. At the outer wall surface, forced convection, due to a 5 m/s wind, produces a higher convective heat transfer coefficient than at the inner surface, where only natural convection, due to local cooling of the air adjacent to the wall, occurs.

Forced convection on outer surface

For forced convection, a number of correlations give h_c in terms of the fluid's thermal properties and velocity. These are nondimensionalized by the Nusselt number

(1.3.3a) $$Nu_L = \frac{h_c L}{k}$$

[Mills §4.2.2], and by the Reynolds number

(1.3.3b) $$Re_L = \frac{\rho V L}{\mu} = \frac{VL}{\nu}$$

respectively. The Reynolds number is the ratio of inertia force to viscous force on the fluid [Gerhart §7.2.5]. In both Nusselt and Reynolds numbers, L is a characteristic dimension--here, the extent in the flow direction.

The fluid properties introduced in the Reynolds number are: density ρ [=] kg/m^3, viscosity μ [=] kg/m·s, and kinematic viscosity $\nu = \mu/\rho$ [=] m^2/s. The remaining fluid property of interest, the specific heat at constant pressure, c_p [=] J/kg·K, is nondimensionalized by the Prandtl number

(1.3.3c) $$Pr = \frac{c_p \mu}{k}$$

[Mills §4.2.2]. Each of Equations 1.3.3 has dimensions 1.

The Prandtl number for air is constant at 0.69 over a broad range of temperatures encountered in buildings. However, as shown by Figure 1.3.2, other thermal properties of air vary significantly even over the range 0°C - 20°C of the sample problem. The temperature dependence of fluid properties can constitute an important nonlinearity in building energy modeling.

T, °C	T, K	k, W/m·K	ρ, kg/m^3	c_p, J/kg·K	μ, kg/m·s	ν, m^2/s	Pr
-13.15	260	0.0242	1.360	1009	16.63E-6	12.23E-6	0.69
-3.15	270	0.0249	1.311	1009	17.12E-6	13.06E-6	0.69
6.85	280	0.0255	1.265	1008	17.60E-6	13.91E-6	0.69
16.85	290	0.0261	1.220	1007	18.02E-6	14.77E-6	0.69
26.85	300	0.0267	1.177	1005	18.43E-6	15.66E-6	0.69
36.85	310	0.0274	1.141	1005	18.87E-6	16.54E-6	0.69

Figure 1.3.2. Selected thermal properties of air [Mills §A.7].

While temperature dependence is important for the sample problem (changes of density with temperature cause the buoyancy forces which drive the natural convection on the interior surface of the wall), it is not modeled explicitly. Instead, fluid properties on each side of the wall are evaluated at representative temperatures. This keeps the equations

tractable. Note the use of tables complicates the analysis, since: (1) the tabular data require interpolation; and (2) partial derivatives of the residual relations, required for the Newton-Raphson solution method of Chapter 2, include partial derivatives of the tabulated relationships as well.

Correlations for forced convection over a flat plate give

$$\text{Nu}_L = \frac{h_c L}{k} = 0.664 \cdot \text{Re}_L^{1/2} \text{Pr}^{1/3} \tag{1.3.4}$$

for laminar flow with $\text{Pr} > 0.5$ [Mills §4.3.2]. Transition to turbulence occurs at $\text{Re}_{crit} \approx 5 \cdot 10^5$. For turbulent forced flow over a flat plate, the relation

$$\text{Nu}_L = 0.664 \cdot \text{Re}_{crit}^{1/2} \text{Pr}^{1/3} + 0.036(\text{Re}_L^{4/5} - \text{Re}_{crit}^{4/5}) \text{Pr}^{0.43} \tag{1.3.5}$$

[Mills §4.3.2] accounts for the laminar starting length. From Equation 1.3.3b, with V = 5 m/s and L = 5 m, the flow is turbulent, even using the largest possible kinematic viscosity from Figure 1.3.2. Therefore only Equation 1.3.5 is needed in the residual relations.

Both Equations 1.3.4 and 1.3.5 evaluate fluid properties at the mean film temperature-- that is, at the average temperature between the outer wall surface and the air in the free stream blowing over it [Mills §4.3.2]. To avoid this nonlinearity, assume the wall temperature is close to the free stream temperature, and evaluate the fluid properties at 275K (≈2°C). Interpolating in Figure 1.3.2, let $k = 0.0252$ W/m·K, $\nu = 13.485 \cdot 10^{-6}$ m²/s, and $\text{Pr} = 0.69$. Substituting in Equation 1.3.5, $h_c = 12.477807$ W/m²K for the turbulent forced convection on the outer wall surface. From the definition of the convective heat transfer coefficient, with $T_{out} = 0$°C, the heat flux

$$q_{out} = 12.48 \cdot T_{wout} \quad \frac{W}{m^2} \tag{1.3.6}$$

from the outer wall surface to the free air stream.

Natural convection on inner surface

For natural convection, a fourth dimensionless group, the Rayleigh number

$$\text{Ra} = \frac{\beta \cdot \Delta T \cdot g \cdot L^3}{\nu^2} \text{Pr} \tag{1.3.7}$$

[Mills §4.2.2] accounts for buoyancy. The volumetric coefficient of thermal expansion, ß [=] 1/K, gives a buoyancy force per unit mass of $\beta \cdot \Delta T \cdot g$ due to density variations in the fluid. For an ideal gas, $\beta = 1/T$ where T is the absolute temperature [Mills §1.3.3].

For laminar natural convection on a vertical wall,

$$\text{Nu}_L = 0.68 + 0.670(\text{Ra}_L \cdot \Psi)^{1/4} \tag{1.3.8}$$

where

$$\Psi = (1 + (0.492/\text{Pr})^{9/16})^{-16/9} \tag{1.3.9}$$

[Mills §4.4.1]. For air, $\Psi = 0.3426$. Since there is no free stream velocity, the Rayleigh number, not the Reynolds number, identifies transition to turbulence. It occurs at $\text{Ra}_{crit} \approx 1 \cdot 10^9$. For turbulent natural convection,

$$\text{Nu}_L = 0.68 + 0.670(\text{Ra}_L \cdot \Psi)^{1/4}(1 + 1.6 \cdot 10^{-8} \text{Ra}_L \cdot \Psi)^{1/12} \tag{1.3.10}$$

[Mills §4.4.1]. Equation 1.3.10 applies for $10^9 < \text{Ra}_L < 10^{12}$.

Note that Equations 1.3.9 and 1.3.10 do not match exactly at $Ra_L = Ra_{crit}$. In fact, for air the turbulent expression calculates a Nusselt number almost 20% larger than does the laminar expression. Such a discontinuity may not imply a modeling error, if in reality the transition from laminar to turbulent flow, with the attendant change in the physical mode of heat transfer, is sudden. However, the discontinuity may cause a numeric solver difficulty, even if the true solution does not lie near Ra_{crit}, since successive iterations may pass from one regime to another.

If the solution is known to lie in a given regime, as in the case of forced convection on the outside surface of the wall, the appropriate equation can be used without regard for discontinuities between adjacent regimes. Unfortunately, for $T_{win} = 19.5°C$ the Rayleigh number is about Ra_{crit}, so Ra_L is likely to be at or slightly above the critical value.

Alternate correlations are available. For turbulent flow along an isothermal vertical plate,

(1.3.11) $$Nu_L = 0.10 \cdot Ra_L^{1/3}$$

[Incropera §9.6] is within 7% of Equation 1.3.10 over the turbulent regime, with the greatest mismatch at the critical Rayleigh number, where it gives a Nusselt number between the laminar and the turbulent estimates. The expression matches the laminar relationship at $Ra_L \approx 3.7 \cdot 10^8$, and for smaller Rayleigh numbers, it consistently underestimates Equation 1.3.9. A natural solution to the discontinuity, then, is to replace Equation 1.3.10 with Equation 1.3.11 for the turbulent regime, and for the laminar regime to use the larger of the two Nu_L calculated from the laminar and the (new) turbulent expressions.

For tractability in the sample problem, however, a single equation is desired. Since the flow regime of the final solution is slightly higher than Ra_{crit}, and since Equation 1.3.11 gives reasonable h_c at Ra_L down to about a decade below Ra_{crit}, use Equation 1.3.11 over the entire range for this sample problem.

As with forced convection, all fluid properties should be evaluated at the mean of T_{win} and T_{in} [Mills §4.4.1], and again to simplify the problem a representative temperature is chosen. For natural convection h_c is not expected to be as high as for forced convection, so the inner wall temperature will deviate from the air temperature by a greater amount. Evaluating all properties at 290K (≈17°C), the Rayleigh number is $1.67178 \cdot 10^9 \cdot |20 - T_{win}|$, where the absolute values are used to allow $T_{win} > 20$ during the search for a solution (the Nusselt number correlation also applies to a heated wall). Then Equations 1.3.11 and 1.3.2 give a heat flux

(1.3.12) $$q_{in} = 1.239 \cdot |20 - T_{win}|^{1/3}(20 - T_{win}) \quad \frac{W}{m^2}$$

from the room air to the inner wall surface.

Energy balance at steady-state

At steady-state, the heat flux calculated at each stage through the wall must be equal. This follows from a surface energy balance on each side of the wall-- at the inner surface, for example, the heat flux q_{in} from the room air must equal the heat flux q_{wall} to the outer surface. Each of these fluxes has been defined positive from right to left, so the energy balances demand $q_{in} = q_{wall}$ and $q_{out} = q_{wall}$. Arranged as residual equations, the energy balance seeks $r_1 = q_{out} - q_{wall} = 0$ and $r_2 = q_{in} - q_{wall} = 0$. Substituting from the heat flux relations,

(1.3.13a) $$r_1 = 13.05 \cdot T_{wout} - 0.5678 \cdot T_{win}$$

(1.3.13b) $$r_2 = 0.5678 \cdot T_{wout} - 0.5678 \cdot T_{win} + (20 - T_{win}) h_{cin}$$

where

(1.3.13c) $$h_{cin} = 1.239 \cdot |20 - T_{win}|^{1/3}$$

1.4 Sample Problem: Flow in a Duct

This three-dimensional algebraic problem models the steady-state flow of air through a duct. The derivation illustrates not only the simplification of the defining relationships, but also modeling strategy considerations for a general component-based building energy simulation tool.

Problem description

Air flows through a circular duct at steady state, with the governing equations simplified by assuming: (1) turbulent flow; (2) fully-developed flow; and (3) air is incompressible, with fixed properties $\rho = 1.2$ kg/m^3 and $\mu = 1.8 \cdot 10^{-5}$ kg/m·s.

With no work or heat interactions across the duct walls, and constant density, an energy balance between cross sections 1 and 2 along the duct gives

(1.4.1) $$\frac{p_1}{\rho} + \tilde{u}_1 + \alpha_1 \frac{V_1^2}{2} + g z_1 = p_2 + \tilde{u}_2 + \alpha_2 \frac{V_2^2}{2} + g z_2$$

[Gerhart §5.4.3] where: (1) $\tilde{u}$ is the specific internal energy (the internal energy per unit mass) of the air; (2) V is the average velocity of air at the cross-section of interest; and (3) the kinetic energy correction factor, α, accounts for the velocity profile, by relating the average kinetic energy at that cross-section to the kinetic energy of a particle with average velocity [Gerhart §5.4.1]. For laminar flow, where the velocity profile is parabolic, $\alpha = 2$. Turbulent flow, with its flatter velocity profiles, has smaller α since the average velocity better represents the overall flow. For fully turbulent flow, $\alpha \approx 1.05$.

Mechanical energy dissipation

Friction forces dissipate mechanical energy down the length of the duct, converting it to internal energy. To convert this internal energy back to mechanical energy requires compressibility [Gerhart §5.4.2], so for constant density air, the added internal energy appears entirely as an increase in temperature. Thus, the converted mechanical energy can not be recovered.

Engineering practice calls this "lost" mechanical energy and reformulates Equation 1.4.1 as a mechanical energy balance. Defining gh_L as the mechanical energy converted per unit mass to thermal energy by friction (the "head loss," h_L, gives the converted mechanical energy as an equivalent increase in elevation),

(1.4.2) $$p_1 + \rho \cdot \alpha_1 \frac{V_1^2}{2} + \rho \cdot g z_1 = p_2 + \rho \cdot \alpha_2 \frac{V_2^2}{2} + \rho \cdot g z_2 + \rho \cdot g h_L$$

Unlike Equation 1.4.1, the mechanical energy balance assumes station 2 is downstream of station 1, since mechanical energy is lost in the flow direction. To reverse the flow, add the $\rho \cdot g h_L$ term (which is always positive) to the energy terms of station 1.

For fully-developed flow the velocity profile does not change down the length of the duct, so $\alpha_1 = \alpha_2$. Then since the duct cross-sectional area and air density do not change, mass conservation requires the same average velocity at each location down the duct. The mechanical energy balance becomes

(1.4.3) $$p_2 - p_1 = \rho \cdot g(z_1 - z_2) - \rho \cdot g h_L$$

Equation 1.4.3 accounts for the pressure drop down the duct by the increase in its height and by the conversion of mechanical to thermal energy due to friction.

To nondimensionalize the friction-induced pressure gradient $\Delta p_L/L$, use the Darcy friction factor

(1.4.4a,b) $$f = \frac{\left(\frac{\Delta p_L}{L}\right) D}{\frac{\rho V^2}{2}} \quad \text{or} \quad g h_L = f\left(\frac{L}{D}\right)\frac{V^2}{2}$$

[Mills §4.2.2]. For laminar flow a momentum balance gives

(1.4.5) $$f = \frac{64}{Re_D}$$

[Gerhart §7.2.4] where the Reynolds number is based on the duct diameter:

(1.4.6) $$Re_D = \frac{\rho V D}{\mu} = \frac{V D}{\nu}$$

The upper limit for laminar flow in a pipe is $Re_D \approx 2300$, although below $Re_D \approx 4000$ the flow may pulse between laminar and turbulent flow [Gerhart §7.1.1].

For turbulent flow, no analytic expression gives the shear force at the duct wall, and correlations to measured data replace the momentum balance when finding the friction factor. The Colebrook formula,

(1.4.7) $$\frac{1}{\sqrt{f}} = -2.0 \cdot \log_{10}\left(\frac{\varepsilon}{3.7D} + \frac{2.51}{Re_D\sqrt{f}}\right)$$

relates the friction factor to the Reynolds number and to the duct diameter and surface roughness ε [Gerhart §7.2.5].

Equation 1.4.7 is valid above $Re_D \approx 4000$ and can be extended to noncircular ducts by using either the hydraulic diameter, or the more accurate effective diameter [Gerhart §7.2.7]. Tables give ε for various materials [Gerhart §7.2.5, Mills §4.7.1]. Note that the relative roughness ε/D can increase by an order of magnitude over time [Gerhart §7.2.5].

Transition from laminar to turbulent flow

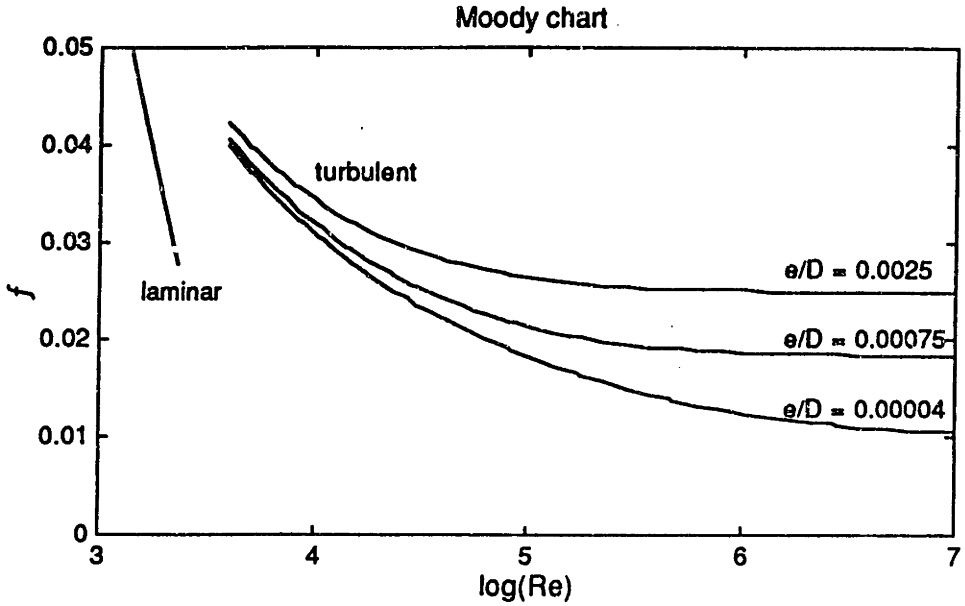

Figure 1.4.1. The three values of relative roughness on this Moody chart correspond to absolute roughness ε = 0.05, 0.9, and 3 mm in a circular duct of diameter 1.2 m.

As with the wall convection problem the equations for the laminar and turbulent regimes do not match. Indeed the range $2300 \le Re_D \le 4000$ does not have a useful defining equation. A Moody chart, Figure 1.4.1, plots friction factor against Reynolds number and shows the mismatch in the critical region of intermediate Re_D.

For simplicity, the sample problem uses only the turbulent relation. A general purpose duct component model, on the other hand, should model the laminar regime since turbulent flow is not guaranteed. The resulting model may have a jump discontinuity which could cause difficulty in solving both algebraic and evolution problems. However, including the (linear) laminar regime in flow component models may serve a numeric as well as a modeling purpose: during initialization, for bidirectional flow elements, the linear relation can help the solver establish flow directions [Walton §2.6].

Problem specification

With gh_L known, Equations 1.4.3 and 1.4.7 give the pressure drop through the duct. Finally the mass flow rate

$$\dot{m} = \rho \frac{\pi D^2}{4} V \tag{1.4.8}$$

A typical problem might: (1) seek the mass flow and pressure drop for a duct with known physical parameters; or (2) seek the duct configuration needed to produce a desired flow and pressure drop. The sample problem mixes physical characteristics and flow values in its unknowns: the duct diameter D, average flow velocity V, and friction factor f are variables. This selection, while atypical, keeps every term of the Colebrook formula active.

Let $\varepsilon = 0.05$ mm (for steel) and D = 1.2 m. To emphasize the nonlinearity of the problem, the flow is made turbulent, but not fully turbulent, so that there is significant change in f with Re_D. From the Moody chart a friction factor 0.025 is appropriate. Then

from the Colebrook formula $Re_D = 23450.2$, and from Equation 1.4.6 with $\rho = 1.2$ kg/m^3 and $\mu = 1.8 \cdot 10^{-5}$ kg/m·s, V = 0.2931275 m/s. Substituting in Equations 1.4.7, 1.4.4b, and 1.4.8 gives, after some rearrangement,

(1.4.9a) $$r_1 = \frac{1}{\sqrt{f}} + 2 \cdot \log_{10}\left(\frac{1}{D}\left[1 + \frac{2.7861}{V\sqrt{f}}\right]\right) - 9.7384634$$

(1.4.9b) $$r_2 = \frac{fV^2}{D} - 0.00179008$$

(1.4.9c) $$r_3 = VD^2 - 0.422104$$

Local versus global solution of local variables

Consider the important modeling decision of how to handle an implicit equation in a general component model.

A typical duct component models mass flow and pressure drop, using the duct size, relative roughness, and so on as parameters. The friction factor depends on flow according to Equation 1.4.5 for the laminar regime, or Equation 1.4.7 for turbulent flow.

If the component model had to deal only with laminar flow, then the relation $f = 64/Re_D$ would appear, if at all, as a helper variable within the model definition. The friction factor is significant within the model-- it is required to find the duct's residuals-- but does not affect interactions between the duct and other components such as fans, mixing boxes, vents, and so forth.

Unfortunately the Colebrook formula, implicit in f, prevents calculating f using a simple assignment. Therefore the component model code must solve for the friction factor by some special technique, e.g. by Newton-Raphson iteration. The modeler may: (1) treat f as a global variable, satisfying the Colebrook formula in the course of the overall solution; or (2) program a solution for f as part of the duct component calculations, satisfying the Colebrook formula at every iteration of the overall solution.

The first choice makes f one of the variables manipulated by the global solver. The Colebrook formula itself provides a residual relation. This is the choice made in the sample problem. In the second case, f remains a helper variable, known only to the duct component, which must solve the Colebrook formula itself.

The defining equations for variables which are wholly internal to a component may always be solved locally or globally. Usually this is a trivial choice, e.g. f in the laminar case. The choice no longer is trivial when: (1) a local variable is defined implicitly; or (2) several local variables are defined by a set of simultaneous equations.

A locally programmed solution reduces the computational load on the global solver. Furthermore the solution technique can be specialized to the equations at hand, possibly resulting in more efficient solution. For example, AIRNET treats the Colebrook formula by Newton-Raphson iteration on $1/\sqrt{f}$, converging in two or three iterations if no previous estimate is available, and in one iteration if the value from a previous nearby solution is available [Walton §3.2].

From the point of view of the global solver, calculating f within the duct component model not only reduces the problem dimension by one, it also changes the residual values returned by the component. This is because the friction factor used to calculate gh_L is consistent with the Colebrook formula at each iteration, whereas if the global solver has charge of f, the Colebrook formula may not be satisfied until the final iteration.

Disadvantages to calculating f locally include: (1) the additional data structures and code required to handle the local variable, particularly if its value from one iteration of the global solver is carried over as an initial guess for the next iteration (for example, the SAVED variables of HVACSIM+ [Clark]; (2) diminished "readability" of the component model formulation; (3) increased difficulty in translating the model definition to new modeling or computational environments; and (4) increased difficulty in finding partial derivatives for the remaining residuals, if these are required by the global solver.

The first two objections, that local solution obscures the model definition, are largely stylistic.

The third objection, that local solution prevents porting the model easily, gives rise to the argument that model specification should not include simulation strategy [Sahlin §2.1]. This arguments asserts that representing a component using a combination of defining equations and algorithmic constructs prevents sharing the model between researchers with different simulation tools or solution strategies [Hensen].

The third, that the model is not easily ported, is the origin of the injunction against mixing simulation strategies into the model specification [Sahlin §2.1]. Even though local solution does not preclude input-output free modeling, it prevents equational expression of the component physics by making the solution strategy part of the component model.

The fourth objection to local solution, that it makes the partial derivatives harder to evaluate, can affect computational efficiency.

Jacobian matrix for global and local solutions

Most of the standard techniques for solving nonlinear simultaneous equations collect the derivatives of the residual equations with respect to the global variables into a Jacobian matrix with $[J]_{i,j} = \partial r_i/\partial x_j$; see Chapter 2. Since solving an implicit equation locally changes the values of the remaining residuals, the corresponding partial derivatives change as well.

Often these derivatives are calculated using analytic expressions given in the component model formulation (although this, too, mixes modeling with solution strategies, and has caused debate in the building simulation community). As formulated in Equations 1.4.9, the duct flow problem has

(1.4.10a)
$$J = \begin{bmatrix} \frac{1}{2f}\left(c_1 - \frac{1}{\sqrt{f}}\right) & \frac{c_1}{V} & \frac{-2}{\ln 10 \cdot D} \\ \frac{V^2}{D} & \frac{2fV}{D} & \frac{-fV^2}{D^2} \\ 0 & D^2 & 2VD \end{bmatrix}$$

where, for example, the first row gives the derivatives of r_1 with respect to f, V, and D respectively, and where

(1.4.10b)
$$c_1 = \frac{-2}{\ln 10}\left(\frac{2.7861}{2.7861 + V\sqrt{f}}\right)$$

The zero entry in the third row reflects the fact that $r_3 = r_3\{V, D\}$ does not depend on f.

Now suppose the component model itself calculates f to satisfy the Colebrook formula at each iteration of the global solver. Then $r_1 = 0$ at every iteration, and r_1 drops out of the global problem along with f. The third residual does not depend on f, so in the

new formulation, r_3, $\partial r_3/\partial V$, and $\partial r_3/\partial D$ are the same as above. However with $f = f\{V, D\}$ calculated locally, Equation 1.4.9b no longer gives r_2.

Call the new residual r_2'. This new residual expresses the same physical relation as r_2 (mechanical energy loss), but with f implicitly defined, r_2' cannot be represented equationally. Furthermore the appropriate Jacobian entries for the second residual become

$$\frac{\partial r_2'}{\partial V} = \frac{\partial r_2}{\partial V} + \frac{\partial r_2}{\partial f}\frac{\partial f}{\partial V} \quad \text{and} \quad \frac{\partial r_2'}{\partial D} = \frac{\partial r_2}{\partial D} + \frac{\partial r_2}{\partial f}\frac{\partial f}{\partial D}$$

because, for example, changing V in r_2' has the same effect as changing V in r_2, plus the effect that changing V induces, through the Colebrook formula, on f.

With f defined implicitly there is no guarantee that a closed expression exists for terms such as $\partial f/\partial V$. If not, the derivative must be estimated when needed, e.g. by finite-difference approximations. For the problem at hand, closed expressions exist, giving

$$J = \begin{bmatrix} \frac{2fV}{D}\left(\frac{1}{1-c_1\sqrt{f}}\right) & \frac{-fV^2}{D^2}\left(1+\frac{4\sqrt{f}}{\ln 10(1-c_1\sqrt{f})}\right) \\ D^2 & 2VD \end{bmatrix} \tag{1.4.11}$$

Deriving Equation 1.4.11 requires considerably more effort than does its three-variable counterpart, Equation 1.4.10. (Furthermore the new component model adds an algorithm to the three equations already used.) The more global variables required to define a helper variable, the greater the effort required to develop its partials analytically. Even if the required derivatives can be found explicitly, as in Equation 1.4.11, the chance of an algebraic or coding error increases, and the modeler is more likely to estimate derivatives by finite differences. (Worse is to use the inapplicable analytic results from Equation 1.4.10. Evaluated at the solution to the equilibrium problem, the partial derivative $\partial r_2'/\partial V$ is off by 13% if evaluated using Equation 1.4.10 instead of Equation 1.4.11, and the partial $\partial r_2'/\partial D$ errs by 19%.)

Finite differences approximate partial derivatives using a numeric form of the defining limit about a point, for example by the forward difference

$$\frac{\partial r_i}{\partial x_j} \approx \frac{r_i\{x_j + \Delta x_j\} - r_i\{x_j\}}{\Delta x_j}$$

where Δx_j is small [Dennis §4.2]. For every global variable x_j of interest, the approximation evaluates r_i at a nearby point in that coordinate direction. These extra evaluations decrease the computational efficiency relative to analytic derivatives (even if the component does not solve any helper variables locally). Local solution procedures exacerbate the problem, by adding to the time spent in these component evaluations.

In fact, local solution procedures may add disproportionately to the computational overhead of a finite-difference evaluation. For the problem at hand, the estimates of $\partial r_2'/\partial V$ and $\partial r_2'/\partial D$ are extremely sensitive to the accuracy of the solution for f. A "noisy" function evaluation of f can seriously affect the finite-difference approximation [Moré82 §1.1]. Consequently, estimating these partial derivatives places a tighter tolerance on the local solution of f than that required merely to zero the global residuals.

1.5 Standard Algebraic Systems

In addition to the wall convection and duct flow problems discussed above, this thesis refers to a number of nonlinear algebraic test problems from the literature. In particular, [Moré81] assembled a standard test suite to: (1) allow users to compare different solvers on a range of problem types; (2) force software testers to consider a variety of starting conditions; and (3) encourage software writers to emphasize reliability and robustness as well as efficiency in their routines. Much of the test bed is intended for optimization software only, but 14 of the 35 problems also define systems of nonlinear equations.

For convenience, this section collects together all the test problems used in the thesis.

Broyden tridiagonal function

The equilibrium problem is

$$r_i\{x\} = (3-2x_i)\cdot x_i - x_{i-1} - 2x_{i+1} + 1 \tag{1.5.1a}$$

where

$$x_0 = x_{n+1} = 0 \tag{1.5.1b}$$

with n variable [Moré81 §3.30]. The standard starting point is $x_{[0]} = (-1, -1, .. , -1)^T$.

The partial derivatives are

$$\frac{\partial r_i}{\partial x_j} = \begin{cases} j = i: & 3 - 4x_i \\ j = i-1: & -1 \\ j = i+1: & -2 \\ \text{else:} & 0 \end{cases} \tag{1.5.1c}$$

The solution depends on the variable count.

Discrete boundary value function

The equilibrium problem is

$$r_i\{x\} = 2x_i - x_{i-1} - x_{i+1} + \frac{h^2}{2}(x_i + i\cdot h + 1)^3 \tag{1.5.2a}$$

where

$$h = \frac{1}{n+1} \quad \text{and} \quad x_0 = x_{n+1} = 0 \tag{1.5.2b}$$

with n variable [Moré81 §3.28]. The standard starting point is $x_{i[0]} = i\cdot h(i\cdot h - 1)$.

The partial derivatives are

$$\frac{\partial r_i}{\partial x_j} = \begin{cases} j = i: & 2 + \frac{3h^2}{2}(x_i + i\cdot h + 1)^2 \\ j = i\pm 1: & -1 \\ \text{else:} & 0 \end{cases} \tag{1.5.2c}$$

The solution depends on the variable count.

Discrete integral equation function

The equilibrium problem is

$$r_i\{x\} = x_i + \frac{h}{2}\left[(1-t_i)\sum_{k=1}^{i} t_k(x_k+t_k+1)^3 + t_i\sum_{k=i+1}^{n}(1-t_k)(x_k+t_k+1)^3\right] \tag{1.5.3a}$$

where

$$h = \frac{1}{n+1} \quad \text{and} \quad t_j = j\cdot h \tag{1.5.3b}$$

with n variable [Moré81 §3.29]. The standard starting point is $x_{i[0]} = t_i(t_i - 1)$.

The partial derivatives are

$$\frac{\partial r_i}{\partial x_j} = \begin{cases} j > i: & \frac{3h}{2}t_i(1-t_j)(x_j+t_j+1)^2 \\ j = i: & \frac{3h}{2}t_j(1-t_j)(x_j+t_j+1)^2 + 1 \\ j < i: & \frac{3h}{2}t_j(1-t_i)(x_j+t_j+1)^2 \end{cases} \tag{1.5.3c}$$

The solution depends on the variable count.

Duct flow problem

The equilibrium problem is

$$r = \begin{pmatrix} \frac{1}{\sqrt{f}} + 2\cdot\log_{10}\left(\frac{1}{D}\left[1 + \frac{2.7861}{V\sqrt{f}}\right]\right) - 9.7384634 \\ \frac{fV^2}{D} - 0.00179008 \\ VD^2 - 0.422104 \end{pmatrix} \tag{1.5.4a}$$

where

$$c_1 = \frac{-2}{\ln 10}\left(\frac{2.7861}{2.7861 + V\sqrt{f}}\right) \tag{1.5.4b}$$

from Section 1.4. With variables ordered (f, V, D), the standard starting points are $x_{[0]} = (0.02, 7, 1)^T$, $(90, 90, 90)^T$, and $(0.001, 0.0039, 34.06)^T$.

The Jacobian is

$$J = \begin{bmatrix} \frac{1}{2f}\left(c_1 - \frac{1}{\sqrt{f}}\right) & \frac{c_1}{V} & \frac{-2}{\ln 10\cdot D} \\ \frac{V^2}{D} & \frac{2fV}{D} & \frac{-fV^2}{D^2} \\ 0 & D^2 & 2VD \end{bmatrix} \tag{1.5.4c}$$

The solution is $x^{**} = (0.025, 0.293127, 1.2)^T$.

Freudenstein and Roth function

The equilibrium problem is

$$r = \begin{pmatrix} x_1 - x_2^3 + 5x_2^2 - 2x_2 - 13 \\ x_1 + x_2^3 + x_2^2 - 14x_2 - 29 \end{pmatrix} \tag{1.5.5a}$$

[Fletcher §6.2]. The standard starting points are $x_{[0]} = (6, 5)^T$ and $(15, -2)^T$.

The Jacobian is

$$J = \begin{bmatrix} 1 & -3x_2^2+10x_2-2 \\ 1 & 3x_2^2+2x_2-14 \end{bmatrix} \tag{1.5.5b}$$

The solution is $x^{**} = (5, 4)^T$.

Powell badly scaled function

The equilibrium problem is

$$r = \begin{pmatrix} 10^4 x_1 x_2 - 1 \\ \varepsilon^{-x_1} + \varepsilon^{-x_2} - 1.0001 \end{pmatrix} \tag{1.5.6a}$$

[Moré81 §3.3]. The standard starting point is $x_{[0]} = (0, 1)^T$.

The Jacobian is

$$J = \begin{bmatrix} 10^4 x_2 & 10^4 x_1 \\ -\varepsilon^{-x_1} & -\varepsilon^{-x_2} \end{bmatrix} \tag{1.5.6b}$$

The solution is $x^{**} = (1.09816 \cdot 10^{-5}, 9.10615)^T$ or $(9.10615, 1.09816 \cdot 10^{-5})^T$.

Powell singular function

The equilibrium problem is

$$r = \begin{pmatrix} x_1 + 10x_2 \\ \sqrt{5} \cdot (x_3 - x_4) \\ (x_2 - 2x_3)^2 \\ \sqrt{10} \cdot (x_1 - x_4)^2 \end{pmatrix} \tag{1.5.7a}$$

[Moré81 §3.13]. The standard starting point is $x_{[0]} = (3, -1, 0, 1)^T$.

The Jacobian is

$$J = \begin{bmatrix} 1 & 10 & 0 & 0 \\ 0 & 0 & \sqrt{5} & -\sqrt{5} \\ 0 & 2(x_2-2x_3) & -4(x_2-2x_3) & 0 \\ 2\sqrt{10}(x_1-x_4) & 0 & 0 & -2\sqrt{10}(x_1-x_4) \end{bmatrix} \tag{1.5.7b}$$

The solution is $x^{**} = (0, 0, 0, 0)^T$.

Rosenbrock function

The equilibrium problem is

$$r_i(x) = \begin{cases} \text{i odd:} & 10(x_{i+1} - x_i^2) \\ \text{i even:} & 1 - x_{i-1} \end{cases} \tag{1.5.8a}$$

with n variable but even [Dennis §B.1]. The standard starting point is $x_{[0]} = (-1.2, 1, .., -1.2, 1)^T$.

The partial derivatives are

$$\frac{\partial r_i}{\partial x_j} = \begin{cases} \text{i odd and j = i:} & -20x_i \\ \text{i odd and j = i+1:} & 10 \\ \text{i even and j = i-1:} & -1 \\ \text{else:} & 0 \end{cases} \tag{1.5.8b}$$

The solution is $x^{**} = (1, 1, .., 1, 1)^T$.

In particular, the two-dimensional version is

$$r = \begin{pmatrix} 10(x_2 - x_1^2) \\ 1 - x_1 \end{pmatrix} \tag{1.5.8c}$$

[Moré81 §3.1] with

$$J = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix} \tag{1.5.8d}$$

and $x^{**} = (1, 1)^T$.

Trigonometric function

The equilibrium problem is

$$r_i\{x\} = n - \sum_{k=1}^{n} \cos(x_k) + i\cdot[1 - \cos(x_i)] - \sin(x_i) \tag{1.5.9a}$$

with n variable [Moré81 §3.26]. The standard starting point is $x_{[0]} = (1/n, 1/n, .., 1/n)^T$.

The partial derivatives are

$$\frac{\partial r_i}{\partial x_j} = \begin{cases} j = i: & (1+i)\sin(x_j) - \cos(x_j) \\ j \neq i: & \sin(x_j) \end{cases} \tag{1.5.9b}$$

The solution depends on the variable count.

In particular, the two-dimensional version is

$$r = \begin{pmatrix} 3 - 2\cos(x_1) - \sin(x_1) - \cos(x_2) \\ 4 - 3\cos(x_2) - \sin(x_2) - \cos(x_1) \end{pmatrix} \tag{1.5.9c}$$

with

$$J = \begin{bmatrix} 2\sin(x_1)-\cos(x_1) & \sin(x_2) \\ \sin(x_1) & 3\sin(x_2)-\cos(x_2) \end{bmatrix} \tag{1.5.9d}$$

and $x^{**} = (i\cdot 2\pi, j\cdot 2\pi)^T$ or $x^{**} = (0.243064 + i\cdot 2\pi, 0.612676 + j\cdot 2\pi)^T$ for i, j integers.

Wall convection problem

The equilibrium problem is

$$r = \begin{pmatrix} 13.05\cdot T_{wout} - 0.5678\cdot T_{win} \\ 0.5678\cdot T_{wout} - 0.5678\cdot T_{win} + (20 - T_{win})h_{cin} \end{pmatrix} \tag{1.5.10a}$$

where

(1.5.10b) $$h_{cin} = 1.239 \cdot |20 - T_{win}|^{1/3}$$

from Section 1.3. With variables ordered (T_{wout}, T_{win}), the standard starting point is $x_{[0]} = (2, 18)^T$.

The Jacobian is

(1.5.10c) $$J = \begin{bmatrix} 13.05 & -0.5678 \\ 0.5678 & -(0.5678+4 \cdot h_{cin}/3) \end{bmatrix}$$

The solution is $x^{**} = (0.684948, 15.7425)^T$.

CHAPTER 2
THE LINEARIZED EQUILIBRIUM PROBLEM

This chapter discusses the basic technique for solving nonlinear algebraic systems-- Newton-Raphson's method-- and establishes the notation used throughout the thesis.

The first two sections define terms, introduce notation, and review the method of Newton-Raphson. Section 2.2 also introduces a geometric picture of the linearization and solution stages of the Newton-Raphson procedure.

Section 2.3 discusses computational aspects, focusing on matrix factorization and the solution of a system of linear equations.

2.1 Framework for Equilibrium Problems

Any nonlinear algebraic system may be expressed as a set of residual equations, $r\{x\}$, whose solution, x^{**}, satisfies

$$r\{x^{**}\} = 0 \tag{2.1.1}$$

For an n-dimensional problem (n equations in n unknowns), both r and x are vectors of length n. Equation 2.1.1 defines an equilibrium problem, as opposed to the evolution problem of Equation 1.1.4.

Other expressions of the algebraic system include the fixed-point form $x = f\{x\}$, and any mixed or algorithmic form defining the original system. The theory presented below treats only the residual form of Equation 2.1.1.

If x^{**} cannot be established directly, e.g. by Gaussian elimination for a linear system, an iterative solution technique generates a succession of guesses $x_{[0]}$, $x_{[1]}$, $x_{[2]}$, and so on. If successful, the sequence finds x^{**} (or an approximation to it), so that $r_{[0]}$, $r_{[1]}$, $r_{[2]}$, ... terminates at $r\{x^{**}\} = 0$. The central task of any iterative algorithm is to select the $x_{[k]}$ as effectively as possible, balancing the computational cost of choosing successive trial solutions against the expected improvement in the estimate of x^{**}.

2.2 Linearization and Newton-Raphson's Method

To solve a general system of nonlinear equations, Newton-Raphson's method linearizes the equations at the current best estimate of the solution, then solves that linear system to produce a new estimated solution. The linear models capture the local behavior of each equation using a tangent surface, which has the same function value and first derivatives as the residual at the current iterate. Newton-Raphson's method seeks the common intersection of these tangent planes in the plane $r = 0$ [Dahlquist §1.2].

Some variation on this basic technique occurs: (1) in every building energy solver described in [Axley, Clark, Hensen, Klein, Sahlin, Walton] (note some references call the technique Newton's method); (2) in the majority of the 50 airflow network and general building energy solvers surveyed in [Feustel]; (3) in five of the eight general-purpose algebraic equation solvers tested in [Hiebert82]; and (4) in the majority of the algorithms currently listed in problem F2 classification (solution of a system of nonlinear equations) of the Guide to Available Mathematical Software [Boisvert].

Linearizing a single residual

Consider a single residual relation, r_i where $1 \le i \le n$. Evaluating or estimating its first derivatives at the current iterate, $x_{[k]}$, and expanding by Taylor's formula,

$$(2.2.1) \qquad r_i\{x_{[k]} + \Delta x\} \approx r_i\{x_{[k]}\} + \Delta x_1 \cdot \left.\frac{\partial r_i}{\partial x_1}\right|_{x_{[k]}} + \Delta x_2 \cdot \left.\frac{\partial r_i}{\partial x_2}\right|_{x_{[k]}} + \ldots + \Delta x_n \cdot \left.\frac{\partial r_i}{\partial x_n}\right|_{x_{[k]}}$$

[Dahlquist §6.9.2]. Equation 2.2.1 estimates $r_i\{x\}$ in the neighborhood of $x_{[k]}$. For nonlinear residual functions, the expansion loses accuracy for large steps Δx.

Defining the gradient vector

$$(2.2.2) \qquad \nabla r_i = \begin{pmatrix} \frac{\partial r_i}{\partial x_1} \\ \frac{\partial r_i}{\partial x_2} \\ \vdots \\ \frac{\partial r_i}{\partial x_n} \end{pmatrix}$$

[Dahlquist §10.5.1], the model can be written more compactly as

$$(2.2.3a) \qquad r_i\{x\} \approx r_{i[k]} + \nabla r_{i[k]}^T \Delta x_{[k]}$$

where

$$(2.2.3b) \qquad \Delta x_{[k]} = x - x_{[k]}$$

Alternately, define

$$(2.2.4) \qquad \hat{r}_{i[k]}\{x\} = r_{i[k]} + \nabla r_{i[k]}^T (x - x_{[k]})$$

The notation $\hat{r}_{i[k]}$ represents the models of r_i formed at the k^{th} iteration of a nonlinear equation-solving algorithm. Equation 2.2.4, tangent to the i^{th} residual equation at $x_{[k]}$ [Ellis §13.7], estimates $r_i\{x\}$ nearby; that is, $\hat{r}_{i[k]}\{x\} \approx r_i\{x\}$ in the neighborhood of $x_{[k]}$.

Example: linearization in one dimension

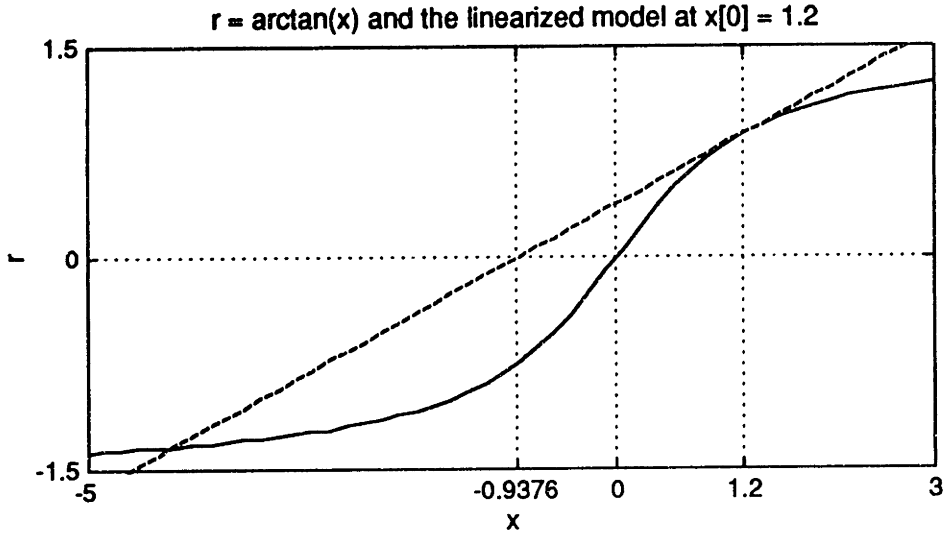

Figure 2.2.1. The residual equation r{x} = arctan(x), and its linearized model at $x_{[0]}$ = 1.2. Newton-Raphson's method finds the next iterate, $x_{[1]}$ = -0.9376, by following the linearized model to the point where it crosses the x axis.

For the case n=1 (one equation in one unknown), consider

(2.2.5a) $$r\{x\} = \arctan(x)$$

[after Dennis §2.4]. The solution x** = 0, since arctan(0) = 0. The gradient

(2.2.5b) $$\frac{\partial r}{\partial x} = \frac{dr}{dx} = \frac{1}{1 + x^2}$$

giving a linearized model

(2.2.5c) $$\hat{r}_{[k]} = \arctan(x_{[k]}) + \frac{x - x_{[k]}}{1 + x_{[k]}^2}$$

at $x_{[k]}$.

For an initial guess $x_{[0]}$ = 1.2, Equation 2.2.5c gives $\hat{r}_{[0]} = 0.3843 + 0.4098x$. Figure 2.2.1 shows the arctan function and this linearized model. The figure also suggests how to use the model, by taking the point where $\hat{r}_{[0]} = 0$ as the next trial solution, $x_{[1]}$. Setting $\hat{r}_{[k]} = 0$ in Equation 2.2.5c gives

(2.2.5d) $$x_{[k+1]:NR} = x_{[k]} - (1 + x_{[k]}^2)\cdot\arctan(x_{[k]})$$

so for the sample solution depicted, $x_{[1]:NR}$ = -0.9376.

Jacobian matrix

Repeating the Taylor series expansion for each residual linearizes the entire equilibrium problem:

(2.2.6) $$\hat{r}_{[k]}(x) = r_{[k]} + \begin{bmatrix} \nabla r_{1[k]}^T \\ \nabla r_{2[k]}^T \\ \vdots \\ \nabla r_{n[k]}^T \end{bmatrix} (x - x_{[k]})$$

Strictly the model is affine, not linear, since a linear subspace passes through the origin [Dennis §2.2]. However, models such as Equation 2.2.6, which truncate the Taylor series after the first derivative, often are called linear.

Equation 2.2.6 defines the Jacobian matrix of partial derivatives of the residuals. The Jacobian has elements

(2.2.7a) $$[J]_{i,j} = \frac{\partial r_i}{\partial x_j}$$

[Strang86 §5.1]. Its i^{th} row is the gradient of a particular residual equation r_i. Conversely, the j^{th} column of the Jacobian holds the partial derivatives of all the residuals with respect to a particular independent variable x_j:

(2.2.7b) $$J = (\nabla r)^T = \begin{bmatrix} \nabla r_1^T \\ \nabla r_2^T \\ \vdots \\ \nabla r_n^T \end{bmatrix} = \begin{bmatrix} \frac{\partial r}{\partial x_1} & \frac{\partial r}{\partial x_2} & .. & \frac{\partial r}{\partial x_n} \end{bmatrix}$$

The Jacobian definition makes $\Delta r = J\Delta x$ for small excursions Δx about the point where it is evaluated. (Linear residual equations have a constant Jacobian, and $\Delta r = J\Delta x$ for all Δx.) As in Equation 2.2.6, adding the current value of the residuals completes the linearized model:

(2.2.8) $$\hat{r}_{[k]}(x) = r_{[k]} + J_{[k]}(x - x_{[k]})$$

Newton-Raphson's method

Following the one-dimensional example, Newton-Raphson's method chooses the next iterate $x_{[k+1]}$ in order to zero the linearized residuals:

$$\hat{r}_{[k]}(x_{[k+1]}) = 0 = r_{[k]} + J_{[k]}(x_{[k+1]} - x_{[k]})$$

or

(2.2.9a) $$J_{[k]}\Delta x_{[k]:NR} = -r_{[k]}$$

where

(2.2.9b) $$\Delta x_{[k]:NR} = x_{[k+1]:NR} - x_{[k]}$$

[Dahlquist §6.9.2]. The notation $x_{[k+1]:NR}$ states that $x_{[k+1]}$ is the Newton-Raphson choice for the next iterate; later algorithms will admit other selections of $x_{[k+1]}$. Mathematically,

(2.2.10) $$x_{[k+1]:NR} = x_{[k]} - J_{[k]}^{-1} r_{[k]}$$

but in practice the inverse is not calculated [Dennis §3.2]. Matrix factorization requires fewer floating-point operations to solve Equation 2.2.9; this holds in general for any numeric operation involving the inverse [Golub §3.4.11].

Close to x**, Newton-Raphson converges rapidly. For example, the inverse tangent problem of Equation 2.2.5 effectively converges by the fifth iteration, as shown in the first two columns of Figure 2.2.2.

Starting point 1		Iteration	Starting point 2	
$x_{[k]}$	$r_{[k]}$	[k]	$x_{[k]}$	$r_{[k]}$
1.2	0.8761	0	1.5	0.9828
-0.9376	-0.7532	1	-1.694	-1.038
0.4777	0.4457	2	2.321	1.164
-0.06965	-0.06954	3	-5.114	-1.378
0.0002251	0.0002251	4	32.30	1.540
$-7.599 \cdot 10^{-12}$	$-7.599 \cdot 10^{-12}$	5	-1575.	-1.570

Figure 2.2.2. Newton-Raphson's method applied to the inverse tangent problem. In the first two columns, $r_{[k]} \approx x_{[k]}$ by k=3 since the sequence converges to the solution, $x^{**} = 0$, where the linearized model is $\hat{r} = x$. In the last two columns, where $x_{[0]}$ is far from the solution, the sequence diverges.

As shown by the second set of iterations in Figure 2.2.2, Newton-Raphson can diverge as well. In this example, $x_{[0]} = 1.5$ starts too far from the solution, and successive tangents project the trial solution to increasingly distant points, each on the opposite side of x**.

Typically, equation-solving algorithms complement Newton-Raphson's method with some means of controlling divergence, and choose $x_{[k+1]}$ by alternate means when necessary.

Controlling divergence

Consider again the second sequence of iterations in Figure 2.2.2. By inspection, the sequence diverges-- the trial solutions jump from one side of the root to the other. The increasing magnitude of the residual at each iteration verifies this divergence.

In multiple dimensions the first type of divergence, in the variables x, may be difficult to detect or even to define appropriately. The second sign of divergence, in the residual magnitude, easily generalizes to multiple dimensions by taking a norm of the residual vector. An algorithm can compare the value of the norm before and after each Newton-Raphson step, taking appropriate action if the norm increases (or, more ambitiously, if it fails to decrease by a sufficient amount). Typically if the Newton-Raphson step fails, the global strategy generates a new trial step based on the problem of minimizing the residual norm.

The most common norm, the square of the Euclidean length [Press §9.7], or r-square,

$$\|r\|_2^2 = \sum_{i=1}^{n} r_i^2\{x\} = r^T r$$

has convenient properties for use in a minimization-based technique [Dennis §6.5]. Chapter 4 considers this and other choices.

Introducing a descent requirement on a norm is not the only way to guard against divergence. For instance, homotopy or continuation methods imbed the original system into an auxiliary system such as

$$\theta \cdot r\{x\} + (1 - \theta)(x - x_{[0]}) = 0$$

with a trivial solution when the scalar parameter $\theta = 0$. Then increasing θ slightly towards 1 defines a new nonlinear system, for which a good initial guess, from the solution for the previous value of θ, is known [Fletcher §6.2, Dahlquist §6.9.3]. The methods may use Newton-Raphson's technique to solve the auxiliary system at each new value of θ.

According to the current Guide to Available Mathematical Software [Boisvert], nonlinear equation solvers in the CONTIN and HOMPACK libraries use homotopy methods. However the majority of the equation-solving algorithms listed there, including those from the CMLIB, IMSLM, HARWELL, MINPACK, NAG, PORT, and SLATEC libraries, control divergence by applying minimization techniques, usually variations of Powell's hybrid method, to r-square (see also [Hiebert82 §2, Strang86 §A]). In the building energy community, the IDA solver uses a homotopy technique, solving the sequential auxiliary systems by Newton-Raphson's method, for initial-value calculations; and unmodified Newton-Raphson iterations for algebraic systems during the rest of an evolution problem [Sahlin §2.4]. Other solvers, such as HVACSIM+, use minimization methods. This thesis focuses exclusively on the minimization-based techniques; Chapter 3 describes one variation on Powell's hybrid method, the double dogleg algorithm.

Terminology

Some authors abbreviate the term Newton-Raphson to Newton's method [Strang86 §5.1], and occasionally apply the name Newton-Raphson to algorithms for minimizing a scalar cost function [Dahlquist §10.5.3]. Strictly Newton-Raphson's and Newton's methods are different, although the strong correspondence, mathematical and conceptual, between them blurs the distinction. For the development to follow, the distinction is important, so this thesis adopts the following naming conventions [Fletcher §6.2]:

(1) Newton-Raphson's method. To solve a system of equations $r\{x^{**}\} = 0$, find $J = (\nabla r)^T$, and solve $J\Delta x = -r$.

(2) Newton's method. To minimize a function f, find its gradient $g = \nabla f$ and solve $g\{x^*\} = 0$ by finding the Hessian $G = \nabla g$ and solving $G\Delta x = -g$.

(3) The Gauss-Newton method. To solve an overdetermined system of equations $r\{x\}$, find $J = (\nabla r)^T$, form the function $r^T r$, and apply Newton's method with $g = 2J^T r$ and $G = 2J^T J$.

Example: wall convection problem

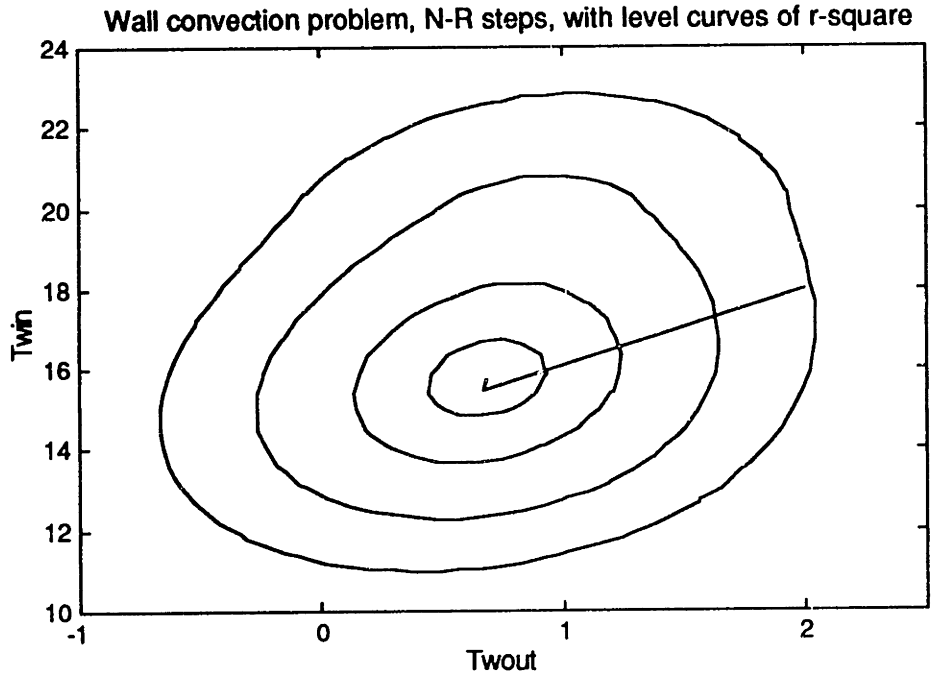

Figure 2.2.3. Two Newton-Raphson steps for the wall convection problem. The level curves $r^T r$ = 300, 150, 50, and 10 indicate progress towards the solution.

To demonstrate Newton-Raphson's method, consider the wall convection problem,

$$r_1 = 13.05 \cdot T_{wout} - 0.5678 \cdot T_{win}$$
$$r_2 = 0.5678 \cdot T_{wout} - 0.5678 \cdot T_{win} + (20 - T_{win}) \cdot h_{cin}$$
$$h_{cin} = 1.239 \cdot |20 - T_{win}|^{1/3}$$

of Chapter 1. An initial guess $x_{[0]} = (2, 18)^T$ gives $r_{[0]} = (15.88, -5.963)^T$. Figure 2.2.3 shows the first two iterations of the method, along with some level curves of $r^T r$.

The first Jacobian row follows trivially from the already-linear function $r_1\{x\}$. Furthermore, each Newton-Raphson step sets $r_1 = 0$ exactly [Dennis §5.1], since it zeros r_1 along with a linearized r_2.

For the second row, $\partial r_2/\partial T_{wout} = 0.5678$ and $\partial r_2/\partial T_{win} = (20 - T_{win}) \cdot \partial h_{cin}/\partial T_{win} - (0.5678 + h_{cin})$. Evaluating $\partial h_{cin}/\partial T_{win}$ separately for the cases $T_{win} \le 20$ and $T_{win} > 20$, the Jacobian can be expressed for either case as

(2.2.11)
$$J = \begin{bmatrix} 13.05 & -0.5678 \\ 0.5678 & -(0.5678 + 4 \cdot h_{cin}/3) \end{bmatrix}$$

The determinant $D = 0.5678^2 - 13.05 \cdot 4 \cdot h_{cin}/3 - 13.05 \cdot 0.5678$ shows that $h_{cin} = -0.4073$ gives a singular Jacobian. However h_{cin} is always positive and therefore the inverse always exists. Here,

$$J^{-1} = \begin{bmatrix} -(0.5678 + 4 \cdot h_{cin}/3) & 0.5678 \\ -0.5678 & 13.05 \end{bmatrix} \cdot \frac{1}{D}$$

and Equation 2.2.10 gives

$$\begin{pmatrix} T_{wout} \\ T_{win} \end{pmatrix}_{[k+1]:NR} = \begin{pmatrix} -0.5678 \\ -13.05 \end{pmatrix} \frac{h_{cin[k]} \cdot (20 + T_{win[k]}/3)}{D_{[k]}}$$

for each Newton-Raphson step. Thus from $x_{[0]} = (2, 18)^T$ the method sets $x_{[1]} = (0.6729, 15.46)^T$, where $r_{[1]} = (0, 0.9023)^T$. See Figure 2.2.3.

Geometric interpretation of a Newton-Raphson iteration

Figure 2.2.1 shows a Newton-Raphson step for the problem $r = \arctan\{x\}$, starting from $x_{[0]} = 1.2$. The graphical elements of the figure depict the steps of the method for one-dimensional problems: (1) find the value and slope of the residual function at the current guess; (2) draw the tangent line, that is, the line with the same slope and function value; and (3) follow the tangent line to $r = 0$.

The method is the same for simultaneous equations, although with vector $x_{[k]}$ the geometric picture takes n+1 dimensions: n dimensions for the independent variables x, and an added dimension to depict the value of the residual function at each point. From Equation 2.2.4, model each residual by

$$\hat{r}_{i[k]}\{x\} = r_{i[k]} + \nabla r_{i[k]}^T (x - x_{[k]}) \tag{2.2.12}$$

(From Equation 2.2.7b the transposed gradient $\nabla r_{i[k]}^T$ is just the i^{th} row of the Jacobian.)

For example, the wall convection problem at $x_{[0]} = (2, 18)^T$ has

$$r_{[0]} = (15.88, -5.963)^T \quad \text{and} \quad J_{[0]} = \begin{bmatrix} 13.05 & -0.5678 \\ 0.5678 & -2.649 \end{bmatrix}$$

The affine model of its first residual is

$$\hat{r}_{1[0]}\{x\} = 15.88 + (13.05 \quad -0.5678) \begin{pmatrix} T_{wout}-2 \\ T_{win}-18 \end{pmatrix} \tag{2.2.13a}$$

This plane, with the same function value and slope as has r_1 at the current iterate $x_{[0]}$, models r_1 at that point. In fact since r_1 is linear in the temperatures, Equation 2.2.13a simply rearranges its terms.

The second, nonlinear, residual has a tangent plane

$$\hat{r}_{2[0]}\{x\} = -5.963 + (0.5678 \quad -2.649) \begin{pmatrix} T_{wout}-2 \\ T_{win}-18 \end{pmatrix} \tag{2.2.13b}$$

at $x_{[0]}$. Figure 2.2.4 shows r_2 and $\hat{r}_{2[0]}$ in the region of interest. The figure marks the point of tangency, at $(x_{[0]}, r_{2[0]})$, on both the residual surface and its tangent plane.

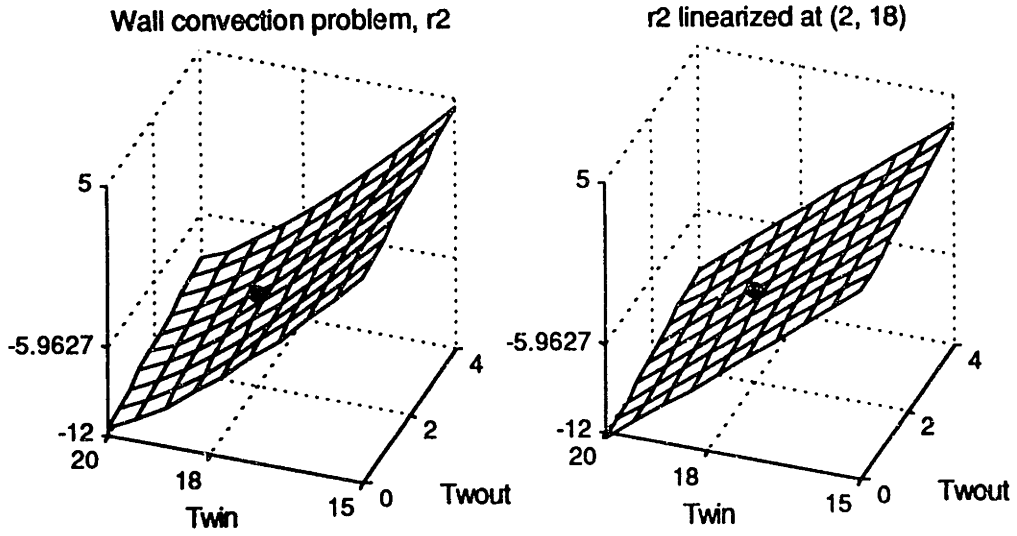

Figure 2.2.4. The second residual of the wall convection problem, and its affine model formed at $(2, 18)^T$. *A solid dot marks the point of tangency on the residual surface, and the corresponding point on the tangent plane.*

Finally the Newton-Raphson iteration follows the tangent approximation to r = 0-- a straightforward task for the one-dimensional problem, since the line in Figure 2.2.1 dictates $x_{[k+1]}$ unambiguously. A tangent plane such as $\hat{r}_{2[0]}$, on the other hand, intersects r = 0 in a line, allowing a range of points for $x_{[1]}$, any of which zeros the affine model. Setting $\hat{r}_{2[0]} = 0$ in Equation 2.2.13b and rearranging,

$$0.5678(T_{wout} - 2) - 2.649(T_{win} - 18) = 5.963$$

or

(2.2.14) $$T_{win} = 0.2143 \cdot T_{wout} + 15.32$$

Any point on this line-- for example $\hat{x}_{[k+1]} = (-2, 14.89)^T$, $(4, 16.18)^T$, and $(100, 36.75)^T$-- satisfies $\hat{r}_{2[0]} = 0$.

Of course the preferred choice of $x_{[k+1]}$ sets $\hat{r}_{1[0]} = 0$ as well. (The three points above give, respectively, $\hat{r}_{1[0]} = -34.56$, 43.01, and 1284.) The tangent plane to r_1 at $x_{[0]}$ also intersects r = 0 in a line $\hat{r}_{1[0]} = 0$, or

$$13.05 \cdot T_{wout} - 0.5678 \cdot T_{win} = 0$$

since $\hat{r}_{1[0]} = r_1$. Substituting Equation 2.2.14 for T_{win} and solving gives $x_{[1]} = (0.6729, 15.46)^T$. Figure 2.2.3 depicts this first Newton-Raphson step. Evaluated at the new iterate, the actual residuals have $r_{[1]} = (0, 0.9023)^T$, instead of the expected $(0, 0)^T$, due to nonlinearities in the second equation.

Increasing the problem dimension does not change the picture. Each affine model is a hyperplane tangent to its respective residual, and the method seeks $x_{[k+1]}$ where these surfaces intersect on r = 0. Mathematically this follows because Equation 2.2.12 zeros each tangent hyperplane $\hat{r}_{i[k]}$ where

$$r_{i[k]} + \nabla r_{i[k]}^T (x - x_{[k]}) = 0$$

or

$$\nabla r_{i[k]}^T (x - x_{[k]}) = -r_{i[k]} \tag{2.2.15}$$

Equation 2.2.15 just gives the i^{th} row of the Newton-Raphson solution $J_{[k]}\Delta x_{[k]} = -r_{[k]}$. Thus, the Newton-Raphson procedure finds the point which simultaneously zeros all the affine models of the residual relations at $x_{[k]}$, and calls this point $x_{[k+1]}$ [Dennis §5.1].

Geometric interpretation of a singular Jacobian

In the geometric interpretation, a Newton-Raphson step follows the tangent planes of the residuals to r = 0, and seeks their intersection in that plane. The wall convection problem always has a unique solution since: (1) neither plane can have zero slope, so that both always intersect r = 0 in a line; and (2) with h_{cin} positive the two planes never cross r = 0 in parallel lines, so that they must intersect at one point, rather than in zero or in infinitely many points.

Other linearized systems may not avoid these difficulties. For example, a tangent surface with zero slope never intersects r = 0 (unless the residual is zero, when the tangent plane itself is r = 0). This gives no Newton-Raphson intersection (or infinitely many). Geometrically, the residual function has a local minimum or maximum, and thus a zero slope, at such a point. Mathematically, the hyperplane enters a zero row into the Jacobian, making the Jacobian singular. The zero row prevents a unique solution because the corresponding row of $J\Delta x$ is zero, and hence $J\Delta x = -r$ cannot match a nonzero residual in that row (and if the residual is zero already then many choices of Δx are possible).

Similarly, two parallel hyperplanes never intersect (no Newton-Raphson solution), unless they are the same hyperplane (infinitely many solutions). The Jacobian is singular because one of its rows is the same as another. In an elimination step this introduces a zero row to the equivalent linear system. If the corresponding residuals are different, the tangents are distinct hyperplanes which never intersect; if the same, then the tangents are the same hyperplane. Elimination sets one of the residuals in the equivalent linear system to zero and, as above, the zero residual allows infinitely many solutions.

Two hyperplanes need not be parallel to have no intersection at r = 0. In a two-dimensional problem, the tangent planes may cross r = 0 in parallel lines, and intersect in a line above or below it. The Jacobian is singular because one of its rows is a scalar multiple of the other.

Finally, when n > 2 each tangent surface may intersect every other tangent in r = 0, but with no unique point at which they all come together [Strang88 §1.2]. For n = 3, where r = 0 is a three-dimensional surface, picture a prism of infinite extent. Each side of the prism is the intersection of a tangent hyperplane with r = 0. Each hyperplane intersects the other two, but since the prism never tapers to a point, there is no common point of intersection.

In this last case, at least one of the hyperplanes is a linear combination of some of the others, so that movement along those other surfaces produces a parallel movement along the degenerate one. (Of course, any one of the linearly-dependent hyperplanes can be expressed as the linear combination of the others; the system as a whole, not a particular equation, is degenerate.) The Jacobian is singular because its rows are linearly dependent; elimination finds a zero row in some equivalent linear system.

Instability near singular points

Newton-Raphson's method is unstable near points where the linearized system gives a singular Jacobian [Fletcher §6.2]. Near these points, some tangent hyperplanes in some equivalent linear system are nearly parallel, and slight changes in the planes produce large shifts in the estimated solution. Therefore linearizing the system at two points near the singularity can give vastly different Newton-Raphson steps. In this case, a stabilizing descent requirement prevents divergence, but also traps the search at a local minimum in the norm [Fletcher §6.2].

Chapter 4 treats this point in greater detail. However a special case, where the singularity arises from a local extremum in one residual equation, follows directly from the geometric interpretation given above.

Instability near local residual extrema

Because a residual function has zero slope at a local maximum or minimum, near the extremum its tangent hyperplanes have shallow slopes. While this does not prevent solving Equation 2.2.9, it destabilizes Newton-Raphson's method by making the Jacobian models formed in the neighborhood of the extremum indicate greatly different Newton-Raphson steps from one another.

A residual function with a small gradient magnitude $\|\nabla r_{i[k]}\|$ compared to its value $|r_{i[k]}|$ at the model point has a tangent hyperplane of relatively small slope, which requires a long run before it intersects the plane $r = 0$. Equation 2.2.9, placing $\Delta x_{[k]:NR}$ in order to zero every residual model, needs a long step to reach this intersection. This long step magnifies changes in the angle between the shallow hyperplane and $r = 0$, making the Newton-Raphson step sensitive to slight changes in the hyperplanes with relatively small slopes.

If the shallow slope comes from forming the model near an extremum, then the Newton-Raphson step formed at nearby points can easily change directions since the slope passes through zero at the extremum. Residual surfaces with saddle points and troughs can cause similar instabilities, though these should not trap a method, as might a local minimum in $|r_i|$.

The Newton-Raphson direction steps away from a local maximum in $|r_i|$, but for a local minimum in $|r_i|$-- that is, for a local minimum when $r_i > 0$ and for a local maximum when $r_i < 0$-- the Newton-Raphson direction steps towards the singularity. If a divergence test, such as a descent requirement, forces the search to backtrack along the Newton-Raphson direction, it may halt the search before escaping the extremum, and may end by stepping closer to the extremum. Then the next tangent hyperplane, of even smaller slope, dominates the Newton-Raphon direction even more strongly. In fact, if the step places $x_{[k+1]}$ on the opposite side of the extremum, the new hyperplane reverses the direction of the Newton-Raphson step. If successive descent steps move the solution closer towards the extremum, the method eventually discovers a singular Jacobian at the extremum itself, or it stagnates, unable to take a step fine enough to satisfy the divergence test.

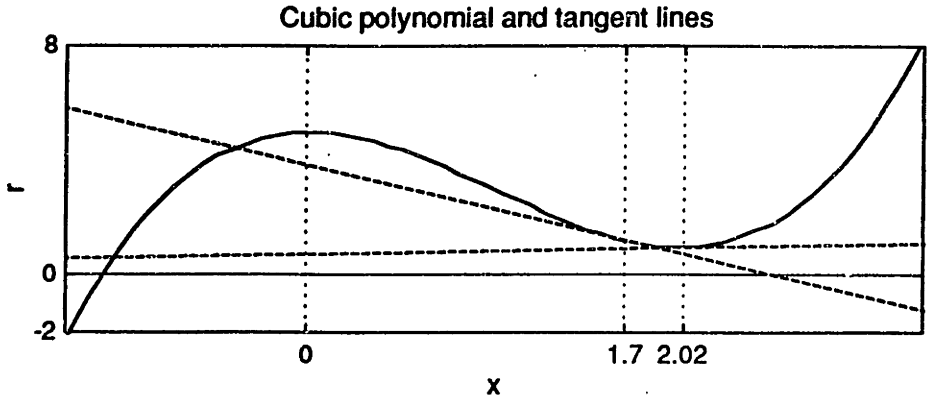

Figure 2.2.5. The cubic $r = x^3 - 3x^2 + 5$, and two tangent lines drawn on opposite sides of the local minimum at x = 2 The singularity makes Newton-Raphson's method unstable, since the direction and step length indicated by the linearized model varies greatly for small changes in x.

Figure 2.2.5 illustrates the problem in one dimension [after Fletcher §6.2]. The cubic $r = x^3 - 3x^2 + 5$ has a local minimum at x = 2. Nearby, small changes in x induce large changes in the Newton-Raphson solution, because tangents drawn near the minimum project long steps in both directions. Forming the model closer to the minimum decreases the slope of the tangent and gives longer Newton-Raphson steps. Some of these, e.g. the tangent line drawn at x = 1.7, give a Newton-Raphson step to the right; others, e.g. that at x = 2.02, dictate a search to the left.

In this one-dimensional case, the local minimum may trap the search, e.g. if it starts to the left of x = 2, steps too far to the right, and backtracks into the minimum. However forming a model on the right, e.g. at x = 2.02 as shown, may allow a solution, depending on how the algorithm chooses the step length.

In multiple dimensions, an extremum in one residual function still destabilizes the Newton-Raphson solution, by placing a nearly zero row in the Jacobian. However, the descent requirement in the residual norm can stabilize the method without trapping the search, provided the norm does not have a minimum at the local minimum in $|r_i|$. In multiple dimensions, minima in the residual norm, rather than in one residual, trap the descent-based search. If they do not correspond to a solution, where the norm is zero, these minima occur at singularities in the linearized equations, destabilizing the Newton-Raphson solution as described above.

Convergence near solution

If $x_{[k]}$ is near the solution, and if J is nonsingular in the neighborhood of x**, then Newton-Raphson's method converges quadratically [Strang86 §5.1]; that is, it squares the magnitude of the error at each step. A detailed proof [Dennis §5.2], sketched below, establishes a testable criterion for determining possible divergence of the method.

Assume: (1) that r{x} is continuously differentiable, and J{x} is Lipschitz continuous, within a radius r_d of x**; (2) that $J^{**-1} = J^{-1}\{x^{**}\}$ exists, with its induced norm

(2.2.16a) $$\|J^{**-1}\| \leq \beta$$

and (3) that for some as yet undefined ε,

(2.2.16b) $$\|x_{[k]} - x^{**}\| \leq \varepsilon$$

By definition, the Lipschitz continuity of J means that within r_d of x^{**},

$$\|J_{[k]} - J^{**}\| \leq \gamma \cdot \|x_{[k]} - x^{**}\| \tag{2.2.16c}$$

for some constant γ [Dennis §4.1]. Note that Equation 2.2.16c requires $\varepsilon \leq r_d$. A further implication of Lipschitz continuity is

$$\|r^{**} - r_{[k]} - J_{[k]}(x^{**} - x_{[k]})\| \leq \frac{\gamma}{2} \|x^{**} - x_{[k]}\|^2 \tag{2.2.17}$$

[Dennis §4.1].

For the induced matrix norm,

$$\|J^{**-1}(J_{[k]} - J^{**})\| \leq \|J^{**-1}\| \cdot \|J_{[k]} - J^{**}\|$$

so from Equations 2.2.16,

$$\|J^{**-1}(J_{[k]} - J^{**})\| \leq \beta \cdot \gamma \cdot \|x_{[k]} - x^{**}\| \leq \beta \cdot \gamma \cdot \varepsilon \tag{2.2.18}$$

By an auxiliary perturbation result, if $\beta\gamma\varepsilon \leq \frac{1}{2}$ then

$$\|J_{[k]}^{-1}\| \leq \frac{\|J^{**-1}\|}{1 - \|J^{**-1}(J_{[k]} - J^{**})\|}$$

[Dennis §3.1] and hence

$$\|J_{[k]}^{-1}\| \leq 2 \cdot \|J^{**-1}\| \leq 2\beta \tag{2.2.19}$$

Note that in addition to the limit $\varepsilon \leq r_d$ established above, the auxiliary result requires $2\beta\gamma\varepsilon \leq 1$, so that

$$\varepsilon = \min\{r_d, \frac{1}{\beta\gamma\varepsilon}\} \tag{2.2.20}$$

Now the Newton-Raphson step, Equation 2.2.10, gives

$$x_{[k+1]} - x^{**} = x_{[k]} - x^{**} - J_{[k]}^{-1} r_{[k]}$$

or

$$\|x_{[k+1]} - x^{**}\| \leq \|J_{[k]}^{-1}\| \cdot \|r^{**} - r_{[k]} - J_{[k]}(x_{[k]} - x^{**})\|$$

since $r^{**} = 0$. Then from Equations 2.2.17 and 2.2.19,

$$\|x_{[k+1]} - x^{**}\| \leq \beta \cdot \gamma \cdot \|x_{[k]} - x^{**}\|^2 \tag{2.2.21}$$

This quadratic convergence is subject to the assumptions of Equations 2.2.16, and the restriction of Equation 2.2.20, which limits the radius around x^{**} from which the Newton-Raphson sequence converges quadratically.

From Equations 2.2.16b and 2.2.20, when $x_{[k]}$ is sufficiently close to x^{**} then

$$\|x_{[k+1]} - x^{**}\| \leq \frac{1}{2} \|x_{[k]} - x^{**}\| \tag{2.2.22}$$

and, at worst, each step halves the distance to the solution. Equation 2.2.22 implies a divergence check for a Newton-Raphson sequence, one which tests the angles between and lengths of successive steps, rather than testing a residual norm.

Higher-order residual models

Equation-solving algorithms are not restricted to first-order models of the residuals. For instance, a minimization-based algorithm could use the second-order Taylor series to form n quadratic residual models. However the computational expense mitigates against this approach. Not only does forming and storing the requisite models add to the algorithm's overhead, but their increased complexity also adds to the burden of extracting information from them.

Specifically, a solver carrying second-order information about each residual no longer can use Newton-Raphson's method to find the step which best reduces each residual. Indeed, since the second-order models may preclude any common zero, and may introduce multiple zeros even for nonsingular systems, they leave open the question how to define the best step in the model. One means of choosing a step again introduces function minimization to the nonlinear equilibrium problem. The solver can model a norm, such as r-square, by combining the n quadratic residual models, then choose the step which minimizes the model. In exchange for calculating, storing, and processing the order-n^3 data required to support the quadratic models (as opposed to the order-n^2 data for the linear models), the solver better estimates this minimum.

In practice, a solver probably would not calculate and store the full second-order behavior for each residual. For instance, it can estimate a tensor of second derivatives of the residual vector, by applying a secant method to observed residual values at previous iterations [Feng §2; note Hiebert81 §2 discusses a similar technique for nonlinear least squares minimizations]. This thesis does not consider these techniques, but uses linear residual models only.

2.3 Direct Techniques for General Linear Systems

Newton-Raphson's method requires solving a linear system $J\Delta x = -r$ for Δx at each iteration. This section discusses direct (factorization) techniques for solving a linear system $Ax = b$. Chapter 4 extends these methods to nonlinear problems.

Direct versus indirect techniques

Newton-Raphson's method does not specify how to solve $J\Delta x = -r$. Techniques for solving the canonical linear system $Ax = b$ are either: (1) direct, finishing after a fixed number of prescribed steps; or (2) indirect, terminating according to a convergence criteria or an externally-imposed limit [Press §2.0].

The direct methods factor A into a product of special matrices whose properties allow the calculation of x using relatively simple manipulations of the right-hand side, b.

The indirect methods search for x using iterative algorithms, and are built around matrix-vector multiplications with an unfactored A [Golub §10.1]. A prematurely-terminated iterative method provides an estimated solution. In contrast, prematurely terminating a direct method yields no solution.

When applying Newton-Raphson's method to nonlinear equilibrium problems, the distinction between direct and indirect methods is blurred by the need to iterate on Equation 2.2.9a, $J_{[k]}\Delta x_{[k]} = -r_{[k]}$. However even for linear systems there is some overlap. Two salient examples are: (1) iterative improvement of a solution; and (2) the conjugate gradient method. During iterative improvement, the same matrix factorization used to compute a solution $\hat{x}$ to $Ax = b$, is used to compute a correction to $\hat{x}$; this is done by calculating $\hat{b} = A\hat{x}$ and then solving $A(x - \hat{x}) = (b - \hat{b})$ [Dahlquist §5.5.6, Golub §3.5.3].

Iterative improvement, recommended to combat the loss of precision due to roundoff errors in solving linear systems [Press §2.5], can be seen as an application of the Newton-Raphson method, in which J does not change from one iteration to the next.

The conjugate gradient method often is considered direct, because with exact arithmetic on a linear problem it finds a solution in a fixed number of steps [Dahlquist §10.4.4]. However, it is best treated as an iterative method since rounding errors can make the method converge either before or after the step count implied by exact analysis [Golub §10.2.7]. The conjugate gradient method applies directly to the system $Ax = b$ when A is symmetric and positive-definite, and applies to $A^TAx = A^Tb$ otherwise. However forming A^TA squares the condition number of the system of interest, increasing the iteration count and decreasing the accuracy obtainable [Press §2.7]. For general linear systems, [Press §2.7] gives a biconjugate gradient method, related to the conjugate gradient method, but which does not use A^TA.

Choice of technique

For linear problems, direct methods are preferred unless the matrix is large and sparse, that is, containing a relatively large number of zero elements. The spatial discretization of partial differential equations is a common source of sparsity [Golub §10.1]. Building energy models can produce sparse Jacobians since an important means of energy transport is by fluid flow through components in series, e.g. air through a ducted ventilation system. In such systems, a component's residual relations usually depend only on variables describing that component and its proximate neighbors. Hence the derivatives with respect to the complete set of variables tend mostly to be zero.

For full matrices, direct methods are more efficient [Dahlquist §5.3], but as the problem dimension grows the roundoff errors introduced by factoring A can become significant. Factorization can fail for nearly-singular matrices [Press §2.0], and as roundoff errors accumulate, even a nonsingular matrix can attain a "singular" factorization [Press §2.5]. Typically a 32-bit (single-precision) floating-point computing environment can support problem dimensions in the range of 20-50, and a double-precision environment in the range of a few hundred [Press §2.0].

Sparse matrices should not be treated using unmodified general matrix factorization algorithms because: (1) the resulting factored matrices are not sparse [Golub §10.1], increasing storage requirements; and (2) most of the operations, performed on zero elements, do not affect the solution [Press §2.7], and therefore needlessly increase the calculation time. Similarly, general matrix-vector multiplication is inefficient for a sparse matrix, since many of the scalar products are zero.

Matrix-vector multiplication is easily specialized to account for sparsity, making the iterative methods more attractive [Dahlquist §5.4.4]. Matrix factorization can be specialized to a particular pattern of sparsity by choosing data structures and pivoting strategies to match [Golub §10.1], but the treatment required to retain sparsity in the factored representation tends to decrease numerical stability [Press §2.7]. Whether to employ direct or iterative techniques for a sparse matrix depends on the pattern of sparsity [Dahlquist §5.3]; specific cases, including special full matrices, are discussed in [Golub §4.1-§4.7], [Dahlquist §5.4], and [Press §2.4, §2.7, §2.8].

When the problem dimension is large, even for a full matrix, iterative methods may be useful because they avoid the accumulation of roundoff errors which can affect factorization. Further, if only an approximate solution to a problem is desired [Dennis

§1.2], and the time required to factor a large matrix is prohibitive [Press §2.0], a partial iterative solution may be sufficient.

Similar considerations apply for nonlinear problems, although the fact that the Jacobian changes at each iteration affects the particular decisions made. This is discussed more fully below.

LU factorization

Gaussian elimination solves a linear system Ax = b by successive subtractions of multiples of one row from those below it [Strang88 §1.3]. The multipliers are chosen to zero the coefficients of a selected variable in the other rows, i.e. to zero the matrix elements in a partial column below the current row. Eventually this produces an equivalent triangular system whose subdiagonal elements are all zero [Golub §3.2]. The resulting system can be solved trivially. For example,

(2.3.1a)
$$\begin{bmatrix} 3 & 2 \\ 6 & -7 \end{bmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}=\begin{pmatrix} -8 \\ -5 \end{pmatrix}$$

could be solved by subtracting twice the first row from the second (the elimination step):

(2.3.1b)
$$\begin{bmatrix} 3 & 2 \\ 0 & -11 \end{bmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}=\begin{pmatrix} -8 \\ 11 \end{pmatrix}$$

from which $x_2 = -1$, and then $x_1 = -2$, follow directly (the substitution step).

Alternatively, Equation 2.3.1a could be solved by subtracting 3/6 of the second row from the first, by subtracting -7/2 of the first row from the second, or by subtracting -2/7 of the second row from the first.

LU factorization formalizes this process by storing the multipliers at the same time as it performs the subtractions [Strang88 §1.5]. The multipliers prescribe how to reconstitute the original system of equations, by successive additions of rows of the reduced system to each other. Placing the multipliers in a lower triangular matrix L, with unity diagonal elements, these additions are performed by matrix multiplication with the reduced upper triangular system, U:

(2.3.2)
$$A = L \cdot U$$

Thus, the sample Jacobian of Equation 2.3.1a factors into

$$\begin{bmatrix} 3 & 2 \\ 6 & -7 \end{bmatrix}=\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 3 & 2 \\ 0 & -11 \end{bmatrix}$$

In practice, each multiplier is stored at the matrix element zeroed by the row subtraction; these zeros, and the ones on the diagonal of L, are implicit, and require no additional storage [Press §2.3].

The LU factorization of A does not depend on a particular right hand side, b. Therefore to solve Ax = b requires two substitution steps: (1) Solve Lb' = b for b'; and (2) Solve Ux = b' for x [Strang88 §1.5]. Storing the multipliers and adding a substitution step is wasteful when there is only one right hand side b, but for multiple problems involving the same matrix, LU factorization is more efficient than performing elimination on every system [Press §2.3].

PLU factorization

LU factorization breaks down when the current row to be subtracted-- called the pivot row-- has a zero in the current column. Then the multipliers are undefined since their denominator is zero. The solution is to select a new, nonzero, pivot element. Complete pivoting chooses the new pivot element from any column of any row which has not already been used as a pivot row. By contrast, partial pivoting restricts the pivot element selection to the current column of the remaining rows [Dahlquist §5.3.3]. In Equation 2.3.1a, complete pivoting considers each of the elements 3, 2, 6, and -7 as pivots; partial pivoting considers only the 3 and the 6.

If there is no suitable nonzero pivot element, then the original system had linearly dependent rows, and the matrix is singular [Strang88 §1.5]. This follows directly for complete pivoting, because at least one row has been zeroed by subtracting multiples of the rows above it. The result holds for partial pivoting as well. An all-zero pivot column means the remaining nonzero submatrix has one fewer columns than rows on which to perform elimination; the last row cannot be independent of those above it.

A pivoting strategy effectively exchanges rows (for partial pivoting), or rows and columns (for full pivoting), in the original matrix, as if the exchanges occurred before factorization. Mathematically, row exchanges are accomplished by a permutation matrix, P, which premultiplies A so that

$$P \cdot A = L \cdot U \tag{2.3.3}$$

[Strang88 §1.5]. In practice, the permutation matrix is stored as a vector of row exchanges rather than as a full matrix [Press §2.3]. Complete pivoting requires a postmultiplying permutation matrix as well, to effect the column exchanges [Strang88 §1.7]. However, complete pivoting is rarely used, since partial pivoting is regarded as adequate for numerical stability [Strang88 §1.7, Golub §3.4.6]. Further discussion assumes partial pivoting only.

Even if the next pivot element during normal LU factorization is not zero, row exchanges may be used to prevent roundoff errors due to division by a relatively small pivot element, which would lead to growth in the size of the elements in the lower rows [Golub §3.4, Dahlquist §5.5.4].

Simply choosing the pivot element with the largest magnitude controls the roundoff error [Strang88 §1.5]. Thus in Equation 2.3.1a, the rows would be swapped to pivot on the second row. However, under this strategy any row could be turned into the pivot row simply by scaling it by a large factor, even though the actual values subtracted during elimination would not change under such a scaling [Press §2.1]. Therefore an equilibration step is recommended before elimination, to scale each row to have the same maximum element magnitude [Golub §3.5.2, Dahlquist §5.5.5]. This scaling can be done implicitly, by storing the maximum magnitude in each row before elimination begins, and selecting pivot elements based on this scaling [Press §2.3]. Under this strategy, the rows of Equation 2.3.1a would not be swapped since the relative pivot 3/3 = 1 of the first row is greater than the relative pivot 6/7 of the second.

Numeric approaches to PLU factorization

In a few special cases, pivoting is unnecessary [Dahlquist §5.3.3]: (1) if A is diagonally dominant-- that is, if the magnitude of each diagonal element is greater than or equal to the sum of magnitudes of the other row elements; and (2) if A is symmetric and positive-definite (defined in Chapter 3). In the second case, the pivots are positive; in fact a

symmetric matrix is positive-definite if and only if it can be LU factored, without row exchanges, using positive pivots only [Dahlquist §5.4.1]. Furthermore, when a symmetric matrix can be LU factored without row exchanges, symmetry is preserved and $U = L^T$ [Strang88 §1.6]. This halves the computational requirements of the factorization [Dahlquist §5.4.1].

Algorithmically, LU decomposition need not follow Gaussian elimination exactly. In [Golub §3.2.6], classical Gaussian elimination is identified as an outer product formulation because it can be seen as updating a submatrix of A by subtracting the outer product of a vector of multipliers (in the pivot column) times a vector of multiplicands (in the pivot row). Inner product formulations, variations of which are called Crout and Doolittle [Golub §3.2.10], express the factorization as updates of individual elements by subtracting the inner product of a vector of previous multiples (to the left in the current row) times a vector of previously updated elements (above in the current column) [Press §2.3, Dahlquist §5.3.5].

Finally, [Golub §3.2.11] shows block LU strategies in which the matrix is decomposed by factoring successive submatrices in groups. The operation count for each of these algorithmic variations is the same; their relative merit depends on implementation details and the machine architecture [Golub §1.3, §1.4].

QR factorization

Where LU factorization expresses each successive row of A as the sum of multiples of the previous rows (with the multiples in L), plus a unique remainder (in U), QR factorization expresses each column of A as the linear sum of the orthogonal parts of the columns up to and including the current column. It results from a Gram-Schmidt orthogonalization of the columns of A [Srang88 §3.4].

To illustrate, let $A = [a_1\ a_2\ ..\ a_n]$. The orthogonalization starts by normalizing the first column to $q_1 = a_1/\|a_1\|$. For later use, note that $\|a_1\| = q_1^T a_1$, and therefore $a_1 = (q_1^T a_1)q_1$.

Next Gram-Schmidt breaks up the second column, a_2, into its component in the q_1 direction, plus an orthogonal component. The projection of a_2 onto q_1 is

$$\frac{q_1^T a_2}{q_1^T q_1} q_1$$

[Strang88 §3.2]. Since $\|q_1\| = 1$, the component of a_2 in the q_1 direction is $(q_1^T a_2)q_1$, and the remaining component, which must be at right angles to the projection, is

$$a_2 - (q_1^T a_2)q_1$$

Normalizing this orthogonal component of a_2 defines a new direction,

$$q_2 = \frac{a_2 - (q_1^T a_2)q_1}{\|a_2 - (q_1^T a_2)q_1\|}$$

which in turn can be used, along with q_1, to orthogonalize a_3. Proceeding through the n columns develops the orthogonal directions q_1, q_2, and so on, up to q_n.

The orthogonalization may fail. If A is singular, one of its columns a_j is the linear sum of the proceeding columns, and can be expressed as the sum of its projections onto the orthogonal parts of those columns [Strang88 §3.4]. Then the corresponding $q_j = 0$, since there is nothing left of a_j after subtracting away the orthogonal projections onto the previous q_i.

Suppose the orthogonalization finds n independent nonzero q_i, each normalized so that $q_i^T q_i = 1$. Constructed to lie at right angles to each other, $q_i^T q_j = 0$ for $j \neq i$. Defining $Q = [q_1\ q_2\ ..\ q_n]$, this matrix has the special property

(2.3.4a,b) $$Q^T Q = I \quad \text{or} \quad Q^{-1} = Q^T$$

[Strang88 §3.4]. Q is called orthogonal or orthonormal.

Given the lengths of the projections, the original matrix A can be reconstructed from Q. For example,

$$a_1 = (q_1^T a_1) q_1$$
$$a_2 = (q_1^T a_2) q_1 + (q_2^T a_2) q_2$$
$$a_j = \sum_{i=1}^{j} (q_i^T a_j) q_i$$

Of course the last term in each sum comes not from the projection $q_j^T a_j$, since q_j is not known at that stage of the construction of Q. Instead, the last term is the length of the remainder a_j' after subtracting the known projections:

(2.3.5a) $$a_j' = a_j - \sum_{i=1}^{j-1} (q_i^T a_j) q_i$$

(2.3.5b) $$q_j = \frac{a_j'}{\|a_j'\|}$$

Equation 2.3.5 describes the Gram-Schmidt orthogonalization. Collecting the lengths,

(2.3.6a) $$A = [a_1\ a_2\ ..\ a_n] = [q_1\ q_2\ ..\ q_n] \begin{bmatrix} q_1^T a_1 & q_1^T a_2 & \cdot & q_1^T a_n \\ & q_2^T a_2 & \cdot & q_2^T a_n \\ & & & \vdots \\ & & & q_n^T a_n \end{bmatrix}$$

or

(2.3.6b) $$A = Q \cdot R$$

[Strang88 §3.4].

A variety of QR factorization techniques provide alternate procedures to Gram-Schmidt orthogonalization, with more favorable stability properties. See [Golub §5.2, Press §2.10].

To solve the system Ax = b with A = QR, first solve Qy = b, using $y = Q^T b$ from Equation 2.3.4b; then solve Rx = y using the same substitutional approach with the upper triangular R that solved Ux = b' in PLU factorization.

PLU versus QR factorization

To choose between PLU and QR factorization, first compare their computational cost. PLU factorization requires $n^3/3$ multiplications and $n^3/3$ additions, versus $2n^3/3$ multiplications and $2n^3/3$ additions for QR factorization [Dennis §3.2], where n is the problem dimension.

Excepting the computational expense, several considerations favor QR over PLU factorization: (1) QR is unconditionally stable, and more robust when solving an ill-

conditioned system [Golub §5.7.1]; and (2) the QR factorization is more amenable to matrix updating techniques [Dennis §3.4, Press §2.10]. In practice, an equation-solving algorithm may not re-evaluate the Jacobian at each iteration, but may modify the old Jacobian using, for example, a secant update based on the results of the most recent step [Dahlquist §6.4, Dennis §8, Press §9.7]. In this case, a QR factored matrix may be modified directly, thus avoiding the cost of refactoring the new Jacobian.

CHAPTER 3
FUNCTION MINIMIZATION

This chapter presents methods and results for minimizing a general scalar cost function. For the most part it treats the subject independently from equilibrium problems, although occasionally noting parallels and distinctions between the two bodies of theory. Later chapters apply these results to solving systems of nonlinear equations.

The first three sections develop the underlying mathematical model-- the quadratic Taylor series approximation-- and the simplest line search algorithm-- the method of steepest descent-- which uses the model. The fourth section develops a geometric interpretation of the model. Taken together these sections form the basis for all the function minimization techniques applied in the thesis. The geometric picture culminates by demonstrating that the expected net direction of travel after two steepest descent minimizations on the quadratic model is itself a valid search direction.

Section 3.5 presents higher-order search methods, and serve as background for Section 3.6, which reviews the double dogleg method of function minimization, a standard algorithm on which many nonlinear equilibrium problem solvers are based.

3.1 Background

Chapter 2 suggests how Newton-Raphson's method for solving an equilibrium problem $r\{x\}$ might be supplemented by a simple divergence test based on a residual norm. It also indicates how, once admitted, the norm can actively guide the search for a new iterate. In both cases, the strategy seeks to reduce the residual norm with each iteration.

The three common residual norms

$$\|r\|_1 = |r_1| + |r_2| + .. + |r_n|$$
$$\|r\|_2 = (r_1^2 + r_2^2 + .. + r_n^2)^{1/2}$$
$$\|r\|_\infty = \max\{|r_1|, |r_2|, .. , |r_n|\}$$

all have a minimum value of zero at the solution to the equilibrium problem. The cost function need not be a residual norm, so long as it has a minimum at $r = 0$. More generally, a scalar cost function, $\Ø\{x\}$, defines a descent-based method when

$$\Ø\{x_{[k+1]}\} < \Ø\{x_{[k]}\} \tag{3.1.1}$$

For applications to equilibrium problems, $\Ø = \Ø\{r\}$ where $r = r\{x\}$.

Role of residual norm in algebraic systems

Suppose $\Ø\{x\}$ is a residual norm from an equilibrium problem. As defined, a descent-based method might generate a trial solution independently of $\Ø$-- for example by Gaussian elimination on the linearized equilibrium problem-- then accept the test point if it satisfies Equation 3.1.1, or reject it if it does not.

Since the solution to the linearized equilibrium problem at $x_{[k]}$ suggests the direction towards the solution, the norm may also serve as a selection mechanism. If the $x_{[k+1]:NR}$ from Equation 2.2.9 gives too large a norm, a line search along the ray connecting $x_{[k]}$ to $x_{[k+1]:NR}$ may find an acceptable value for $x_{[k+1]}$.

Beyond its roles as a test for accepting a point, and as a criterion for selecting the best from among a number of points, the residual norm may be used to generate trial points. Collecting information about its behavior, both from previous iterations and by direct analysis at $x_{[k]}$, an algorithm can develop search directions and trial points directly from the residual norm itself. To do so requires a general theory of function minimization.

This chapter considers methods for minimizing a function when there is no underlying equilibrium problem.

Role of cost function in function minimization

Suppose $\emptyset\{x\}$ is a function whose minimum is of interest in its own right. That is, the problem is to find the minimizing x^* such that

$$\emptyset\{x^*\} < \emptyset\{x \neq x^*\} \tag{3.1.2}$$

Again, $\emptyset$ is a scalar function of an n-dimensional vector x, only now there is no intervening vector function $r\{x\}$ for which $r\{x^{**}\} = 0$. Two immediate consequences for the minimization problem are: (1) the minimum value of $\emptyset$ is not known; and (2) trial solutions cannot be generated by solving a linearized residual problem. The first consequence affects the termination criteria-- no absolute test can determine whether a point really is x^*-- while the second affects the search itself. The cost function no longer can be used merely to accept an $x_{[k+1]}$, or to select an $x_{[k+1]}$ from among a range of choices; the trial solutions and the search directions must come from $\emptyset$ itself.

All the methods of function minimization considered below model the cost function by a truncated Taylor series, and develop trial solutions directly from the model. In the steepest descent, conjugate gradient, and Newton's methods, the trial solutions lie along a line; the double dogleg algorithm searches a trajectory defined by combining directions from the other methods.

Line search

Starting at a point $x_{[k]}$, a search direction d sweeps out a ray of points

$$x = x_{[k]} - \mu \cdot d \tag{3.1.3}$$

where the scalar stepsize μ determines the distance along the search direction. (Strictly the search is in the -d direction, since μ is presumed positive.) To minimize $\emptyset\{x\}$ by a succession of line searches, at each iteration the stepsize μ is chosen so that

$$x_{[k+1]} = x_{[k]} - \mu_{[k]} d_{[k]} \tag{3.1.4}$$

minimizes $\emptyset$ along the search direction $d_{[k]}$. Variations do not require strict minimization at each step, for example accepting the first trial step to satisfy some criteria such as $\emptyset < \emptyset_{[k]}$ [Dahlquist §10.5.2]. From the new location $x_{[k+1]}$, a new search direction is chosen and the line searches continue until a minimum is found.

Parallels with equation solving

The theory and methods of function minimization strongly parallel those of equation solving. As in Newton-Raphson iteration, the minimization algorithms: (1) model the function behavior at an iterate $x_{[k]}$ using local function value and derivative information; (2) use the model to find an $x_{[k+1]}$ with improved function values; and (3) seek the global solution through a succession of such local models.

Since the minimization methods begin with a scalar function $\emptyset\{x\}$ instead of a vector function $r\{x\}$, the first derivatives form a vector (the gradient, defined below) instead of a matrix (the Jacobian of Chapter 2). To a certain extent this gradient vector corresponds with the residual vector in algebraic problems-- both map the independent variables x onto a vector space of the same dimension, both are zero at the solution, and both have a matrix of derivatives whose inverse may be used to pick trial solutions. However the gradient elements, connected by a common "parent" function, $\emptyset$, have a symmetric matrix of derivatives [Strang86 §5.1]; this in general is not true for the Jacobian, since the residual elements in general are not derivatives of a common function. Furthermore, while the gradient is zero at the solution, a zero gradient does not guarantee the solution has been found; of course for equilibrium problems a zero residual vector defines the solution.

While it may be possible to solve an equilibrium problem by minimizing a residual norm, the gradient vector of a residual norm is not tied to the problem as closely as the residuals, nor the matrix of its derivatives as closely as the Jacobian. Therefore it is best not to forsake the underlying structure imposed by the residual functions themselves [Press §9.6]. Chapter 4 treats this observation further.

Unless a specific reference is made to equation solving, the methods discussed below apply strictly to function minimization.

3.2 Steepest Descent and the Quadratic Approximation

The simplest derivative-based line search follows the gradient direction, the direction of greatest local change $\partial\emptyset/\partial\mu$.

Descent directions

For any search direction it may be possible, by searching sufficiently far from $x_{[k]}$, to find a smaller $\emptyset$. When $\emptyset$ decreases locally, that is, for arbitrarily small μ in Equation 3.1.3, then d is called a descent direction [Dahlquist §10.5.2]. This requires

$$\frac{d\emptyset}{d\mu} < 0$$

or

$$\frac{\partial\emptyset}{\partial x_1}\frac{dx_1}{d\mu} + \frac{\partial\emptyset}{\partial x_2}\frac{dx_2}{d\mu} + .. + \frac{\partial\emptyset}{\partial x_n}\frac{dx_n}{d\mu} < 0$$

at the point $x_{[k]}$. (As in Chapter 2, the notation x_1 gives the first entry in vector x; its i^{th} entry at the k^{th} iteration is $x_{i[k]}$.) Since $x_{[k]}$ is fixed, $dx_1/d\mu = -d_1$, and so on for each component of d. Defining the gradient of $\emptyset$,

(3.2.1)
$$g \equiv \nabla\emptyset = \begin{pmatrix} \frac{\partial\emptyset}{\partial x_1} \\ \frac{\partial\emptyset}{\partial x_2} \\ \vdots \\ \frac{\partial\emptyset}{\partial x_n} \end{pmatrix}$$

the slope of Ø at any point along any search direction is

$$\frac{d\emptyset}{d\mu} = -g^T d \tag{3.2.2}$$

[Fletcher §1.2]. Note that the magnitude of $g^T d$ depends on the length of d. Since the relative power of μ to change x in the search direction also depends on this length (see Equation 3.1.3), Equation 3.2.2 is self-consistent. However to compare slopes in different directions, d should be normalized so that $\|dx\| = |d\mu|$ in each direction.

If d is a descent direction, then $d\emptyset/d\mu$ must be negative at $x_{[k]}$, so that

$$g_{[k]}^T d > 0 \tag{3.2.3}$$

for a descent direction. Since $g^T d = \|g\| \cdot \|d\| \cdot \cos\theta$ [Strang88 §3.2], Equation 3.2.3 states that for d to be a descent direction, the angle θ between $g_{[k]}$ and d must be less than 90 degrees. Geometrically this follows because a zonzero gradient of a function is perpendicular to its level curve [Ellis §13.6], so a search at right angles to $g_{[k]}$ does not change Ø in the neighborhood of $x_{[k]}$. Also, the gradient points in the direction of increasing Ø, so the search for a minimum must take the opposite direction, and $d_{[k]}$, which is the negative search direction, should be nearly colinear with the gradient.

Following the same reasoning, if Ø has been minimized along the previous search direction then a local change along $\pm d_{[k-1]}$ will not change Ø, so the gradient at $x_{[k]}$ is perpendicular to the former search direction. That is,

$$d_{[k-1]}^T g_{[k]} = 0 \tag{3.2.4}$$

when $x_{[k]}$ minimizes Ø along $d_{[k-1]}$.

Steepest descent method

For normalized $\|d\|$, the greatest initial decrease in Ø occurs when $d\emptyset/d\mu = -\|g_{[k]}\| \cdot \|d\| \cdot \cos\theta$ is most negative-- that is, when $\theta = 0$. Thus the method of steepest descent chooses

$$d_{[k]:SD} = g_{[k]} \tag{3.2.5}$$

The steepest descent direction is recommended when the current best position $x_{[k]}$ is far from a minimizing x* [Dahlquist §10.5.3]. However, convergence to x* may be slow or impossible. First, constraining $d_{[k]}$ to be perpendicular to $d_{[k-1]}$ means the search cannot respond to curvature in Ø; in the classic illustration of this problem, a line search, encountering an elliptical level curve at an angle to its major axis, never aligns to the axis, but crosses back and forth over it at each iteration. See Figure 3.2.1, below. Second, roundoff errors may prevent the method from ever reaching x* [Strang86 §5.1]. In particular this happens when the minimum along the steepest descent direction occurs so close to $x_{[k]}$, or has a function value so close to $\emptyset_{[k]}$, that the machine precision is insufficient to resolve $x_{[k+1]}$.

Curvature in the cost function is a second derivative effect. A second-order model of Ø, obtained by Taylor series expansion, characterizes the curvature. In addition it accounts for roundoff errors and establishes conditions on line minima and local minima.

Hessian matrix

The second derivatives of Ø are collected in the Hessian matrix, $G \equiv \nabla^2 Ø = \nabla g$. The component function derivatives enter the Hessian by columns (rather than by rows, as they do in the Jacobian):

(3.2.6a)
$$G = \nabla g = [\nabla g_1 \; \nabla g_2 \; .. \; \nabla g_n] = \begin{bmatrix} \frac{\partial}{\partial x_1} g^T \\ \frac{\partial}{\partial x_2} g^T \\ \vdots \\ \frac{\partial}{\partial x_n} g^T \end{bmatrix}$$

Where the gradient has components $g_j = \partial Ø / \partial x_j$, the Hessian has elements

(3.2.6b)
$$[G]_{i,j} = \frac{\partial g_j}{\partial x_i} = \frac{\partial^2 Ø}{\partial x_i \partial x_j}$$

[Strang88 §6.1]. If the second partial derivatives of Ø are continuous, then the order of differentiation is immaterial [Ellis §13.3] and G is symmetric. Therefore $G^T = G$ and the approximate relation $\Delta g \approx G^T \Delta x$ is used below as $\Delta g \approx G \Delta x$.

Since the gradient is perpendicular to the level curve of Ø at a point, and since the Hessian gives the change of g with x, the Hessian accounts for curvature in the contours of Ø. The curvature of $Ø\{x_{[k]} - \mu \cdot d\}$ at $x_{[k]}$ is

(3.2.7)
$$\frac{d^2 Ø}{d\mu^2} = \frac{d}{d\mu} \frac{dØ}{d\mu} = -d^T \nabla (-g_{[k]}{}^T d) = d^T G_{[k]} d$$

for any search direction d [Fletcher §1.2].

Second-order models (Taylor series expansion)

Using the Hessian, a Taylor series expansion of Ø about $x_{[k]}$ is written

$$Ø\{x_{[k]} + \Delta x\} = Ø_{[k]} + \Delta x^T g_{[k]} + \tfrac{1}{2} \Delta x^T G_{[k]} \Delta x + (\text{third-order terms})$$

[Strang88 §6.1]. In the limit as $\|\Delta x\|$ goes to zero, the higher-order terms also go to zero [Fletcher §1.2]. The resulting limiting form is used in two ways. The first substitutes $\Delta x = -\mu \cdot d$ from Equation 3.1.3 to give the quadratic approximation

(3.2.8)
$$Ø \approx Ø_{[k]} - \mu d^T g_{[k]} + \tfrac{1}{2} \mu^2 d^T G_{[k]} d$$

for small μ along any search direction d. A related but distinct use defines the quadratic function

(3.2.9a)
$$f_{[k]}\{x\} = Ø_{[k]} + \Delta x^T g_{[k]} + \tfrac{1}{2} \Delta x^T G_{[k]} \Delta x$$

where $\Delta x = x - x_{[k]}$.

Generally the quadratic approximation (Equation 3.2.8) is used to analyze convergence properties of a method when $x_{[k]}$ is near a minimum, while the quadratic function (Equation 3.2.9a) is formed as part of a minimization strategy which places a trial

iterate $\hat{x}_{[k+1]}$ at the minimum of $f_{[k]}$. In the latter case, it may be more convenient to treat Equation 3.2.9a directly in x rather than in Δx. Substituting $\Delta x = x - x_{[k]}$,

$$f_{[k]}\{x\} = \frac{1}{2} x^T G_{[k]} x - x^T (G_{[k]} x_{[k]} - g_{[k]}) + c_{[k]} \tag{3.2.9b}$$

Note that the constant term $c_{[k]} = \emptyset_{[k]} - x_{[k]}^T g_{[k]} + \frac{1}{2} x_{[k]}^T G_{[k]} x_{[k]}$ affects the function value at a minimum, but not its location.

The quadratic function has the same function value, gradient, and Hessian as has $\emptyset$ at $x_{[k]}$, and for small Δx, $f_{[k]} \approx \emptyset$. Using the notation of Chapter 2 (where for example $\hat{r}$ is the truncated Taylor series estimate of r), f would be written $\hat{\emptyset}$; the notation $f_{[k]}$ is adopted for convenience.

Line minima

Suppose $x_{[k]}$ minimizes $\emptyset$ along a previous search direction $d_{[k-1]}$. Then the quadratic approximation, formulated at $x_{[k]}$, must give $\emptyset > \emptyset_{[k]}$ for $d = d_{[k-1]}$ and μ both positive and negative. Thus from Equation 3.2.8

$$- \mu d_{[k-1]}^T g_{[k]} + \frac{1}{2} \mu^2 d_{[k-1]}^T G_{[k]} d_{[k-1]} > 0$$

for all μ. With μ small so that $\mu^2 \ll \mu$, this requires $d_{[k-1]}^T g_{[k]} = 0$, the orthogonality condition already described above as Equation 3.2.4. Then the remaining term must have

$$d_{[k-1]}^T G_{[k]} d_{[k-1]} > 0 \tag{3.2.10}$$

when $x_{[k]}$ minimizes $\emptyset$ along $d_{[k-1]}$.

Next the search continues in a new direction, $d_{[k]}$. Taking $d = d_{[k]}$ in the quadratic approximation, and setting $d\emptyset/d\mu = -d_{[k]}^T g_{[k]} + \mu d_{[k]}^T G_{[k]} d_{[k]} = 0$ to minimize $\emptyset$, an initial estimate for the optimizing stepsize is

$$\hat{\mu}_{[k]} = \frac{d_{[k]}^T g_{[k]}}{d_{[k]}^T G_{[k]} d_{[k]}} \tag{3.2.11a}$$

where the expected function value is

$$\hat{\emptyset}_{[k+1]} = f_{[k]}\{\hat{\mu}_{[k]}\} = \emptyset_{[k]} - \frac{1}{2} \frac{(d_{[k]}^T g_{[k]})^2}{d_{[k]}^T G_{[k]} d_{[k]}} \tag{3.2.11b}$$

after $\emptyset$ has been minimized along $d_{[k]}$.

Cauchy point

Applying these results to the steepest descent direction, the quadratic model estimates a minimizing stepsize

$$\hat{\mu}_{[k]:SD} = \frac{g_{[k]}^T g_{[k]}}{g_{[k]}^T G_{[k]} g_{[k]}}$$

The Cauchy point, where the quadratic model predicts the cost function has a minimum in its steepest descent direction [Dennis §6.4.2], lies at $x_{[k]} - \hat{\mu}_{[k]:SD} \cdot g_{[k]}$.

Roundoff error at line minimum

These line minimizations cannot be performed exactly in a finite-precision computing environment. If $x_{[k]}$ minimizes $\emptyset$ along some direction $d_{[k-1]}$, then $d_{[k-1]}^T g_{[k]} = 0$ and the Taylor series expansion along the former search direction gives

$$\emptyset \approx \emptyset_{[k]} + \frac{1}{2}\mu^2 d_{[k-1]}^T G_{[k]} d_{[k-1]}$$

at that point. Suppose ε is the smallest fraction of $|\emptyset_{[k]}|$ which can be represented by the machine. Then to insure $\emptyset$ evaluates as greater than $\emptyset_{[k]}$ requires

$$\frac{1}{2}\mu^2 d_{[k-1]}^T G_{[k]} d_{[k-1]} > \varepsilon \cdot |\emptyset_{[k]}| \tag{3.2.12}$$

Let $m_{[k]}$ be the spectral radius of $G_{[k]}$, that is, the maximum magnitude of its eigenvalues [Strang88 §7.4]. Then $d^T G_{[k]} d \le m_{[k]} d^T d = m_{[k]} \|d\|^2$ [Strang88 §5.2] and

$$m_{[k]} \cdot \mu^2 \cdot \|d_{[k-1]}\|^2 \ge \mu^2 d_{[k-1]}^T G_{[k]} d_{[k-1]} > 2\varepsilon \cdot |\emptyset_{[k]}|$$

is required to detect the step μ in the calculated function values. With $\Delta x = \mu \cdot d$, $\|\Delta x\|^2 = \mu^2 \|d\|^2$ and

$$\|\Delta x\| > \sqrt{\frac{2\varepsilon \cdot |\emptyset_{[k]}|}{m_{[k]}}} \tag{3.2.13}$$

is required to distinguish $\emptyset\{x_{[k]} + \Delta x\}$ from $\emptyset_{[k]}$ along $d_{[k-1]}$. A one-dimensional version of Equation 3.2.13 is given in [Press §10.1].

Equation 3.2.13 relates the minimum resolvable step to the largest eigenvalue of the Hessian matrix. In search directions dominated by the eigenvectors corresponding to smaller eigenvalues, larger steps are required, and the result does not consider roundoff errors at intermediate steps in the calculation of $\emptyset$.

Significantly, the precision obtained by the line search is limited not only by the eigenvalues, but also by the square root of the machine precision, ε.

Steepest descent convergence

The roundoff error implied by Equation 3.2.13 prevents exact minimization in a line search, potentially stopping the search short of (or beyond) the true minimum, within the bounds of $\|\Delta x\|$ [Press §10.1]. Furthermore, finite-precision effects can have the opposite effect, of identifying a minimum where the cost function is only very flat; this happens when small changes in $\emptyset\{x\}$ cannot be detected to machine precision [Dennis §1.3]. Finally, if noise in the function evaluations overwhelms meaningful changes in the cost function, the line search can falsely identify a minimum where none exists [Moré82 §1.1].

Nevertheless, within the limits of the computational environment the steepest descent method converges either to a local minimizer of $\emptyset$, or to a saddle point [Dennis §6.2] (a saddle point has no local curvature, and a zero gradient, even though it is a minimum only for some search directions).

(All these roundoff effects-- placing minimum bounds on the step length, and introducing spurious minima-- continue to hold when enforcing a descent requirement on an equilibrium problem.)

Local minima

Suppose a series of line minimizations leads to a unique local minimum at x^*. As above, if $\emptyset\{x^*\}$ is a minimum along direction d then $d^T g^* = 0$ and $d^T G^* d > 0$. However,

at x* this must hold not only along the last search direction, but for any d. Therefore at a unique local minimum x*,

(3.2.14) $$g^* = 0 \quad \text{and} \quad d^T G^* d > 0$$

for all $d \neq 0$ [Strang88 §6.1]. When $g\{x^*\} = 0$, x* is called a stationary point, and when $d^T G^* d > 0$ for any $d \neq 0$, G* is called positive-definite [Strang88 §6.2].

A zero gradient does not in itself guarantee a minimum-- for example the stationary point may be a maximum, in which case $d^T G d < 0$ for all nonzero d (and G is negative-definite); or a saddle point, in which case $d^T G d$ is positive along some search directions and negative along others (and G is indefinite). Nor does Equation 3.2.14 guarantee that x* gives the best minimum point of the cost function, since G* positive-definite means only that the quadratic approximation curves upward in a small region about x*. That is, x* may be a local but not a global minimum.

(When solving an equilibrium problem, the local minimum is global, and x* = x**, when $r\{x^*\} = 0$. When minimizing a general cost function, the value of the global minimum usually is not known, and the theory presented below makes no such claim.)

When G* is positive-semidefinite, that is, if $d^T G d \geq 0$ for all $d \neq 0$, then a stationary point x* may lie along a continuum of maxima or minima of equal value. It is a local minimum of Ø when G* is Lipschitz continuous [Dennis §4.3]. Thus G* positive-semidefinite is a necessary condition, and G* positive-definite is a sufficient condition, for optimality of any x* where g* = 0.

Example: quadratic problem

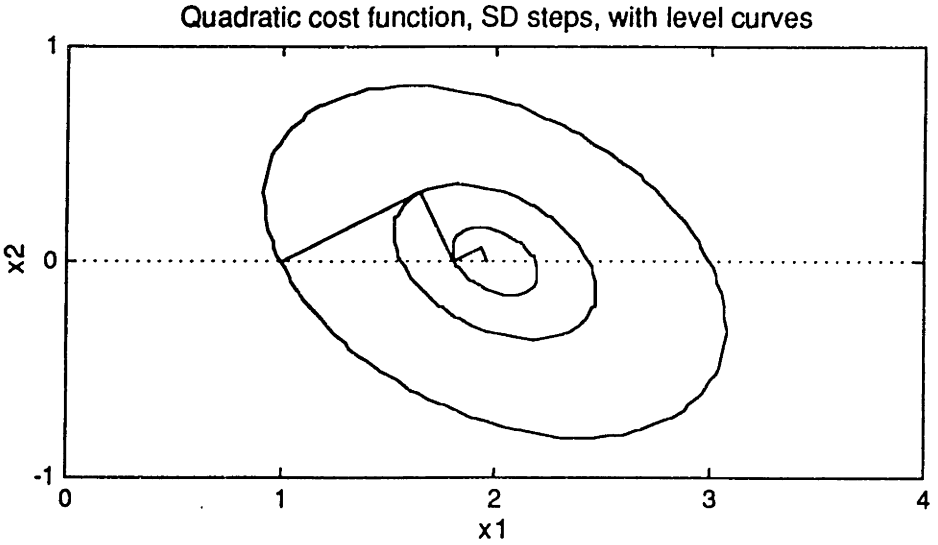

Figure 3.2.1. Level curves of Equation 3.2.15a. Each steepest descent step begins at right angles to the level curve (by Equation 3.2.5) and ends just tangent to another level curve (by Equation 3.2.4).

To illustrate the work above, let

(3.2.15a) $$Ø = \frac{6}{7}(x_1 - 2)^2 + \frac{1}{7}(x_1 - 2 + \frac{7}{2}x_2)^2$$

By inspection the minimum is $x^* = (2, 0)^T$. Figure 3.2.1 shows some level curves of the function, and the first few iterates of a steepest descent solution.

The gradient and Hessian are

$$g = \begin{pmatrix} 2x_1 + x_2 - 4 \\ x_1 + 3.5x_2 - 2 \end{pmatrix} \quad \text{and} \quad G = \begin{bmatrix} 2 & 1 \\ 1 & 3.5 \end{bmatrix} \tag{3.2.15b,c}$$

The desired stationary point can be found directly by setting $g\{x^*\} = 0$ and solving the trivial linear system $Gx^* = (4, 2)^T$. To motivate the solution by a descent method, suppose that the cost function is available only as a "black box" $\emptyset\{x\}$, from which the gradient is determined, when needed, numerically.

From an initial point $x_{[0]} = (1, 0)^T$, where $\emptyset_{[0]} = 1$, the first iteration of a steepest descent search is:

(1) Calculate $g_{[0]} = (-2, -1)^T$.

(2) Set $d_{[0]} = g_{[0]}$ according to Equation 3.2.5 for steepest descent.

(3) Search for $x_{[1]} = x_{[0]} - \mu_{[0]}d_{[0]}$ according to Equation 3.1.4, choosing $\mu_{[0]}$ to minimize $\emptyset\{x_{[1]}\}$. For this quadratic problem, Equation 3.2.11a is exact. It gives

$$\hat{\mu}_{[0]} = \frac{5}{15.5} = 0.3226$$

$$x_{[1]} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} - 0.3226 \begin{pmatrix} -2 \\ -1 \end{pmatrix} = \begin{pmatrix} 1.645 \\ 0.3226 \end{pmatrix}$$

$$\emptyset_{[1]} = \emptyset\{x_{[1]}\} = 0.1935$$

For a nonquadratic problem, $\hat{\mu}_{[0]}$ would serve as an initial guess, with $\mu_{[0]}$ to be determined by trial and error. In effect, selecting $\mu_{[0]}$ is a separate iterative task buried within the larger iteration from $x_{[0]}$ to $x_{[1]}$. Moreover, since the steepest descent method does not require the Hessian, it might be more expensive to calculate or estimate $G_{[0]}$ than would be justified by a (potentially poor) guess at $\mu_{[0]}$. In this case, even the initial guess at $\mu_{[0]}$ would be largely uninformed.

3.3 Positive-Definite Matrices

The symmetric positive-definite matrix is central to the study of descent methods of function minimization. The (symmetric) Hessian matrix must be positive-definite at a local minimum. Beyond this, the Hessian at a current iterate $x_{[k]}$ is assumed positive-definite, both in order to study a particular choice of descent direction, and to motivate the selection of others. This section develops some of the properties of positive-definite matrices.

Connection to quadratic cost function

Suppose $\emptyset$ is the quadratic function

$$f\{x\} = \frac{1}{2} x^T A x - x^T b + c \tag{3.3.1}$$

With A symmetric, the gradient and Hessian are

$$\nabla f = Ax - b \quad \text{and} \quad \nabla^2 f = A \tag{3.3.2a,b}$$

[Strang88 §6.1]. If A is not symmetric, direct application of the gradient and Hessian definitions gives

$$\nabla f = A_S x - b \quad \text{and} \quad \nabla^2 f = A_S \tag{3.3.2c,d}$$

where

(3.3.2e) $$A_S = \frac{1}{2}(A + A^T)$$

A_S is called the symmetric part of A; the remainder, $A - A_S = \frac{1}{2}(A - A^T)$, is its skew part [Golub §4.2.2]. According to Equation 3.2.14, A_S, not A, must be positive-definite in order for f{x} to have a local minimum. Indeed, some authors associate the property of positive-definiteness only with symmetric matrices [Strang88 §6.2], though it can be applied meaningfully to asymmetric matrices [Golub §4.2.2].

When A is not symmetric, Equation 3.3.1 can be written

$$f\{x\} = \frac{1}{2}x^T\left(\frac{1}{2}[A + A^T] - \frac{1}{2}[A^T - A]\right)x - x^Tb + c$$

$$f\{x\} = \frac{1}{2}x^TA_Sx - x^Tb + c - \frac{1}{4}\Delta x^T(A^T - A)\Delta x$$

The last term is a number whose transpose equals the negative of itself:

$$(\Delta x^T[A^T - A]\Delta x)^T = \Delta x^T(A - A^T)\Delta x = -\Delta x^T(A^T - A)\Delta x$$

and can only be zero. Therefore any quadratic of the form of Equation 3.3.1 can be rewritten

(3.3.3) $$f\{x\} = \frac{1}{2}x^TAx - x^Tb + c = \frac{1}{2}x^TA_Sx - x^Tb + c$$

without changing its function, gradient, or Hessian values. Henceforth it is assumed, without loss of generality, that A in Equation 3.3.1 is symmetric.

Connection to symmetric linear systems

If f{x} is to have a local minimum, it must have a positive-definite Hessian and a zero gradient at the minimizing point x*. Then $g^* = Ax^* - b = 0$, and the solution to the linear system Ax = b also is the minimum of the quadratic:

(3.3.4) $$x^* = A^{-1}b$$

where A is assumed symmetric. The inverse of A exists, and the algebraic problem can be solved, if and only if the columns of A are linearly independent [Strang88 §2.4]. This is also a condition for A to be positive-definite. Suppose the columns of a square symmetric matrix A are not linearly independent. Then by definition there exists a vector $x \neq 0$ such that Ax = 0 [Strang88 §2.3]. Hence $x^TAx = 0$ for some $x \neq 0$, and A is not positive-definite.

(3.3.5) A symmetric positive-definite matrix
has linearly independent columns.

Using the known solution $x^* = A^{-1}b$, f can be rewritten to bring out the positive-definite requirement on A [Strang86 §1.4]:

$$f\{x\} = \frac{1}{2}x^TAx - x^Tb + c = \frac{1}{2}(x - A^{-1}b)^TA(x - A^{-1}b) - \frac{1}{2}b^TA^{-1}b + c$$

which follows since $(A^{-1})^T = (A^T)^{-1}$ [Strang88 §1.6]. Substituting x*,

(3.3.6) $$f\{x\} = \frac{1}{2}(x - x^*)^TA(x - x^*) - \frac{1}{2}x^{*T}b + c$$

Clearly if x^* minimizes f, then $(x - x^*)^T A(x - x^*)$ is positive for $x \neq x^*$; that is, A must be positive-definite. Moreover this result gives the minimum function value as

$$f\{x^*\} = c - \frac{1}{2} b^T A^{-1} b \tag{3.3.7}$$

The fact that minimizing a quadratic cost function solves a linear equation does not mean any linear system $Ax = b$ can be solved by imbedding it directly into a quadratic function for minimization. Since A in general is not symmetric, minimizing the quadratic sets the gradient $A_S x^* - b = 0$, so that instead of solving $Ax = b$ the method solves

$$\frac{1}{2}(A + A^T)x = b$$

Furthermore the stationary point is not necessarily a minimum; it might also be a maximum or a saddle point.

The standard means of avoiding this problem is to premultiply $Ax = b$ by A^T [Strang86 §1.4], yielding a system $A^TAx = A^Tb$ which corresponds to a least-squares minimization of $r = b - Ax$. The matrix A^TA is at worst positive-semidefinite, since $xA^TAx = (Ax)^T(Ax) = \|Ax\|^2$ is never negative [Strang86 §1.4]. When the columns of A are independent, $Ax = 0$ only for $x = 0$, and A^TA is positive-definite:

(3.3.8) A^TA is positive-definite when the columns of A are independent; otherwise it is positive-semidefinite.

The converse, that any positive-definite matrix can be expressed as A^TA where A has independent columns, is proved below.

Eigenvectors of a symmetric matrix

The eigenvalues λ and eigenvectors s of a general n by n matrix, A, are defined by

$$As = \lambda s \tag{3.3.9a}$$

[Strang88 §5.1]. Rewriting as $(A - \lambda I)s = 0$, λ is an eigenvalue of A if $(A - \lambda I)$ has a null space, so that $(A - \lambda I)$ is singular, and its determinant is zero:

$$\det\{A - \lambda I\} = 0 \tag{3.3.9b}$$

This n-degree characteristic polynomial has n roots, the eigenvalues of A. Taking the eigenvalues and their corresponding eigenvectors together, Equation 3.3.9a may be written

$$AS = S\Lambda \tag{3.3.10a}$$

where

$$S = [s_1\ s_2\ ..\ s_n] \quad \text{and} \quad \Lambda = \mathrm{diag}\{\lambda_1, \lambda_2, .., \lambda_n\} \tag{3.3.10b,c}$$

[Strang88 §5.2]. If the eigenvectors are independent, S can be inverted and

$$A = S\Lambda S^{-1} \tag{3.3.11}$$

Equation 3.3.11 gives the diagonal form of A.

The eigenvalues collected in Λ may be positive or negative, real or complex, and may occur in any multiplicity. The implications for the eigenvectors are considered for four particular cases: (1) zero eigenvalues; (2) complex eigenvalues; (3) distinct eigenvalues; and (4) repeated eigenvalues.

Zero eigenvalues

When one of the $\lambda_i = 0$, $\det\{A\} = 0$ by Equation 3.3.9b. Therefore A already is singular and cannot be inverted [Strang88 §4.2].

(3.3.12) A matrix is singular if and only if it has a zero eigenvalue.

However, singularity does not prevent diagonalization as in Equation 3.3.11 [Strang88 §5.2]. When a $\lambda_i = 0$, Λ is singular: $[\Lambda^{-1}]_{i,i} = 1/[\Lambda]_{i,i} = 1/\lambda_i$ does not exist. Therefore A cannot be inverted, even if it can be diagonalized, because the inverse of its diagonal form does not exist.

Complex eigenvalues

Even for a real matrix, the roots of the characteristic polynomial may be complex (the complex roots occur in conjugate pairs) [Dahlquist §6.8.1]. This leads to complex entries in the corresponding eigenvectors. For a complex vector x, the positive-definite test x^TAx does not necessarily give a real number; neither does x^Tx, which for x real is the square of its Euclidean length. The length is given physical meaning by using $\bar{x}$, the complex conjugate of x [Strang88 §5.5]: $\bar{x}^Tx$ is real since each component x_i of x multiplies its complex conjugate $\bar{x}_i$, adding a real number-- the square of the length of x_i-- to the sum.

Using the complex conjugate, $As = \lambda s$ becomes $\bar{s}^TAs = \lambda\bar{s}^Ts$, with the eigenvalue multiplied by a real number on the right. Again the product $\bar{x}^TAx$ is not necessarily real. However, if $A = A^T$ then $(\bar{x}^TAx)^T = x^TA\bar{x}$, making $\bar{x}^TAx$ the complex conjugate of itself, and therefore real. Then the eigenvalue must be real:

(3.3.13) A real symmetric matrix has
real eigenvalues and real eigenvectors.

[Strang88 §5.5].

Henceforward it is assumed that the λ_i are real and are ordered $\lambda_1 \le \lambda_2 \le .. \le \lambda_n$. Even with this choice, the matrix S in Equation 3.3.9 is not unique, since any scalar multiple of an eigenvector is itself an eigenvector.

Distinct eigenvalues

If all n eigenvalues of A are distinct, diagonalization cannot fail. Suppose A has j distinct eigenvalues, and that a linear combination of their corresponding eigenvectors is zero: $c_1s_1 + c_2s_2 + .. + c_js_j = 0$. Premultiplying by $(A - \lambda_jI)$,

$$c_1(\lambda_1 - \lambda_j)s_1 + c_2(\lambda_2 - \lambda_j)s_2 + .. + c_{j-1}(\lambda_{j-1} - \lambda_j)s_{j-1} = 0$$

Since the λ_i are different, by induction the c_i must be zero in order for the linear combination of the eigenvectors to be zero. Therefore for any matrix,

(3.3.14) The eigenvectors associated with distinct
eigenvalues are linearly independent.

[Strang88 §5.2].

For real symmetric matrices, these eigenvectors are not merely independent, they are orthogonal. For distinct real eigenvalues $\lambda_1 \neq \lambda_2$,

$$\lambda_1 s_1^T s_2 = (As_1)^T s_2 = s_1^T (As_2) = s_1^T (\lambda_2 s_2)$$
$$\lambda_1 s_1^T s_2 = \lambda_2 s_1^T s_2$$

from which $s_1^T s_2 = 0$ [Strang86 §1.5]. Therefore the distinct eigenvalues of a real symmetric matrix produce orthogonal eigenvectors [Strang88 §5.5].

Repeated eigenvalues

When an eigenvalue repeats-- that is, when the characteristic equation of a general matrix A has a root of multiplicity greater than one-- it is possible that there is only one associated independent eigenvector. In this case the matrix S of eigenvectors has linearly dependent columns and is not invertible. With repeated eigenvalues, the diagonalization of Equation 3.3.11 may fail [Strang88 §5.2]. (It need not fail. For example, all n eigenvalues of the n by n identity matrix are equal to one, yet I is not singular, and already is diagonalized.)

For real symmetric matrices, a perturbation argument shows that the eigenvectors remain orthogonal even with repeated eigenvalues [Strang86 §1.5]. A slightly perturbed matrix $A + \varepsilon B$ with distinct eigenvalues has orthogonal eigenvectors. Placing these into S as above, $(A + \varepsilon B)S = S\Lambda$. Since S is nonsingular, $S^{-1}AS + \varepsilon S^{-1}BS = \Lambda$. It can be shown [Dahlquist §5.8] that the eigenvalues of the perturbed matrix are within an order of ε of the eigenvalues of A. Therefore in the limit as ε tends to zero, the eigenvalues approach each other while the eigenvectors remain orthogonal.

(3.3.15) A real symmetric matrix has a complete set of orthogonal eigenvectors.

[Strang88 §5.6]. This holds regardless of repeated eigenvalues or zero eigenvalues.

Orthonormal diagonal form

Suppose the n orthogonal eigenvectors of a real symmetric matrix are normalized to length 1. Rename the collected matrix from S to Q. Then $q_i^T q_i = 1$ since the eigenvectors have unit length, and $q_i^T q_j = 0$ for $i \neq j$, since the eigenvectors are orthogonal. Therefore $Q^T Q = I$ or

$$Q^{-1} = Q^T \tag{3.3.16}$$

Q is called orthonormal or orthogonal [Strang88 §3.4]. Note that while this orthogonal Q shares the same properties with the Q of QR factorization in Chapter 2, its columns are the eigenvectors of a real symmetric matrix, and are unique; the Q of Equation 2.3.4 is constructed by projections during matrix factorization, and its columns depend on the order of orthogonalization of the columns of J. Substituting in Equation 3.3.11, the diagonal form is

$$A = Q\Lambda Q^T \tag{3.3.17}$$

for any real symmetric A. This result, called the principal axis theorem or the spectral theorem [Strang88 §5.5], establishes conditions on the eigenvalues of a symmetric positive-definite matrix.

Eigenvalues of a positive-definite matrix

By definition, a symmetric matrix A is positive-definite if $x^T Ax > 0$ for all $x \neq 0$. Suppose x is an eigenvector q_i of A. Then $q_i^T Aq_i = \lambda_i$ and the corresponding eigenvalue

must be positive for A to be positive-definite ($\lambda_i \geq 0$ is required for A to be positive-semidefinite). Now suppose x is not an eigenvector. Since A has a full set of orthogonal eigenvectors, x may be expressed as a linear combination $x = c_1q_1 + c_2q_2 + .. + c_nq_n$. Each c_i is the length of the projection of x onto q_i, or $c_i = x^Tq_i/q_i^Tq_i = x^Tq_i$ [Strang88 §3.2]. Collecting the c_i in a vector $c = (c_1, c_2, .. , c_n)^T$, any x may be written $x = Qc$. Then

$$x^TAx = (Qc)^TA(Qc) = c^TQ^TQ\Lambda Q^TQc = c^T\Lambda c$$

$$x^TAx = \sum_{i=1}^{n} c_i^2\lambda_i \tag{3.3.18}$$

If the λ_i are positive, x^TAx must be positive:

(3.3.19) A symmetric matrix is positive-definite if and only if all its eigenvalues are positive.

[Strang88 §6.2]. If the eigenvalues are nonnegative, A is positive-semidefinite; by Equation 3.3.12, a positive-semidefinite matrix, having a zero eigenvalue, is singular.

Since the eigenvalues of a positive-definite matrix are positive, Λ can be split into the product of two symmetric matrices of the square roots of the λ_i. Then $A = Q\Lambda Q^T = (Q\sqrt{\Lambda})(Q\sqrt{\Lambda})^T = R^TR$ where $R = \sqrt{\Lambda}\cdot Q^T$. Thus a positive-definite matrix can be expressed R^TR. Equation 3.3.8 states that R^TR is positive-definite when the columns of R are independent. Hence

(3.3.20) A symmetric matrix is positive-definite if and only if it can be expressed as R^TR where R has independent columns.

[Strang88 §6.2].

Diagonally-dominant symmetric matrix

Suppose a symmetric matrix A is strictly diagonally dominant,

$$|a_{ii}| > \sum_{j=1,\, j\neq i}^{n} |a_{ij}| \quad \text{for} \quad i = 1..n$$

[Golub §3.4.10]. From the discussion of LU factorization, pivoting is unnecessary and $A = LL^T$. Then since A is symmetric, it can be expressed $A = R^TR$ where $R = L$, and by Equation 3.3.20 it is positive-definite provided L has independent columns (from the same discussion, this will be true if and only if the pivots are positive).

A simpler test, which does not rely on LU factorization, supposes further that the diagonal elements are A are positive. The definition $As = \lambda s$ requires for any row i of A that $\sum a_{ij}s_j = \lambda s_i$ where for the argument s_i represents the i^{th} row of s, not the i^{th} eigenvector. Rewriting,

$$\lambda = a_{ii} + \sum_{j=1,\, j\neq i}^{n} a_{ij}\frac{s_j}{s_i}$$

In particular, this holds for the row i for which $|s_i| \geq |s_{j\neq i}|$. Then the most negative value possible for λ is

$$\lambda = a_{ii} - \sum_{j=1,\, j\neq i}^{n} |a_{ij}|$$

which is still positive under the assumptions above.

(3.3.21) A symmetric, strictly diagonally dominant matrix with positive diagonal elements is positive-definite.

[Golub §4.2.1].

3.4 Geometric Interpretation of the Hessian

The principal axis theorem, Equation 3.3.17, decomposes a real symmetric matrix into the weighted sum of the outer products of its eigenvectors, $A = \lambda_1 q_1 q_1^T + \lambda_2 q_2 q_2^T + .. + \lambda_n q_n q_n^T$ [Strang88 §5.5]. The orthogonal eigenvectors represent an alternate coordinate system, rotated from the original coordinate axes, along which the ellipsoidal level curves of a quadratic term x^TAx are aligned.

Axes of an ellipsoid

The pure quadratic $x^TAx = x^TQ\Lambda Q^Tx$ by the principal axis theorem. Each term of the vector Q^Tx is q_i^Tx, the length of the projection of x onto q_i, and gives the coordinate of x along the new axis, q_i.

Expanding,

$$x^TAx = \lambda_1(Q^Tx)_1^2 + \lambda_2(Q^Tx)_2^2 + .. + \lambda_n(Q^Tx)_n^2$$

When the λ_i are positive (when A is positive-definite), the quadratic has a minimum at $Q^Tx^* = 0$. Since the columns of Q are independent this requires $x^* = 0$.

Now consider the level curve $x^TAx = 1$. From the expansion above, when $(Q^Tx)_1 = \pm 1/\sqrt{\lambda_1}$, the other components must be zero for the point to lie on the curve. Thus $x = \pm q_1 \Lambda R(\lambda_1)$ lies on $x^TAx = 1$. Since q_1 is normalized to length one, the point is a distance $1/\sqrt{\lambda_1}$ from the center of the level curve.

The same holds in any of the eigenvector directions; the point $x = \pm q_i/\sqrt{\lambda_i}$ lies on the level curve $x^TAx = 1$ at a distance $1/\sqrt{\lambda_i}$ from its center. Thus the level curve is an ellipsoid whose orthogonal axes are the eigenvectors of A [Strang88 §6.2]. Its longest axis corresponds to the smallest eigenvalue, λ_1. Similarly, the shortest axis of the ellipsoid is the eigenvector q_n corresponding to the largest eigenvalue λ_n.

To associate a minimum principle with this ellipsoidal level curve, note that on $x^TAx = 1$ the largest value of x^Tx occurs in the direction of q_1. Therefore the Rayleigh quotient

(3.4.1) $R\{x\} = \frac{x^TAx}{x^Tx}$ is minimized by an eigenvector corresponding to the smallest eigenvalue λ_1 of a positive-definite A, and its value there is λ_1.

[Strang88 §6.4]. The function value follows regardless of the scaling of the eigenvector since for any $s_1 = |s_1|q_1$, the Rayleigh quotient $R\{s_1\} = s_1^TAs_1/s_1^Ts_1 = \lambda_1 s_1^Ts_1/s_1^Ts_1$. A maximum principle, similarly derived, states that $R\{x\}$ finds its largest value, λ_n, for the choice $x = q_n$ (or some multiple of q_n).

Geometric interpretation of a quadratic cost function

To extend these results from the pure quadratic to the quadratic cost function of Equation 3.3.1, shift the center of reference from the origin, where x^TAx is zero, to $x^* = A^{-1}b$, the minimizing point of $f\{x\}$. From Equations 3.3.6 and 3.3.7,

$$f\{x\} = \tfrac{1}{2} x^T A x - x^T b + c = \tfrac{1}{2} (x - x^*)^T A (x - x^*) - f^*$$

where $f^* = f\{x^*\}$ is the minimum value (assuming A is positive-definite).

The axes of the ellipsoidal level curves intersect at x^*. The relative position with respect to x^* may be expressed in terms of the normalized eigenvectors as

(3.4.2a,b) $$x = x^* + Qy \quad \text{or} \quad y = Q^T(x - x^*)$$

where y acts as an independent variable in the shifted, rotated coordinate system. Just as $Q^T x$ projects x onto the eigenvectors of A, so $Q^T(x - x^*)$ projects the position relative to x^* onto the eigenvectors. Therefore y expresses x in a coordinate system whose origin is at x^* and whose axes point along the axes of the ellipsoidal level curves of f.

With $A = Q\Lambda Q^T$, the cost function in the new coordinate system is

(3.4.3) $$f\{x^* + Qy\} = \tfrac{1}{2} y^T \Lambda y + f^*$$

In principle the results for $x^T A x = 1$ hold for $f\{y\}$, though the scale factor 1/2 and the possibility of a nonzero minimum f^* move the intersections between the level curves and the orthogonal axes. In fact the level curve $f\{x\} = f_{lc}$ intersects the ith eigenvector at the points

(3.4.4) $$y_i = \pm \sqrt{\frac{2(f_{lc} - f^*)}{\lambda_i}} \quad \text{or} \quad x = x^* \pm q_i \sqrt{\frac{2(f_{lc} - f^*)}{\lambda_i}}$$

Example: principal axis decomposition of quadratic problem

Returning to the sample quadratic of Equation 3.2.15a, the Hessian, with its eigenvalues 1.5 and 4, is diagonalized by

$$\begin{bmatrix} 2 & 1 \\ 1 & 3.5 \end{bmatrix} = \begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix} \begin{bmatrix} 1.5 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} \\ 1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}$$

Since its eigenvalues are positive, G is positive-definite. The level curves of Ø are ellipses, centered at $x^* = (2, 0)^T$, oriented with the major axis in the $(2, -1)^T$ direction as given by q_1, and with the minor axis along $(1, 2)^T$. Note the axes are orthogonal. Figure 3.2.1 demonstrates these relationships.

Steepest descent step

Consider an iterative line search for the minimum of a quadratic cost function

(3.4.5a) $$f_{[k]} = \tfrac{1}{2} x_{[k]}^T A x_{[k]} - x_{[k]}^T b + c = \tfrac{1}{2} y_{[k]}^T \Lambda y_{[k]} + f^*$$

A general cost function Ø, expanded by Taylor series as in Equation 3.2.9b, gives a quadratic of this form. Sufficiently close to a minimum, the Hessian is positive-definite and the diagonal elements of Λ in $G = Q\Lambda Q^T$ are positive. Collecting results from above, the minimum is

(3.4.5b,c) $$x^* = A^{-1} b \quad \text{where} \quad f^* = f\{x^*\} = c - \tfrac{1}{2} b^T A^{-1} b$$

and the relative position in terms of the eigenvectors is

(3.4.5d) $$y_{[k]} = Q^T(x - x^*) \quad \text{or} \quad x_{[k]} = x^* + Q y_{[k]}$$

Finally the gradient and Hessian are

(3.4.5e) $$g_{[k]} = Ax_{[k]} - b = Q\Lambda y_{[k]}$$

(3.4.5f) $$G = A = Q\Lambda Q^T$$

Equation 3.4.5 defines f and its derivatives at a current iterate. A line search with the steepest descent choice $d_{[k]} = g_{[k]}$ gives:

$$x_{[k+1]:SD} = x_{[k]} - \mu_{[k]}A(x_{[k]} - A^{-1}b)$$
$$x_{[k+1]:SD} - x^* = x_{[k]} - x^* - \mu_{[k]}A(x_{[k]} - x^*)$$
$$Q^T(x_{[k+1]:SD} - x^*) = Q^T(I - \mu_{[k]}A)(Qy_{[k]})$$

(3.4.6a) $$y_{[k+1]:SD} = (I - \mu_{[k]}\Lambda)y_{[k]}$$

where the last step follows since $Q^TAQ = \Lambda$. For a quadratic cost function, Equation 3.2.11a gives $\mu_{[k]}$ exactly as

$$\mu_{[k]:SD} = \frac{g_{[k]}^T g_{[k]}}{g_{[k]}^T A g_{[k]}} = \frac{(y_{[k]}^T \Lambda Q^T)(Q\Lambda y_{[k]})}{(y_{[k]}^T \Lambda Q^T)(Q\Lambda Q^T)(Q\Lambda y_{[k]})}$$

(3.4.6b) $$\mu_{[k]:SD} = \frac{y_{[k]}^T \Lambda^2 y_{[k]}}{y_{[k]}^T \Lambda^3 y_{[k]}} = \frac{(\Lambda y_{[k]})^T(\Lambda y_{[k]})}{(\Lambda y_{[k]})^T \Lambda (\Lambda y_{[k]})}$$

The second form of Equation 3.4.6b gives the stepsize $\mu_{[k]}$ as the inverse of the Rayleigh quotient $R\{\Lambda y_{[k]}\}$ for the matrix Λ. According to Equation 3.4.1, if Λ is positive-definite the minimum of the Rayleigh quotient is the smallest eigenvalue of Λ, and its maximum is the largest eigenvalue of Λ. Choosing $Q = I$ in Equation 3.3.17 shows that Λ is a symmetric positive-definite matrix whose eigenvalues are just the λ_i from A. Therefore

(3.4.7) $$\frac{1}{\lambda_n} \le \mu_{[k]:SD} \le \frac{1}{\lambda_1}$$

From the geometric interpretation of the cost function, the maximum stepsize comes from the longest axis of the ellipse-- that is, when $x_{[k]} - x^*$ lies along the eigenvector q_1 corresponding to the smallest eigenvalue. For a diagonal matrix such as Λ the eigenvectors are just the unit vectors in each of the coordinate directions. For convenience these are denoted e_i; for example $e_1 = (1\ 0\ ..\ 0)^T$ is an eigenvector corresponding to the smallest eigenvalue of Λ, λ_1. Returning to the steepest descent search, the maximum stepsize occurs when $\Lambda y_{[k]} = ce_1$ or $y_{[k]} = ce_1/\lambda_1$, where c is an arbitrary constant. Then $x_{[k]} = x^* + Qy_{[k]} = x^* + cq_1/\lambda_1$ and the maximum value of $\mu_{[k]}$ in a steepest descent search, $1/\lambda_1$, occurs when $x_{[k]}$ lies in the $\pm q_1$ direction from x^*. Similarly $\mu_{[k]}$ attains its smallest value, $1/\lambda_n$, when $x_{[k]} - x^*$ is a scalar multiple of q_n.

More generally, when $x_{[k]}$ lies in an eigenvector direction from x^*, so that $y_{[k]} = ce_i$, the gradient $g_{[k]} = Q\Lambda y_{[k]} = c\lambda_i q_i$ points along the axis as well. Then choosing $\mu_{[k]} = 1/\lambda_i$ sets $y_{[k+1]} = 0$; see Equation 3.4.6a. Barring roundoff and truncation effects, a steepest descent search will find x^* exactly at $x_{[k+1]}$.

On the other hand, when $x_{[k]}$ is removed from x^* by some combination of the eigenvectors, $y_{[k]}$ has more than one nonzero entry. Then Equation 3.4.6a defines one step of an iteration towards x^*. The effectiveness of this single step can be evaluated either by finding the decrease in the cost function, or by considering changes to the individual components of $y_{[k]}$.

Steepest descent improvement in cost function

From Equations 3.4.5a and 3.4.6a, and with $(I - \mu_{[k]}\Lambda)$ symmetric, the relative improvement of the error in the function minimum is

$$\frac{f_{[k+1]} - f^*}{f_{[k]} - f^*} = \frac{y_{[k+1]}^T \Lambda y_{[k+1]}}{y_{[k]}^T \Lambda y_{[k]}} = \frac{(\sqrt{\Lambda} y_{[k]})^T (I - \mu_{[k]}\Lambda)^2 (\sqrt{\Lambda} y_{[k]})}{(\sqrt{\Lambda} y_{[k]})^T (\sqrt{\Lambda} y_{[k]})} \tag{3.4.8}$$

in the steepest descent direction. The ratio may be seen as a Rayleigh quotient $R\{\sqrt{\Lambda} y_{[k]}\}$ for the matrix $(I - \mu_{[k]}\Lambda)^2$, a diagonal matrix with elements $(1 - \mu_{[k]}\lambda_i)^2$ in the [i,i] position and zeros elsewhere. This is a positive-definite matrix with its eigenvalues on the diagonal. Therefore from Equation 3.4.1 the ratio is bounded by

$$\frac{f_{[k+1]} - f^*}{f_{[k]} - f^*} \leq \max\{(1 - \mu_{[k]}\lambda_1)^2, (1 - \mu_{[k]}\lambda_2)^2, .. , (1 - \mu_{[k]}\lambda_n)^2\} \tag{3.4.9}$$

The eigenvectors of the diagonal matrix $(I - \mu_{[k]}\Lambda)^2$ are the same as those of Λ: the unit vectors e_i in the new coordinate directions. By the Rayleigh analysis, one of these eigenvectors-- the one corresponding to the maximum eigenvalue $(1 - \mu_{[k]}\lambda_i)^2$-- should maximize the ratio of function values in Equation 3.4.8. However, starting from along any of the eigenvectors makes $y_{[k+1]} = 0$, and $f_{[k+1]} = f^*$, in a steepest descent search. The vector which is supposed to maximize the Rayleigh quotient is exactly a vector which makes it zero.

The Rayleigh quotient analogy breaks down since the results for a constant matrix A in x^TAx/x^Tx do not hold when $A = A\{x\}$. This is the case for $(I - \mu_{[k]}\Lambda)^2$ since $\mu_{[k]} = \mu\{y_{[k]}\}$ according to Equation 3.4.6b. Nonetheless the analogy does guarantee an upper bound on the ratio of function values, since for a given choice of $\mu_{[k]}$ the ratio may not exceed the maximum eigenvalue, even if the corresponding eigenvector produces an unexpected result for special choices of $\mu_{[k]}$.

Suppose the $\lambda_1 .. \lambda_n$ are known. From Equation 3.4.7 the smallest $\mu_{[k]}$ is $1/\lambda_n$, where the eigenvalues of $(I - \mu_{[k]}\Lambda)^2$ range from a maximum of $(1 - \lambda_1/\lambda_n)^2$ to a minimum of zero. As $\mu_{[k]}$ increases towards $1/\lambda_1$, the bound, set by the first eigenvalue, decreases. Meanwhile the n^{th} eigenvalue, $(1 - \mu_{[k]}\lambda_n)^2$, increases. Thus the maximum eigenvalue of $(I - \mu_{[k]}\Lambda)^2$ decreases until

$$\begin{aligned} (1 - \mu_{[k]}\lambda_1)^2 &= (1 - \mu_{[k]}\lambda_n)^2 \\ 1 - \mu_{[k]}\lambda_1 &= \mu_{[k]}\lambda_n - 1 \\ \mu_{[k]} &= \frac{2}{\lambda_1 + \lambda_n} \end{aligned}$$

After this point the bound increases again. Therefore if $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$, Equation 3.4.9 guarantees

$$\frac{f_{[k+1]} - f^*}{f_{[k]} - f^*} \leq \left(\frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1}\right)^2 \tag{3.4.10}$$

in a steepest descent search.

When its eigenvalues are substantially the same, the Hessian defines nearly spheroidal level curves, and a single steepest descent method converges quickly. Eigenvalues with greatly different magnitudes generate elongated ellipsoids, slowing convergence (see Figure 3.2.1).

From Equation 3.4.6b, $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$ when $\Lambda y_{[k]} = (c, 0, .. , 0, c)^T$ or $y_{[k]} = (c/\lambda_1, 0, .. , 0, c/\lambda_n)^T$ where c is an arbitrary constant. Substituting in Equation 3.4.8 gives

Equation 3.4.10 as an equality. Following the Rayleigh analysis, equality is expected only for the appropriate eigenvector of $(I - \mu_{[k]}\Lambda)^2$, which by the discussion above is expected to yield $f_{[k+1]} - f^* = 0$. However, the special choice $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$ creates a repeated eigenvalue $(\lambda_n - \lambda_1)^2/(\lambda_n + \lambda_1)^2$ in the [1,1] and [n,n] positions of $(I - \mu_{[k]}\Lambda)^2$. With this repeated eigenvalue, any linear combination of the eigenvectors e_1 and e_n also is an eigenvector of $(I - \mu_{[k]}\Lambda)^2$.

Equation 3.4.10 was derived using the choice $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$. In a steepest descent search constrained to use a constant stepsize, this would be the ideal choice. Normally the eigenvalues are unknown, and if the quadratic is a local approximation to a nonlinear cost function, the effective eigenvalues change at each iteration. In any event, $\mu_{[k]}$ is not constant but is determined by a line search along $g_{[k]}$-- or by Equation 3.4.6b in the quadratic case-- precisely to minimize the ratio of Equation 3.4.10. Therefore the bound established by that equation for $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$ holds for the $\mu_{[k]}$ chosen by a steepest descent search.

Steepest descent improvement in estimate

Similar arguments may be made for the relative improvement of the distance from the minimizing point x*:

$$\frac{\|x_{[k+1]} - x^*\|^2}{\|x_{[k]} - x^*\|^2} = \frac{y_{[k+1]}^T y_{[k+1]}}{y_{[k]}^T y_{[k]}} = \frac{y_{[k]}^T (I - \mu_{[k]}\Lambda)^2 y_{[k]}}{y_{[k]}^T y_{[k]}}$$

Again the choice $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$ minimizes the maximum eigenvalue of $(I - \mu_{[k]}\Lambda)^2$ and therefore

$$\frac{\|x_{[k+1]} - x^*\|}{\|x_{[k]} - x^*\|} \le \frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1} \quad \text{when} \quad \mu_{[k]} = \frac{2}{\lambda_1 + \lambda_n} \tag{3.4.11}$$

[Dennis §6.2]. However, the $\mu_{[k]}$ which minimizes $f_{[k+1]}$ may not satisfy the condition on Equation 3.4.11. For example, the first steepest descent step shown in Figure 3.2.1 has $\|x_{[1]} - x^*\|/\|x_{[0]} - x^*\| = 0.480$ which is greater than $(\lambda_n - \lambda_1)/(\lambda_n + \lambda_1) = 0.455$. Note however that $(f_{[1]} - f^*)/(f_{[0]} - f^*) = 0.194$ is less than the upper bound $(\lambda_n - \lambda_1)^2/(\lambda_n + \lambda_1)^2 = 0.207$ established by Equation 3.4.10.

Steepest descent effect on components of estimate

A steepest descent step on a quadratic cost function, with its stepsize chosen to minimize the function value at $x_{[k+1]}$, may not satisfy the bound of Equation 3.4.11, which holds only for the special case $\mu_{[k]} = 2/(\lambda_1 + \lambda_n)$. However, for any stepsize, some bounds can be placed on the components of y.

Equation 3.4.6a gives $y_{[k+1]} = (I - \mu_{[k]}\Lambda)y_{[k]}$ for the steepest descent step. For a single component of y this means

$$y_{i[k+1]:SD} = (1 - \mu_{[k]}\lambda_i)y_{i[k]} \tag{3.4.12}$$

Recall that the components of y give the distance to x* along each of the eigenvector directions. Since the eigenvalues are ordered $\lambda_1 \le \lambda_2 \le .. \le \lambda_n$, y_1 corresponds to the longest axis of the ellipsoidal level curves of the cost function, while y_n gives the distance along the shortest axis.

For the q_1 direction, $y_{1[k+1]} = (1 - \mu_{[k]}\lambda_1)y_{1[k]}$. Since $\mu_{[k]}$ is at most $1/\lambda_1$, $y_{1[k+1]}$ always has the same sign and a smaller magnitude than $y_{1[k]}$. That is, a steepest descent

step always reduces the distance along the long (q_1) axis, without changing to the opposite direction along it.

Since a nonzero $y_{1[k+1]}$ has the same sign as $y_{1[k]}$, and since Equation 3.2.4 requires $g_{[k]}^T g_{[k+1]} = y_{[k]}^T \Lambda^2 y_{[k+1]} = \Sigma \lambda_i^2 y_{i[k]} y_{i[k+1]} = 0$, at least one component of $y_{[k]}$ must take the opposite sign in $y_{[k+1]}$.

In particular the n^{th} component of $y_{[k+1]}$, if it is not zero, changes sign. In the q_n direction, $y_{n[k+1]} = (1 - \mu_{[k]}\lambda_n) y_{n[k]}$. If $\mu_{[k]} = 1/\lambda_n$, which occurs only when $y_{[k]}$ is exactly a multiple of q_n, $y_{[k+1]} = 0$. For any other starting point, $\mu_{[k]} > 1/\lambda_n$ and $y_{n[k+1]}$ has the opposite sign from $y_{n[k]}$. In fact if $\mu_{[k]} > 2/\lambda_n$, then $1 - \mu_{[k]}\lambda_n < -1$ and $y_{n[k+1]}$ has a greater magnitude than $y_{n]k]}$.

Two consequences follow. First, since $\mu_{[k]} \leq 1/\lambda_1$, if $\lambda_1 > \lambda_n/2$ then $\mu_{[k]} < 2/\lambda_n$ and the magnitude of y_n decreases with each steepest descent step. By similar reasoning, when $\lambda_1 > \lambda_n/2$ each iteration reduces the magnitude of every component of y.

By contrast, if the spread in the eigenvalues is greater than a factor of two, one or more components of $y_{[k+1]}$ can increase in magnitude at a given iteration.

Second, steepest descent steps are trapped by the q_1 axis. Viewed in the q_1-q_n plane, where the level curve is at its most elongated, each steepest descent step crosses the q_1 (long) axis of the ellipse and never crosses the q_n (short) axis. Successive steps work their way down the q_1 axis towards x*. Figure 3.4.1 illustrates this behavior for three starting points on the level curve $\frac{1}{2} y^T \Lambda y = 1$, where $\Lambda = \text{diag}\{1.5, 4\}$ from the sample quadratic of Equation 3.2.15a.

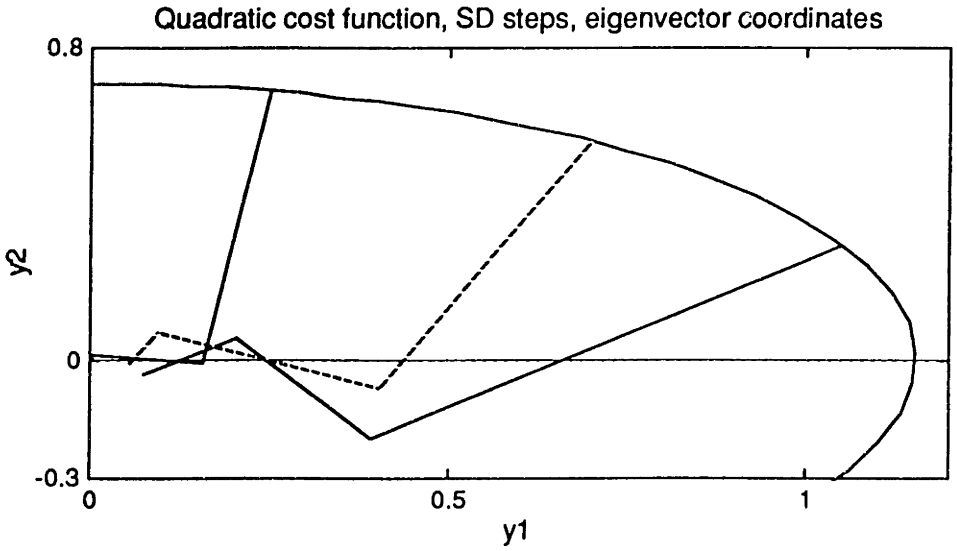

Figure 3.4.1. The level curve $0.5 \cdot y^T \Lambda y = 1$, and some steepest descent steps for three different starting points, for $\Lambda = diag\{1.5, 4\}$. Successive iterates follow q_1, the long axis. Successive search directions do not appear to be perpendicular because the axes are not drawn to the same scale.

Steepest descent net direction

The fact that steepest descent method tends to follow q_1 and to alternate signs on the q_n axis implies that the net direction traveled by successive steps points more in the direction of q_1, and less in that of q_n, than do the gradients themselves. Since q_1 is the long axis of the level curves, down which successive steps must travel, this in turn

suggests that the net direction is itself a likely search direction. (Powell's direction set method [Press §10.5] uses a similar strategy in searching the net direction after n minimizations along the coordinate axes.)

For a two-dimensional problem, the net direction after two steps goes right to x*. Suppose the iterate $x_{[k+2]}$ was reached by two steepest descent steps. By Equations 3.2.4 and 3.2.5, after an exact line search the gradients $g_{[k+2]}$ and $g_{[k+1]}$ are perpendicular. Similarly, $g_{[k]}$ is perpendicular to $g_{[k+1]}$, and therefore is parallel to $g_{[k+2]}$,

$$c \cdot g_{[k+2]} = g_{[k]}$$

With $g = Q\Lambda y$, it follows

$$c \cdot y_{[k+2]} = y_{[k]}$$

so that $y_{[k+2]}$ lies along the ray from the center of the ellipsoid, $x = x^*$ or $y = 0$, to the original starting point $y_{[k]}$. Therefore a search in the net direction from $y_{[k]}$ to $y_{[k+2]}$ will, with exact arithmetic, pass directly through the minimum. The search direction,

$$d_{[k+2]} = x_{[k]} - x_{[k+2]} \tag{3.4.13}$$

points from $x_{[k+2]}$ back to $x_{[k]}$ since Equation 3.1.4 gives the negative search direction.

For a quadratic cost function of more than two variables, and for a nonquadratic cost function whose Hessian does not change too rapidly, the net direction after two steepest descent steps should still be a valid choice.

Estimated net direction

Equation 3.4.13 implies two exact line minimizations in gradient directions, followed by a third minimization, from $x_{[k+2]}$, in the net direction. Suppose instead that $x_{[k+2]}$ is estimated by the projected iterate $x'_{[k+2]}$ resulting from two steepest descent steps on the quadratic model of Ø at $x_{[k]}$. Then a single search can be made, from $x_{[k]}$, in the estimated net direction. (The notation x' is adopted over $\hat{x}$ to avoid confusion, e.g. between the stepsize $\mu'_{[k]}$ for the quadratic model in the steepest descent direction, and a trial stepsize $\hat{\mu}_{[k]}$ for Ø in the estimated net direction.)

Given $g_{[k]}$ and a positive-definite $G_{[k]}$, form the quadratic model of Equation 3.2.9b. A steepest descent step on this quadratic finds

$$x'_{[k+1]} = x_{[k]} - \mu'_{[k]} g_{[k]} \tag{3.4.14}$$

The minimizer of the quadratic model in the steepest descent direction, $x'_{[k+1]}$ is called the Cauchy point [Dennis §6.4.2]. From Equation 3.4.5e, the gradient at $x'_{[k+1]}$ is $g'_{[k+1]} = A(x_{[k]} - \mu'_{[k]} g_{[k]}) - b$, where $A = G_{[k]}$ and $b = G_{[k]} x_{[k]} - g_{[k]}$ are identified from Equation 3.2.9b. Substituting,

$$g'_{[k+1]} = g_{[k]} - \mu'_{[k]} G_{[k]} g_{[k]}$$

Note if $g_{[k]}$ is an eigenvector of $G_{[k]}$ then $g'_{[k+1]} = 0$, and $x'_{[k+1]}$ minimizes the quadratic model. Otherwise the next steepest descent iteration on the quadratic finds $x'_{[k+2]} = x'_{[k+1]} - \mu'_{[k+1]} g'_{[k+1]}$ or

$$x'_{[k+2]} = x_{[k]} - \mu'_{[k]} g_{[k]} - \mu'_{[k+1]} g'_{[k+1]}$$

Therefore the net search direction from $x_{[k]}$ for two steepest descent steps on the quadratic model is

(3.4.15a) $$d_{[k]:net} = \mu'_{[k]}g_{[k]} + \mu'_{[k+1]}g'_{[k+1]}$$

where

(3.4.15b) $$\mu'_{[k]} = \frac{g_{[k]}^T g_{[k]}}{g_{[k]}^T G_{[k]} g_{[k]}}$$

from Equation 3.2.11a, and where

(3.4.15c,d) $$g'_{[k+1]} = g_{[k]} - \mu'_{[k]}G_{[k]}g_{[k]} \quad \text{and} \quad \mu'_{[k+1]} = \frac{g'^T_{[k+1]} g'_{[k+1]}}{g'^T_{[k+1]} G_{[k]} g'_{[k+1]}}$$

Now use this estimated net direction in a line search directly from $x_{[k]}$. From Equation 3.2.2 the initial slope

$$\left.\frac{d\emptyset}{d\mu}\right|_{x_{[k]}:net} = -g_{[k]}^T(\mu'_{[k]}g_{[k]} + \mu'_{[k+1]}g'_{[k+1]})$$

or

(3.4.16) $$\left.\frac{d\emptyset}{d\mu}\right|_{x_{[k]}:net} = -\mu'_{[k]}g_{[k]}^T g_{[k]}$$

since $g_{[k]}^T g'_{[k+1]} = 0$ by Equation 3.2.4. Note that the initial slope in the steepest descent direction itself is $-g_{[k]}^T g_{[k]}$. If $G_{[k]}$ is positive-definite then $\mu'_{[k]} > 0$ and $d_{[k]:net}$ is a descent direction.

3.5 Other Line Search Directions

The steepest descent method, considering only first derivative information at the current iterate, limits convergence of an iterative function minimization routine. The methods of this section improve the serach direction, either by using information available from the last search, or by using second derivative information at the current iterate.

Conjugate directions

Rather than choosing $d_{[k]}$ perpendicular to $d_{[k-1]}$, as in the steepest descent method, conjugate direction methods choose the new search direction so that, in the neighborhood of $x_{[k]}$, the gradient remains perpendicular to $d_{[k-1]}$ as μ increases [Press §10.5]. A Taylor series expansion of the gradient near $x_{[k]}$ is

(3.5.1) $$\nabla\emptyset\{x_{[k]} + \Delta x\} \approx g_{[k]} + G_{[k]}\Delta x$$

where G replaces G^T in $\Delta g \approx G^T\Delta x$ since the Hessian is symmetric. To keep this gradient perpendicular to $d_{[k-1]}$ requires $d_{[k-1]}^T(g_{[k]} + G_{[k]}\Delta x) = 0$. With $d_{[k-1]}^T g_{[k]} = 0$ from the previous minimization, and with $\Delta x = -\mu d_{[k]}$, the conjugate direction satisfies

(3.5.2) $$d_{[k-1]}^T G_{[k]} d_{[k]} = 0$$

When Equation 3.5.2 holds the two search directions are called G-conjugate [Press §10.5] or G-orthogonal [Strang86 §5.1].

When G is constant-- for example, in the quadratic $f = \frac{1}{2}x^TAx - x^Tb + c$ where A is a symmetric, positive-definite matrix and G = A-- then $d_{[k]}$ can be chosen to be G-

conjugate not only to $d_{[k-1]}$, but to all previous search directions [Strang86 §5.3]. That is, $d_{[k]}$ can be chosen to set

(3.5.3) $$D_{[k-1]}{}^T G d_{[k]} = 0$$

where $D_{[k-1]} = [d_{[0]}\ d_{[1]}\ ..\ d_{[k-1]}]$ is the rectangular matrix of the k previous search directions.

Search directions satisfying Equation 3.5.3 for G positive-definite are linearly independent. Suppose $d_{[k]}$ is a linear combination of the previous serach directions, so that $d_{[k]} = D_{[k-1]}y$. Then $d_{[k]}{}^T G d_{[k]} = y^T D_{[k-1]}{}^T G d_{[k]} = 0$. But $d_{[k]}{}^T G d_{[k]} = 0$ for G positive-definite implies $d_{[k]} = 0$. Therefore $d_{[k]}$ is independent of the previous search directions. This implies that, using exact arithmetic, the minimum of the quadratic is found within n iterations [Strang86 §5.3], since it is not possible to pick n+1 independent search directions for an n-dimensional problem. Mathematically this occurs because the condition $D_{[k-1]}Gd_{[k]} = 0$ decouples the search in the $d_{[k]}$ direction from the k-1 searches preceding it [Golub §10.2.3]. Thus, for a quadratic function the optimizing stepsize in a G-conjugate search direction is independent of the order in which the search directions are tried.

Conjugate gradient method

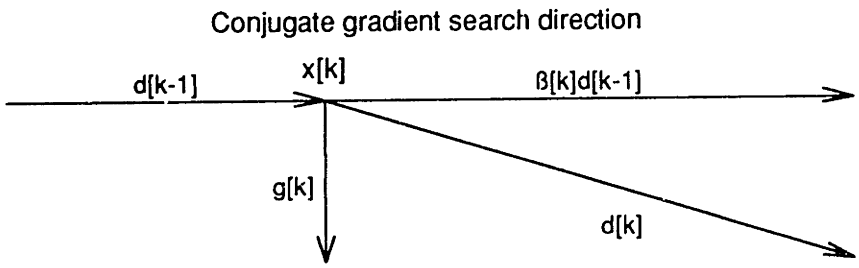

Figure 3.5.1. The conjugate gradient search direction. The gradient $g_{[k]}$ is orthogonal to the previous search direction $d_{[k-1]}$ by Equation 3.2.4.

Equation 3.5.2 defines a conjugate direction for a general function Ø without specifying how to choose $d_{[k]}$. The conjugate gradient method uses

(3.5.4) $$d_{[k]:CG} = g_{[k]} + ß_{[k]}d_{[k-1]}$$

See Figure 3.5.1. The first direction is the gradient, $d_{[0]} = g_{[0]}$ [Golub §10.2.5].

The scalar $ß_{[k]}$ in Equation 3.5.4 is chosen to satisfy Equation 3.5.2. Transposing, this requires

$$0 = d_{[k]}{}^T G_{[k]}{}^T d_{[k-1]} = \frac{1}{\mu_{[k-1]}} d_{[k]}{}^T G_{[k]}{}^T (x_{[k-1]} - x_{[k]})$$

Substituting $\Delta g \approx G^T \Delta x$,

$$0 = d_{[k]}{}^T (g_{[k-1]} - g_{[k]})$$

Substituting Equation 3.5.4 and recognizing $d_{[k-1]}{}^T g_{[k]} = 0$ from the previous line minimization,

$$0 = g_{[k]}^T(g_{[k-1]} - g_{[k]}) + \beta_{[k]} d_{[k-1]}^T g_{[k-1]}$$

Again from Equation 3.5.4, $d_{[k-1]} = g_{[k-1]} + \beta_{[k-1]} d_{[k-2]}$, so $d_{[k-1]}^T g_{[k-1]} = g_{[k-1]}^T g_{[k-1]}$ and

$$\beta_{[k]} = \frac{g_{[k]}^T(g_{[k]} - g_{[k-1]})}{g_{[k-1]}^T g_{[k-1]}} \tag{3.5.5}$$

the Polak-Ribiere formula [Press §10.6].

Conjugate gradients for quadratic cost function

Suppose G is constant. Then the direction generated by Equation 3.5.5 is G-conjugate to every previous direction [Press §10.6]. An inductive proof [Golub §10.2.5] supposes the previous search directions are G-conjugate, $d_{[j]}^T G d_{[i]} = 0$ for $i \neq j$ and $i,j \leq k-1$. Then

$$g_{[k]}^T d_{[i]} = (Gx_{[k]} - b)^T d_{[i]} = (Gx_{[i+1]} - b - \sum_{j=i+1}^{k-1} \mu_{[j]} G d_{[j]})^T d_{[i]}$$

$$g_{[k]}^T d_{[i]} = g_{[i+1]}^T d_{[i]} - \sum_{j=i+1}^{k-1} \mu_{[j]} d_{[j]}^T G d_{[i]}$$

$$g_{[k]}^T d_{[i]} = 0$$

and the gradient at $x_{[k]}$ is perpendicular to every previous search direction $d_{[i]}$. Since the previous search directions span the same space as the previous gradients, the gradient at $x_{[k]}$ is perpendicular to every previous gradient $g_{[i]}$:

$$0 = g_{[k]}^T d_{[i]} = g_{[k]}^T(g_{[i]} + \beta_{[i]} d_{[i-1]})$$

$$g_{[k]}^T g_{[i]} = 0 \tag{3.5.6}$$

when G is constant.

Finally, with $\beta_{[k]}$ has been chosen to make $d_{[k]}$ G-conjugate to $d_{[k-1]}$, $d_{[k]}$ is G-conjugate to all previous directions:

$$d_{[k]}^T G d_{[i]} = (g_{[k]} + \beta_{[k]} d_{[k-1]})^T G d_{[i]} = g_{[k]}^T G(x_{[i]} - x_{[i+1]})/\mu_{[i]}$$

$$d_{[k]}^T G d_{[i]} = g_{[k]}^T(g_{[i]} - g_{[i+1]})/\mu_{[i]}$$

$$d_{[k]}^T G d_{[i]} = 0$$

Therefore, when G-conjugacy holds for the k-1 previous search directions, it holds for the new search direction $d_{[k]}$, satisfying Equation 3.5.3. With G constant the conjugate gradient choice yields conjugate directions, and ideally the method minimizes the quadratic function in n steps, as shown above. Note the method requires exact minimization at each step. However, roundoff errors during computation prevent full orthogonalization [Golub §10.2.7], and in practice the method is iterative.

By Equation 3.5.6, when G is constant Equation 3.5.5 simplifies to

$$\beta_{[k]} = \frac{g_{[k]}^T g_{[k]}}{g_{[k-1]}^T g_{[k-1]}} \tag{3.5.7}$$

the Fletcher-Reeves formula [Press §10.6]. When Ø is not quadratic, the Polak-Ribiere form is preferred [Strang86 §5.3].

Newton's method

The second order Taylor series model of Ø, Equation 3.2.9b,

$$f_{[k]}\{x\} = \frac{1}{2} x^T G_{[k]} x - x^T(G_{[k]}x_{[k]} - g_{[k]}) + c_{[k]}$$

has the same function value, gradient, and Hessian as has Ø at $x_{[k]}$. If Ø is quadratic, its Hessian $\nabla^2\text{Ø} = G$ is constant and $f_{[k]}$ exactly equals Ø for all x. For nonquadratic Ø, $f_{[k]} \approx \text{Ø}$ in the neighborhood of $x_{[k]}$.

Similarly, $\nabla f_{[k]} \approx g$ in the neighborhood of $x_{[k]}$. Following Equation 3.3.2a,

$$\nabla f_{[k]}\{x\} = G_{[k]}x - (G_{[k]}x_{[k]} - g_{[k]}) = G_{[k]}(x - x_{[k]}) + g_{[k]} \tag{3.5.8}$$

Equation 3.5.8 has $\nabla f_{[k]}\{x_{[k]}\} = g_{[k]}$, as expected. Equation 3.2.14 states that $f_{[k]}$ has a stationary point-- a minimum if $\nabla^2 f_{[k]} = G_{[k]}$ is positive-definite-- at $\nabla f_{[k]} = 0$. Since the stationary point in $f_{[k]}$ estimates the location of the stationary point in Ø, Newton's method chooses $x_{[k+1]}$ to set $\nabla f_{[k]}\{x_{[k+1]}\} = 0$:

$$G_{[k]}\Delta x_{[k]:N} = -g_{[k]} \tag{3.5.9a}$$

where

$$\Delta x_{[k]:N} = x_{[k+1]:N} - x_{[k]} \tag{3.5.9b}$$

[Dennis §5.5]. As with Newton-Raphson's method for nonlinear equations, matrix factorization is preferred to inversion when solving Equation 3.5.9a for Δx. Special factorizations for symmetric matrices are more efficient than the general LU and QR methods used for the Jacobian [Dennis §3.2].

Variations on Newton's method

By Equation 3.2.14 the stationary point in $f_{[k]}\{x\}$ is a unique minimum of $f_{[k]}$ only if $\nabla^2 f_{[k]}$ is positive-definite. Therefore setting $\nabla f_{[k]} = 0$ may not find a minimum of f, and Newton's method may converge to a $\nabla\text{Ø} = 0$ where Ø has a maximum. To avoid this, the method of Levenberg-Marquardt perturbs $G_{[k]}$ to insure it is positive-definite:

$$G_{[k]} + \nu_{[k]}I$$

[Fletcher §5.2]. Equation 3.3.21 asserts that such a $\nu_{[k]}$ can be found, although a $\nu_{[k]}$ chosen to satisfy Equation 3.3.21 may be unnecessarily large (as $\nu_{[k]}$ gets large, the perturbed Hessian tends towards a scaled identity matrix, and the direction tends towards the gradient).

Even with $G_{[k]}$ positive-definite, Newton's method may not converge if $x_{[k]}$ is far from x*. The Newton step therefore gives a trial iterate, which may be accepted if $\text{Ø}\{x_{[k+1]:N}\} < \text{Ø}\{x_{[k]}\}$. Otherwise, a search in the Newton direction

$$d_{[k]:N} = -\Delta x_{[k]:N} \tag{3.5.10}$$

may be used. The initial slope is

$$\left.\frac{d\text{Ø}}{d\mu}\right|_{x_{[k]:N}} = -g_{[k]}{}^T G_{[k]}{}^{-1} g_{[k]}$$

Note $G_{[k]}{}^{-1} = (Q_{[k]}\Lambda_{[k]}Q_{[k]}{}^T)^{-1} = Q_{[k]}\Lambda_{[k]}{}^{-1}Q_{[k]}{}^T$. Since the $[\Lambda_{[k]}{}^{-1}]_{ii} = 1/[\Lambda_{[k]}]_{ii}$, by Equation 3.3.19, if $G_{[k]}$ is positive-definite then so is $G_{[k]}{}^{-1}$, and then Equation 3.5.10 defines a descent direction.

An exact line search can be used with Equation 3.5.10 even if the full Newton step, at $\mu = 1$, reduces Ø.

The damped Newton method follows the Newton direction unless the expected decrease in Ø is greater for the gradient direction. By Equation 3.2.11b the search follows the gradient when

$$g_{[k]}{}^T G_{[k]}{}^{-1} g_{[k]} < \frac{(g_{[k]}{}^T g_{[k]})^2}{g_{[k]}{}^T G_{[k]} g_{[k]}} \tag{3.5.11}$$

[Dahlquist §10.5.3].

Both the Levenberg-Marquardt and damped Newton methods may be seen as precursors of the double dogleg algorithm of the next section, in that all three define a continuous search path which varies in some sense between the Newton and gradient directions.

Newton versus steepest descent

All three of the steepest descent, conjugate gradient, and Newton's methods choose trial iterates using a local model of the behavior of interest. In steepest descents, the model

$$\hat{\emptyset} = \emptyset_{[k]} - \mu d^T g_{[k]}$$

(a truncated Equation 3.2.8) is sufficient to identify $d_{[k]} = g_{[k]}$ as an appropriate search direction. In Newton's method, the behavior of interest is the gradient of Ø, not Ø itself, and the model

$$\hat{g} = g_{[k]} + G_{[k]} \Delta x$$

(a rewritten Equation 3.5.8) is solved for the Δx which sets $\hat{g} = 0$. Therefore Newton's method requires the Hessian, and must invert or factor it, while steepest descents uses the Hessian only to investigate the model-- for instance to estimate the minimizing stepsize-- and then only in multiplication. (Conjugate gradients can do without the Hessian as well, though to develop the Polak-Ribiere and Fletcher-Reeves formulae requires the quadratic model.)

Because Newton's method chooses $x_{[k+1]}$ directly, it only implies a search direction; the method of steepest descent begins by choosing a search direction, and picks $x_{[k+1]}$ from along it by trial and error. However, the two steps can be compared by assuming the quadratic model is exact (and $G_{[k]} = G$ is constant).

In the Newton direction, the step $\Delta x_{[k]:N} = -G^{-1} g_{[k]}$. In the notation of Section 3.4, $G^{-1} = (Q \Lambda Q^T)^{-1} = Q \Lambda^{-1} Q^T$ and $g_{[k]} = Q \Lambda y_{[k]}$. Then

$$\Delta x_{[k]:N} = -G^{-1} g_{[k]} = -Q y_{[k]}$$

This result follows immediately from the definition of $y_{[k]}$ as the vector of distances from x^* to $x_{[k]}$ in the eigenvector directions; see Equation 3.4.5d. Therefore the length of a Newton step is the length of $y_{[k]}$:

$$\|\Delta x_{[k]:N}\| = (\Delta x_{[k]:N}{}^T \Delta x_{[k]:N})^{1/2} = (y_{[k]}{}^T y_{[k]})^{1/2} \tag{3.5.12}$$

In the steepest descent direction, the step

$$\Delta x_{[k]:SD} = x_{[k+1]:SD} - x_{[k]} = -\mu_{[k]:SD} g_{[k]}$$

where $x_{[k+1]:SD}$ is the Cauchy point, at which the quadratic model finds its minimum in the steepest descent direction (see Equation 3.4.14). If the quadratic model is exact, then by Equations 3.2.11a and 3.4.6b

$$\mu_{[k]:SD} = \frac{g_{[k]}^T g_{[k]}}{g_{[k]}^T G g_{[k]}} = \frac{y_{[k]}^T \Lambda^2 y_{[k]}}{y_{[k]}^T \Lambda^3 y_{[k]}}$$

With $\|g_{[k]}\| = (g_{[k]}^T g_{[k]})^{1/2} = (y_{[k]}^T \Lambda^2 y_{[k]})^{1/2}$ from Equation 3.4.5e, the distance to the Cauchy point is

$$\|\Delta x_{[k]:SD}\| = \frac{(g_{[k]}^T g_{[k]})^{3/2}}{g_{[k]}^T G g_{[k]}} = \frac{(y_{[k]}^T \Lambda^2 y_{[k]})^{3/2}}{y_{[k]}^T \Lambda^3 y_{[k]}} \tag{3.5.13}$$

From the geometric argument of Section 3.4, if $x_{[k]}$ lies along an eigenvector from x^*, then the steepest descent direction goes right to the maximum, and has the same length as the Newton step (combinations of eigenvectors from a repeated eigenvalue share this property). Thus, using $y_{[k]} = c \cdot e_i$ gives $\|\Delta x_{[k]}\| = c$ in both Equations 3.5.12 and 3.5.13.

If the steepest descent direction does not pass through x^*, it makes an angle $\theta_{[k]} \neq 0$ with the Newton direction. This angle is given by [Strang88 §3.2]

$$\Delta x_{[k]:SD}^T \Delta x_{[k]:N} = \|\Delta x_{[k]:SD}\| \cdot \|\Delta x_{[k]:N}\| \cdot \cos\theta_{[k]}$$

Substituting from above,

$$\mu_{[k]:SD}(Q\Lambda y_{[k]})^T(Q y_{[k]}) = \mu_{[k]:SD}(y_{[k]}^T \Lambda^2 y_{[k]})^{1/2} \cdot \|\Delta x_{[k]:N}\| \cdot \cos\theta_{[k]}$$
$$y_{[k]}^T \Lambda y_{[k]} = \|\Delta x_{[k]:N}\| \cdot (y_{[k]}^T \Lambda^2 y_{[k]})^{1/2} \cdot \cos\theta_{[k]}$$

Combining with Equation 3.5.13,

$$\frac{\|\Delta x_{[k]:SD}\|}{\|\Delta x_{[k]:N}\|} = \frac{(y_{[k]}^T \Lambda^2 y_{[k]})^2}{(y_{[k]}^T \Lambda^3 y_{[k]})(y_{[k]}^T \Lambda y_{[k]})} \cos\theta_{[k]} \tag{3.5.14a}$$

Assuming distinct eigenvalues, the expression on the right equals one only when $y_{[k]} = c \cdot e_i$, for then the algebraic terms cancel and $\cos\theta_{[k]} = \cos(0) = 1$. For any other $y_{[k]}$, $\cos\theta_{[k]} < 1$, while for any y

$$\frac{(y^T \Lambda^2 y)(y^T \Lambda^2 y)}{(y^T \Lambda^3 y)(y^T \Lambda y)} = \frac{\lambda_1^4 y_1^4 + 2\lambda_1^2 \lambda_2^2 y_1^2 y_2^2 + .. + \lambda_n^4 y_n^4}{\lambda_1^4 y_1^4 + (\lambda_1^3 \lambda_2 + \lambda_1 \lambda_2^3) y_y^2 y_2^2 + .. + \lambda_n^4 y_n^4} \leq 1$$

since every term $\lambda_i^4 y_i^4$ in the numerator matches an identical term in the denominator, and since every term $2\lambda_i^2 \lambda_j^2 y_i^2 y_j^2$ in the numerator matches a term $(\lambda_i^3 \lambda_j + \lambda_i \lambda_j^3) y_i^2 y_j^2$ in the denominator. The denominator must be at least as large as the numerator since the geometric mean of λ_i^2 and λ_j^2 is less than or equal to their arithmetic mean [Ellis §1.1]:

$$\sqrt{\lambda_i^2 \lambda_j^2} \leq \frac{\lambda_i^2 + \lambda_j^2}{2}$$
$$2\lambda_i \lambda_j \leq \lambda_i^2 + \lambda_j^2$$
$$2\lambda_i \lambda_j \leq \lambda_i^3 \lambda_j + \lambda_i \lambda_j^3$$

Therefore

$$\|\Delta x_{[k]:SD}\| \leq \|\Delta x_{[k]:N}\| \tag{3.5.14b}$$

and for a quadratic cost function the length of a steepest descent step is less than the Newton step length, unless the directions are colinear [Dennis §6.4.2]. In practice their ratio can be much less than one [Schnabel §2.2].

Example: quadratic problem

Applying Newton's method to the quadratic cost function of Equation 3.2.15a finds the minimum in one iteration, since the quadratic has a constant Hessian. Furthermore, using exact arithmetic the conjugate gradient method finds the minimum in two iterations, for this two-dimensional problem (the first iterate is the steepest descent step, exactly as shown in Figure 3.2.1).

Newton versus Newton-Raphson

Of all the minimization methods, Newton's method most strongly resembles Newton-Raphson because it models an n-dimensional vector-- the gradient-- rather than the scalar Ø itself. Both methods zero the model, $\hat{r} = 0$ or $\hat{g} = 0$, resulting in an equation, $J\Delta x = -r$ or $G\Delta x = -g$, whose solution requires matrix factorization or inversion.

However, for function minimization finding $g_{[k]} = 0$ merely suggests that $x_{[k]}$ may be a solution-- Newton's method may converge to a maximum or to a saddle point, or it may converge to a local but not a global minimum. For equilibrium problems, $r = 0$ defines the solution, so Newton-Raphson's method has no such caveats.

3.6 The Double Dogleg Algorithm

Suppose a full Newton step with a positive-definite $G_{[k]}$ increases the cost function. Since $d_{[k]:N}$ is a descent direction, some $\mu < 1$ makes $Ø < Ø_{[k]}$ in the Newton direction. If the resulting step length is less than that to the Cauchy point, the steepest descent direction is likely to reduce Ø more effectively than does the Newton direction.

The double dogleg algorithm searches the steepest descent direction for small step lengths, and cuts back to the Newton direction for longer steps. (Because the search directions have different lengths, the extent of a step is measured by its length $\|\Delta x\|$ rather than by some composite stepsize μ). Thus the double dogleg curve combines the globally-convergent, but potentially slow, gradient direction with the fast, locally-convergent, Newton step [Dennis §6.1].

The development in this section is not rigorous; see [Dennis §6.4, Fletcher §5.2] for a more complete discussion, and [Dennis §A.IV] for computational details.

Relation to Powell's hybrid method

A precursor to the double dogleg method, Powell's hybrid method, uses a dogleg curve with a single turning, at the Cauchy point in the steepest descent direction, from which it proceeds directly to the Newton point [Dennis §6.4.2; for more detail see Schnabel §2.2, Moré82 §2.1].

Optimal trajectory

It can be shown that to minimize the quadratic model $f_{[k]}$, subject to a desired step length $\|\Delta x_{[k]}\| = \delta_{[k]}$, requires finding the unique scalar $v_{[k]}$ such that the step

(3.6.1a) $$\Delta x_{[k]:opt} = -(G_{[k]} + v_{[k]}I)^{-1} g_{[k]}$$

satisfies

(3.6.1b) $$\|\Delta x_{[k]:opt}\| = \delta_{[k]}$$

[Dennis §6.4], provided the desired step length $\delta_{[k]} \leq \|\Delta x_{[k]:N}\|$.

Equation 3.6.1 defines the Levenberg-Marquardt curve. It starts out in the gradient direction, since for large $\nu_{[k]}$, $(G_{[k]} + \nu_{[k]}I)^{-1}g_{[k]} \approx g_{[k]}/\nu_{[k]}$ has a small norm. As $\delta_{[k]}$ increases towards $\|\Delta x_{[k]:N}\|$, $\nu_{[k]}$ decreases towards zero, where Equation 3.6.1a gives the Newton step $-G_{[k]}^{-1}g_{[k]}$.

The trajectory defined by Equation 3.6.1, optimal for the quadratic model, also defines a search path for minimizing Ø. This is the basis for the hook, or locally constrained optimal step, algorithm [Dennis §6.4.1]. However, the computational cost of placing $\Delta x_{[k]}$ on the Levenberg-Marquardt curve is high, since the method requires iterations on Equation 3.6.1a to pick $\nu_{[k]}$ for a given $\delta_{[k]}$, and iterations on Equation 3.6.1b to pick an appropriate $\delta_{[k]}$, even without exact minimization.

The double dogleg method replaces the hook trajectory with a piecewise linear approximation to the Levenberg-Marquardt curve. Points along the double dogleg curve are only marginally inferior to the corresponding hook curve points at minimizing $f_{[k]}$, and can be calculated directly for a given $\delta_{[k]}$. Therefore the dogleg is preferred unless the cost of evaluating Ø is high [Dennis §6.4.2].

Double dogleg curve

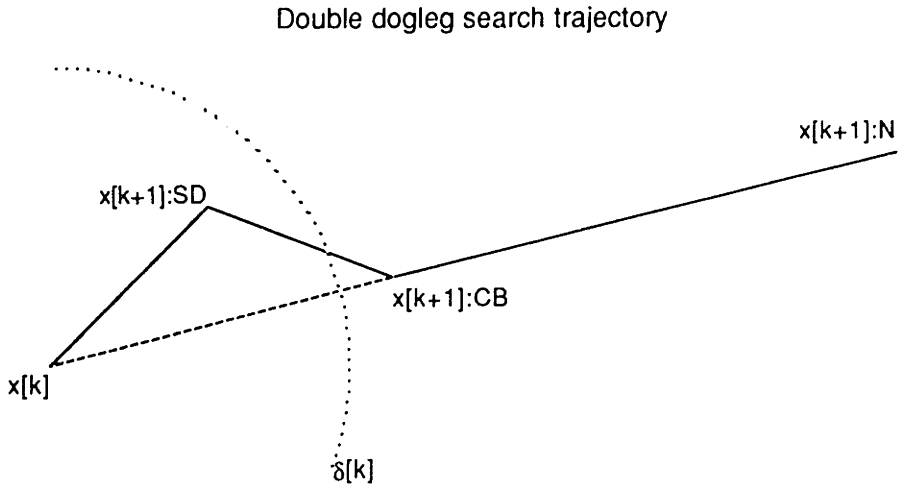

Figure 3.6.1. The double dogleg search path, with a sample choice of $\delta_{[k]}$.

The double dogleg curve extends from $x_{[k]}$ to the Cauchy point, then turns to connect with the Newton direction at a step length somewhere between $\|\Delta x_{[k]:SD}\|$ and $\|\Delta x_{[k]:N}\|$. Call this point the cutback point, $x_{[k+1]:CB}$. From the cutback point the curve continues on to $x_{[k+1]:N}$. See Figure 3.6.1.

Placing the cutback point at

(3.6.2a) $$x_{[k+1]:CB} = x_{[k]} + \eta \cdot \Delta x_{[k]:N}$$

requires $\eta \leq 1$ to keep it between $x_{[k]}$ and the Newton point (the single dogleg curve sets $\eta = 1$ [Schnabel §2.2]). Also requiring $\eta \geq \|\Delta x_{[k]:SD}\|/\|\Delta x_{[k]:N}\|$ makes the curve: (1) well defined, because for any trust length it specifies a unique point a distance $\delta_{[k]}$ from $x_{[k]}$; and (2) reasonable, because the value of $f_{[k]}$ decreases monotonically along the curve [Dennis §6.4.2]. Based on computational experience, the algorithm puts

(3.6.2b) $$\eta = 0.2 + 0.8 \frac{(g_{[k]}{}^T g_{[k]})^2}{(g_{[k]}{}^T G_{[k]} g_{[k]}) \cdot (-g_{[k]}{}^T \Delta x_{[k]:N})}$$

[Dennis §6.4.2; note $H_c^{-1}\nabla f\{x_c\} = -\Delta x_{[k]:N}$]. Following logic similar to that for Equation 3.5.14b, this choice always satisfies $\|\Delta x_{[k]:SD}\|/\|\Delta x_{[k]:N}\| \leq \eta \leq 1$.

Model trust region

The double dogleg algorithm does not minimize Ø exactly along its search path (neither does the hook algorithm). Rather than varying $\delta_{[k]}$ continuously along the curve at each iteration, trust region methods carry over the expected length of a successful step from one iteration to another, updating $\delta_{[k]}$ by heuristic rules [Dennis §6.4]. Trust region methods also are called restricted step methods, since $\delta_{[k]}$ limits the step which may be taken [Fletcher §5.1].

In response to an unsuccessful trial step, the algorithm shortens the trust region, pushing the trial iterate back along the double dogleg curve. As $\delta_{[k]}$ gets small with respect to the length of the Newton step, the trial steps bias towards the gradient direction, recommended as part of a global strategy when far from x* [Dahlquist §10.5.3]. As the iterates approach x*, the algorithm adjusts the trust region to take full Newton steps.

Geometrically, the trust region $\delta_{[k]}$ defines a spherical region about $x_{[k]}$, within which the quadratic model adequately captures the behavior of $Ø\{x\}$ [Dennis §6.4]. If $f_{[k]} \approx Ø$ inside this trust region, then minimizing $f_{[k]}$ on $\delta_{[k]}$ also approximately minimizes Ø on this region. The hook step exactly minimizes $f_{[k]}$; the double dogleg approximates the hook; for both, $f_{[k]}$ decreases monotonically along the curve. Therefore placing $x_{[k+1]}$ at a step length $\delta_{[k]}$ along the double dogleg curve approximately minimizes Ø within the trust region.

The trust region need not be spherical. For example, scaling x by a diagonal matrix D, x' = Dx, defines ellipsoids in the scaled system x'. See [Dennis §7.1] for a discussion of scaling in the algorithm. Note also that while the trust region usually limits $\|\Delta x\|_2$, other choices are possible. In particular, the boxstep method constrains $\|\Delta x\|_\infty \leq \delta_{[k]}$ [Fletcher §5.1].

After choosing $x_{[k+1]}$, the algorithm may expand or contract the trust region based on how well $f_{[k]}\{x_{[k+1]}\}$ predicts $Ø\{x_{[k+1]}\}$ [Dennis §6.4.3]. It may update the trust region for the current iteration (adjust $\delta_{[k]}$) or change it in preparation for the next iteration (choose $\delta_{[k+1]}$), as described below.

Actual and predicted decreases

Trust region methods compare the actual decrease in the cost function to that predicted by the quadratic model, expanding the trust region if the prediction is close to the actual value, and reducing the trust region for a poor prediction. The actual change in Ø at a point $x_{[k+1]}$,

(3.6.3) $$\Delta Ø_{[k]} = Ø\{x_{[k+1]}\} - Ø_{[k]}$$

is negative when a step improves the cost function-- ΔØ gives the actual increase in Ø, not its actual decrease. Though perhaps counter-intuitive, this definition is consistent with the Taylor series expansions where e.g. $\Delta x_{[k]} = x_{[k+1]} - x_{[k]}$. Similarly,

(3.6.4a) $$\Delta f_{[k]} = f_{[k]}\{x_{[k+1]}\} - Ø_{[k]}$$

is the predicted increase in $\emptyset$ (it is also the actual increase in the model $f_{[k]}$, since $f_{[k]}\{x_{[k]}\} = \emptyset\{x_{[k]}\} = \emptyset_{[k]}$). From Equation 3.2.9a,

$$\Delta f_{[k]} = \Delta x_{[k]}^T g_{[k]} + \frac{1}{2} \Delta x_{[k]}^T G_{[k]} \Delta x_{[k]} \tag{3.6.4b}$$

Updating current trust region

Suppose a trust region algorithm, inheriting $\delta_{[k]}$ from the previous iteration, finds a trial point $\hat{x}_{[k+1]}$ on the double dogleg curve at a step length $\|\hat{x}_{[k+1]} - x_{[k]}\| = \delta_{[k]}$. If $\hat{x}_{[k+1]}$ is unacceptable, the algorithm reduces $\delta_{[k]}$ and finds a new $\hat{x}_{[k+1]}$. If on the other hand $f_{[k]}\{\hat{x}_{[k+1]}\}$ estimates $\emptyset\{\hat{x}_{[k+1]}\}$ well, the algorithm may increase $\delta_{[k]}$, after storing $\hat{x}_{[k+1]}$ in case the new trust region oversteps the model's predictive range.

Note that although $\delta_{[k]}$ may be updated iteratively within the larger iteration from $x_{[k]}$ to $x_{[k+1]}$, no special notation is adopted to make the sub-iterative task explicit.

To accept $\hat{x}_{[k+1]}$, the algorithm requires that the actual decrease in $\emptyset$, $-\Delta\emptyset_{[k]}$, is at least some fraction α of the decrease which would obtain if the initial rate of decrease held over the entire step length. Let $d = -\Delta x_{[k]}$ in Equation 3.2.2. Then at $x_{[k]}$ the slope $d\emptyset/d\mu = g_{[k]}^T \Delta x_{[k]}$. Continuing at this initial slope over the entire step, from $\mu = 0$ to $\mu = 1$, would decrease $\emptyset$ by $-g_{[k]}^T \Delta x_{[k]}$. Therefore the algorithm accepts a step $\Delta x_{[k]} = \hat{x}_{[k+1]} - x_{[k]}$ when

$$\Delta\emptyset_{[k]} \leq \alpha \cdot g_{[k]}^T \Delta x_{[k]} \tag{3.6.5a}$$

[Dennis §6.4.3]. A second interpretation of Equation 3.6.5 states that the average rate of change of $\emptyset$ must be less than a fraction α of its initial rate of change,

$$\frac{\Delta\emptyset_{[k]}}{\|\Delta x_{[k]}\|} \leq \alpha \frac{g_{[k]}^T \Delta x_{[k]}}{\|\Delta x_{[k]}\|}$$

[Dennis §6.3]. The fraction α is on (0, 1); the double dogleg algorithm uses

$$\alpha = 10^{-4} \tag{3.6.5b}$$

With this choice, Equation 3.6.5 requires little more than that $\emptyset\{x_{[k+1]}\} < \emptyset_{[k]}$.

If a trial step fails Equation 3.6.5, the algorithm backtracks by reducing the trust region for the current iteration, multiplying it by a factor λ on $[\frac{1}{10}, \frac{1}{2}]$. To select λ, fit a parabola to the points (λ=0, y=$\emptyset_{[k]}$) and (λ=1, y=$\emptyset\{\hat{x}_{[k+1]}\}$), and to the slope $y'\{\lambda=0\} = g_{[k]}^T \Delta x_{[k]}$ (λ is analogous to μ in a line search, so $dy/d\lambda = -g^T d$ where $d = -\Delta x_{[k]}$). The parabola has a minimum where $y' = 0$, or at

$$\lambda = \frac{g_{[k]}^T \Delta x_{[k]}}{2(g_{[k]}^T \Delta x_{[k]} - \Delta\emptyset_{[k]})} \tag{3.6.6}$$

[Dennis §6.4.3]. This value is increased to $\frac{1}{10}$ or decreased to $\frac{1}{2}$ as necessary, and $\delta_{[k]}$ reset to $\lambda \cdot \delta_{[k]}$.

If $\hat{x}_{[k+1]}$ satisfies Equation 3.6.5, and if $\delta_{[k]} < \|\Delta x_{[k]:N}\|$, the algorithm tests whether to extend $\delta_{[k]}$. Calculating $\emptyset$ at a new $\hat{x}_{[k+1]}$ is less expensive than evaluating the gradient and Hessian at an accepted $x_{[k+1]}$, so in general if the model agrees well with $\emptyset$, the current trust region should be extended. Of course the current trust region is not

extended if it was reduced during the same step in order to satisfy Equation 3.6.5, or if it already takes the full Newton step.

The algorithm doubles $\delta_{[k]}$ when the predicted change is within ten per cent of the actual change, $|\Delta f_{[k]} - \Delta\emptyset_{[k]}| \le 0.1 \cdot |\Delta\emptyset_{[k]}|$. Since $\hat{x}_{[k+1]}$ satisfies Equation 3.6.5, $\Delta\emptyset_{[k]} < 0$, so the algorithm doubles $\delta_{[k]}$ when an acceptable step has

$$|\Delta f_{[k]} - \Delta\emptyset_{[k]}| \le -0.1 \cdot \Delta\emptyset_{[k]} \tag{3.6.7}$$

[Dennis §6.4.3]. In addition, the algorithm tries to double the trust region when the actual reduction is greater than the reduction predicted by the initial slope alone. This corresponds to using $\alpha = 1$ in Equation 3.6.5a. Thus the algorithm also doubles $\delta_{[k]}$ when an acceptable step has

$$\Delta\emptyset_{[k]} \le g_{[k]}^T \Delta x_{[k]} \tag{3.6.8}$$

[Dennis §6.4.3]. Geometrically, Equation 3.6.8 requires negative curvature in the search direction $d = -\Delta x_{[k]}$, since a concave-up plot of $\emptyset$ versus positive μ, where $d\emptyset/d\mu > 0$, lies above the tangent line drawn at $\mu = 0$.

To extend the current trust region, the algorithm: (1) stores the current, successful, values of $\hat{x}_{[k+1]}$ and $\emptyset\{\hat{x}_{[k+1]}\}$; (2) doubles $\delta_{[k]}$; and (3) rejects the new trial iterate if it fails Equation 3.6.5 or if it gives a cost function more positive than that stored for the previous value of $\delta_{[k]}$.

Choosing next trust region

Once the algorithm accepts an iterate $x_{[k+1]}$, it sets the trust region for the next step, $\delta_{[k+1]}$, to twice, half, or the same as $\delta_{[k]}$.

If the algorithm increased the trust region during the current iteration, then rejected that increased trust length due to a worse cost function, or for failing Equation 3.6.5a, the unchanged trust length carries to the next iteration.

If the model overestimates the actual decrease in $\emptyset$ by more than a factor of ten, the algorithm halves the trust region for the next step. That is, it sets $\delta_{[k+1]} = \delta_{[k]}/2$ when $-10 \cdot \Delta\emptyset_{[k]} < -\Delta f_{[k]}$, or when $\Delta\emptyset_{[k]} > 0.1 \cdot \Delta f_{[k]}$.

Conversely, when the model underestimates the actual decrease in $\emptyset$, $-\Delta f_{[k]} < -\Delta\emptyset_{[k]}$, the algorithm doubles the trust region for the next step. It also doubles the trust region if the model overestimates the reduction, provided the actual reduction is within 75 per cent of the predicted one. Combining these two conditions, finding $-0.75 \cdot \Delta f_{[k]} \le -\Delta\emptyset_{[k]}$, or $\Delta\emptyset_{[k]} \le 0.75 \cdot \Delta f_{[k]}$, doubles the trust region.

Taken together, the double dogleg algorithm updates the trust region by

$$\delta_{[k+1]} = \begin{cases} 2 \cdot \delta_{[k]} \text{ fails tests:} & \delta_{[k]} \\ \Delta\emptyset_{[k]} > 0.1 \cdot \Delta f_{[k]}: & \dfrac{\delta_{[k]}}{2} \\ \Delta\emptyset_{[k]} \le 0.75 \cdot \Delta f_{[k]}: & 2 \cdot \delta_{[k]} \\ \text{else:} & \delta_{[k]} \end{cases} \tag{3.6.9}$$

[Dennis §6.4.3].

CHAPTER 4
APPLYING FUNCTION MINIMIZATION TO ALGEBRAIC PROBLEMS

To control divergence in Newton-Raphson's method by a descent requirement on a residual norm, this chapter integrates the material of Chapters 2 and 3. Section 4.1 discusses general aspects of treating algebraic problems using the methods of function minimization. Requiring each step of a Newton-Raphson-based solver to reduce a residual norm controls divergence, but subjects the solver to stagnation, convergence to non-roots, and slowed convergence to real roots of the algebraic system.

Section 4.2 treats the sum of squares of the residuals, or r-square, a common and important norm. It distinguishes two sources of second-order (curvature) information in the model of the cost function, and shows that neglecting one of these sources allows the straightforward adaptation of Hessian-based methods of function minimization to the solution of algebraic problems. Finally it adapts the double dogleg algorithm to r-square, rewriting the necessary expressions purely in terms of the variables defined in the equilibrium problem framework.

Section 4.3 considers a general residual norm. Since each norm forms a scalar cost function using unique combinations of the residuals, the function minimization results depend not only on the derivatives of the residual functions with respect to the independent variables x, but also on the derivatives of the norm with respect to the residuals. The Jacobian matrix used by Newton-Raphson's method already contains the first set of derivatives; Section 4.3 defines a vector, p, containing the second set. Expressing the results of function minimization in terms of these collections of derivatives gives a geometric interpretation of the norm's direction of steepest descent, provides tests for reasonableness of any particular norm, and establishes exact conditions under which the Newton step on a norm equals the Newton-Raphson step on the underlying algebraic problem.

The last two sections apply the general theory to two particular norms. Section 4.4 considers the one-norm, and shows from the general framework why this norm generally is less tractable than r-square. Section 4.5 discusses a weighted r-square, and gives expressions comparable to those of Section 4.2 for r-square itself.

4.1 Minimization and Equation Solving

Chapter 2 presents Newton-Raphson's method for solving the algebraic system

$$r\{x^{**}\} = 0 \tag{4.1.1}$$

and indicates how instability in the sequence of Newton-Raphson iterates may require constraining each step by a descent requirement

$$\emptyset\{r_{[k+1]}\} < \emptyset\{r_{[k]}\} \tag{4.1.2}$$

for an appropriate scalar cost function $\emptyset\{r\}$ (the residual norm).

Designed to minimize a residual norm, the solver becomes a function minimization algorithm, specialized to equilibrium problems by: (1) the choice of a norm [illegible] computational details derived in this chapter; and (3) a special termination test for a zero residual vector, described in Chapter 5. Unfortunately, while the descent requirement

controls divergence, it introduces convergence problems not associated with a pure Newton-Raphson method. This section describes these problems, explicated more fully in later sections and chapters.

Residual norms

Enforcing a descent requirement requires the cost function to have a minimum at r = 0. This suggests a vector norm, most commonly the Euclidean length or 2-norm,

$$\|r\|_2 = \sqrt{\sum_{i=1}^{n} r_i^2} \tag{4.1.3a}$$

and its variations. Many algorithms use the square of the Euclidean length,

$$\emptyset_{(2)}\{x\} = \sum_{i=1}^{n} r_i^2\{x\} = r^T r \tag{4.1.3b}$$

sometimes introducing a factor of one half for algebraic convenience [Dennis §6.2, Press §9.7]. This thesis writes the Euclidean norm $\|r\|_2$ simply as $\|r\|$, and refers to its square as r-square or as the sum of squares (of the residuals).

The 1-norm

$$\emptyset_{(1)}\{x\} = \|r\|_1 = \sum_{i=1}^{n} |\, r_i \,| \tag{4.1.4}$$

and the infinity-norm or sup-norm

$$\emptyset_{(\infty)}\{x\} = \|r\|_\infty = \max_{1\le i\le n}\{|\, r_i \,|\} \tag{4.1.5}$$

[Dennis §3.1] find common use in linear programming [Strang86 §8.5]. While r-square is a natural choice for overdetermined systems [Dahlquist §5.7], the 1-norm may be useful in ill-conditioned problems [Fletcher §6.2], and the ∞-norm lends itself to efficient gradient calculations since it typically involves only one residual relation [Shor §3.5].

Unlike the 1-norm and the ∞-norm, the 2-norm and its square are everywhere differentiable [Golub §5.3], in particular at r = 0, the ultimate destination of an algebraic equation solver. By contrast, the derivative of the i^{th} term of $\emptyset_{(1)}$ is discontinuous at r_i = 0.

Working algorithms use $\emptyset_{(2)}$ rather than $\|r\| = \sqrt{\emptyset_{(2)}}$ when possible, to avoid the square root operation. However the smaller magnitude of the 2-norm, closer to that of the 1- and ∞-norms, makes it more convenient for graphing. The three norms bound each other by

$$\|r\|_\infty \le \|r\| \le \|r\|_1 \le \sqrt{n}\cdot\|r\| \le n\cdot\|r\|_\infty \tag{4.1.6}$$

where n is the problem dimension [Dennis §3.1].

Other cost functions weight the residuals differently, for example the Huber loss function

(4.1.7a)
$$\emptyset_{(H)}\{x\} = \sum_{i=1}^{n} \rho_{(H)}\{r_i\}$$
$$\rho_{(H)}\{r_i\} = \begin{cases} |r_i| \le h: & r_i^2/2 \\ |r_i| > h: & h|r_i| - h^2/2 \end{cases}$$

and the Beaton-Tukey function

(4.1.7b)
$$\emptyset_{(BT)}\{x\} = \sum_{i=1}^{n} \rho_{(BT)}\{r_i\}$$
$$\rho_{(BT)}\{r_i\} = \begin{cases} |r_i| \le \sqrt{h}: & h(1 - (1 - r_i^2/h)^3)/6 \\ |r_i| > \sqrt{h}: & h/6 \end{cases}$$

For overdetermined systems these cost functions reduce the influence of outlying data [Dennis §10.4], as does the 1-norm [Fletcher §6.1].

Strictly, the cost function need not be a residual norm. For example, $\emptyset_{(2)}$ fails the scaling property $\|c \cdot r\| = |c| \cdot \|r\|$ required of a vector norm [Dennis §3.1], since $\emptyset_{(2)}\{c \cdot r\} = c^2 \cdot \emptyset_{(2)}\{r\}$. Nevertheless, the discussion below often refers to a "residual norm" rather than to a "cost function," in order to emphasize the application to nonlinear equation solving rather than to function minimization.

Overview of algorithms

Suppose an equilibrium problem solver rejects the Newton-Raphson solution $x_{[k+1]:NR}$-- perhaps because it induces an error in a residual evaluation, or because, violating Equation 4.1.2, it increases a residual norm. As in Chapter 3, the algorithm can perform a line search,

(4.1.8)
$$x_{[k+1]} = x_{[k]} - \mu_{[k]} d_{[k]}$$

choosing $\mu_{[k]}$ to find a point which sets $\emptyset_{[k+1]} < \emptyset_{[k]}$, and possibly which satisfies some other conditions meant to insure sufficient decrease. Besides the gradient, conjugate gradient, and Newton search directions, the equation solver can use the Newton-Raphson direction, $d_{[k]:NR} = -\Delta x_{[k]:NR}$, or

(4.1.9)
$$d_{[k]:NR} = x_{[k]} - x_{[k+1]:NR} = J_{[k]}^{-1} r_{[k]}$$

(Recall that the vector $d_{[k]}$ points from $x_{[k+1]}$ back to $x_{[k]}$ because with μ positive the search takes the $-d_{[k]}$ direction.)

With $\hat{\mu}_{[k]} = 1$ as an initial stepsize estimate, Equations 4.1.8 and 4.1.9 prescribe a range of possible points from which to chose $x_{[k+1]}$, e.g. as the exact minimizer of the residual norm. In practice, the computational expense of an exact line search is unjustified far from x**, while near x** the Newton-Raphson iteration converges rapidly unassisted. Therefore a typical algorithm accepts a trial $\hat{x}_{[k+1]}$ if its associated residual norm satisfies some acceptable descent criterion. Otherwise, the algorithm backtracks, reducing μ to find an acceptable point [Dennis §6.3.2].

Trust region methods carry the expected length of a successful step forward from one iteration to the next. This avoids taking an overly ambitious step if the previous search required backtracking [Dennis §6.4]. In addition, the methods extend the trust length when the model performs well at a trial step. Trust region methods for equation solving adapt the methods of Chapter 3 not only by choosing a residual norm for use as a cost

function, but also by replacing the Newton with the Newton-Raphson direction [Dennis §6.5]. Section 4.2 shows that with the residuals linearized in the Jacobian, the Newton direction for $\emptyset_{(2)}$ is identical to the Newton-Raphson direction; this does not hold for all residual norms.

As in Chapter 3, an algorithm may generate search directions directly from the residual norm as well. Far from the solution, Newton-Raphson may diverge, so that only the descent requirement keeps the algorithm stable [Fletcher §6.2]. See for example the second starting point for the inverse tangent problem, presented in Figure 2.2.2. An equation solver cannot simply backtrack along the Newton-Raphson direction to satisfy the descent requirement, since a series of such minimizations may discover a singular Jacobian, making the algorithm converge to a non-stationary point (with a nonzero gradient, and a minimum only in some search directions) [Fletcher §6.1, §6.2]. Thus, the search path should incorporate the norm's direction of steepest descent for short steps, as for instance the hook and double dogleg paths of Section 3.6. Of course for the algebraic system a number of residual norms, and hence gradient directions, are possible.

The double dogleg method of equation solving [Dennis §6.4.2] combines the Newton-Raphson direction with the gradient of $r^T r/2$, selecting the actual step by a trust region algorithm. Because the double dogleg method combines the trust region and gradient direction approaches, and because it is widely used as a standard solution method for algebraic equations as well as for function minimization [Fletcher §6.2; see also the current Guide to Available Mathematical Software as described in Boisvert], this thesis uses it as a benchmark equation-solving algorithm.

Pitfalls

Constraining each iterate to have $\emptyset_{[k+1]} < \emptyset_{[k]}$ gives an equation solver the same convergence properties as the function minimization routine on which it is based. It may converge to a stationary point x*, at which

$$g^* = \nabla\emptyset\{x^*\} = 0$$

[Dennis §6.3.1], or it may stagnate in a flat region where the residual norm does not provide sufficient decrease to satisfy the algorithm's step selection mechanism, forcing it to take such small steps that successive iterates do not differ appreciably. Furthermore, requiring a descent at each step forces the iterative sequence to follow contours in the residual norm, and may slow convergence to a solution that an unmodified Newton-Raphson sequence would find directly.

These last two possibilities-- stagnation, and contour-following-- introduce practical objections to using the descent requirement, and later chapters discuss them in the context of the observed behavior of implemented algorithms. The first effect-- convergence to a stationary point-- represents both the utility and the disincentive of the descent requirement. Clearly a well-constructed and reasonable norm has a minimum, and therefore a stationary point, at $r = 0$. However, the norm may have other stationary points $x^* \neq x^{**}$ which do not solve the algebraic system. Equation 3.2.14 states that x* is a unique local minimum of $\emptyset$ when G* is positive-definite, but does not guarantee that the descent-based method finds the norm's global minimum (which solves the equations), or even that it finds a minimum-- it may also find a saddle point, after the manner of steepest descent convergence [Dennis §6.2]. Therefore a local minimum or saddle point can trap the search away from the solution x** [Fletcher §6.2, Dennis §6.5, Press §9.6].

Unfortunately any descent strategy is likely to seek out such points [Dennis §6.5]. Near an $x^* \neq x^{**}$, the algorithm may step free of (or "burrow out" of) the local minimum, depending on its rules for selecting a step length and direction. That is, it can escape local minima provided its step selection rules happen onto a satisfactory point away from x^*. If not, the algorithm steps closer to x^* in order to meet the descent requirement.

To compound the problem, the linearized system becomes singular at such local minima (the converse may not be true-- the system may be singular without creating a local minimum in a particular norm). In the common r-square norm, this occurs because $g_{(2)} = 2J^T r$ [Fletcher §6.1], so $g_{(2)} = 0$ implies either $r = 0$ or J singular. Note however that a singular Jacobian does not force $g_{(2)} = 0$. This possibility forces a descent-based Newton-Raphson method to include the gradient direction in its search path, so that diverging Newton-Raphson steps near a singular point (discussed in Section 2.2) do not trap the search at a nonstationary point [Fletcher §6.2].

These comments apply to norms other than r-square. Section 4.3 shows that the algebraic form of any residual norm $\emptyset\{r\}$ couples its gradient to the Jacobian. Thus any norm may have stationary points where the Jacobian is singular, and which attract a descent-based solver. Newton-Raphson's method becomes unstable near any such point, so the descent requirement, instituted to avoid divergence in individual Newton-Raphson iterations, may destabilize the method as a whole.

Example: Freudenstein and Roth function

The Freudenstein and Roth function, Equation 1.5.5a, illustrates some difficulties of tying the equilibrium problem to a residual norm. It has local minima in $\emptyset_{(1)}$, $\emptyset_{(2)}$, and $\emptyset_{(\infty)}$, all at $(11.41, -0.8968)^T$. In addition, the Jacobian there is singular [Fletcher §6.2].

By inspection, the Jacobian in Equation 1.5.5b is singular when $-3x_2^2+10x_2-2 = 3x_2^2+2x_2-14$, or when $x_2 = (2\pm\sqrt{22})/3$.

For $\emptyset_{(2)} = r_1^2 + r_2^2$, the first component of the gradient $g_{1(2)} = \partial\emptyset_{(2)}/\partial x_1 = 2r_1(\partial r_1/\partial x_1) + 2r_2(\partial r_2/\partial x_1)$. Substituting derivatives from the first column of Equation 1.5.5b, $g_{1(2)} = 2(r_1 + r_2)$. Therefore to find a stationary point, $g_{(2)} = 0$, the first component requires $r_2 = -r_1$.

Similarly $g_{2(2)} = 2r_1(-3x_2^2+10x_2-2) + 2r_2(3x_2^2+2x_2-14)$. Substituting $r_2 = -r_1$, stationary points occur at $r_1 = r_2 = 0$, and wherever $r_2 = -r_1$ and $-3x_2^2+10x_2-2 = 3x_2^2+2x_2-14$. But this is just the same condition which gives a singular Jacobian. With $x_2 = (2\pm\sqrt{22})/3$, to have $r_2 = -r_1$ requires $x_1 = (53\pm4\sqrt{22})/3$.

Collecting, the point $x_{[k]} = (11.41, -0.8968)^T$ finds both a singular Jacobian and a local minimum in $\emptyset_{(2)}$. At this point, $r_{[k]} = (4.949, -4.949)^T$.

For $\emptyset_{(1)} = |r_1| + |r_2|$, $g_{1(1)} = \partial\emptyset_{(1)}/\partial x_1 = \text{sign}\{r_1\}\cdot(\partial r_1/\partial x_1) + \text{sign}\{r_2\}\cdot(\partial r_2/\partial x_1) = \text{sign}\{r_1\} + \text{sign}\{r_2\}$. Again a stationary point requires $r_2 = -r_1$. Similarly $g_{2(1)} = 0$ when $r_2 = -r_1$ and $x_2 = (2\pm\sqrt{22})/3$. The point $(11.41, -0.8968)^T$ is stationary in $\emptyset_{(1)}$ as well.

Finally for $\emptyset_{(\infty)} = \max\{|r_i|\}$, if $|r_1| > |r_2|$ then $g_{1(\infty)} = \text{sign}\{r_1\} = \pm1$, and if $|r_1| < |r_2|$ then $g_{1(\infty)} = \text{sign}\{r_2\} = \pm1$. However if $|r_1| = |r_2|$ then $g_{1(\infty)} = \text{sign}\{r_1\} + \text{sign}\{r_2\}$. Again if $r_2 = -r_1$ then the first component of the gradient is zero. Furthermore $g_{2(\infty)} = g_{2(1)}$ by the same reasoning, and the given point is stationary in $\emptyset_{(\infty)}$.

Application to building energy solvers

Chapter 2 motivates the descent requirement, Equation 4.1.2, in a Newton-Raphson based equilibrium problem solver by arguing the need to estimate the quality of a trial

solution $\hat{x}_{[k+1]}$. Once established, the norm provides a natural means of evaluating small changes in the estimated solution. This has immediate applications to building energy modeling, which uses piecewise residuals and component-based model definitions.

For a piecewise residual, whose defining relationship can change at discrete points along $d_{[k]}$, the residual norm allows meaningful comparison between the regimes of operation. Computationally this requires that the particular relationship defining each $r_i\{x\}$ must be determined completely by the current variables $x_{[k]}$. That is, a residual evaluation routine must detect the regime of operation using only the continuous variables appearing in the Jacobian. If not, then the residual evaluation requires an integer or enumeration variable to indicate the regime, placing the solution strategy outside the pure Newton-Raphson framework. Note that in a component-based modeling environment, one component may pass integer or enumeration variables to others, without violating the Newton-Raphson conditions; this requires only that the solver evaluate the component models in their proper order.

A more sophisticated solver can react to changes in residual regimes explicitly, even without controlling the regime directly. This requires that the residual routines signal mode changes to the solver, which might for example isolate the exact point at which the regime changes, then investigate new search directions on either side of the shift. In this case the solver must have some means to signal an accepted regime change back to the residual evaluation routine, so that successive guesses $\hat{x}_{[k+1]}$ all have their mode changes reported relative to $x_{[k]}$, rather than relative to the last attempt at $x_{[k+1]}$ [Sahlin §4].

The residual norm also allows the solver to evaluate the effect of an individual variable x_i explicitly. This suggests that a solver in a modular (component-based) environment can take advantage of sparsity in the system of equations, e.g. by changing a subset of the global variables and re-evaluating only the affected residuals. Thus, it could solve a component's local variables-- for example the friction factor f in the duct component of Section 1.4-- at every iteration, without special intervention by the component modeler.

4.2 R-Square

The standard trust region algorithms for nonlinear systems use r-square as their cost function [Dennis §6.4], as does the conjugate gradient method for linear systems [Strang86 §1.4]. These algorithms require the gradient and Hessian of r-square.

Gradient of r-square

Equation 3.2.1 defines the gradient of $\emptyset_{(2)} = \sum r_i^2$ as

$$g_{(2)} \equiv \nabla \emptyset_{(2)} = \begin{pmatrix} \frac{\partial \emptyset_{(2)}}{\partial x_1} \\ \frac{\partial \emptyset_{(2)}}{\partial x_2} \\ \vdots \\ \frac{\partial \emptyset_{(2)}}{\partial x_n} \end{pmatrix} = \begin{pmatrix} 2 \sum_{i=1}^{n} r_i \frac{\partial r_i}{\partial x_1} \\ 2 \sum_{i=1}^{n} r_i \frac{\partial r_i}{\partial x_2} \\ \vdots \\ 2 \sum_{i=1}^{n} r_i \frac{\partial r_i}{\partial x_n} \end{pmatrix}$$

The Jacobian matrix collects the $\partial r_i/\partial x_j$. For example, the first column of J contains the $\partial r_i/\partial x_1$. Thus the first element of $g_{(2)}$ is twice the inner product of r with the first column of J. The other elements follow suit, and

$$g_{(2)} = 2J^T r \tag{4.2.1}$$

[Fletcher §6.1; note $A = J^T$]. From Chapter 3, the gradient indicates slope in the cost function (see e.g. Equation 3.2.2); Equation 4.2.1 relates this slope not only to the algebraic system, represented by J, but also to the structure imposed by the chosen residual norm, loosely represented by the factor of 2 and by the inner product with r.

Note from Equation 4.2.1 that a local minimum $x^* \neq x^{**}$ of $\Phi_{(2)}$ has a singular Jacobian [Fletcher §6.2], since if $r \neq 0$ then $g_{(2)}^* = 0$ implies $J^T\{x^*\}$ has linearly dependent columns. Furthermore the method of Newton-Raphson becomes unstable in the neighborhood of x^* [Fletcher §6.2], as described in Chapter 2. However, a singular Jacobian does not necessarily give an extremum in $\Phi_{(2)}$. For a means of detecting nearly singular $J_{[k]}$ see [Dennis §6.5].

Initial slope

For a line search in an arbitrary direction $d_{[k]}$, Equation 3.2.2 gives the slope of Φ with respect to μ at any point along a search direction $d_{[k]}$ as $-g^T d_{[k]}$. The initial slope in $\Phi_{(2)}$, that is, its rate of change at $x_{[k]}$, has $g = g_{(2)[k]} = 2J_{[k]}^T r_{[k]}$. Combining,

$$\left.\frac{d\Phi_{(2)}}{d\mu}\right|_{x_{[k]}} = -2\cdot(J_{[k]}^T r_{[k]})^T d_{[k]} = -2\cdot r_{[k]}^T (J_{[k]} d_{[k]}) \tag{4.2.2a}$$

In the Newton-Raphson direction, $d_{[k]:NR} = J_{[k]}^{-1} r_{[k]}$ and

$$\left.\frac{d\Phi_{(2)}}{d\mu}\right|_{x_{[k]:NR}} = -2r_{[k]}^T r_{[k]} = -2\Phi_{(2)[k]} \tag{4.2.2b}$$

[Dennis §6.5; note $f = \frac{1}{2} r^T r$]. Thus the Newton-Raphson direction, if it exists, is a descent direction for $\Phi_{(2)}$. Geometrically this follows because the Newton-Raphson step zeros the local models of all the residuals, and so reduces the magnitude of every residual in the neighborhood of $x_{[k]}$. This property makes the Newton-Raphson choice a descent direction for any reasonable cost function, not just for $\Phi_{(2)}$.

In the steepest descent direction, $d_{[k]:SD} = g_{(2)[k]}$, the initial slope

$$\left.\frac{d\Phi_{(2)}}{d\mu}\right|_{x_{[k]:SD}} = -g_{(2)[k]}^T g_{(2)[k]} = -4\cdot\|J_{[k]}^T r_{[k]}\|^2 \tag{4.2.2c}$$

Hessian of r-square

Equation 3.2.6b gives the Hessian elements of $\Phi_{(2)}$ as

$$[G_{(2)}]_{i,j} = \frac{\partial g_j}{\partial x_i} = \frac{\partial}{\partial x_i}\left(2\sum_{m=1}^{n} r_m \frac{\partial r_m}{\partial x_j}\right) = 2\sum_{m=1}^{n}\left(\frac{\partial r_m}{\partial x_i}\frac{\partial r_m}{\partial x_j} + r_m\frac{\partial^2 r_m}{\partial x_i \partial x_j}\right) \tag{4.2.3a}$$

Therefore

(4.2.3b) $$G_{(2)} \equiv \nabla^2 \emptyset_{(2)} = 2\left(J^T J + \sum_{m=1}^{n} r_m \nabla^2 r_m\right)$$

[Fletcher §6.1]. Equation 4.2.3b expresses the curvature in $\emptyset_{(2)}$ as the sum of: (1) the curvature $2J^TJ$ due to the linearized residuals; and (2) the curvature $2\sum r_m \nabla^2 r_m$ due to second-order effects in the residuals [Dennis §10.1]. [Hiebert81 §2] discusses this model Hessian in connection with the least-squares minimization of overdetermined systems.

Truncating the Taylor series representation of the residuals after the second term, as in Equation 2.2.1 or Equation 2.2.8, drops their second-order effects and higher. When only the Jacobian derivative information remains, the Hessian of $\emptyset_{(2)}$ becomes

(4.2.4) $$G_{(2)} \approx 2J^TJ$$

Newton-Raphson's method, linearizing the residuals in J, ignores curvature in the r_i [Fletcher §6.2], and its residual models support only the approximate Hessian of Equation 4.2.4.

Note that both Equations 4.2.3b and 4.2.4 give symmetric matrices, as expected. In addition, by Equation 3.3.20 the matrix J^TJ is positive-definite if J is nonsingular, that is, if it has independent columns. This follows directly from the definition, since $x^T(J^TJ)x = (Jx)^T(Jx)$ is positive for all $x \neq 0$ unless Jx can be zero for nonzero x-- that is, unless J is singular [Strang88 §2.4]. By the same reasoning, even if J is singular the Hessian approximation of Equation 4.2.4 is positive-semidefinite.

If $J_{[k]}$ is nearly singular, [Dennis §6.5] recommends perturbing the model Hessian.

Neglected second-order terms

Suppose a single residual relation r_i is linear. Then its Hessian $\nabla^2 r_i = 0$, and it contributes nothing to the second-order curvature terms $S = \sum r_m \nabla^2 r_m$. Similarly, if $r_i = 0$, its contribution to S is zero. Therefore the common justification for neglecting S to approximate $G_{(2)} \approx 2J^TJ$ asserts either: (1) that the residual relations are nearly linear; or (2) that the residuals are small enough in magnitude to make the terms negligible.

A typical discussion of these neglected terms occurs in the context of the Gauss-Newton method for fitting data to a function in an overdetermined problem [Dahlquist §10.5.4, Dennis §10.1, Dennis §10.3, Fletcher §6.1]. For this method an additional justification is that, for a successful model, the r_i tend to be uncorrelated with x, and hence the terms in S tend to cancel one another [Press §15.5]. However, when fitting data to a parameterized equation, the structure of the $\nabla^2 r_i$ is identical for every residual. The argument does not apply when solving systems of nonlinear equations, since in general the residual equations bear no relation to each other.

Now suppose a descent-based equation solver requires $G_{(2)}$-- for example to compare the actual change in $\emptyset_{(2)}$ against its expected value, as in the double dogleg algorithm of Chapter 3. To include the second-order effects requires estimating n matrices $\nabla^2 r_i$, dramatically increasing the computational cost of estimating $G_{(2)}$ [Dennis §10.3]. Therefore the algorithm is likely to estimate $G_{(2)} \approx 2J^TJ$, resulting in a different quadratic model of $\emptyset_{(2)}$ than would be used by Newton's method to minimize $\emptyset_{(2)}$ [Dennis §6.5].

From Equation 3.2.9a, the quadratic model for pure minimization methods is

(4.2.5) $$f_{(2)[k]}\{x\} = \emptyset_{(2)[k]} + \Delta x_{[k]}^T g_{(2)[k]} + \frac{1}{2} \Delta x_{[k]}^T G_{(2)[k]} \Delta x_{[k]}$$

where $\Delta x_{[k]} = x - x_{[k]}$. When a descent-based equilibrium problem solver estimates the Hessian from the Jacobian, it models

$$f_{(2)[k]}\{x\} = r_{[k]}{}^T r_{[k]} + 2\Delta x_{[k]}{}^T (J_{[k]}{}^T r_{[k]}) + \Delta x_{[k]}{}^T J_{[k]}{}^T J_{[k]} \Delta x_{[k]} \tag{4.2.6}$$

Numerically these models are identical in the first and second terms, which use function and first derivative information only; but differ in the third, Hessian, term, since $\nabla(r^T r) \neq 2J^T J$.

Computationally the models may be quite different, even in the first and second terms. Newton's method defines neither r nor J. If the residuals exist at all, they exist only as "helper" variables in the routine which evaluates $\emptyset_{(2)}$. Since Newton's method treats $\emptyset_{(2)}$ directly, rather than forming it out of an underlying equilibrium problem, it has no option of approximating $G_{(2)}$ according to Equation 4.2.4.

Convergence to nonstationary point

A descent-based method searching only the Newton-Raphson direction may converge to a fixed point in $f_{(2)}$ which is not a stationary point in $\emptyset_{(2)}$ [Fletcher §6.2]. As the method converges to such a point, the Newton-Raphson steps become numerically orthogonal to $g_{(2)}$ until finally the method makes no progress and the Jacobian becomes singular to machine precision [Fletcher §6.1]. Again, from Equation 4.2.1 a singular Jacobian does not imply a stationary point in $\emptyset_{(2)}$, though the converse is true.

The fact that descent-based searches in the Newton-Raphson direction may converge to a nonstationary point in $\emptyset_{(2)}$ suggests the gradient $g_{(2)}$ as an alternate search direction. This motivates the double dogleg algorithm without refering to the minimization theory of Chapter 3. In the double dogleg algorithm, searching the steepest descent direction for small step lengths ensures convergence to a stationary point of $\emptyset_{(2)}$.

Newton versus Newton-Raphson direction

Suppose a nonlinear algebraic equation solver approximates $G_{(2)}$ by Equation 4.2.4, forming the quadratic model of Equation 4.2.6. A Newton step on this model finds $\Delta x_{(2)[k]:N}$ by solving $G_{(2)[k]} \Delta x_{(2)[k]:N} = -g_{(2)[k]}$ or

$$2J_{[k]}{}^T J_{[k]} \Delta x_{(2)[k]:N} = -2J_{[k]}{}^T r_{[k]}$$

If the Newton-Raphson step exists, $J_{[k]}{}^{-1}$ exists and then $(J_{[k]}{}^T)^{-1} = (J_{[k]}{}^{-1})^T$ exists [Strang88 §1.6]. Therefore the Newton step for $\emptyset_{(2)}$ solves

$$J_{[k]} \Delta x_{(2)[k]:N} = -r_{[k]}$$

when $G_{(2)}$ is approximated by the linearized residuals. But this is just the Newton-Raphson step defined in Equation 2.2.9a. Therefore

$$\Delta x_{(2)[k]:N} = \Delta x_{(2)[k]:NR} \tag{4.2.7}$$

and, under the linearized residual approximation, the Newton's method minimizer of $f_{(2)[k]}$ is the same as the Newton-Raphson solution of $\hat{r}_{[k]}$. Chapter 5 uses this fact to adapt the double dogleg method of function minimization to equation solving, without modifying the algorithm to ensure the Cauchy length does not exceed the length of the Newton-Raphson step.

Algebraically, the Newton-Raphson step minimizes Equation 4.2.6 because dropping the second-order terms $\nabla^2 r_m$ from the Hessian gives $f_{(2)[k]} = \hat{r}^T \hat{r}$. Thus the solution to $\hat{r}_{[k]} = 0$ minimizes $f_{(2)[k]}$. This in itself does not guarantee Equation 4.2.7--

the Newton-Raphson solution minimizes any function of $\hat{r}$ constructed to have a minimum at $\hat{r} = 0$. For a Newton step on a function $\emptyset\{\hat{r}\}$ to equal the Newton-Raphson step, it must find the minimum of $\emptyset$ in one step.

Since one iteration of Newton's method exactly minimizes only a pure quadratic-- that is, a function with a constant Hessian-- the Newton step equals the Newton-Raphson step only for quadratic $\emptyset\{\hat{r}\}$. Equation 4.2.7 holds because $\hat{r}^T\hat{r}$ has constant curvature. Thus, performing a Newton step on $\emptyset_{(2)}$ without first dropping the $\nabla^2 r_m$ in the model Hessian-- that is, minimizing Equation 4.2.5 rather than Equation 4.2.6-- does not give the Newton-Raphson step.

The point is important because without equality the Newton-Raphson step may not substitute meaningfully for the Newton step in a second-order minimization algorithm. The literature does not stress this precondition: [Fletcher §6.2] does not state the equality; while [Dennis §6.5; note $M_c = \hat{r}$ and $\hat{m}_c = \hat{r}^T\hat{r}/2$] asserts that because the Newton-Raphson step gives the root of the linearized residuals, it uniquely minimizes $\emptyset\{\hat{r}\}$ (correct), and therefore matches the Newton step on $\emptyset\{\hat{r}\}$ (incorrect without the coupled assertion that $\emptyset$ is quadratic).

Computationally, finding the Newton-Raphson step is easier since it does not require forming $J_{[k]}^T J_{[k]}$. Furthermore, since the condition number of $J_{[k]}^T J_{[k]}$ is the square of the condition number of $J_{[k]}$ [Press §2.7], solving for $\Delta x_{(2)[k]:N}$ introduces greater roundoff error than does solving for $\Delta x_{[k]:NR}$ [Strang88 §7.2]. Thus, the Newton-Raphson solution is preferred computationally.

Reformulated quadratic model

The common subexpressions from Equation 4.2.6, $J^T r$ and $J\Delta x$, recur throughout the results derived from the quadratic model. A computational scheme should take advantage of these subexpressions. For instance, calculating the third term $\Delta x^T J^T J \Delta x$ as $b^T b$ where $b = J\Delta x$ requires one matrix-vector multiplication, one vector-vector inner product, and additional storage for one vector. A literal reading of Equation 4.2.6 would either: (1) calculate $\Delta x^T(J^T b)$, adding another matrix-vector multiplication; or (2) calculate $\Delta x^T(J^T J)\Delta x$, adding a matrix-matrix multiplication and storage for a matrix.

Many of the expressions presented below can be calculated two ways, one favoring $J^T r$ and the other $J\Delta x$. Since presumably matrix-vector multiplication is more efficient with the matrix untransposed, and since $J\Delta x$ necessarily appears in the second-order term, the expressions below favor $J\Delta x$ over $J^T r$. To bring out the structure in these equations, the subexpressions are grouped as e.g. $(J\Delta x)_{[k]}$ rather than as $J_{[k]}\Delta x_{[k]}$. In this notation the quadratic model becomes

$$f_{(2)[k]}\{x\} = r_{[k]}^T r_{[k]} + 2\Delta x_{[k]}^T (J^T r)_{[k]} + (J\Delta x)_{[k]}^T (J\Delta x)_{[k]} \tag{4.2.8a}$$

or

$$f_{(2)[k]}\{x\} = r_{[k]}^T r_{[k]} + 2(J\Delta x)_{[k]}^T r_{[k]} + (J\Delta x)_{[k]}^T (J\Delta x)_{[k]} \tag{4.2.8b}$$

where $\Delta x_{[k]} = x - x_{[k]}$. In addition, substituting $\Delta x_{[k]} = -\mu d_{[k]}$ from Equation 4.1.8 provides a third form, useful for line searches.

Predicted stepsize

Using Equation 4.2.4 in Equation 3.2.11a, the estimated stepsize which minimizes $f_{(2)[k]}$ along a descent direction $d_{[k]}$ is

(4.2.9a) $$\hat{\mu}_{(2)[k]} = \frac{d_{[k]}{}^T(J^Tr)_{[k]}}{(Jd)_{[k]}{}^T(Jd)_{[k]}} = \frac{(Jd)_{[k]}{}^Tr_{[k]}}{(Jd)_{[k]}{}^T(Jd)_{[k]}}$$

For the Newton-Raphson direction, $d_{[k]} = J_{[k]}{}^{-1}r_{[k]}$ and

(4.2.9b) $$\hat{\mu}_{(2)[k]:NR} = 1$$

This result follows directly from Equation 4.1.9 since the full Newton-Raphson step, where $\mu = 1$, zeros the individual residual models.

In the steepest descent direction, $d_{[k]} = g_{(2)[k]} = 2J_{[k]}{}^Tr_{[k]}$, and

(4.2.9c) $$\hat{\mu}_{(2)[k]:SD} = \frac{1}{2}\frac{\|(J^Tr)_{[k]}\|^2}{\|(JJ^Tr)_{[k]}\|^2}$$

Note the new subexpression, JJ^Tr. The estimated minimizer in the steepest descent direction lies at $\Delta x_{(2)[k]:SD} = -\hat{\mu}_{(2)[k]:SD}\cdot g_{(2)[k]}$ or

(4.2.10) $$\Delta x_{(2)[k]:SD} = \frac{-\|(J^Tr)_{[k]}\|^2}{\|(JJ^Tr)_{[k]}\|^2}(J^Tr)_{[k]}$$

The notation $\Delta x_{(2)[k]:SD}$ represents the Cauchy point of $f_{(2)[k]}$. Recall the Cauchy point minimizes the quadratic model, rather than the cost function itself, in the steepest descent direction. Furthermore in this case the Cauchy point minimizes the approximate quadratic, with $G_{(2)} \approx 2J^TJ$.

Predicted change in cost function

Following Equation 3.6.4a, the predicted change in $\emptyset_{(2)}$, $f_{(2)[k]}\{x_{[k+1]}\} - \emptyset_{(2)[k]}$, is the expected increase, not the expected decrease, in the cost function. From Equation 4.2.8b,

(4.2.11a) $$\Delta f_{(2)[k]} = 2(J\Delta x)_{[k]}{}^Tr_{[k]} + (J\Delta x)_{[k]}{}^T(J\Delta x)_{[k]}$$

In the Newton-Raphson direction, $\Delta x = -\mu J^{-1}r$, this gives $-2\mu r_{[k]}{}^Tr_{[k]} + \mu^2 r_{[k]}{}^Tr_{[k]}$, or

(4.2.11b) $$\Delta f_{(2)[k]:NR} = (\mu^2 - 2\mu)\cdot\emptyset_{(2)[k]}$$

Note at $\mu = 1$, $\Delta f_{(2)[k]:NR} = -\emptyset_{(2)[k]}$ predicts $\emptyset_{(2)[k+1]} = 0$. Again, the Newton-Raphson step minimizes the quadratic model of Equation 4.2.6.

In the steepest descent direction, $\Delta x = -2\mu J^Tr$,

(4.2.11c) $$\Delta f_{(2)[k]:SD} = 4\mu^2\cdot\|(JJ^Tr)_{[k]}\|^2 - 4\mu\cdot\|(J^Tr)_{[k]}\|^2$$

Special forms for double dogleg algorithm

The double dogleg algorithm of Chapter 3 minimizes a general cost function $\emptyset$. For later use in adapting the algorithm to the solution of algebraic systems, let $\emptyset = \emptyset_{(2)}$ and replace the Newton with the Newton-Raphson step; by Equation 4.2.7, the resulting step is identical provided the Newton step estimates $\emptyset_{(2)}$ using the linearized residuals (i.e., provided it ignores the $\nabla^2 r_m$ in Equation 4.2.3b).

The algorithm defines the curve by linear combinations of the Cauchy and Newton-Raphson steps, choosing trial points on the curve to match the trust length (see Figure 3.6.1). The first break point on the curve occurs at the Cauchy point, the steepest descent minimizer of the model, $f_{(2)[k]}$. From Equation 4.2.10, the Cauchy length

(4.2.12) $$\|\Delta x_{(2)[k]:SD}\| = \frac{\|(J^T r)_{[k]}\|^3}{\|(JJ^T r)_{[k]}\|^2}$$

The cutback point, where the double dogleg curve rejoins the Newton-Raphson direction, lies at $\Delta x_{[k]:CB} = \eta_{[k]} \cdot \Delta x_{[k]:NR}$ and has length $\eta_{[k]} \cdot \|\Delta x_{[k]:NR}\|$; there is no closed-form expression for the length of the Newton-Raphson step. From Equation 3.6.2b, the algorithm sets

(4.2.13) $$\eta_{[k]} = 0.2 + 0.8 \frac{\|(J^T r)_{[k]}\|^4}{\|(JJ^T r)_{[k]}\|^2 \cdot \emptyset_{(2)[k]}}$$

When the trust length falls between the Cauchy and cutback lengths, the double dogleg algorithm chooses

(4.2.14a) $$\Delta x_{[k]} = (1 - \lambda_{[k]}) \cdot \Delta x_{(2)[k]:SD} + \lambda_{[k]} \cdot \eta_{[k]} \cdot \Delta x_{[k]:NR}$$

where $0 < \lambda < 1$. Given a target step length $\delta_{[k]}$ between the Cauchy and cutback lengths, take the squared lengths of both sides of Equation 4.2.14a and rearrange to find

(4.2.14b) $$\lambda_{[k]}^2 \cdot \left(1 + \frac{\|\Delta x_{[k]:CB}\|^2}{\|\Delta x_{(2)[k]:SD}\|^2} - 2\eta_{[k]} \frac{\|(JJ^T r)_{[k]}\|^2}{\|(J^T r)_{[k]}\|^4} \emptyset_{(2)[k]}\right) + \lambda_{[k]} \cdot \left(2\eta_{[k]} \frac{\|(JJ^T r)_{[k]}\|^2}{\|(J^T r)_{[k]}\|^4} \emptyset_{(2)[k]} - 2\right) + \left(1 - \frac{\delta_{[k]}^2}{\|\Delta x_{(2)[k]:SD}\|^2}\right) = 0$$

since $\Delta x_{(2)[k]:SD}{}^T \Delta x_{[k]:NR} = 2 \cdot \hat{\mu}_{(2)[k]:SD} \cdot \emptyset_{(2)[k]}$. Identifying the quadratic as

$$\lambda^2 \cdot (c_1 - c_2) + \lambda \cdot c_2 + c_3 = 0$$

note that with $\|\Delta x_{SD}\| < \delta < \|\Delta x_{CB}\|$ it follows that $c_3 < 0$, that $c_1 > 0$, and that $c_1 > -c_3$. To relate c_2, note that

$$2\eta \frac{\|JJ^T r\|^2}{\|J^T r\|^4} \emptyset_{(2)} = 2\eta \frac{\|JJ^T r\|^2}{\|J^T r\|^4} (r^T J \cdot J^{-1} r) \le 2\eta \frac{\|JJ^T r\|^2}{\|J^T r\|^3} \|J^{-1} r\|$$

$$1 + \frac{\|\Delta x_{CB}\|^2}{\|\Delta x_{SD}\|^2} - 2\eta \frac{\|JJ^T r\|^2}{\|J^T r\|^4} \emptyset_{(2)} \ge 1 - 2\eta \frac{\|\Delta x_{NR}\|}{\|\Delta x_{SD}\|} + \frac{\|\Delta x_{CB}\|^2}{\|\Delta x_{SD}\|^2}$$

$$c_1 - c_2 \ge \left(1 - \frac{\|\Delta x_{CB}\|}{\|\Delta x_{SD}\|}\right)^2 > 0$$

so that $c_1 > c_2$.

Applying the quadratic formula,

$$\lambda = \frac{-c_2 \pm \sqrt{c_2^2 + 4c_2c_3 - 4c_1c_3}}{2(c_1 - c_2)}$$

Since $c_1 > -c_3$, and with $c_3 < 0$, $-4c_1c_3 > 4c_3^2$. Therefore the determinant $D > c_2^2 + 4c_2c_3 + 4c_3^2 = (c_2 + 2c_3)^2 > 0$. Thus the square root is always defined.

Now consider whether to take the positive or the negative square root. To find $\lambda > 0$ with $c_1 > c_2$ requires a positive numerator, $-c_2 \pm |\sqrt{D}| > 0$. If $c_2 > 0$ this requires the

positive square root. If on the other hand $c_2 < 0$, then $|\sqrt{D}| > -(c_2 + 2c_3)$. Taking the negative square root gives $-c_2 + (c_2 + 2c_3) = 2c_3 < 0$. Therefore, over the cutback part of the double dogleg curve, find λ from Equation 4.2.14b using the positive square root in the quadratic formula.

Compared to the rule used to find λ for a general minimization problem (i.e., compared to the double dogleg algorithm for a general cost function), Equation 4.2.14b saves a vector-vector inner product, a vector sum of squares, and two scalar vector operations [Dennis §A.IV.3.A6.4.4].

The double dogleg algorithm evaluates the progress of a step by projecting the initial slope over the entire step length, to find $g^T\Delta x$. Consider the linear combination of the Cauchy and Newton-Raphson steps,

$$\Delta x_{[k]} = \rho_{1[k]} \cdot \Delta x_{(2)[k]:SD} + \rho_{2[k]} \cdot \Delta x_{[k]:NR} \tag{4.2.15}$$

On the first leg of the double dogleg curve, $\rho_2 = 0$, while on the last leg $\rho_1 = 0$. For the cutback path between the two break points, $\rho_1 = 1 - \lambda$ and $\rho_2 = \lambda \cdot \eta$. Then with $\Delta x_{(2)[k]:SD} = -\hat{\mu}_{(2)[k]:SD} \cdot g_{(2)[k]}$ and $\Delta x_{[k]:NR} = -J_{[k]}^{-1} r_{[k]}$,

$$g_{(2)[k]}^T \Delta x_{[k]} = -2\rho_{1[k]} \frac{\|(J^T r)_{[k]}\|^4}{\|(JJ^T r)_{[k]}\|^2} - 2\rho_{2[k]} \cdot \emptyset_{(2)[k]} \tag{4.2.16}$$

for a linear combination of the Cauchy and Newton-Raphson steps.

The term $g_{(2)[k]}^T \Delta x_{[k]}$ recurs in the expressions for $\Delta f_{(2)[k]}$:

$$\Delta f_{(2)[k]} = g_{(2)[k]}^T \Delta x_{[k]} + \|(J\Delta x)_{[k]}\|^2$$

Substituting Equation 4.2.15, the second term expands as

$$\|(J\Delta x)_{[k]}\|^2 = \|(-2\rho_{1[k]} \cdot \hat{\mu}_{(2)[k]:SD} \cdot (JJ^T r)_{[k]} - \rho_{2[k]} \cdot r_{[k]})\|^2$$

or as

$$\|(J\Delta x)_{[k]}\|^2 = 4(\rho_{1[k]} \cdot \hat{\mu}_{(2)[k]:SD})^2 \cdot \|(JJ^T r)_{[k]}\|^2 + 4\rho_{1[k]} \cdot \rho_{2[k]} \cdot \hat{\mu}_{(2)[k]:SD} \cdot \|(J^T r)_{[k]}\|^2 + \rho_{2[k]}^2 \cdot \emptyset_{(2)[k]}$$

Note that if $\rho_1 = 0$, the result matches Equation 4.2.11b for the Newton-Raphson direction, since $\mu = \rho_2$ in that equation. Furthermore, if $\rho_2 = 0$, the result matches Equation 4.2.11c for the steepest descent direction, since $\mu = \rho_1 \cdot \hat{\mu}_{(2)[k]:SD}$ in that equation. Substituting Equation 4.2.9c for the optimizing stepsize in the steepest descent direction,

$$\Delta f_{(2)[k]} = g_{(2)[k]}^T \Delta x_{[k]} + (\rho_{1[k]}^2 + 2\rho_{1[k]} \cdot \rho_{2[k]}) \frac{\|(J^T r)_{[k]}\|^4}{\|(JJ^T r)_{[k]}\|^2} + \rho_{2[k]}^2 \cdot \emptyset_{(2)[k]} \tag{4.2.17a}$$

or, using Equation 4.2.16,

$$\Delta f_{(2)[k]} = \rho_{1[k]} \cdot (\rho_{1[k]} + 2\rho_{2[k]} - 2) \frac{\|(J^T r)_{[k]}\|^4}{\|(JJ^T r)_{[k]}\|^2} + (\rho_{2[k]}^2 - 2\rho_{2[k]}) \cdot \emptyset_{(2)[k]} \tag{4.2.17b}$$

for a linear combination of the Cauchy and Newton-Raphson steps. Compared to the defining relations, Equations 4.2.16 and 4.2.17 save a matrix-vector product, a vector-vector inner product, and a vector sum of squares.

4.3 Unspecified Residual Norm

Consider enforcing the descent requirement on a single iteration of a nonlinear equation-solving algorithm. Rather than r-square, use a general residual norm, $\emptyset$. To make explicit the role that the linear residual models from equation-solving theory play in the function minimization methods, this section reformulates the minimization results to include the Jacobian matrix. In this it follows Section 4.2, which expresses some useful results from minimization theory directly in terms of the Jacobian, for the specific norm r-square.

Expanding $\emptyset$ in r

The minimization methods require the derivatives $\partial\emptyset/\partial x_i$ and $\partial^2\emptyset/\partial x_i\partial x_j$ of the norm with respect to the independent variables, x. Since the Jacobian matrix already contains the derivatives $\partial r_i/\partial x_j$, reformulating the minimization results to show the Jacobian requires the derivatives such as $\partial\emptyset/\partial r_i$ and $\partial^2\emptyset/\partial r_i\partial r_j$. Accordingly, let ∇_r represent the gradient with respect to the variables r-- the operator ∇ still represents the gradient with respect to the independent vector space, x. Then the gradient in r,

(4.3.1)
$$p \equiv \nabla_r\emptyset = \begin{pmatrix} \frac{\partial\emptyset}{\partial r_1} \\ \frac{\partial\emptyset}{\partial r_2} \\ \vdots \\ \frac{\partial\emptyset}{\partial r_n} \end{pmatrix}$$

is zero at a minimum of $\emptyset\{r\}$. Following Equation 3.2.6, the Hessian of $\emptyset$ with respect to r is

(4.3.2a)
$$P \equiv \nabla_r p = [\nabla_r p_1\ \nabla_r p_2\ ..\ \nabla_r p_n]$$

with elements

(4.3.2b)
$$[P]_{i,j} = \frac{\partial p_j}{\partial r_i} = \frac{\partial^2\emptyset}{\partial r_i\partial r_j}$$

Like $G = \nabla^2\emptyset\{x\}$, $P = \nabla_r^2\emptyset\{r\}$ is symmetric when the appropriate second partial derivatives of $\emptyset$ are continuous. Unlike G, where continuity of the derivatives depends on the residual equations, continuity for P depends only on the algebraic form of the residual norm.

Since p and P depend only on $\emptyset\{r\}$, expanding $\emptyset$ by Taylor series in terms of the residuals gives

(4.3.3)
$$\emptyset\{r_{[k+1]}\} \approx \emptyset\{r_{[k]}\} + \Delta r_{[k]}^T p_{[k]} + \tfrac{1}{2}\Delta r_{[k]}^T P_{[k]}\Delta r_{[k]}$$

where $\Delta r_{[k]} = r_{[k+1]} - r_{[k]}$.

Linearized residual approximation

Linearizing the residual equations using Equation 2.2.8,

$$\hat{r}_{[k]}\{x\} = r_{[k]} + J_{[k]}(x - x_{[k]})$$

gives $\Delta r_{[k]} \approx J_{[k]}\Delta x_{[k]}$. Substituting in Equation 4.3.3,

(4.3.4) $$\emptyset_{[k+1]} \approx \emptyset_{[k]} + \Delta x_{[k]}{}^T(J_{[k]}{}^T p_{[k]}) + \frac{1}{2}\Delta x_{[k]}{}^T(J_{[k]}{}^T P_{[k]} J_{[k]})\Delta x_{[k]}$$

where $\Delta x_{[k]} = x_{[k+1]} - x_{[k]}$. Comparing to Equation 3.2.9a, the linearized residual approximation gives $g = J^T p$ and $G = J^T P J$. These relations are developed from the defining equations below.

General gradient

From Equation 3.2.1, the gradient has elements

$$(g)_j = \frac{\partial \emptyset}{\partial x_j} = \sum_{m=1}^{n} \frac{\partial \emptyset}{\partial r_m}\frac{\partial r_m}{\partial x_j}$$

The j^{th} element is the inner product of p with the j^{th} column of the Jacobian, so

(4.3.5) $$g = J^T p \quad \text{or} \quad \nabla \emptyset = J^T \nabla_r \emptyset$$

This result exactly matches the gradient implied by the linearized residual approximation in Equation 4.3.4.

Equation 4.3.5 gives the derivatives g of $\emptyset\{x\}$, in terms of: (1) the derivatives J of $r\{x\}$; and (2) the derivatives p of $\emptyset\{r\}$. Where J represents the algebraic form of the residual equations, p represents the way in which $\emptyset$ combines the residuals to form its scalar measure of the residual vector.

Significantly, if a descent-based method finds a stationary point in $\emptyset$ where $g = J^T p = 0$, then $p \neq 0$ implies a singular Jacobian. A descent-based method, seeking out local minima in $\emptyset$, may also seek out singularities in the Jacobian.

Geometric interpretation of general gradient

Equation 4.3.5, derived by considering a single component of g, also specifies how to treat a single component of p: its i^{th} element multiplies the i^{th} column of J^T, or the i^{th} row of J. Equation 2.2.7b gives this row as the gradient of the i^{th} residual, so

(4.3.6) $$g = \sum_{i=1}^{n} p_i \cdot \nabla r_i$$

In other words, the gradient of the residual norm is the weighted sum of gradients of the residuals, with the weights determined by the algebraic form of the residual norm.

Each of the gradients ∇r_i gives the direction of steepest descent for an independent function $r_i\{x\}$. Equation 4.3.6 interprets the net gradient-- that is, the direction of steepest descent for the cost function-- as the weighted vector sum of these gradients, with the weights coming from the corresponding element of p. Different cost functions simply weight the residual gradients differently in the summation.

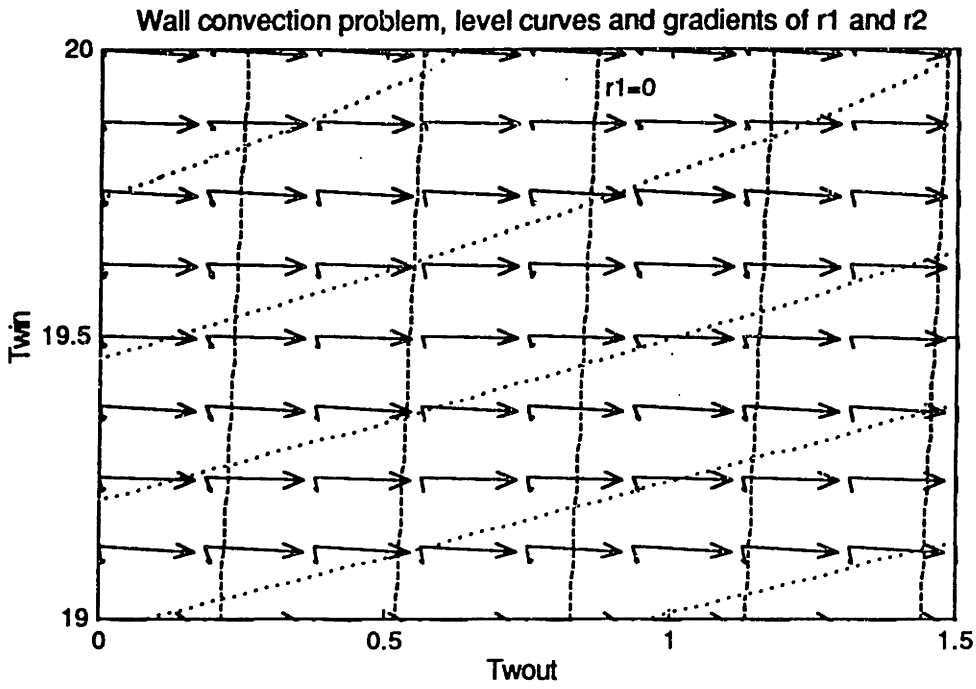

Figure 4.3.1. Sample gradients and iso-residual lines for the wall convection problem. The dashed and dotted lines give level curves of r_1 and r_2, respectively. The gradients, respective rows of the Jacobian matrix, are scaled to fit the frame of the drawing.

Figure 4.3.1 shows some gradients, along with contours, for the two residuals of the wall convection problem of Chapter 1. The gradients are perpendicular to the level curves, according to the discussion in Chapter 3, and point towards more positive residual values. For example, the first, linear, residual, $r_1 = 13.05 \cdot T_{wout} - 0.5678 \cdot T_{win}$ from Equation 1.5.10a, is more positive for more positive T_{wout}; in Figure 4.3.1 the gradients of r_1, $\nabla r_1 = (13.05, -0.5678)^T$, are parallel, constant, and point towards more positive T_{wout}. (The gradients of r_2, a nonlinear relation, are neither parallel nor of constant magnitude; nor are its level curves parallel.)

According to Equation 4.3.6, to find the gradient of a cost function constructed from r_1 and r_2, sum their gradients at each point in Figure 4.3.1 according to the weights in p. From Equation 4.2.1, setting $p_i = p_{i(2)} = 2 \cdot r_i$ gives the gradient of $\emptyset_{(2)}$ at any point. Figure 4.3.2 shows the weighted sums, along with some contours of $\emptyset_{(2)}$.

Comparing Figures 4.3.1 and 4.3.2, note that on opposite sides of the level curve $r_1 = 0$, the gradients of $\emptyset_{(2)}$ point in opposite directions, since the weighting of ∇r_1 changes sign. Furthermore, farther from $r_1 = 0$, as $|r_1|$ increases, the length of the gradients of $\emptyset_{(2)}$ increases, and their direction becomes dominated by the direction of ∇r_1. Conversely, near $r_1 = 0$, the direction of ∇r_2 dominates that of $\nabla \emptyset_{(2)}$.

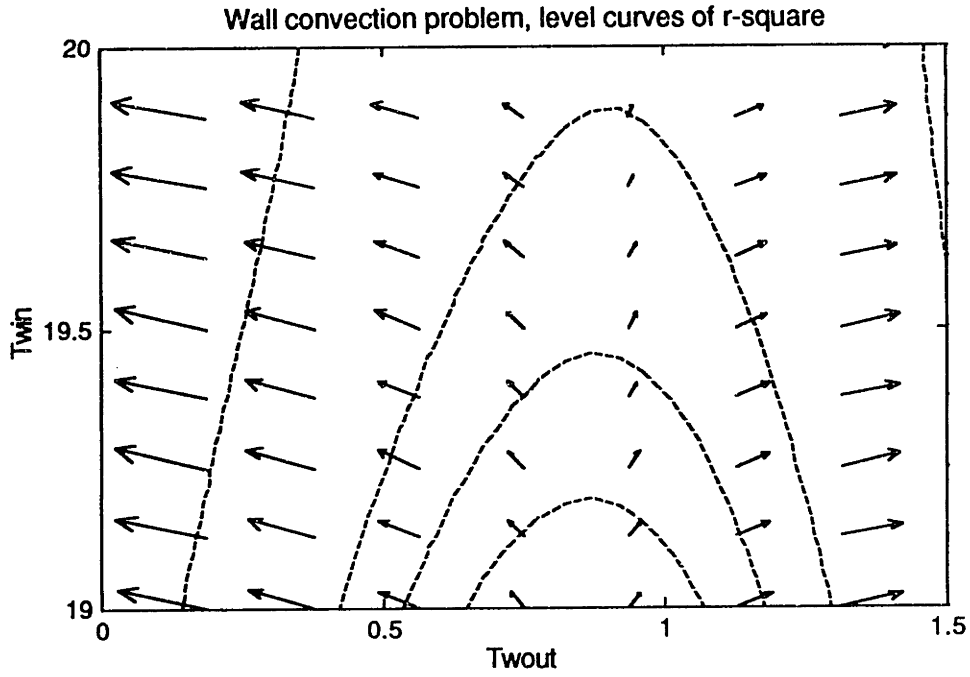

Figure 4.3.2. Level curves and scaled sample gradients for $\emptyset_{(2)}$ on the wall convection problem. For a contour plot depicting a larger area, see Figure 2.2.3.

Constructing a norm from its gradient

The geometric interpretation of Equation 4.3.6 suggests the possibility constructing a residual norm by choosing an appropriate weighting scheme $p\{r\}$, and integrating to obtain $\emptyset$ (see Equation 4.3.1).

Considering Figures 4.3.1 and 4.3.2, a reasonable weighting scheme: (1) gives p_i the same sign as r_i, so that at least the contribution of ∇r_i to the net gradient tends to reduce the magnitude of r_i, even if the net gradient direction does not; (2) sets $p_i = 0$ when $r_i = 0$, so that at least the contribution of ∇r_i to the net gradient does not tend to change r_i, even if the net gradient direction does change r_i; and (3) makes p_i monotonically nondecreasing with $|r_i|$, so that at least the contribution of ∇r_i to the net gradient does not diminish as the need to travel in that direction increases.

These conditions, along with the desire to keep the integration from p to $\emptyset$ tractable, imply another property, that a reasonable weighting scheme: (4) makes $p_i = p_i\{r_i\}$, to avoid dependence on the other residuals. In this case, each differential equation $p_i = \partial\emptyset/\partial r_i$ may be integrated independently of the others, making the cost function a sum of n terms, each depending on exactly one residual:

$$\emptyset = \sum_{i=1}^{n} t_i\{r_i\} \tag{4.3.7}$$

Note that a $\emptyset\{r\}$ satisfying Equation 4.3.7, and which has a minimum only at $r = 0$, satisfies the property that $p_i = 0$ when $r_i = 0$, since any minimum makes $\partial\emptyset/\partial r_i = p_i = 0$, so long as the derivative exists.

The rule $p_{i(2)} = 2 \cdot r_i$ inferred from Equation 4.2.1 for $\emptyset_{(2)}$ meets these tests for reasonableness. In particular, comparing Figures 4.3.1 and 4.3.2 illustrates the advisability

of keeping sign{p_i} = sign{r_i}, since the residual gradients point in the same direction on either side of the contour $r_1 = 0$, while the gradients of $\emptyset_{(2)}$ in general should point away from $\emptyset_{(2)} = 0$, at the intersection of $r_1 = 0$ and $r_2 = 0$.

Escaping a stationary point

Suppose a descent-based method converges to a local minimum x^* for which $r\{x^*\} \neq 0$. From Equation 4.3.5, a stationary point $g = 0$ can occur: (1) when $p = 0$; or (2) when p is in the left null space of J, that is, when $J^Tp = 0$ for nonzero p [Strang88 §2.4].

The first possibility can be prevented by choosing $\emptyset\{r\}$ reasonably, i.e., with a unique minimum at $r = 0$. Then $p = 0$ implies $x^* = x^{**}$. The second possibility for $g = 0$, that $J^Tp = 0$, indicates the weighted vector sum of the residual gradients is zero. This possibility gives rise to those local minima which can trap a descent-based equation solver away from the solution.

Suppose the solver converges to such a minimum, for example a local minimum in $\emptyset_{(2)}$, such that $J_{[k]}{}^Tp_{(2)[k]} \approx 0$. If $r_{[k]} \neq 0$ and $J_{[k]} \neq 0$ then the solver need only switch to a new residual norm for which p is not in the left null space of J.

The case $J_{[k]} = 0$ indicates a local minimum in every residual equation. Otherwise, it must be possible to find such a function, since if the Jacobian has r independent equations then its left null space has dimension n-r [Strang88 §2.4]; with $r \geq 1$, there is some vector p which is not in the null space of $J_{[k]}{}^T$. (Of course the vector p may not correspond to a residual norm, and may be inappropriate for evaluating the convergence of a Newton-Raphson step. For example, in the Freudenstein and Roth function at $x_{[k]} = (11.41, -0.8968)^T$, choosing $\emptyset = |r_1|$ or $\emptyset = |r_2|$ escapes the stationary point, although neither cost function is a residual norm and the resulting $x_{[k+1]}$ may have a worse norm, according to $\emptyset_{(1)}$, $\emptyset_{(2)}$, and $\emptyset_{(\infty)}$, than had $x_{[k]}$.)

In a finite-precision computing environment, having p nearly in the left null space of J is sufficient to trap the solution-- not necessarily because J^Tp might evaluate to zero, but because realistic termination criteria declare convergence when some measure of the step length $\|\Delta x_{[k]}\| \approx 0$. Therefore for a Jacobian nearly zero to machine precision, it may not be possible to find such an alternate weighting.

Initial slope

From Equation 3.2.2 the initial slope along a search direction $d_{[k]}$

$$\left.\frac{d\emptyset}{d\mu}\right|_{x_{[k]}} = -g_{[k]}{}^Td_{[k]} = -p_{[k]}{}^T(Jd)_{[k]} \tag{4.3.8}$$

In the Newton-Raphson direction, $d_{[k]:NR} = J_{[k]}{}^{-1}r_{[k]}$ and

$$\left.\frac{d\emptyset}{d\mu}\right|_{x_{[k]:NR}} = -p_{[k]}{}^Tr_{[k]} \tag{4.3.9a}$$

To evaluate a step in the Newton-Raphson direction, a general cost function should: (1) have a positive inner product between $p_{[k]}$ and $r_{[k]}$; and (2) allow any element of $p_{[k]}$ to be nonzero. Note that choosing a cost function with sign{p_i} = sign{r_i}, as suggested above, ensures $p_{[k]}{}^Tr_{[k]} > 0$.

In the steepest descent direction, $d_{[k]} = J_{[k]}{}^Tp_{[k]}$ and

$$\left.\frac{d\emptyset}{d\mu}\right|_{x_{[k]:SD}} = -\|(J^Tp)_{[k]}\|^2 \tag{4.3.9b}$$

The steepest descent direction exists so long as $p_{[k]}$ does not lie in the left null space of $J_{[k]}$, an implicit requirement of the linearized residual approximation (see for example Equation 4.3.5).

General Hessian

From Equation 3.2.6b, the Hessian elements

$$[G]_{i,j} = \frac{\partial^2 \emptyset}{\partial x_i \partial x_j} = \sum_{m=1}^{n} \frac{\partial r_m}{\partial x_j} \frac{\partial}{\partial x_i}\left(\frac{\partial \emptyset}{\partial r_m}\right) + \sum_{m=1}^{n} \frac{\partial \emptyset}{\partial r_m} \frac{\partial^2 r_m}{\partial x_i \partial x_j}$$

The first term is the inner product of the j^{th} column of the Jacobian ($\partial r/\partial x_j$ from Equation 2.2.7b) and the i^{th} row of a matrix whose [i,m] element is

$$\frac{\partial}{\partial x_i}\left(\frac{\partial \emptyset}{\partial r_m}\right) = \frac{\partial p_m}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial p_m}{\partial r_k} \frac{\partial r_k}{\partial x_i}$$

That is, it is the i^{th} row of J^TP times the j^{th} column of J, or $[J^TPJ]_{i,j}$. In the second summation, note $\partial \emptyset/\partial r_m = p_m$ and $\partial^2 r_m/\partial x_i \partial x_j = [\nabla^2 r_m]_{i,j}$. Therefore

$$G = J^TPJ + \sum_{m=1}^{n} p_m \nabla^2 r_m \tag{4.3.10}$$

This result expresses the curvature in $\emptyset$ as the sum of: (1) the curvature J^TPJ due to the linearized residuals; and (2) the weighted curvatures $p_m\nabla^2 r_m$ due to second-order effects in the residuals. As with Equation 4.2.3b for $G_{(2)}$, when the residuals are nearly linear, $\nabla^2 r_m \approx 0$, or when an iterate is near a minimum, $p_m \approx 0$, then the second term is nearly zero and

$$G \approx J^TPJ \tag{4.3.11}$$

As shown by Equation 4.3.4, the linearized residual approximation results in the same Hessian expression.

Hessian properties under linearized residual approximation

The Hessian matrix of Equation 4.3.11, J^TPJ, should be: (1) symmetric always; and (2) positive-definite in the neighborhood of a minimum. To prove symmetry, note that $P = \nabla_r^2 \emptyset$ is symmetric, and transpose Equation 4.3.11: $(J^TPJ)^T = J^TP^TJ = J^TPJ$, and the approximate Hessian is symmetric.

By definition, to have J^TPJ positive-definite requires $x^TJ^TPJx > 0$ for all $x \neq 0$. Let $y = Jx$. Then the Hessian approximation is positive-definite if: (1) $y^TPy > 0$ for all $y \neq 0$; and (2) $y = 0$ only when $x = 0$. The first condition, P positive-definite, depends only on the choice of $\emptyset\{r\}$. The second condition requires $Jx \neq 0$ for $x \neq 0$; that is, J must have independent columns [Strang88 §2.4]. Note this is the same requirement imposed by Newton-Raphson's method, since to be invertible J must be nonsingular; that is, its rows and therefore its columns must be independent.

By Equation 3.3.20, P is symmetric positive-definite if and only if it can be written $P = R^TR$ where R has independent columns (R bears no relation to the residual vector r). When R does not have independent columns, R^TR is positive-semidefinite, because $Rx = 0$ for some $x \neq 0$, so that $x^TR^TRx = \|Rx\|^2 = 0$ for some $x \neq 0$. However, since $\|Rx\|^2 \geq 0$ for all x, x^TR^TRx is never negative. By similar reasoning, the Hessian approximation of Equation 4.3.11 is at worst positive-semidefinite.

Newton versus Newton-Raphson direction

Perform a Newton step on Equation 4.3.4, the quadratic model of $\emptyset\{x\}$ resulting from linearizing the residuals. The method attempts to minimize $\emptyset\{x\}$ by solving $G_{[k]}\Delta x_{[k]:N} = -g_{[k]}$ or

$$J_{[k]}^T P_{[k]} J_{[k]} \Delta x_{[k]:N} = -J_{[k]}^T p_{[k]}$$

If the Newton-Raphson solution exists, $(J_{[k]}^T)^{-1}$ exists and

(4.3.12) $$P_{[k]} J_{[k]} \Delta x_{[k]:N} = -p_{[k]}$$

under the linearized residual approximation.

Equation 4.2.7 shows that the Newton step for $\emptyset_{(2)}$ approximated by linearized residuals gives the Newton-Raphson step exactly. Suppose $\Delta x_{[k]:N} = \Delta x_{[k]:NR}$ holds generally, i.e. for a general $\emptyset$. Then $J\Delta x_{[k]:N} = -r_{[k]}$ and $P_{[k]} r_{[k]} = p_{[k]}$ or

$$r_1 \frac{\partial p_1}{\partial r_j} + r_2 \frac{\partial p_2}{\partial r_j} + .. + r_n \frac{\partial p_n}{\partial r_j} = p_j$$

for $1 \le j \le n$. Suppose further that $\emptyset$ satisfies Equation 4.3.7, $\emptyset = \sum t_i\{r_i\}$. (Strictly $\emptyset = \sum t_i\{\hat{r}_i\}$ for this discussion, since Equation 4.3.12 approximates G using the linearized residuals.) Then $p_j = \partial\emptyset/\partial r_j = \partial t_j/\partial r_j$ is a function of r_j only, and for the Newton step to match the Newton-Raphson step requires

$$r_j \frac{dp_j}{dr_j} = p_j$$

for $1 \le j \le n$. Since $p_j = p_j\{r_j\}$, solve this separable differential equation [Ellis §B.2] by integrating:

$$\frac{1}{p_j} dp_j - \frac{1}{r_j} dr_j = 0$$

$$\ln\{p_j\} - \ln\{r_j\} = c_{1j}$$

$$\frac{p_j}{r_j} = \varepsilon^{c_{1j}}$$

$$p_j = \frac{\partial \emptyset}{\partial r_j} = \frac{dt_j}{dr_j} = \varepsilon^{c_{1j}} \cdot r_j$$

Finally

$$t_j = c_{2j} \cdot r_j^2 + c_{3j}$$

where c_{1j}, c_{2j}, and c_{3j} are constants. Note that $c_{2j} = \frac{1}{2}\varepsilon^{c_{1j}} > 0$. This $t_j\{r_j\}$ makes a Newton step on the cost function of Equation 4.3.7, with r approximated by the linearized model $\hat{r}$, identical to the Newton-Raphson step.

Since c_{3j} changes only the value of $\emptyset$ at the minimum, let $c_{3j} = 0$ for $1 \le j \le n$. Then

(4.3.13) $$\emptyset = \sum_{i=1}^{n} c_i \cdot r_i^2$$

with $c_i > 0$ for $1 \le i \le n$ makes $\Delta x_{[k]:N} = \Delta x_{[k]:NR}$ for a cost function which: (1) sums n terms, each depending on exactly one residual; and (2) approximates the Hessian by neglecting the second-order terms of the residual expansion. With identical Newton and

Newton-Raphson steps, Equation 3.5.14b guarantees that a Cauchy step on Equation 4.3.13 is shorter than the Newton-Raphson step.

For other cost functions, the Newton step may not be the same as the Newton-Raphson step. The Newton-Raphson step exactly minimizes a reasonable cost function built up from the $\hat{r}$, so the Newton step misses the minimizer. Geometrically this occurs because the Newton step assumes constant curvature (in the sense that for a given choice of $\hat{r}_{[k]}$ the Hessian approximation based on the residual models does not depend on x). Since $G \approx J^T P J$, and with $J_{[k]}$ fixed for a given choice of $\hat{r}_{[k]}$, constant curvature in the model $f_{[k]}$ requires P independent of r. This occurs only for the sum of squares formulation and scaled variations.

R-square revisited

Return to $\Phi_{(2)} = \Sigma r_i^2$. The $\partial\Phi_{(2)}/\partial r_j = 2r_j$ and thus

$$p_{(2)} = 2r \tag{4.3.14}$$

as inferred from Equation 4.2.1. Note $p_{(2)} = 0$ only at $r = 0$. Also $\partial^2\Phi_{(2)}/\partial r_i \partial r_j = 2$ when i=j, and zero otherwise. Therefore

$$P_{(2)} = 2I \tag{4.3.15}$$

Note $P_{(2)}$ is positive-definite and independent of r. Substituting in the general gradient and Hessian relations gives $g_{(2)} = 2J^T r$ and $G_{(2)} \approx 2J^T J$, as expected.

4.4 The One-Norm

The one-norm, $\Phi_{(1)} = \Sigma|r_i|$, is not differentiable when any particular residual $r_j = 0$, because $|r_j|$ has no tangent there [Ellis §3.1]. Moreover the Hessian of a piecewise-linear function is almost everywhere zero [Shor §3.1], so $P_{(1)} = 0$ even when no $r_j = 0$.

Gradient with no zero residuals

Applied to the one-norm, Equation 4.3.1 gives

$$p_{j(1)} = \frac{\partial\Phi_{(1)}}{\partial r_j} = \frac{\partial|r_j|}{\partial r_j} = \begin{cases} r_j>0: & 1 \\ r_j<0: & -1 \\ r_j=0: & \text{undefined} \end{cases} \tag{4.4.1}$$

Therefore if no $r_j = 0$, Equation 4.3.6 gives

$$g_{(1)} = \sum_{i=1}^{n} \text{sign}\{r_i\} \cdot \nabla r_i$$

weighting each Jacobian row by the sign of the corresponding residual. Figure 4.4.1 shows the resulting gradient field, along with some level curves, for the wall convection problem.

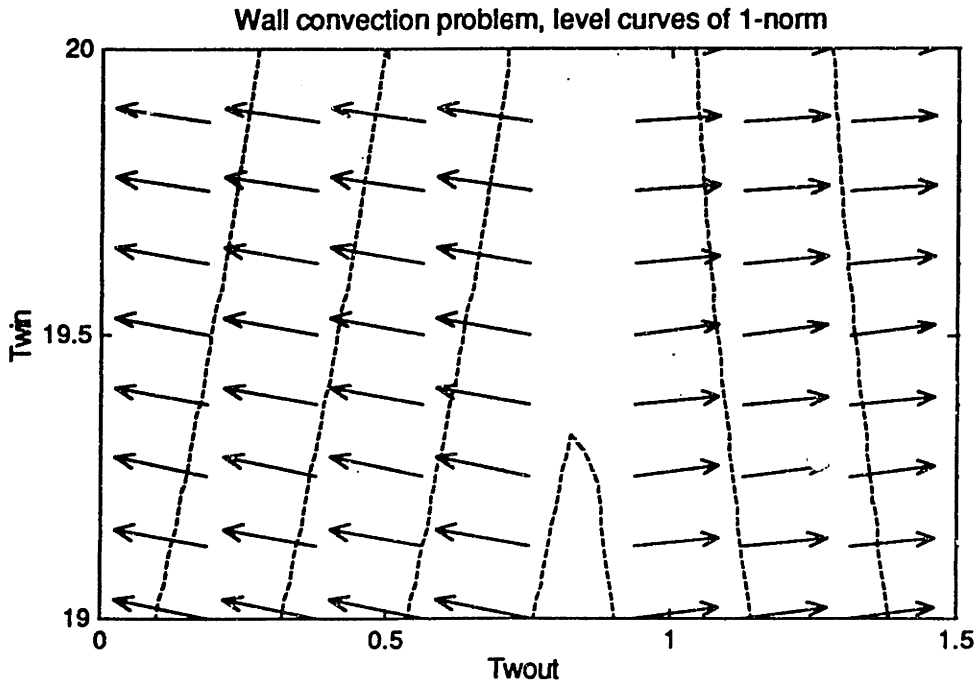

Figure 4.4.1. Level curves and scaled sample gradients for $\emptyset_{(1)}$ on the wall convection problem. The gradients are not drawn to the same scale as those of Figure 4.3.2.

Compared to the gradients of $\emptyset_{(2)}$ in Figure 4.3.2, those of $\emptyset_{(1)}$ do not vary with the residual magnitude. The resulting discontinuity in the gradient field, where the residual changes sign, accounts for the undefined behavior of Equation 4.4.1 at $r_j = 0$. Figure 4.4.1 suggests this discontinuity at the level curve $r_1 = 0$ (and Figure 4.3.1 shows the level curve itself).

Hessian with no zero residuals

Since no component of $p_{(1)}$ depends on any of the residuals, Equation 4.3.2b gives $P_{(1)} = 0$. Geometrically this follows because $\emptyset_{(1)}$ combines the residuals linearly, and $P_{(1)}$ gives the curvature in $\emptyset_{(1)}$ due to the linearized residuals, i.e., the curvature in $f_{(1)}$ (see Equation 4.3.10). A linear combination of linear functions, $f_{(1)}$ is itself linear and has no curvature. Therefore any curvature in $\emptyset_{(1)}$ is due to second-order effects in the residuals themselves, and cannot be estimated from the first-order information in the Jacobian.

Without an estimate of $G_{(1)}$ at its k^{th} iteration, an algorithm can predict neither the optimizing stepsize $\hat{\mu}_{(1)[k]}$ nor the change $\Delta f_{(1)[k]}$ in the cost function using the results of Chapter 3. However with no curvature in $f_{(1)}$, the linear model gives their values directly.

Line search with no zero residuals

Consider a line search for the minimum of $f_{(1)} = \sum|r_i|$ where each of the r_i is linear in x. Following Equation 2.2.8, let $r = r_{[k]} + J_{[k]}(x - x_{[k]})$. From Equation 3.1.3 the search chooses $x = x_{[k]} - \mu d_{[k]}$. Combining,

(4.4.2) $$r = r_{[k]} - \mu J_{[k]} d_{[k]}$$

By definition, a descent direction $d_{[k]}$ decreases $f_{(1)[k]}$ for $\mu > 0$. Now suppose no component of $r_{[k]}$ is zero, so Equation 4.4.1 defines $g_{[k]} = (J^T p)_{[k]}$. The initial slope $df_{(1)[k]}/d\mu = -g_{[k]}^T d_{[k]} = -p_{[k]}^T (Jd)_{[k]}$. Since $f_{(1)}$ has no curvature, this rate of decrease holds until one of the residual components reaches zero. (The magnitude of at least one component of r must decrease along the search direction, because $\Sigma|r_i|$ decreases.)

Let the j^{th} element of r reach zero at some $\hat{\mu}_j > 0$. For smaller μ, $|r_j|$ decreases with μ and $d|r_j|/d\mu < 0$. Beyond $\hat{\mu}_j$, $|r_j|$ increases with μ, at the same rate that it decreased before the critical point, since $dr_j/d\mu$ is constant for the linear equation.

Now consider $df_{(1)}/d\mu$ in the neighborhood of $\hat{\mu}_j$. For smaller μ, $df_{(1)}/d\mu = \Sigma(d|r_i|/d\mu)$ receives a negative contribution from r_j. For larger μ, the other terms remain constant but the contribution from r_j changes sign. Therefore the critical point at $\hat{\mu}_j$ makes $df_{(1)}/d\mu$ more positive by $2 \cdot |dr_j/d\mu| = 2 \cdot |(Jd)_{j[k]}|$.

From Equation 4.4.2, these critical points occur when $r_{j[k]} - \hat{\mu}_{j[k]} \cdot (Jd)_{j[k]} = 0$ or

$$\hat{\mu}_{j[k]} = \frac{r_{j[k]}}{(Jd)_{j[k]}} \tag{4.4.3}$$

where $(Jd)_{j[k]}$ denotes the j^{th} element of $J_{[k]}d_{[k]}$. Equation 4.4.3 holds unless this j^{th} component is zero. This happens if: (1) $d_{[k]}$ is perpendicular to the j^{th} row of $J_{[k]}$; or (2) the j^{th} row of $J_{[k]}$ is zero. Equation 2.2.7b gives this j^{th} row as the gradient of r_j, so in neither case does the linearized residual change along $d_{[k]}$. A component $(Jd)_{j[k]} = 0$ does not contribute a critical stepsize.

Beyond the first critical point, the net effect of other decreasing residuals may keep $df_{(1)}/d\mu < 0$. Again if $f_{(1)}$ decreases then so does at least one component of r, and again $f_{(1)}$ changes at a constant rate until the next critical point, where another component of r crosses zero. At this next critical point, the slope again bends upward by an increment $2 \cdot |(Jd)_{j[k]}|$.

Visiting each positive critical stepsize in order from smallest to largest, while the slope remains negative the cost function decreases. When the slope changes from negative to positive (or possibly from negative to zero), the corresponding critical stepsize minimizes $f_{(1)}$ in the $d_{[k]}$ direction (or possibly marks the beginning of a continuum of equal-valued minima). Note that beyond the last critical stepsize, the magnitude of every residual increases and so $df_{(1)}/d\mu > 0$. Therefore $f_{(1)}$ has a minimum as long as $d\emptyset_{(1)}/d\mu < 0$ at $x_{[k]}$.

Since the minimum of $f_{(1)[k]}$ occurs at one of the critical stepsizes of Equation 4.4.3, the result for a linear system finds one of the $r_{j[k+1]} = 0$, and Equation 4.4.1 no longer suffices to choose the gradient $g_{(1)[k+1]}$.

To estimate the change in $\emptyset_{(1)}$, either:

(4.4.4a) Predict $r_{[k+1]}$ by substituting the predicted minimizing stepsize $\hat{\mu}_{[k]}$ in Equation 4.4.2, then find $f_{(1)[k+1]}$ directly from the definition;

or

(4.4.4b) Accumulate the products $\frac{df_{(1)}}{d\mu} \cdot \Delta\mu$ between critical stepsizes, visiting them in order from $\mu = 0$, where $\Delta f_{(1)} = 0$.

Equation 4.4.4b may be performed during the search for the minimizing stepsize.

The example below illustrates this second procedure.

Example: duct flow problem

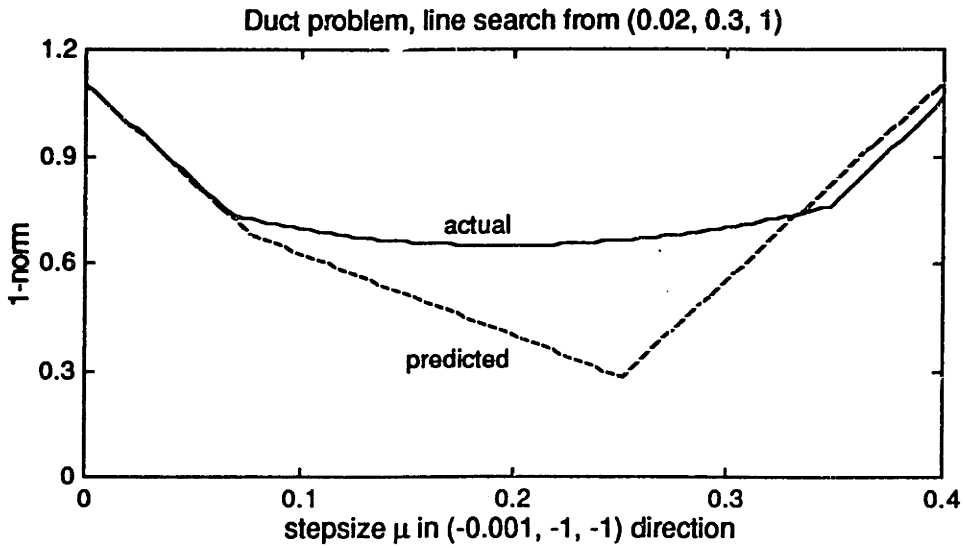

Figure 4.4.2. The linearized model, $f_{(1)[k]}$, predicts a minimum in $\emptyset_{(1)}$ where one of the linearized residuals crosses zero in the $d_{[k]}$ direction.

Figure 4.4.2 shows a line search for the duct flow problem, from a point $x_{[k]} = (0.02, 0.3, 1)^T$. At this point,

$$r_{[k]} = (0.9805, 9.920 \cdot 10^{-6}, -0.1221)^T \quad \text{and} \quad g_{(1)[k]} = (-198.1, -3.840, -1.470)^T$$

Here $\emptyset_{(1)[k]} = 1.103$. Choose a search direction

$$d_{[k]} = (-0.001, -1, -1)^T \quad \text{where} \quad J_{[k]}d_{[k]} = (3.919, -0.01029, -1.6)^T$$

Then Equation 4.4.3 gives the critical stepsizes as

$$\hat{\mu}_{[k]} = (0.2502, -9.640 \cdot 10^{-4}, 0.07632)^T$$

At $x_{[k]}$, the slope $df_{(1)}/d\mu = -g_{(1)[k]}d_{[k]} = -5.508$. The dashed line in Figure 4.4.2 follows this slope until the first critical stepsize, $\hat{\mu}_{3[k]} = 0.07632$, by which point the change $\Delta f_{(1)} = -5.508 \cdot 0.07632 = -0.4204$. At the critical stepsize itself the slope increases by 2·|-1.6|, to -2.308.

This new slope in $f_{(1)}$ holds until the second critical stepsize, $\hat{\mu}_{1[k]} = 0.2502$. In the interval, $\Delta f_{(1)[k]} = -2.308 \cdot (0.2502 - 0.07632) = -0.4014$, for an overall change of -0.8218. Passing the critical point, the slope increases by 2·|3.919|, to 5.529. Since the slope changes from negative to positive, this second critical point minimizes $f_{(1)[k]}$ in the given search direction.

At $\hat{\mu}_{[k]} = 0.2502$ the linearized residuals $\hat{r}_{[k+1]} = (0, 0.002585, 0.2782)^T$, where $f_{(1)} = 0.2808 = \emptyset_{(1)[k]} - 0.8218$ as predicted. Figure 4.4.2 shows both the predicted curve $f_{(1)}$ and the actual norm $\emptyset_{(1)}$. The curves differ due to nonlinearity in the residual relations.

Gradient with zero residuals

In the work above, where no component of the residual vector is zero, Equation 4.4.1 defines the gradient $g_{(1)} = J^T p_{(1)}$ unambiguously. When some $r_j = 0$, however, the value of $\partial|r_j|/\partial r_j$ depends on the search direction.

The case is important for several practical reasons. Besides the chance that a particular selection of $x_{[k]}$ happens to find a zero residual, each full Newton-Raphson step exactly solves any linear residual relations, creating zero residuals at the next iteration. Moreover, to estimate the net direction for two steepest descent steps on $\emptyset_{(1)}$ from $x_{[k]}$, after the manner of Equation 3.4.15, requires a gradient choice from the projected minimizer of $\emptyset_{(1)}$ in the first steepest descent direction. Even if $r_{[k]}$ has no zero elements, the projected residual vector after the first steepest descent step, $r'_{[k+1]}$, has at least one, since by the analysis above the predicted minimum occurs at a critical stepsize where some $r_{j[k+1]} = 0$.

Suppose an $r_{j[k]} = 0$. Then if $(Jd)_{j[k]} > 0$ and $\mu > 0$ in Equation 4.4.2, $r_j < 0$ and $p_{j(1)} = \partial|r_j|/\partial r_j = -1$. If on the other hand $(Jd)_{j[k]} < 0$ then $p_{j(1)} = 1$ for a search with $\mu > 0$ (for negative μ the signs reverse).

As above, if $(Jd)_{j[k]} = 0$ then either $d_{[k]}$ is perpendicular to the gradient of r_j at $x_{[k]}$, or the gradient of r_j is zero. In either case, the linearized residual does not change along the search direction, and $\partial\emptyset_{(1)}/\partial r_j = 0$. Combining these results with Equation 4.4.1, for a given search direction d, find the gradient using

$$p_{j(1)} = \begin{cases} r_j>0: & 1 \\ r_j<0: & -1 \\ r_j=0 \text{ and } \mu\cdot(Jd)_j>0: & -1 \\ r_j=0 \text{ and } \mu\cdot(Jd)_j<0: & 1 \\ r_j=0 \text{ and } (Jd)_j=0: & 0 \end{cases} \tag{4.4.5}$$

Equation 4.4.5 holds in a particular search direction as soon as $\mu \neq 0$, that is, as soon as movement along the search direction gives magnitude to the zero residual components (unless $(Jd)_j = 0$, as discussed above). Therefore it gives the gradient $g_{(1)} = J^T p_{(1)}$ suitable for evaluating the initial slope $d\emptyset_{(1)}/d\mu = -g_{(1)}{}^T d$ in the specified search direction.

However, since there exist any number of search directions, Equation 4.4.5 implies a multiplicity of gradients at any point $x_{[k]}$ where one or more $r_{j[k]} = 0$. Therefore the simple assignment of Equation 3.2.5, $d_{(1)[k]:SD} = g_{(1)[k]}$, no longer specifies a unique direction when an $r_j = 0$ (nor does it necessarily specify a descent direction, as shown below).

Steepest descent direction

Each zero component of the residual vector admits three possible values for $p_{j(1)}$-- 1, -1, or 0, depending on $(Jd)_j$. To find the direction of steepest descent requires: (1) finding all the local gradient directions, called subgradients [Shor §1.2], e.g. by taking every combinatorial possibility $\hat{p}_{(1)}$ given the zero residual elements in r; and (2) choosing the best from among these gradient directions.

Two complications arise in using Equation 3.2.2,

$$\frac{d\emptyset_{(1)}}{d\mu} = -g_{(1)}{}^T d$$

to decide which subgradient gives the steepest descent. First, to compare the slopes in several directions requires normalized directional lengths $\|d\|$, since μ changes x more rapidly for longer subgradients. Second, the particular subgradient appropriate for evaluating movement along a given gradient direction may not be the self-same gradient. That is, setting d to a subgradient in Equation 3.2.2 does not guarantee that the same subgradient stands in for $g_{(1)}$. Thus $d\emptyset_{(1)}/d\mu \neq -\|g_{(1)}\|^2$ in a general subgradient direction (although equality holds when no $r_j = 0$).

To show why equality may not hold, rearrange Equation 3.2.2 with $g_{(1)} = J^T d$:

$$\frac{d\emptyset_{(1)}}{d\mu} = -p_{(1)}^T(Jd) \tag{4.4.6}$$

If $r_j = 0$, then $p_{j(1)} = -\text{sign}\{(Jd)_j\}$, making the j^{th} term of the inner product $-p_{(1)}^T(Jd)$ positive. Geometrically this follows because if moving in the d direction changes an $r_j = 0$, it can only increase in magnitude, making a positive contribution to $\emptyset_{(1)}$.

Now suppose this j^{th} element of Jd dominates the sum of the other elements in magnitude (or simply exceeds the weighted sum of the other elements, with the weights in $p_{(1)}$). Then as soon as movement along d sets $r_j \neq 0$, the increase in r_j exceeds the net decrease in the other elements, and the unambiguous direction of steepest descent implies movement back towards $r_j = 0$. The gradient opposes motion along d. When d is itself a subgradient-- that is, when a subgradient $J^T\hat{p}_{(1)}$ has a $(Jd)_j = (JJ^T\hat{p}_{(1)})_j$ of large enough magnitude to overwhelm the signed sum of its other elements-- then the gradient appropriate for evaluating a subgradient direction is not identical to the subgradient.

Note however that when no $r_j = 0$, $p_{(1)}$ is fixed, and finding the direction of steepest descent requires finding a d which maximizes $(p_{(1)}^T J)d/\|d\| = \|g_{(1)}\|\cdot\cos\theta$ where θ is the angle between d and $g_{(1)}$. Setting $d = g_{(1)}$ sets $\theta = 0$, maximizing the expression and making $d\emptyset_{(1)}/d\mu = -g_{(1)}^T g_{(1)}$ as expected. In this case $g_{(1)}$ holds at $x_{[k]}$, instead of coming into force only after movement from $x_{[k]}$ changes the critical residual components from zero. Therefore $d\emptyset_{(1)}/d\mu = -\|g_{(1)}\|^2$ holds, and $g_{(1)}$ is the direction is steepest descent, when no $r_j = 0$.

Example: duct flow problem

The duct flow problem at $x_{[k]} = (0.02, 0.3, 3.0919)^T$ has $r_{[k]} = (0, -0.001208, 2.446)^T$. Suppressing the subscript [k] for clarity, an arbitrary search direction can give any of

$$\hat{p}_{(1)} = \begin{cases} (1, -1, 1)^T \\ (-1, -1, 1)^T \\ (0, -1, 1)^T \end{cases} \quad \text{for which} \quad \hat{g}_{(1)} = \begin{cases} (-198.2, 6.704, 1.574)^T \\ (198.1, 12.41, 2.136)^T \\ (-0.02911, 9.556, 1.855)^T \end{cases}$$

depending on the sign of the first element of Jd. The corresponding subgradients $\hat{g}_{(1)} = J^T\hat{p}_{(1)}$ all are candidates as the steepest descent direction, with the choice decided by: (1) finding $p_{(1)}$ using $d = \hat{g}_{(1)}$ in Equation 4.4.5; (2) calculating $g_{(1)} = J^T p_{(1)}$; and (3) selecting the $\hat{g}_{(1)}$ for which $d\emptyset_{(1)}/d\|x\| = -g_{(1)}^T\hat{g}_{(1)}/\|\hat{g}_{(1)}\|$ is most negative.

From Equation 4.4.5, with

$$J\hat{g}_{(1)} = \begin{cases} (39260, -5.743, 67.01)^T \\ (-39300, 5.815, 122.6)^T \\ (-22, 0.03589, 94.8)^T \end{cases}$$

the first element of $p_{(1)}$ would be -1, 1, and 1, respectively. Note that in general guessing $\hat{p}_{j(1)} = 0$ does not yield $p_{j(1)} = 0$ since this would require that $(JJ^T\hat{p}_{(1)})_j = 0$. In the example, the magnitude of the first component of $J\hat{g}_{(1)}$, which corresponds to the zero residual, exceeds the sum of magnitudes of the other components for two of the subgradients; as discussed above, these cannot be descent directions.

The candidate steepest descent directions have gradients and slopes

$$g_{(1)} = \begin{cases} (198.1, 12.41, 2.136)^T \\ (-198.2, 6.704, 1.574)^T \\ (-198.2, 6.704, 1.574)^T \end{cases} \quad \text{and} \quad \frac{d\emptyset_{(1)}}{d\|x\|} = \begin{cases} 197.6 \\ 197.4 \\ -7.474 \end{cases}$$

respectively. As expected, two of the subgradients $\hat{g}_{(1)}$ are not descent directions. They lie at nearly 180° to each other, and form a reciprocal pair of search directions and their gradients. That is, to evaluate $d\emptyset_{(1)}/d\mu$ in the first $\hat{g}_{(1)}$ direction requires the second gradient, while to evaluate movement along the second $\hat{g}_{(1)}$ direction requires the first.

This reciprocal situation does not apply in every case. For many starting points with a zero residual it is possible to find a subgradient which is its own gradient choice (and which, of course, defines a descent direction).

Existence of a descent direction

The example shows that with an $r_j = 0$, not every subgradient defines a descent direction. Note, however, that, except at the solution to the algebraic system, a descent direction exists provided the Jacobian is not singular.

Imagine choosing $\hat{p}_{(1)}$ with a zero in every element corresponding to a zero residual. Call this particular choice $\tilde{p}_{(1)}$. The resulting subgradient corresponds to a search direction $\bar{d}$ for which

$$(J\bar{d})_j = 0 \quad \text{when} \quad r_j = 0 \tag{4.4.7}$$

If $(J\bar{d})_j = 0$, movement along $\bar{d}$ does not change r_j, and $d\emptyset/d\mu$ depends only on the nonzero residuals. Given at least one nonzero residual, and a nonsingular Jacobian, it must be possible to choose a search direction satisfying Equation 4.4.7.

As shown above, $\tilde{p}_{(1)}$, with its zero element whenever $r_j = 0$, gives the appropriate subgradient $\tilde{g}_{(1)}$. With $\bar{d}$ a descent direction, $-\tilde{g}^T\bar{d} < 0$ and therefore $\tilde{g}$ is not perpendicular to $\bar{d}$. Thus to find a search direction satisfying Equation 4.4.7, find $\tilde{g}_{(1)}$ and subtract from it its projections onto each row of the Jacobian which corresponds to a zero residual, a Gram-Schmidt orthogonalization of $\tilde{g}_{(1)}$ to the unwanted rows of J. (Note $\tilde{g}_{(1)}$ contains only contributions from the fixed part of $p_{(1)}$, as do all the other subgradients. Thus in general any subgradient may be orthogonalized into $\bar{d}$.)

The first orthogonalization sets

$$\bar{d} = \tilde{g}_{(1)} - \frac{\nabla r_j^T \tilde{g}}{\nabla r_j^T \nabla r_j} \nabla r_j$$

for the first j for which $r_j = 0$. Subsequent orthogonalization of $\bar{d}$ to another Jacobian row ∇r_i must first orthogonalize ∇r_i to the previous ∇r_j, to avoid adding an unwanted component of any of those Jacobian rows back into $\bar{d}$ [Strang88 §3.2, §3.4]. With the orthogonalization complete, $(J\bar{d})_j = 0$, as desired.

For a geometric interpretation of the orthogonalized search direction, consider the gradient $\nabla r_{j[k]}$ of a residual component $r_{j[k]} = 0$. This Jacobian row is perpendicular to

the level curve $r_j = 0$ at the point $x_{[k]}$, making the orthogonalized search direction tangent to that level curve at $x_{[k]}$. The orthogonalization chooses a search direction, not over the entire n-dimensional vector space x, but over the n-1 dimensions on the level curve $r_j = 0$. For a linear residual relation, following a series of such orthogonalized directions searches for $r = 0$ by moving along the surface $r_j = 0$; for nonlinear r_j the search direction touches the level curve only at the point of tangency and $r_{j[k+1]} \neq 0$.

Note that finding a descent direction $\bar{d}$ does not guarantee that $\tilde{g}_{(1)}$ is a descent direction. Suppose $-\tilde{g}_{(1)}^T \bar{d} < 0$ or $-\tilde{p}_{(1)}^T (J\bar{d}) < 0$. Since $(J\bar{d})_j = 0$ this result depends only on the fixed part of $p_{(1)}$; the elements of $p_{(1)}$ corresponding to zero residuals can take arbitrary finite values without effect. Thus any $\bar{d}$ satisfying Equation 4.4.7 is a descent direction according to every subgradient $\hat{g}_{(1)}$, not just to $\tilde{g}_{(1)}$, and $\tilde{g}_{(1)}$ has no special properties in this regard.

Uncertainty of steepest descent direction

The difficulty of finding a steepest descent direction for $\emptyset_{(1)}$ when an $r_j = 0$ indicates the possible instability of that choice for any residual vector. Where the steepest descent direction for $\emptyset_{(2)}$ weights the gradient of a residual component according to its magnitude, thereby assuring that a nearly-zero residual component exerts relatively little influence on the overall directional choice, the steepest descent direction of $\emptyset_{(1)}$ does not scale the ∇r_i in its summation. Therefore a residual r_j with a relatively large gradient magnitude $|\nabla r_j|$ dominates the direction of steepest descent for $\emptyset_{(1)}$, regardless of the advisability of travel in the ∇r_j direction.

Suppose a row ∇r_j^T of the Jacobian consistently exceeds its other rows in magnitude. Then every steepest descent search on $\emptyset_{(1)}$ is likely to favor reducing just that r_j, especially in a trust region or similar method with inexact line minimization, which leaves r_j with nonzero magnitude at each iteration. In fact the only way to escape the influence of ∇r_j in the search direction is to zero r_j in one iteration, and in the next to orthogonalize the search direction to ∇r_j using the Gram-Schmidt technique mentioned above. Even following such a direction, for nonlinear r_j the search finds a nonzero magnitude at the next iterate, and ∇r_j dominates the next steepest descent choice.

When an r_j is not zero, but has very small magnitude, then a search in a gradient direction $g_{(1)}$ dominated by ∇r_j is not likely to proceed far before r_j changes sign, increasing $d\emptyset_{(1)}/d\mu$ by $2 \cdot |(Jg_{(1)})_j| \approx 2 \cdot \|\nabla r_j\|^2$ (the last follows assuming $g_{(1)} \approx \nabla r_j$). Furthermore the positive turn in the slope at $r_j = 0$ is likely to establish the line minimum there.

Geometrically, the strong gradient of r_j creates a deep valley in $|r_j|$. Summing this curve with the other $|r_i\{x\}|$ is likely to produce a similarly deep valley, aligned to the contour $r_j = 0$, in $\emptyset_{(1)}$. Such a valley traps searches in any direction crossing it (even one orthogonal to $\nabla r_{j[k]}$, provided r_j is nonlinear so that the valley curves).

After stepping down a direction dominated by ∇r_j, the next steepest descent direction, again dominated by ∇r_j, takes either nearly the same course or nearly its opposite, depending on the sign of r_j at the conclusion of the first step (assuming inexact line minimization). In this situation a succession of steepest descent steps follows the level curve $r_j = 0$, crossing it periodically depending on the stepsize selection algorithm. Thus each search direction, while steep, yields neither long steps nor dramatic decreases in $\emptyset_{(1)}$. Furthermore each step may undo the greater part of the movement the previous step provided.

Special case: Newton-Raphson direction

The difficulties encountered in a general search direction starting from a point with a zero residual element, do not apply for a search in the Newton-Raphson direction.

Because Newton-Raphson zeros every linearized residual, it does not change the value of the linear model of a residual element already zeroed by the previous iteration. This resolves the ambiguity in Equation 4.4.5, since $r_{j[k]} = 0$ implies $p_{j(1)} = 0$ in the Newton-Raphson search. Mathematically this follows because $d_{[k]:NR} = J_{[k]}^{-1} r_{[k]}$ makes $(J_{[k]}d_{[k]})_j = (r_{[k]})_j = r_{j[k]}$, so $r_{j[k]} = 0$ gives $p_{j(1)[k]} = 0$ in Equation 4.4.5.

From Equation 4.3.9a, the initial slope

$$\left.\frac{d\emptyset_{(1)}}{d\mu}\right|_{x_{[k]:NR}} = -p_{(1)[k]}^T r_{[k]} = -\emptyset_{(1)} \tag{4.4.8a}$$

and so a step $\Delta x_{[k]} = \mu \cdot d_{[k]:NR}$ has a projected initial slope $g_{(1)[k]}^T \Delta x_{[k]} = -\mu \cdot \emptyset_{(1)[k]}$.

Next consider the linearized model $f_{(1)[k]}$. In the Newton-Raphson direction, every nonzero residual has a critical stepsize at $\hat{\mu} = 1$. For the linearized residuals, the initial slope $d\emptyset_{(1)}/d\mu = -\emptyset_{(1)}$ holds until $\mu = 1$, where $\Delta f_{(1)} = -\emptyset_{(1)}$ (i.e., where $f_{(1)} = 0$). At points between,

$$\Delta f_{(1)[k]:NR} = -\mu \cdot \emptyset_{(1)[k]} \tag{4.4.8b}$$

since the linear combination of the linearized residuals gives a linear estimated cost function.

Use in double dogleg algorithm

Consider using $\emptyset_{(1)}$ in a descent-based search for the solution to a nonlinear algebraic system, in particular by adapting the double dogleg algorithm of Chapter 3. The new algorithm must: (1) find the steepest descent direction, or indeed even a descent direction, when r has a zero component; (2) decide whether the steepest descent direction is of any value, since it may be steep but have negative slope only for very short step lengths; and (3) estimate the model, $f_{(1)}$, at arbitrary step lengths, in order to verify how well it captures the actual behavior of the norm, $\emptyset_{(1)}$.

At the least, this third task implies an additional n-dimensional vector of storage, for the critical stepsizes calculated by Equation 4.4.3. Another vector, of predicted values of $f_{(1)}$ at the critical stepsizes, could be added for computational efficiency, as could interpolation and ordering algorithms to speed the calculation. The more modest computational and storage requirements of $\emptyset_{(2)}$, embodied in Equations 4.2.11, make this norm more attractive from the point of view of algorithmic simplicity, even if for some particular problem the one-norm is better-suited for evaluating progress in the Newton-Raphson direction, or for choosing the steepest descent direction.

Quadratic approximations

Given the algorithmic complexity associated with using $f_{(1)}$ in a descent-based equation solver, consider replacing it with an approximating quadratic, $F_{(1)}$, at each iteration.

One possibility fits $F_{(1)[k]}\{\mu\} = a_{[k]}\mu^2 + b_{[k]}\mu + c_{[k]}$ to three observations along a search path: (1) to the initial cost function value, $F_{(1)[k]}\{0\} = \emptyset_{(1)[k]}$; (2) to Equation 4.4.6 for the initial slope, $F_{(1)[k]}'\{0\} = -p_{(1)[k]}^T (Jd)_{[k]}$; and (3) to the predicted minimizing stepsize, $F_{(1)[k]}\{\hat{\mu}\} = f_{(1)[k]}\{\hat{\mu}\}$.

Figure 4.4.2 suggests the dangers of this scheme-- for example, the lack of symmetry of $f_{(1)}$ about the line minimum, as opposed to the symmetry of the parabolic $F_{(1)}$. Moreover, should $\hat{\alpha}$ coincide with the smallest positive critical stepsize from Equation 4.4.3, the three observations define a line, and $a_{[k]} = 0$, leaving no parabola. Finally, since $F_{(1)}$ depends on the choice of search direction, it does not solve the problem of choosing an appropriate search direction when one or more residual components are zero.

Suppose, then, that instead of matching the extrapolated behavior of $f_{(1)[k]}$ along a particular search direction, $F_{(1)[k]}$ captures the local behavior of $f_{(1)[k]}$ in every direction-- i.e., suppose it models the function value and gradient of $\emptyset_{(1)}$ at $x_{[k]}$. Then in the quadratic expansion

$$F_{(1)[k]} \approx \emptyset_{(1)[k]} + \Delta x^T g_{(1)[k]} + \frac{1}{2} \Delta x^T G_{(1)[k]}' \Delta x$$

the Hessian $G_{(1)[k]}'$ does not match the actual Hessian $G_{(1)[k]}$ (which as discussed above is almost everywhere zero), but follows from the algebraic form of $F_{(1)[k]}$.

To make $F_{(1)}$ a quadratic, follow Equation 4.3.13 and let

$$F_{(1)[k]} = b_{[k]} + \sum_{i=1}^{n} c_{i[k]} \cdot r_i^2 \tag{4.4.9a}$$

where the $c_{[k]}$ make $\nabla F_{(1)[k]} = g_{(1)[k]}$ at each iteration, and where $b_{[k]}$ sets $F_{(1)[k]} = \emptyset_{(1)[k]}$. With $J_{[k]}$ constant, Equation 4.3.5 shows that matching the gradients requires $\nabla_r F_{(1)[k]} = p_{(1)[k]}$, or $2c_{j[k]} \cdot r_{j[k]} = \text{sign}\{r_{j[k]}\}$ unless $r_{j[k]} = 0$. Thus choosing $c_{j[k]} = 1/(2 \cdot |r_{j[k]}|)$ matches the gradient of $F_{(1)[k]}$ to that of $\emptyset_{(1)}$ at $x_{[k]}$ when the residual has no zero components. With this choice, $b_{[k]} = \emptyset_{(1)[k]}/2$ matches the initial function values. Alternately, match the gradient direction, though not the gradient length, by setting

$$c_{j[k]} = \frac{1}{|r_{j[k]}|} \tag{4.4.9b}$$

when the residual has no zero components. This choice allows

$$b_{[k]} = 0 \tag{4.4.9c}$$

Equations 4.4.9 describe a weighted sum of squares, whose weights, though constant during each iteration, vary from one iteration to the next. The weighted quadratic captures the essential local behavior of $\emptyset_{(1)}$ at $x_{[k]}$ when $r_{[k]}$ has no zero component. To define $F_{(1)[k]}$ meaningfully even if an $r_{j[k]} = 0$, consider perturbing the denominator by a very small positive number, or limiting it to a small minimum magnitude; either method may result in a large c_j which does not allow much travel in the search direction. In any event, using $F_{(1)}$ in place of $\emptyset_{(1)}$ allows essentially comparable gradient choices, sidesteps the problem of zero residual components, and allows more convenient cost function estimation at arbitrary stepsizes. (However the cost function does not aproximate $f_{(1)[k]}$ away from $x_{[k]}$, since it combines the residuals differently. It captures the local behavior of $\emptyset_{(1)}$ at $x_{[k]}$, but not beyond, since it has a nonzero Hessian.)

Section 4.5 treats the weighted sum of squares more generally.

Infinity-norm

Like $\emptyset_{(1)}$, the ∞-norm combines the residuals piecewise-linearly, so its model $f_{(\infty)}$, based on the linearized residuals, has $\nabla^2 f_{(\infty)} = 0$. At a point $x_{[k]}$, the gradient in $\emptyset_{(\infty)}$

typically depends on a single residual component, that with the maximum magnitude. Therefore its slope depends on a single row of the Jacobian. When more than one residual component shares the same maximum magnitude, more rows of the Jacobian come into force; the Freudenstein and Roth function at $(11.41, -0.8968)^T$ exemplifies this case.

One advantage of $\Phi_{(\infty)}$ over $\Phi_{(1)}$, its independence from zero residual components, circumvents the problem of directionally-dependent gradients, and hence defines unambiguously its direction of steepest descent. Since an exact minimization on the linearized function $f_{(\infty)[k]}$ ends at a point $x_{[k+1]}$ where two components of the residual have equal magnitude, predicting the minimizing stepsize $\hat{\mu}_{[k]}$ and the value of $f_{(\infty)}$ at that stepsize requires element-wise processing of the vectors $r_{[k]}$ and $(Jd)_{[k]}$, in a manner similar to that for predicting the behavior of $\Phi_{(1)}$. This procedure is not treated here.

4.5 Weighted R-Square

Consider the weighted sum of squares of residuals,

$$\Phi_{(w)} = \sum_{i=1}^{n} w_i \cdot r_i^2 \tag{4.5.1}$$

where the w_i are constant. If $w_i > 0$ for $1 \le i \le n$, then from Equation 4.3.13, substituting the linearized residuals and calculating the Newton minimizer for the cost function $\Phi_{(w)}$ yields the same step as the Newton-Raphson solution of the equilibrium problem.

While an iterative algorithm could change the w_i from iteration to iteration, they remain constant during each iteration. An algorithm might use $w_i = w_{i[k]}$, changing weights between iterations in order: (1) to escape local minima in residual norms with other weights; (2) to accomodate changing Jacobian row magnitudes; or (3) to make a predetermined search direction a steepest descent direction. Chapter 8 explores these possibilities further.

For this cost function,

$$p_{j(w)} = 2w_j \cdot r_j \quad \text{and} \quad [P]_{i,j(w)} = \begin{cases} i=j: & 2w_j \\ \text{else:} & 0 \end{cases} \tag{4.5.2a,b}$$

For notational convenience, define the diagonal weighting matrix

$$W = \text{diag}\{w_1, w_2, .. , w_n\} \tag{4.5.3}$$

Note that the weighting matrix performs a different function than the scaling matrix D of [Dennis §7.1]: W applies to the residuals, while D scales the independent variables x.

For $\Phi_{(2)}$, the $w_i = 1$ and $W = I$. Thus, substituting I for W and $\Phi_{(2)}$ for $\Phi_{(w)}$ in any expression below reduces it to the corresponding result for $\Phi_{(2)}$ in Section 4.2.

In terms of W,

$$\Phi_{(w)} = r^T W r \tag{4.5.4a}$$

and

$$p_{(w)} = 2Wr \quad \text{and} \quad P_{(w)} = 2W \tag{4.5.4b,c}$$

from which Equations 4.3.5 and 4.3.11 give

$$g_{(w)} = 2J^T W r \quad \text{and} \quad G_{(w)} \approx 2J^T W J \tag{4.5.5a,b}$$

respectively.

Initial slope

From Equation 3.2.2, a line search in an arbitrary search direction $d_{[k]}$ has initial slope $-g_{[k]}^T d_{[k]} = -2(J^T W r)_{[k]}^T d_{[k]}$. Since the diagonal $W = W^T$,

$$\left.\frac{d\emptyset_{(w)}}{d\mu}\right|_{x_{[k]}} = -2 \cdot r_{[k]}^T (WJd)_{[k]} = -2(Wr)_{[k]}^T (Jd)_{[k]} \tag{4.5.6a}$$

In the Newton-Raphson direction, $d_{[k]:NR} = J_{[k]}^{-1} r_{[k]}$ and

$$\left.\frac{d\emptyset_{(w)}}{d\mu}\right|_{x_{[k]:NR}} = -2 r_{[k]}^T (Wr)_{[k]} = -2\emptyset_{(w)[k]} \tag{4.5.6b}$$

while in the steepest descent direction, $d_{[k]:SD} = g_{(w)[k]}$,

$$\left.\frac{d\emptyset_{(w)}}{d\mu}\right|_{x_{[k]:SD}} = -4 \cdot \|(J^T W r)_{[k]}\|^2 \tag{4.5.6c}$$

As expected, both the Newton-Raphson and steepest descent choices give descent directions. Note that Equations 4.5.6 treat W as $W_{[k]}$, i.e., as if the weights might change from iteration to iteration. This notation continues below.

Predicted stepsize

From Equation 3.2.11a, the estimated optimizing stepsize

$$\hat{\mu}_{(w)[k]} = \frac{(Jd)_{[k]}^T (Wr)_{[k]}}{(Jd)_{[k]}^T W_{[k]} (Jd)_{[k]}} \tag{4.5.7a}$$

minimizes the model $f_{(w)[k]}$ along a descent direction $d_{[k]}$. As expected, substituting $d_{[k]:NR} = J_{[k]}^{-1} r_{[k]}$ gives

$$\hat{\mu}_{(2)[k]:NR} = 1 \tag{4.5.7b}$$

since the Newton-Rapson step zeros the residual models. In the steepest descent direction,

$$\hat{\mu}_{(w)[k]:SD} = \frac{1}{2} \frac{\|(J^T W r)_{[k]}\|^2}{\|(\sqrt{W} J J^T W r)_{[k]}\|^2} \tag{4.5.7c}$$

where $\sqrt{W}$ has the roots of the weights on its diagonal (in practice, calculating the denominator does not require actually finding the square roots, because the inner product multiplies $\sqrt{W}^T \sqrt{W} = W$). Using Equations 4.5.7c and 4.5.5a, the steepest descent minimizer of the quadratic model (the Cauchy point) lies at $\Delta x_{(w)[k]:SD} = -\hat{\mu}_{(w)[k]:SD} \cdot g_{(w)[k]}$ or

$$\Delta x_{(w)[k]:SD} = \frac{-\|(J^T W r)_{[k]}\|^2}{\|(\sqrt{W} J J^T W r)_{[k]}\|^2} (J^T W r)_{[k]} \tag{4.5.8}$$

Predicted change in cost function

Equation 3.2.9a gives the expected increase in the cost function $\Delta f_{(w)[k]} = f_{(w)[k]}\{x_{[k+1]}\} - \Phi_{(w)[k]}$ as

$$\Delta f_{(w)[k]} = \Delta x^T g_{(w)[k]} + \frac{1}{2}\Delta x^T G_{(w)[k]} \Delta x$$

or

$$\Delta f_{(w)[k]} = 2(J^T W r)_{[k]}^T \Delta x_{[k]} + (J\Delta x)_{[k]}^T W_{[k]} (J\Delta x)_{[k]} \tag{4.5.9a}$$

In the Newton-Raphson direction, $\Delta x = -\mu J^{-1} r$, and

$$\Delta f_{(w)[k]:NR} = (\mu^2 - 2\mu)\cdot\Phi_{(w)[k]} \tag{4.5.9b}$$

In the steepest descent direction, $\Delta x = -2\mu J^T W r$, and

$$\Delta f_{(w)[k]:SD} = 4\mu^2 \cdot \| (\sqrt{W} J J^T W r)_{[k]} \|^2 - 4\mu \cdot \| (J^T W r)_{[k]} \|^2 \tag{4.5.9c}$$

Special forms for double dogleg algorithm

Suppose the double dogleg algorithm of Chapter 3 is to be adapted to equilibrium problems with $\Phi_{(w)}$ as the cost function. Equation 4.3.13 guarantees that the Newton-Raphson solution to the equilibrium problem gives the Newton step for $\Phi_{(w)}$ under the linearized residual approximation. Therefore substituting the computationally preferable Newton-Raphson solution does not change the algorithm.

The first break point in the double dogleg curve, the Cauchy point, occurs at a distance

$$\| \Delta x_{(w)[k]:SD} \| = \frac{\| (J^T W r)_{[k]} \|^3}{\| (\sqrt{W} J J^T W r)_{[k]} \|^2} \tag{4.5.10}$$

from $x_{[k]}$. The second breakpoint, the cutback point, lies at $\eta_{[k]} \cdot \Delta x_{[k]:NR}$. From Equation 3.6.2b,

$$\eta_{[k]} = 0.2 + 0.8 \frac{\| (J^T W r)_{[k]} \|^4}{\| (\sqrt{W} J J^T W r)_{[k]} \|^2 \cdot \Phi_{(w)[k]}} \tag{4.5.11}$$

When the trust length falls between the Cauchy and cutback lengths, the double dogleg algorithm chooses

$$\Delta x_{[k]} = (1 - \lambda_{[k]}) \cdot \Delta x_{(w)[k]:SD} + \lambda_{[k]} \cdot \eta_{[k]} \cdot \Delta x_{[k]:NR} \tag{4.5.12a}$$

where $0 < \lambda < 1$. Setting $\delta_{[k]}^2 = \Delta x_{[k]}^T \Delta x_{[k]}$ gives

$$\begin{aligned} \lambda_{[k]}^2 \cdot \left(1 + \frac{\| \Delta x_{[k]:CB} \|^2}{\| \Delta x_{(w)[k]:SD} \|^2} - 2\eta_{[k]} \frac{\| (\sqrt{W} J J^T W r)_{[k]} \|^2}{\| (J^T W r)_{[k]} \|^4} \Phi_{(w)[k]} \right) \\ + \lambda_{[k]} \cdot \left(2\eta_{[k]} \frac{\| (\sqrt{W} J J^T W r)_{[k]} \|^2}{\| (J^T W r)_{[k]} \|^4} \Phi_{(w)[k]} - 2 \right) \\ + \left(1 - \frac{\delta_{[k]}^2}{\| \Delta x_{(w)[k]:SD} \|^2} \right) = 0 \end{aligned} \tag{4.5.12b}$$

Following logic identical to that used for Equation 4.2.14b, solve Equation 4.5.12b using the positive square root in the quadratic formula (again, the determinant is always

positive). Using the resulting $\lambda_{[k]}$ in Equation 4.5.12a gives, for a particular trust length satisfying $\|\Delta x_{(w)[k]:SD}\| < \delta_{[k]} < \|\Delta x_{[k]:CB}\|$, the double dogleg choice for $x_{[k]}$.

Next the algorithm requires $g_{(w)[k]}{}^T\Delta x_{[k]}$. Take the linear combination

$$\Delta x_{[k]} = \rho_{1[k]}\cdot\Delta x_{(w)[k]:SD} + \rho_{2[k]}\cdot\Delta x_{[k]:NR} \tag{4.5.13}$$

As with Equation 4.2.15, $\rho_2 = 0$ on the first leg of the curve, $\rho_1 = 1 - \lambda$ and $\rho_2 = \lambda\cdot\eta$ on the second leg of the curve, and $\rho_1 = 0$ beyond the cutback point. Then

$$g_{(w)[k]}{}^T\Delta x_{[k]} = -2\rho_{1[k]}\frac{\|(J^TWr)_{[k]}\|^4}{\|(\sqrt{W}JJ^TWr)_{[k]}\|^2} - 2\rho_{2[k]}\cdot\emptyset_{(w)[k]} \tag{4.5.14}$$

for a linear combination of the Cauchy and Newton-Raphson steps.

Finally, using Equation 4.5.13 for $\Delta x_{[k]}$ and Equation 4.5.5b for $G_{(w)[k]}$, Equation 3.2.9a gives

$$\begin{aligned}\Delta f_{(w)[k]} = {} & g_{(w)[k]}{}^T\Delta x_{[k]} + \rho_{2[k]}{}^2\cdot\emptyset_{(w)[k]} \\ & + (\rho_{1[k]}{}^2 + 2\rho_{1[k]}\cdot\rho_{2[k]})\frac{\|(J^TWr)_{[k]}\|^4}{\|(\sqrt{W}JJ^TWr)_{[k]}\|^2}\end{aligned} \tag{4.5.15a}$$

or

$$\Delta f_{(w)[k]} = \rho_{1[k]}\cdot(\rho_{1[k]} + 2\rho_{2[k]} - 2)\frac{\|(J^TWr)_{[k]}\|^4}{\|(\sqrt{W}JJ^TWr)_{[k]}\|^2} + (\rho_{2[k]}{}^2 - 2\rho_{2[k]})\cdot\emptyset_{(w)[k]} \tag{4.5.15b}$$

for a linear combination of the Cauchy and Newton-Raphson steps.

CHAPTER 5
IMPLEMENTATION DETAILS

This chapter notes some practical consequences of the preceding theory, and discusses its implementation.

The first section, devoted to line minimization, estimates the limits placed on exact minimization by finite-precision effects. In particular, it establishes a rule for estimating the smallest resolvable stepsize difference near the minimum in a line search, and demonstrates by numerical examples the greater accuracy of this rule over a standard relationship.

Section 5.2 summarizes standard termination criteria for both function minimization and equation solving.

Section 5.3 compares two implementations of the double dogleg algorithm for solving systems of nonlinear equations. Both implementations exploit the underlying equilibrium problem by substituting the Newton-Raphson for the Newton direction, and both respond to errors in the residual evaluation by taking shorter steps. The first version implements the standard algorithm with a few modifications; the second replaces unnecessary matrix-vector and vector-vector expressions with their scalar equivalents from Chapter 4. The second implementation, algorithmically equivalent, executes more quickly due to its lower computational overhead; Section 5.4 reports comparative tests.

5.1 Terminating a Line Search

The line searches of Chapter 3 seek the stepsize $\mu_{[k]}$ in

$$x_{[k+1]} = x_{[k]} - \mu_{[k]} d_{[k]}$$

which minimizes the cost function at $x_{[k+1]}$. When the cost function is not quadratic, or when the Hessian is unavailable or can only be approximated, the $\hat{\mu}_{[k]}$ of Equation 3.2.11a at best approximates the minimizing stepsize.

Each line minimization, then, is an iterative process, imbedded in the larger iteration of choosing search directions and minimizing along them. This section discusses some of the practical aspects of the line search. Note however that the equation-solving algorithms used throughout this thesis use a trust region, rather than exact minimizations, to set the step length at each iteration.

Requirements for general line minimization routine

A general line minimization routine: (1) makes no assumptions about the sign of the optimizing stepsize; (2) operates at near-zero and negative as well as positive values of the cost function; (3) can limit the range of stepsizes if necessary, for example in order to respond to errors in the cost function evaluation at extreme stepsizes; and (4) accounts for the effects of finite-precision computation.

Within this framework a practical algorithm should locate the minimum with as few function evaluations as possible. To this end the routine also: (5) allows the user to determine the precision to which the minimum is identified; and (6) limits the number of function evaluations allowed before terminating the minimization attempt.

Finite-precision effects

Figure 5.1.1 shows a continuous function Ø of a single continuous independent variable μ. In a finite-precision computing environment, discretization effects prevent the exact representation of Ø{μ}. Suppose the machine can represent μ exactly, and can calculate Ø{μ} exactly, but can store the result only using a coarse grid of numbers. Then the dashed stairstep curve in the figure might result. This dashed line gives the machine representation of a very flat function, whose range spans an insufficient count of machine numbers to allow its adequate resolution. As shown in the figure, discretization of the function values is most exaggerated about the minimum.

Discretization of the independent variable also affects the line minimization. Again in Figure 5.1.1, suppose the machine can represent only values of μ at the dotted vertical lines. This type of discretization is likely to dominate only for relatively steep functions; nonetheless, it also limits the accuracy to which the minimum can be established.

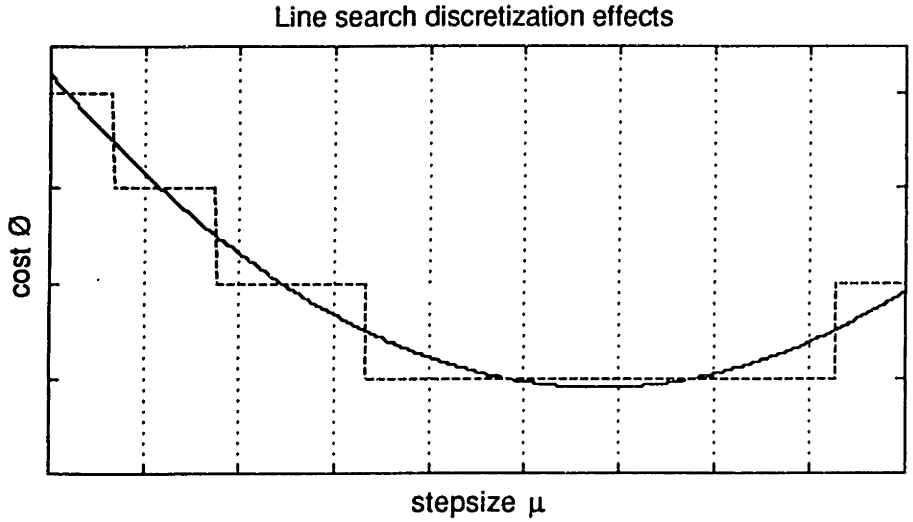

Figure 5.1.1. The solid curve shows a continuous function Ø{μ}, which may be discretized significantly in the function values, in the independent variables, or in both.

Figure 5.1.1 does not depict a third important floating-point effect, since it assumes no rounding errors in calculating Ø. In fact floating-point calculations may introduce spurious minima to regions where a perfectly-discretized Ø{μ} would be flat; see [Moré82 §1.1].

A fourth, minor, discretization effect ignored in Figure 5.1.1 is that machine numbers are not evenly spaced [Dahlquist §2.3.3]. A number $x_1 = m_1 \cdot b^{e1}$ is separated from its neighbors by an increment $m_{min} \cdot b^{e1}$ where m_{min} is the smallest normalized floating-point mantissa available (the base, b, is fixed). Similarly a number $x_2 = b \cdot x_1$ is separated from its neighbors by an increment $m_{min} \cdot b^{1+e1}$, a factor of b greater than the spacing of machine numbers around x_1.

Discretization of independent variables

Discretization of the independent variable places a lower bound on the stepsize increments a line search can make. For example, the stepsize must be large enough that an evaluation of $Ø\{x_{[k]} - \mu d_{[k]}\}$ does not return $Ø\{x_{[k]}\}$ due to rounding of the input vector

back to $x_{[k]}$. Thus the stepsize must be large enough to insure that for some component i of the variable vector x,

$$\mu \cdot |d_{i[k]}| > \varepsilon \cdot |x_{i[k]}|$$

where ε is the smallest fraction of $x_{i[k]}$ which the machine can represent.

Machine epsilon, the smallest fraction of one (but not the smallest fraction) that can be represented in the finite-precision computing environment, gives the difference between the mantissas of two adjacent machine numbers, and is roughly the same as ε when the mantissa of $x_{i[k]}$ is roughly one. For a 24 binary-digit floating-point machine, MACH_EPS $\approx 2^{-23} \approx 1 \cdot 10^{-7}$ is within a factor of two of ε. Therefore the smallest stepsize which can be resolved in the cost function input vector

$$\Delta\mu_{minx[k]} = \text{MACH\_EPS} \cdot \min\left\{ \frac{|x_{1[k]}|}{|d_{1[k]}|}, \frac{|x_{2[k]}|}{|d_{2[k]}|}, \ldots, \frac{|x_{n[k]}|}{|d_{n[k]}|} \right\} \tag{5.1.1}$$

The notation $\Delta\mu_{minx}$ represents the smallest resolvable change in μ due to discretization of the independent variables (the vector x).

To a first approximation this bound applies all along the line, although, as pointed out above, when numbers are represented as $m \cdot b^e$ the absolute differences between adjacent machine numbers depends on their exponent.

Discretization of function values at line minimum

Discretization of the function values may make a function appear flat even over ranges of μ large enough to produce distinct input vectors. As suggested by Figure 5.1.1, this effect is most pronounced near the line minimum, where $d\emptyset/d\mu = 0$ and second-derivative effects dominate.

Suppose the line search algorithm finds an optimizing $\mu_{[k]}$ such that $x_{[k+1]}$ minimizes $\emptyset$ in the search direction. Now consider a search, still in the $d_{[k]}$ direction, but from $x_{[k+1]}$. From Equation 3.2.12, for $\emptyset\{\mu\}$ to evaluate to a different machine number than $\emptyset_{[k+1]}$ requires

$$\frac{1}{2}\mu^2 d_{[k]}{}^T G_{[k+1]} d_{[k]} > \varepsilon \cdot |\emptyset_{[k+1]}|$$

Since μ refers to the same search direction as did $\mu_{[k]}$, the machine cannot calculate meaningful changes in $\emptyset$ for smaller changes in $\mu_{[k]}$, and

$$\Delta\mu_{min\emptyset[k]} = \sqrt{\text{MACH\_EPS}} \cdot \sqrt{\frac{2|\emptyset_{k+1]}|}{d_{[k]}{}^T G_{[k+1]} d_{[k]}}} \tag{5.1.2}$$

The notation $\Delta\mu_{min\emptyset}$ represents the smallest resolvable change in μ due to discretization of the cost function.

For quadratic $\emptyset$, $G_{[k+1]} = G_{[k]} = G$, and Equations 3.2.11a and 3.2.11b give

$$\mu_{[k]} = \frac{d_{[k]}{}^T g_{[k]}}{d_{[k]}{}^T G d_{[k]}} \quad \text{and} \quad \emptyset_{[k]} - \emptyset_{[k+1]} = \frac{1}{2}\frac{(d_{[k]}{}^T g_{[k]})^2}{d_{[k]}{}^T G d_{[k]}}$$

exactly. Then

$$\frac{\Delta\mu_{min\emptyset[k]}}{\mu_{[k]}} = \sqrt{\text{MACH\_EPS}} \cdot \sqrt{\frac{|\emptyset_{[k+1]}|}{\emptyset_{[k]} - \emptyset_{[k+1]}}} \tag{5.1.3}$$

regardless of the choice of search direction. For a 24 binary-digit machine, $\sqrt{\text{MACH\_EPS}} \approx 3 \cdot 10^{-4}$ limits the fractional accuracy to which $\mu_{[k]}$ can be determined.

Equation 5.1.3, exact for a quadratic cost function, is reasonable for general Ø. Write the Taylor series expansion along $d_{[k]}$ at $x_{[k+1]}$,

$$\text{Ø}\{\mu\} \approx \text{Ø}\{0\} + \mu \frac{d\text{Ø}}{d\mu} + \frac{\mu^2}{2} \frac{d^2\text{Ø}}{d\mu^2}$$

At the line minimum, $d\text{Ø}/d\mu = 0$. The second derivative term $d^2\text{Ø}/d\mu^2$ corresponds to $d_{[k]}^T G_{[k+1]} d_{[k]}$ in Equation 5.1.2. Estimating this term as $(\text{Ø}_{[k]} - \text{Ø}_{[k+1]})/\mu_{[k]}^2$, and dropping the factor $\sqrt{2}$, the same expression follows.

A different heuristic argument applied to the one-dimensional case [Press §10.1] and extended directly to multi-dimensional searches [Press §10.5] leads to

$$\frac{\Delta\mu_{\text{min}\text{Ø}[k]}}{\mu_{[k]}} = \sqrt{\text{MACH\_EPS}} \tag{5.1.4}$$

From Equation 5.1.3, this is exact only if $f_{[k+1]}$ is half of a positive $f_{[k]}$ (recall that a cost function may have negative values yet still have a positive-definite Hessian).

Numeric examples

Suppose a line search attempts to identify $\mu_{[k]}$ to within $\pm\Delta\mu_{\text{min}\text{Ø}[k]}$. For the sample steepest descent problem of Section 3.2, where $\text{Ø}_{[0]} = 1$ and $\text{Ø}_{[1]} = 0.1935$, Equation 5.1.3 gives $\Delta\mu_{\text{min}\text{Ø}[0]} \approx 0.5 \cdot \mu_{[0]} \sqrt{\text{MACH\_EPS}}$, about half the smallest change prescribed by Equation 5.1.4. In this case Equation 5.1.3 increases the accuracy of the search, at the expense of added trial stepsizes. Conversely, Equation 5.1.3 can save cost function evaluations (at no expense in accuracy) for search directions which do little to reduce the cost function.

To compare Equations 5.1.3 and 5.1.4 on nonquadratic cost functions, form $\text{Ø}_{(2)} = \Sigma r_i^2$ for one of the equilibrium problems defined in Chapter 1, choose a starting point, and minimize $\text{Ø}_{(2)}$ along some search direction. At the minimum, calculate the smallest resolvable change in $\mu_{[k]}$ for a hypothetical machine with coarser discretization than that used to perform the calculations. Then evaluate $\text{Ø}_{(2)}\{\mu_{[k]} \pm \Delta\mu_{\text{min}\text{Ø}[k]}\}$, and compare this cost function to that at the minimum.

Figure 5.1.2 shows such a comparison for the duct flow problem, for a variety of starting points and search directions, on a hypothetical machine with MACH_EPS = 10^{-4}. Consider the first row. Starting at $(0.02, 0.3, 1)^T$, search in the Newton-Raphson direction. The minimizing stepsize is 1.147.

Equation 5.1.3 estimates that stepsizes smaller than $4.051 \cdot 10^{-4}$ will not produce significant changes in the cost function on a machine with MACH_EPS = 10^{-4}. Evaluating the residuals at $\mu = 1.147 + 4.051 \cdot 10^{-4}$, r-square differs from that at the minimum by 0.82 parts per ten thousand. Ideally, the value differs by 1 part in ten thousand, the machine precision. By contrast, Equation 5.1.4 estimates the smallest resolvable stepsize as $1.147 \cdot 10^{-2}$. The cost function, evaluated at this offset, differs from the minimum value by 610 parts per ten thousand. For this starting point and search direction on the duct flow problem, Equation 5.1.3 gives the more accurate estimate.

$x_{[0]}$, $d_{[0]}$	$\emptyset_{(2)[0]}$	$\mu_{[0]}$, $\Delta\mu_{min}\emptyset_{[0]:5.1.3}$, $\Delta\mu_{min}\emptyset_{[0]:5.1.4}$	$\emptyset_{(2)}(\mu_{[0]})$, $\emptyset_{(2)}(\mu_{[0]}+\Delta\mu_{5.1.3})$, $\emptyset_{(2)}(\mu_{[0]}+\Delta\mu_{5.1.4})$	Difference, parts per 10^4
(0.02, 0.3, 1) $-\Delta x_{[0]:NR}$	0.976196	1.147 4.051E-4 1.147E-2	0.00121538 0.00121548 0.00128989	 0.82 610
(0.02, 0.3, 1) $g_{(2)[0]}$	0.976196	1.542E-5 1.918E-8 1.542E-7	0.0148835 0.0148846 0.0149495	 0.74 44
(0.02, 0.3, 1) (-1, 1, 100)	0.976196	3.664E-3 5.135E-5 3.664E-5	0.646886 0.646947 0.646917	 0.94 0.48
(0.02, 7, 1) $-\Delta x_{[0]:NR}$	46.4893	1.217 2.731E-3 1.217E-2	2.22881 2.22897 2.23197	 0.72 14
(0.02, 7, 1) $g_{(2)[0]}$	46.4893	9.637E-6 3.614E-7 9.637E-8	43.4033 43.4111 43.4039	 1.8 0.14
(0.02, 7, 1) (-1, -1, 5)	46.4893	1.010E-2 7.177E-4 1.010E-4	45.5866 45.5898 45.5867	 0.70 0.01

Figure 5.1.2. The smallest resolvable change in $\mu_{[k]}$ at a line minimum of $\emptyset_{(2)}$, calculated by Equations 5.1.3 and 5.1.4, on the duct flow problem, using MACH_EPS = 10^{-4}. Differences of nearly one part per 10000 indicate the rule correctly estimates the minimum resolvable $\Delta\mu$.

Figure 5.1.3 shows a similar series of starting points and search directions for the trigonometric function with problem dimension 10.

Both figures group the observations by their starting point, arranging those with a common starting point according to the quality of the minimum at $x_{[1]}$.

Note that the greater the fractional difference between $\emptyset_{[k+1]}$ and $\emptyset_{[k]}$-- that is, the better the search direction at minimizing $\emptyset$-- the more Equation 5.1.4 overestimates the smallest resolvable stepsize. Overestimating $\Delta\mu_{min}$ prevents the line search algorithm from identifying the minimum as accurately as possible, by spacing trial stepsizes farther apart than required.

For these very good search directions, Equation 5.1.3 tends to underestimate the smallest resolvable stepsize. Underestimating $\Delta\mu_{min}$ forces the line search algorithm to make more function evaluations than is necessary, by placing trial stepsizes so close together that they do not produce distinct function values. Unlike the overestimating case, a line search algorithm can detect this behavior easily, and increase $\Delta\mu_{min}$ accordingly.

Relatively poor search directions reverse the situation. Equation 5.1.3 tends to overestimate, and Equation 5.1.4 tends to underestimate, the smallest resolvable change in the minimizing stepsize. However Equation 5.1.3 does not overestimate by the large factors observed for Equation 5.1.3 with the profitable search directions.

$x_{[0]}$, $d_{[0]}$	$\emptyset_{(2)[0]}$	$\mu_{[0]}$, $\Delta\mu_{min\emptyset[0]:5.1.3}$, $\Delta\mu_{min\emptyset[0]:5.1.4}$	$\emptyset_{(2)}\{\mu_{[0]}\}$, $\emptyset_{(2)}\{\mu_{[0]}+\Delta\mu_{5.1.3}\}$, $\emptyset_{(2)}\{\mu_{[0]}+\Delta\mu_{5.1.4}\}$	Difference, parts per 10^4
(0.1, 0.1, .. , 0.1) $g_{(2)[0]}$	0.00707576	8.373E-1 4.949E-3 8.373E-3	0.00183178 0.00183206 0.00183257	 1.5 4.3
(0.1, 0.1, .. , 0.1) $-\Delta x_{[0]:NR}$	0.00707576	1.513E-1 2.276E-3 1.1513E-3	0.00490648 0.00490770 0.00490702	 2.5 1.1
(0.1, 0.1, .. , 0.1) (-1,-1,.., -1, -0.01)	0.00707576	2.243E-2 3.779E-4 2.243E-4	0.00523301 0.00523385 0.00523330	 1.6 0.56
(10, 10, .. , 10) $g_{(2)[0]}$	8717.84	5.121E-3 5.098E-6 5.121E-5	85.5592 85.5623 85.8730	 0.36 37
(10, 10, .. , 10) $-\Delta x_{[0]:NR}$	8717.84	6.368E-1 9.097E-4 6.368E-3	174.353 174.358 174.643	 0.34 17
(10, 10, .. , 10) (-100, 1, 1, .. , 1)	8717.84	2.558E-2 7.031E-4 2.558E-4	7698.95 7700.38 7699.14	 1.9 0.25

Figure 5.1.3. The smallest resolvable change in $\mu_{[k]}$ at a line minimum of $\emptyset_{(2)}$, calculated by Equations 5.1.3 and 5.1.4, on the 10-dimensional trigonometric function, using MACH_EPS = 10^{-4}.

Discretization of function values during search

In a search for a line minimum, both Equations 5.1.1 and 5.1.3 limit the accuracy of the search, and an algorithm should place trial stepsizes no closer to each other than the larger of $\Delta\mu_{minx}$ and $\Delta\mu_{min\emptyset}$. However, unlike the minimum spacing derived for the independent variables, $\Delta\mu_{min\emptyset}$ does not apply all along the search line, but only in the immediate vicinity of the minimum.

At a general test point $x_t = x_{[k]} - \mu_t d_{[k]}$, $d_{[k]}^T g_t \neq 0$. The curve of function values, steeper than at the line minimum, provides more opportunity for distinguishing $\emptyset\{\mu_t + \Delta\mu\}$ from $\emptyset\{\mu_t\}$. This fact should not discourage using Equation 5.1.3, however, since placing test points as close as possible to one another wastes function evaluations. A high linear density of test stepsizes far away from the minimum implies an inefficient search strategy. Moreover, suppose the algorithm fits local models, such as parabolae, to existing test points in order to choose successive trial stepsizes [Dennis §6.3.2, Press §10.2]. Then placing trial stepsizes so near to each other that the corresponding function values are only just distinguishable at machine precision implies rounding errors will swamp meaningful differences in the cost function during the data fit.

5.2 Terminating a Search Algorithm

Ideally, after a reasonable number of iterations a minimization method finds a point x* with a zero gradient and a positive-definite Hessian, satisfying Equation 3.2.14 and guaranteeing that, at least locally, x* uniquely minimizes the cost function. Finite-precision effects-- for example, the improbability that a line search terminates where the gradient is exactly zero, and the inexact machine representation of even a zero gradient-- force termination when $g \approx 0$. Then limitations of the search algorithm (for example, the convergence to saddle points as well as to minima, and the inability to distinguish local from global minima) mean that finding $g \approx 0$ does not necessarily indicate an approximate

solution to an equilibrium problem has been found. Finally, if the gradient is discontinuous at the minimum point, Equation 3.2.14 does not apply and the test $g^* \approx 0$ cannot succeed with any computational accuracy [Shor §2.1].

The task simplifies for equilibrium problems, which seek $r = 0$ where the global minimum of the cost function is known. In this case a direct test of the residual norm is appropriate [Golub §10.2.7, Dennis §10.4]. However the possibility of encountering local minima means that a test for stationary points still is required [Dennis §7.2].

Any iterative algorithm should limit the number of passes through its major loop, to force termination in the event it fails to converge [Golub §10.2.7].

Terminating a minimization problem

To detect a stationary cost function, compare its relative change to a preselected tolerance, terminating when

$$\frac{\Ø_{[k+1]} - \Ø_{[k]}}{|\Ø_{[k]}|} < \text{FUNC\_TOL}$$

To better estimate the baseline, substitute the average of the two cost function magnitudes for $|\Ø_{[k]}|$ in the denominator. Then to guard against division by zero, add a small value FUNC_SMALL to the denominator. For speed reasons, floating-point multiplication may be prefered to division; rearranging, terminate the algorithm when

$$2\cdot(\Ø_{[k+1]} - \Ø_{[k]}) < \text{FUNC\_TOL}\cdot(|\Ø_{[k+1]}| + |\Ø_{[k]}| + \text{FUNC\_SMALL}) \tag{5.2.1}$$

[Press §10.6]. (The factor two may be combined with FUNC_TOL in a single division during initialization.) A simpler expression results from dropping the average of the cost function magnitudes, in which case the factor of two, and one of the magnitudes on the right hand side, are omitted. Typical values for the constants are FUNC_TOL $\le 10^{-4}$ and FUNC_SMALL $= 10^{-10}$.

To detect a stationary point directly from the gradient, note its components $g_j = \partial\Ø/\partial x_j$ vary with the scaling of the cost function and of the variables. Therefore define the relative gradient with components

$$\text{relgrad}_j\{x\} = \frac{\text{relative rate of change of } \Ø}{\text{relative rate of change of } x_j} = \frac{\partial\Ø}{\Ø}\cdot\frac{x_j}{\partial x_j}$$

$$\text{relgrad}_j\{x\} = \frac{g_j\cdot x_j}{\Ø} \tag{5.2.2a}$$

[Dennis §7.2]. In practice a typical small value replaces x_j or $\Ø$ if they are too close to zero. This has approximately the same effect as adding the small value FUNC_SMALL to the cost function terms in Equation 5.2.1. Comparing the maximum component of the relative gradient to a predetermined tolerance completes the test: if

$$\|\text{relgrad}\|_\infty < \text{GRAD\_TOL} \tag{5.2.2b}$$

then terminate for a stationary point. A typical GRAD_TOL = (MACH_EPS)$^{1/3}$ [Dennis §A.II.4]. For a 24 binary-digit floating-point machine, this gives GRAD_TOL $\approx 5\cdot10^{-3}$.

In testing for a stationary point, note Equation 5.2.1 compares one number, $\Ø_{[k+1]}$, against another, $\Ø_{[k]}$; whereas Equation 5.2.2 checks each component of the gradient vector. Moreover, it is possible that an optimal point does not have a zero gradient, due to discontinuities in the gradient [Shor §1.4]. Thus any test for $g \approx 0$ should be supplemented by a test for a stationary function value; in fact Equation 5.2.1 might serve

as a screening test to determine whether to perform the more complete test of Equation 5.2.2.

A stalled solution, i.e. an $x_{[k+1]}$ insufficiently different from $x_{[k]}$, may also indicate a stationary point. A test for detecting small changes in the estimated solution is described for equilibrium problems below.

Terminating an equilibrium problem

For an equilibrium problem Ax = b, the residuals r = b - Ax, and a convenient test is

(5.2.3) $$\|r\|_2 < \text{NORM\_TOL} \cdot \|b\|_2$$

[Golub §10.2.7]. A variation with the r-square formulation includes the problem dimension:

(5.2.4) $$\|r\|_2 < \sqrt{n} \cdot \text{NORM\_TOL} \cdot \|b\|_2$$

[Press §2.7] where NORM_TOL = 10^{-6}. For general equilibrium problems, the 1-norm also can be used, e.g. by

(5.2.5) $$\|r\|_1 < \text{CRIT\_ONE\_NORM}$$

[Press §9.6]. Finally the ∞-norm allows the user to specify an absolute magnitude which no single residual component may exceed:

(5.2.6) $$\|r\|_\infty < \text{CRIT\_INF\_NORM}$$

[Dennis §7.2] The suggested default CRIT_INF_NORM = $(\text{MACH\_EPS})^{1/3}$ [Dennis §A.III.4]. Note, however, that for ill-conditioned problems, finding a small residual norm does not insure an accurate estimate of the solution [Dahlquist §5.5.1].

To detect stagnation in the solution, define the relative change in x

(5.2.7a) $$\text{relstep}_i = \frac{|x_{i[k+1]} - x_{i[k]}|}{|x_{i[k+1]}|}$$

where a typical small value of x_i replaces a nearly zero denominator. Alternately, add a small machine number to the denominator. Then compare the maximum of the relative component changes to a predetermined tolerance:

(5.2.7b) $$\|\text{relstep}\|_\infty < \text{STEP\_TOL}$$

[Dennis §7.2]. To find p significant digits in the solution, set STEP_TOL = 10^{-p}. The suggested default STEP_TOL = $(\text{MACH\_EPS})^{2/3}$ [Dennis §A.II.4]. For a 24 binary-digit floating-point machine, this gives STEP_TOL $\approx 2 \cdot 10^{-5}$.

In practice the stagnation test

(5.2.7c) $$|x_{i[k+1]} - x_{i[k]}| < \text{STEP\_TOL} \cdot \max\{|x_{i[k+1]}|, \text{typ}x_i\}$$

or

(5.2.7d) $$|x_{i[k+1]} - x_{i[k]}| < \text{STEP\_TOL} \cdot (|x_{i[k+1]}| + \text{STEP\_SMALL})$$

is performed component by component, terminating as soon as the condition is met.

The code in file TERMCOND.CPP, in Appendix 3, implements Equations 5.2.6 and 5.2.7d, and a variation on Equation 5.2.1 for checking the norm. The file also provides functions for checking the tolerances and setting them to their default values if necessary.

5.3 The Double Dogleg Algorithm

This section describes a single iteration of the double dogleg algorithm, updated for solving nonlinear equations with the cost function $\emptyset_{(2)}$.

Two code files

Appendix 3 lists the code in both STDDOGLG.CPP and DBLDOGLG.CPP. The first file gives the standard implementation, a straightforward adaptation of the algorithm given by [Dennis §6.4.2, §6.5, §A.IV.3]. As discussed in Section 4.2, the Newton-Raphson solution of the residual equations, computationally more direct than Newton's method, gives the same step as does Newton's method applied to the approximate cost function. Besides substituting $\Delta x_{[k]:NR}$ for $\Delta x_{[k]:N}$, the code makes other slight changes, listed below, to the published algorithm.

The second file, DBLDOGLG.CPP, logically identical to STDDOGLG.CPP, departs from it computationally. In particular it replaces matrix-vector and vector-vector expressions with the scalar equivalents developed in Section 4.2. The complete list of changes is given below.

Efficiency

For small problem sizes, avoiding the redundant calculations of $g_{(2)}{}^T\Delta x$ and $\Delta x^T G_{(2)}\Delta x$ can save on the order of 5% of the execution time. For example, solving the three-dimensional duct flow problem from a starting point $x_{[0]} = (90, 90, 90)^T$, the double dogleg algorithm requires 21 Jacobian and 59 residual iterations to satisfy the default termination criteria using double-precision variables on a Macintosh 7100/66. The standard algorithm, i.e. STDDOGLG.CPP, requires 19.7 seconds to solve from this starting point 10,000 times. Replacing the matrix-vector and vector-vector expressions cut the execution time by 4%, to 18.9 seconds. See Figure 5.4.4. Similarly, Figure 5.4.5 shows a 7% decrease in execution time using DBLDOGLG.CPP on the Powell badly scaled function from the standard starting point, $(0, 1)^T$.

For large problems, matrix factorization dominates the computational overhead. Then the execution time saved using the scalar equivalents from Section 4.2 may be negligible-- for example, 1 second out of 590, or 0.2%, when performing 4 Jacobian and 5 residual evaluations on a 1000-dimensional version of the Broyden tridiagonal function. See Figure 5.4.1.

The fact that matrix factorization dominates the solution time for large problems suggests re-arranging the algorithm to avoid calculating the Newton-Raphson solution when the trust length is less than the Cauchy length, i.e., when the trust length is small enough to restrict trial steps to the steepest descent direction. However, this condition was never met for any trial starting point on the test problems. Then the extra logic needed to defer calculating the Newton-Raphson step increases the execution time (to implement this feature requires slight structural changes to the code, and an additional vector of storage to carry $r_{[k]}$ concurrently with $r_{[k+1]}$). Therefore DBLDOGLG.CPP does not defer calculating the Newton-Raphson step.

Double dogleg algorithm

The subsections below describe a single iteration of the algorithm presented in DBLDOGLG.CPP.

Initialize loop variables

At the beginning of the iteration, $x_{[k]}$, $r_{[k]}$, and $\emptyset_{(2)[k]}$ are known. The trust region $\delta_{[k]}$ is known, except on the first iteration. In addition, program constants and user inputs such as CRIT_INF_NORM and STEP_TOL are known and do not vary from iteration to iteration.

Evaluate $J_{[k]}$. Find $(J^T r)_{[k]}$ and $(JJ^T r)_{[k]}$.

Solve $J_{[k]}\Delta x_{[k]:NR} = -r_{[k]}$. Find $\|\Delta x_{[k]:NR}\|$. On the first iteration, set $\delta_{[k]} = \|\Delta x_{[k]:NR}\|$.

Find next iterate

Seek a point at a step length $\delta_{[k]}$ along the double dogleg curve, to approximately minimize the model $f_{(2)[k]}$. Since $\delta_{[k]}$ may increase or decrease in response to the residual evalution at the trial iterate, this defines a new iterative task within the major iteration at $x_{[k]}$:

(1) If $\delta_{[k]} < \|\Delta x_{[k]:NR}\|$ and the double dogleg curve break points are unknown, find the double dogleg curve. See below.

(2) Place the iterate on the double dogleg curve. That is, find $\Delta x_{[k]}$ with length $\delta_{[k]}$ so that $x_{[k+1]}$ lies on the curve. See below.

(3) Evaluate the residuals at $x_{[k+1]}$.

(4) Update the current trust region. That is, change $\delta_{[k]}$ if necessary. See below.

(5) Return to step (1) above, or go on to prepare for the next iteration, as indicated by the trust region update.

Find double dogleg curve

Find $\|(J^T r)_{[k]}\|^2$, $\|(J^T r)_{[k]}\|^2/\|(JJ^T r)_{[k]}\|^2$, and $\|(J^T r)_{[k]}\|^4/\|(JJ^T r)_{[k]}\|^2$.

Find $\|\Delta x_{(2)[k]:SD}\|$ using Equation 4.2.12.

Find $\eta_{[k]}$ using Equation 4.2.13. Find $\|\Delta x_{[k]:CB}\| = \eta_{[k]} \cdot \|\Delta x_{[k]:NR}\|$.

Place iterate on curve

If $\delta_{[k]} \geq \|\Delta x_{[k]:NR}\|$, set $\delta_{[k]} = \|\Delta x_{[k]:NR}\|$. Set $\rho_{1[k]} = 0$ and $\rho_{2[k]} = 1$.

Otherwise, if $\delta_{[k]} \geq \|\Delta x_{[k]:CB}\|$, set $\rho_{1[k]} = 0$ and $\rho_{2[k]} = \delta_{[k]}/\|\Delta x_{[k]:NR}\|$.

Otherwise, if $\delta_{[k]} \leq \|\Delta x_{(2)[k]:SD}\|$, set $\rho_{1[k]} = \delta_{[k]}/\|\Delta x_{(2)[k]:SD}\|$ and $\rho_{2[k]} = 0$.

Otherwise, find $\lambda_{[k]}$ by solving Equation 4.2.14b, taking the positive square root in the quadratic formula. Set $\rho_{1[k]} = 1 - \lambda_{[k]}$. Set $\rho_{2[k]} = \lambda_{[k]} \cdot \eta_{[k]}$.

Find $\Delta x_{[k]}$ using Equations 4.2.10 and 4.2.15. Find $g_{(2)[k]}{}^T \Delta x_{[k]}$ and $\Delta f_{(2)[k]}$ using Equations 4.2.16 and 4.2.17a, respectively.

Set $x_{[k+1]} = x_{[k]} + \Delta x_{[k]}$.

Update current trust region

In case of an error evaluating $r_{[k+1]}$, if the trust region was increased during the current iteration then restore the previous values of $\delta_{[k]}$, $x_{[k+1]}$, $r_{[k+1]}$, and $\emptyset_{(2)[k+1]}$, accept the step, and go on to prepare for the next iteration. Otherwise, halve $\delta_{[k]}$ and return to find the new $\Delta x_{[k]}$.

Find $\emptyset_{(2)[k+1]} = r_{[k+1]}{}^T r_{[k+1]}$. If $r_{[k+1]} \approx 0$ according to Equation 5.2.6, terminate the algorithm.

If $\delta_{[k]}$ was increased during the current iteration, and results in a greater $\emptyset_{[k+1]}$ than before, then restore the previous values of $\delta_{[k]}$, $x_{[k+1]}$, $r_{[k+1]}$, and $\emptyset_{(2)[k+1]}$, accept the step, and go on to prepare for the next iteration.

Find $\Delta\emptyset_{(2)[k]}$ using Equation 3.6.3.

If $\Delta\emptyset_{(2)[k]} > \alpha \cdot g_{(2)[k]}{}^T \Delta x_{[k]}$ the average slope fails Equation 3.6.5. Check Equation 5.2.7, terminating if the solution is stagnated. Otherwise, if $\delta_{[k]}$ was increased during the current iteration, accept the current step and go on to prepare for the next iteration. Otherwise, calculate λ using Equation 3.6.6, adjust it if necessary to place it on $[\frac{1}{10}, \frac{1}{2}]$, reset $\delta_{[k]}$ to $\lambda \cdot \delta_{[k]}$, and return to find the new $\Delta x_{[k]}$.

If $\delta_{[k]}$ was reduced during the current iteration, or if $\delta_{[k]} = \|\Delta x_{[k]:NR}\|$, then accept the current step and go on to prepare for the next iteration.

Consider increasing $\delta_{[k]}$ for the current iteration. If the step satisfies either Equation 3.6.7 or 3.6.8, double $\delta_{[k]}$, store the current values of $\delta_{[k]}$, $x_{[k+1]}$, $r_{[k+1]}$, and $\emptyset_{[k+1]}$, and return to find the new $\Delta x_{[k]}$.

Otherwise, accept the current step and go on to prepare for the next iteration.

Prepare for next iteration

If an earlier test identified either $r_{[k+1]} \approx 0$ or $\Delta x_{[k]} \approx 0$, terminate the search.

Find $\delta_{[k+1]}$ using Equation 3.6.9.

Save $\delta_{[k+1]}$, $x_{[k+1]}$, $r_{[k+1]}$, and $\emptyset_{[k+1]}$ for the next iteration.

Changes in standard algorithm

The code in file STDDOGLG.CPP deviates from the published algorithm [Dennis §A.IV.3] as listed below:

(1.1) It does not scale the independent variable vectors.

(1.2) It sets the initial trust region length to the Newton-Raphson, not the Cauchy, length. This change seems reasonable since the algorithm is specialized for equation solving, rather than serving as a general minimization algorithm.

(1.3) It admits the possibility of residual evaluation errors, reacting by halving the trust region. Note this requires the residual evaluation routine to signal errors (such as divide by zero, square root of a negative number, and so forth) back to the solver.

(1.4) It includes a termination test for $r \approx 0$.

(1.5) If it increases $\delta_{[k]}$ to find a less positive value of $\emptyset_{[k+1]}$ than at the previous trial $x_{[k+1]}$, it does not also require the new point to pass the average slope test. This seems reasonable since the previous trial point passed the average slope test, or else the algorithm would not have attempted to increase the trust length.

(1.6) It does not calculate $\|\Delta x_{[k]}\|$, but uses $\delta_{[k]}$ directly, since the double dogleg algorithm always finds $\Delta x_{[k]}$ with the proper step length. (The published routine, intended for use with the hook algorithm as well, finds the step length from its definition since the hook algorithm does not always set $\|\Delta x_{[k]}\| = \delta_{[k]}$ exactly [Dennis §6.4].)

(1.7) It does not limit the maximum trust region, but allows a full Newton-Raphson step no matter what its length.

(1.8) It makes no provision for updating the Jacobian or its factorization, but re-evaluates the Jacobian at the beginning of every iteration.

(1.9) It does not estimate the condition number of the Jacobian.

(1.10) It does not perturb a Jacobian identified as singular.

(1.11) If it doubles the trust length during an iteration, it stores the previous value rather than assuming the previous value was half the increased value. Thus, if it truncates the increased trust length to the Newton-Raphson length, and if it rejects the Newton-Raphson step, the algorithm restores the correct previous value.

(1.12) It does not defer calculating $(JJ^Tr)_{[k]}$ until it is needed, but calculates it at the same time as $(J^Tr)_{[k]}$. Numerically this multiplies the vector J^Tr by the unfactored, rather than by the factored, Jacobian.

(1.13) It changes the sign of all the steps Δx. Thus it calculates $x_{[k+1]} = x_{[k]} - \Delta x_{[k]}$, rather than $x_{[k+1]} = x_{[k]} + \Delta x_{[k]}$ as defined throughout this thesis. This avoids the vector unary minus operation when solving $J_{[k]}\Delta x_{[k]:NR} = -r_{[k]}$, and avoids the scalar unary minus when calculating $\Delta x_{(2)[k]:SD}$. Note that the description of the algorithm given above uses $x_{[k+1]} = x_{[k]} + \Delta x_{[k]}$, to conform with the thesis text; the change to the negative sign occurs only in the code.

(1.14) It tests the termination criteria using functions in a separate code file. This enforces uniformity in the termination tests from one solution method to another.

(1.15) It tests for the first iteration directly, rather than by initializing the trust length to a nonpositive number.

Changes in modified algorithm

In addition to the changes to the published algorithm listed above, the code in file DBLDOGLG.CPP makes the following modifications:

(2.1) It calculates $g_{(2)[k]}{}^T\Delta x_{[k]}$ and $\Delta f_{(2)[k]}$, and places points on the cutback portion of the curve, using expressions based on the linear combination of the Cauchy and Newton-Raphson steps.

(2.2) It uses the sum of squares of the residuals, rather than half the sum of squares, as its cost function.

(2.3) It does not store $\Delta x_{(2)[k]:SD}$. Since it must store $(J^Tr)_{[k]}$ for use elsewhere, and since the step to the Cauchy point is just a scaled version of this vector, it simply stores the scale factor.

5.4 Numeric Results

The figures below compare the performance of STDDOGLG.CPP and DBLDOGLG.CPP, described above, on a number of equilibrium problems and starting points. In addition, they compare the Newton-Raphson algorithm, modified only to avoid errors in the residual evaluation by halving the step length in the Newton-Raphson direction. File NEWTRAPH.CPP in Appendix 3 lists the code.

All executables were tested on a Macintosh 7100/66, and built using the Symantec C++ Development System, Version 8.0, Release 5, with Version 8.1.0b4 of the MrC++ compiler. For this machine, the default termination tolerances, used for every test reported below, are CRIT_INF_NORM $\approx 6.1 \cdot 10^{-6}$ in Equation 5.2.6, and STEP_TOL $\approx 3.7 \cdot 10^{-11}$ in Equation 5.2.7.

Figure entries

Each figure gives, for a particular problem and a variety of starting points, the number of Jacobian and residual evaluations required to terminate, and the termination reason if not for a satisfactory solution. For significant running times, the figures also give the time in seconds required to solve the problem.

Chapter 1 defines standard starting points for each problem. The test problems from the literature are intended to be tested from multiples of 1, 10, and 100 times the standard starting points [Moré81 §4]; the figures give the multiple used, if appropriate.

Termination

The nonlinear equation solvers terminate for: (1) a zero residual vector, determined by Equation 5.2.6; (2) a stagnated solution vector, determined by Equation 5.2.7d, and denoted in the figures by [stag]; (3) a nondecreasing residual norm [nondec]; (4) an error encountered when evaluating the residual vector or Jacobian matrix [err]; (5) an error solving the linearized system of equations, indicating a singular Jacobian to machine precision [sing]; or (6) exceeding the maximum number of iterations [itns]. These conditions correspond roughly to the return criteria of [Hiebert82 §2], except that a stagnated solution does not count as a root acceptance.

File PROBDEFN.HPP communicates these conditions from the solver to the driver code, while PROBDEFN.CPP diagnoses the termination condition for the user. Appendix 3 lists both files.

In addition to the explicit termination conditions listed above, the calculations may fail due to undiagnosed evaluation errors. The termination depends on the machine. For example, the Symantec environment represents an infinite (to machine precision) residual as INF, and carries on with the calculation. Thus the Newton-Raphson procedure, which does not react to the resulting infinite residual norm, continues to find NAN (not a number) values for the next Newton-Raphson step, and hence NAN values for the next iterate and residuals there. Since NAN tests as a small number, the procedure terminates and reports a zero residual vector. Other machines might not handle the original infinite residual in the same way. The figures below denote this condition by [nan].

Since STDDOGLG.CPP and DBLDOGLG.CPP implement the same algorithm, in general they take the same number of Jacobian and residual evaluations to terminate. However, roundoff differences between the two sometimes cause their solution trajectories to diverge over a large number of iterations, especially when the solution stagnates. See for example Figure 5.4.5, for the Powell badly scaled function.

Broyden tridiagonal function

Figure 5.4.1 shows that the double dogleg algorithm takes the full Newton-Raphson step at each iteration. From the 1000-dimensional case, the computational overhead of the double dogleg algorithm adds about 1.5% to the execution time of the Newton-Raphson algorithm, even using the scalar expressions for evaluating the initial slope and expected decrease in r-square.

dimension	5			50		1000
starting point	std	10·std	100·std	std	100·std	std
NEWTRAPH	4, 5	7, 8	10, 11	4, 5	10, 11	4, 5, 581
STDDOGLG	4, 5	7, 8	10, 11	4, 5	10, 11	4, 5, 590
DBLDOGLG	4, 5	7, 8	10, 11	4, 5	10, 11	4, 5, 589

Figure 5.4.1. Results for the standard algorithms on the Broyden tridiagonal function. Data pairs give the number of Jacobian and residual evaluations, respectively, to solution. Triples also give the run time in seconds.

Discrete boundary value function

Like the Broyden tridiagonal function, and like the discrete integral equation function tabulated below, all the implementations take the full Newton-Raphson step at each iteration, and the trust region requirements of the double dogleg algorithm add slightly to its computational overhead.

dimension	10			100		1000
starting point	std	10·std	100·std	std	100·std	std
NEWTRAPH	2, 3	3, 4	8, 9	1, 2	7, 8	1, 2, 145
STDDOGLG	2, 3	3, 4	8, 9	1, 2	7, 8	1, 2, 147
DBLDOGLG	2, 3	3, 4	8, 9	1, 2	7, 8	1, 2, 147

Figure 5.4.2. Results for the discrete boundary value function.

Discrete integral equation function

dimension	10			100		500
starting point	std	10·std	100·std	std	100·std	std
NEWTRAPH	2, 3	3, 4	8, 9	2, 3	8, 9	2, 3, 30.0
STDDOGLG	2, 3	3, 4	8, 9	2, 3	8, 9	2, 3, 31.0
DBLDOGLG	2, 3	3, 4	8, 9	2, 3	8, 9	2, 3, 31.0

Figure 5.4.3. Results for the discrete integral equation function.

Duct flow problem

Figure 5.4.4 gives, in addition to the number of Jacobian and residual evaluations needed to solve the duct flow problem from a variety of starting points, the time in seconds required to solve the problem from that starting point 10,000 times in a row.

The observation at $x_{[0]} = (0.001, 0.0039, 34.06)^T$ is the only known starting point for which the scalar expression version of the double dogleg algorithm takes longer to run. (Note that this starting point was chosen near a stagnation point in a steepest descent search using exact line minimization on r-square. However, this does not explain why the scalar version of the double dogleg algorithm takes longer to run.)

starting point	$(0.02, 7, 1)^T$	$(0.001, 0.0039, 34.06)^T$	$(60, 60, 60)^T$	$(90, 90, 90)^T$
NEWTRAPH	8, 9, 4.07	14, 64, 8.89	18, 45, 9.48	19, 46, 9.95
STDDOGLG	8, 9, 5.58	8, 34, 8.72	18, 54, 17.7	21, 59, 19.7
DBLDOGLG	8, 9, 5.22	8, 34, 9.12	18, 54, 17.0	21, 59, 18.9

Figure 5.4.4. Results for the duct flow problem. Times are for 10,000 successive runs from the given starting point.

Powell badly scaled function

Besides the points shown in Figure 5.4.5, using 100 times the standard starting point on the Powell badly scaled function makes the Jacobian singular to machine precision.

starting point	std	5·std	10·std	$(-10, -9.9)^T$	$(10, 20)^T$
NEWTRAPH	11, 12, 3.32	7, 8	4, 5	91, 92	2, 3, nan
STDDOGLG	24, 29, 13.2	25, 30	4, 5	33, 59, stag	39, 52, 22.5
DBLDOGLG	24, 29, 12.3	25, 30	4, 5	34, 58, stag	39, 52, 21.3

Figure 5.4.5. Results for the Powell badly scaled function. Times are for 10,000 successive runs.

Powell singular function

starting point	std	10·std	100·std
NEWTRAPH	11, 12	14, 15	18, 19
STDDOGLG	11, 12	14, 15	18, 19
DBLDOGLG	11, 12	14, 15	18, 19

Figure 5.4.6. Results for the Powell singular function.

Rosenbrock function

dimension	2				10	100
starting point	std	10·std	100·std	$(20, 20)^T$	std	std
NEWTRAPH	2, 3	2, 3	2, 3	2, 3	2, 3	2, 3
STDDOGLG	16, 23	3, 5	3, 5	100, 103, itns	16, 23	16, 23
DBLDOGLG	16, 23	3, 5	3, 5	100, 103, itns	16, 23	16, 23

Figure 5.4.7. Results for the extended Rosenbrock function.

Trigonometric function

In addition to the points shown in Figure 5.4.8, solving the trigonometric function with n = 5 from 100 times the standard starting point requires 80 Jacobian evaluations; the double dogleg algorithms each require 14 Jacobian and 20 residual evaluations to solve the problem.

dimension	5			10	50
starting point	std	5·std	10·std	std	std
NEWTRAPH	5, 6	84, 85	49, 50	6, 7	8, 9
STDDOGLG	8, 12	71, 100, stag	75, 103, stag	69, 96, stag	100, 115, itns
DBLDOGLG	8, 12	72, 101, stag	77, 106, stag	71, 97, stag	100, 115, itns

Figure 5.4.8. Results for the trigonometric function.

CHAPTER 6
EXTENDED EXAMPLE

This chapter demonstrates, on the two-dimensional trigonometric function, the application of function minimization to equation solving. It highlights the important distinctions between the methods of Chapters 2 and 3, and reviews the connections that Chapter 4 shows exist between them.

Section 6.1 compares the first-order models formed by Newton-Raphson's method for nonlinear equations, and the second-order model used in function minimization.

Sections 6.2 and 6.3 deal with the linearized residuals only. Section 6.2 shows how equation solvers extract information from these models, and Section 6.3 calculates sample search paths for a descent-based search on three sample residual norms.

Section 6.4 illustrates a complete double dogleg solution of the two-dimensional trigonometric function.

6.1 Linear and Quadratic Models

Newton-Raphson's method for equation solving, and Newton's method for function minimization, model their functions of interest differently. Where Newton-Raphson forms n linear models, one for each residual function, Newton treats only a single cost function, using a quadratic model. This section compares the two models.

Residual behavior

Equation 1.5.9c defines the two-dimensional trigonometric problem:

$$r_1 = 3 - 2\cos(x_1) - \sin(x_1) - \cos(x_2)$$
$$r_2 = 4 - 3\cos(x_2) - \sin(x_2) - \cos(x_1)$$

To apply Newton's method to solve this system would require calculating a norm such as $\emptyset_{(2)} = r_1^2 + r_2^2$ and forming its quadratic model. For simplicity, this section compares the linear and quadratic models of r_1 only; Section 4.2 shows the connection between the quadratic models of r_1 and r_2, and the quadratic model of $\emptyset_{(2)}$.

Figure 6.1.1 shows r_1 for both large and small excursions about $x_{[0]}$. The first subplot, Figure 6.1.1a, shows r_1 over a range of x wide enough to suggest its periodic nature. Because both r_1 and r_2 are periodic in x_1 and x_2, the fundamental roots $x^{**} = (0, 0)^T$ and $(0.2431, 0.6127)^T$ repeat at intervals of $2 \cdot \pi$ in each coordinate direction.

Evaluating r_1 at the starting point $x_{[0]} = (0.1, 0.7)^T$ gives $r_{1[0]} = 0.1453$. The figure marks this initial point $(x_{[0]}, r_{[0]})$ with a solid dot on the surface of r_1. Since the models of r_1 formed at this point represent its local behavior only, Figure 6.1.1b depicts r_1 on a narrower range of x.

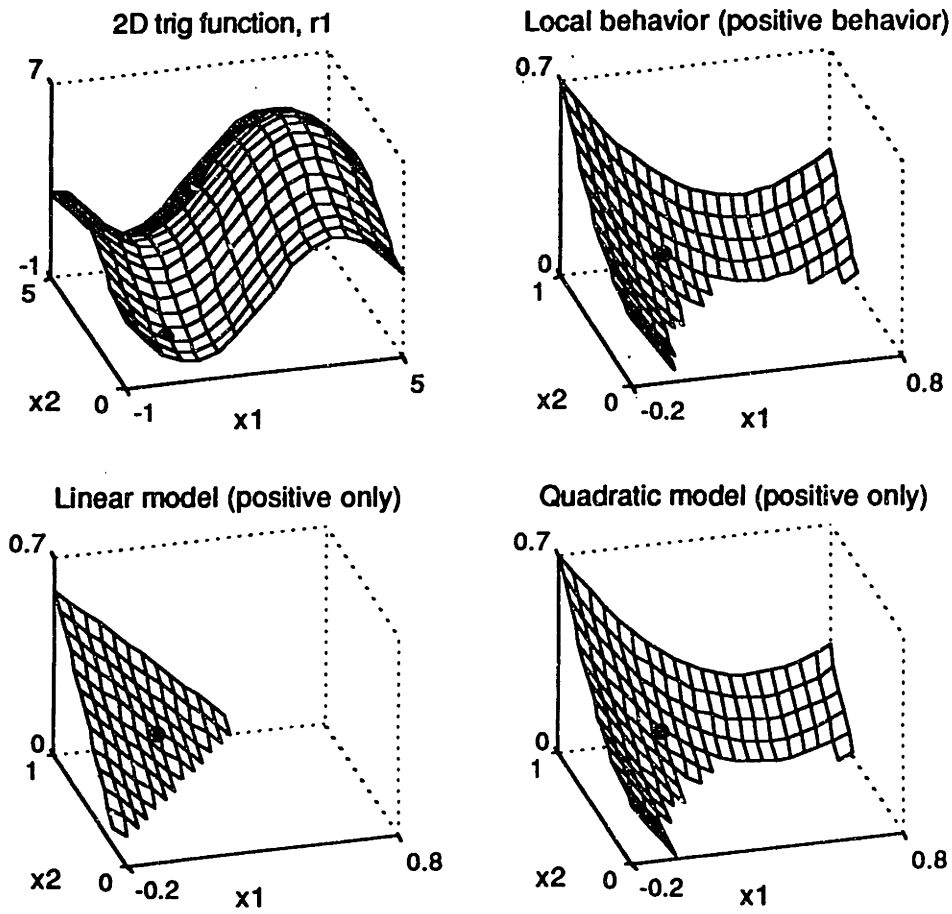

Figure 6.1.1. The first residual equation of the two-dimensional trigonometric function. (a) The residual relation repeats with period $2 \cdot \pi$ in each coordinate direction. (b) The local behavior of r_1 determines the model behavior. (c) The first-order model captures function and slope information only. (d) The second-order model includes curvature as well. A dot marks the point of model formation; i.e. the point of tangency between the models and the actual curve.

Linear model

Newton-Raphson's method expands r_1 in the first-order Taylor series of Equation 2.2.4,

$$\hat{r}_{1[k]}\{x\} = r_{1[k]} + \nabla r_{1[k]}^T (x - x_{[k]})$$

The gradient

$$\nabla r_1 = \begin{pmatrix} \frac{\partial r_1}{\partial x_1} \\ \frac{\partial r_1}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 2\sin(x_1) - \cos(x_1) \\ \sin(x_2) \end{pmatrix}$$

becomes the first row of the Jacobian matrix; see Equation 2.2.7b. Evaluating at $x_{[0]} = (0.1, 0.7)^T$ gives

(6.1.1) $$\hat{r}_{1[0]} = 0.1453 + (-0.7953 \;\; 0.6442)\begin{pmatrix} x_1 - 0.1 \\ x_2 - 0.7 \end{pmatrix}$$

Figure 6.1.1c shows this linear model for positive predicted values. Because the model captures only the local behavior of the actual residual, to aid comparison Figure 6.1.1b shows r_1 in the same neighborhood of $x_{[0]}$, again above the plane $r = 0$. The linear model has the same function value and slope as has r_1 at $x_{[0]}$, so the plane depicted in the third subplot is tangent to the surface depicted in the second subplot, just touching it at the point $(x_{[0]}, r_{[0]})$ marked on both surfaces.

Quadratic model

Newton's method uses a second-order Taylor series expansion of the function of interest. Consider a second-order model of r_1; for convenience call it $R_{1[k]}$. By Equation 3.2.9a,

$$R_{1[k]}\{x\} = r_{1[k]} + \nabla r_{1[k]}{}^T \Delta x_{[k]} + \tfrac{1}{2} \Delta x_{[k]}{}^T \nabla^2 r_{1[k]} \Delta x_{[k]}$$

where $\Delta x_{[k]} = x - x_{[k]}$. The Hessian

$$\nabla^2 r_1 = \begin{bmatrix} \frac{\partial^2 r_1}{\partial x_1{}^2} & \frac{\partial^2 r_1}{\partial x_1 \partial x_2} \\ \frac{\partial^2 r_1}{\partial x_2 \partial x_1} & \frac{\partial^2 r_1}{\partial x_2{}^2} \end{bmatrix} = \begin{bmatrix} 2\cos(x_1)+\sin(x_1) & 0 \\ 0 & \cos(x_2) \end{bmatrix}$$

is symmetric, as expected. In this case the off-diagonal elements are zero since no term of $r_1\{x\}$ mixes x_1 and x_2, but this does not occur in general. Evaluating at $x_{[0]}$,

(6.1.2) $$R_{1[0]} = \hat{r}_{1[0]} + \tfrac{1}{2}(x_1\text{-}0.1 \;\; x_2\text{-}0.7)\begin{bmatrix} 2.090 & 0 \\ 0 & 0.7648 \end{bmatrix}\begin{pmatrix} x_1 - 0.1 \\ x_2 - 0.7 \end{pmatrix}$$

Figure 6.1.1d shows $R_{1[0]} > 0$ in the neighborhood of $x_{[0]}$. By accounting for curvature, Equation 6.1.2 represents r_1 over a greater region than the linear model (it also requires more storage and greater computational effort). In a minimization method, the first-order model suggests only the steepest descent direction-- following the negative gradient towards less positive function values. The second-order model, on the other hand, has a minimum, suggesting not only appropriate search directions, but also the appropriate step lengths.

6.2 Exercising the First-Order Models

Equation 6.1.1 gives the first row of the Jacobian model of Equation 2.2.8,

$$\hat{r}_{[k]}\{x\} = r_{[k]} + J_{[k]}\Delta_{[k]}$$

To complete the model, linearize r_2 about $x_{[0]}$ to find

(6.2.1) $$\hat{r}_{[0]} = \begin{pmatrix} 0.1453 \\ 0.06625 \end{pmatrix} + \begin{bmatrix} -0.7953 & 0.6442 \\ 0.09983 & 1.168 \end{bmatrix}\begin{pmatrix} x_1 - 0.1 \\ x_2 - 0.7 \end{pmatrix}$$

A typical solver uses this linearized model in two ways: (1) in Newton-Raphson's method, to calculate the step $\Delta x_{[0]:NR}$ which zeros both linearized residuals; and (2) in a descent-based method, to choose intermediate points between $x_{[0]}$ and $x_{[1]:NR}$, in the event the full Newton-Raphson step fails (or falls outside the trust length).

Newton-Raphson versus descent-based methods

Newton-Raphson's method seeks to zero the residuals directly, by zeroing their linear models. A descent-based method, on the other hand, tries to zero the residuals as the indirect result of minimizing some residual norm. Therefore modifying Newton-Raphson's method with a descent requirement involves more than simply accepting or rejecting the Newton-Raphson step depending on whether it decreases the norm. The methods of function minimization use a model of the norm to define a search path, and to select appropriate test points along that path. For instance, the double dogleg algorithm defines a search path based on the cost function's steepest descent and Newton directions-- calculated using its second-order model-- then places points along this path by a set of rules which extend the trust length when the model performs well, and reduce it when the model predicts the norm poorly.

Following the techniques of function minimization, a descent-based equation solver models a residual norm, usually by combining the same linear residual models that Newton-Raphson uses. Because it seeks to define and evaluate a search path, rather than to calculate a single trial point, the modified solver extracts additional information from the models.

Extracting model information

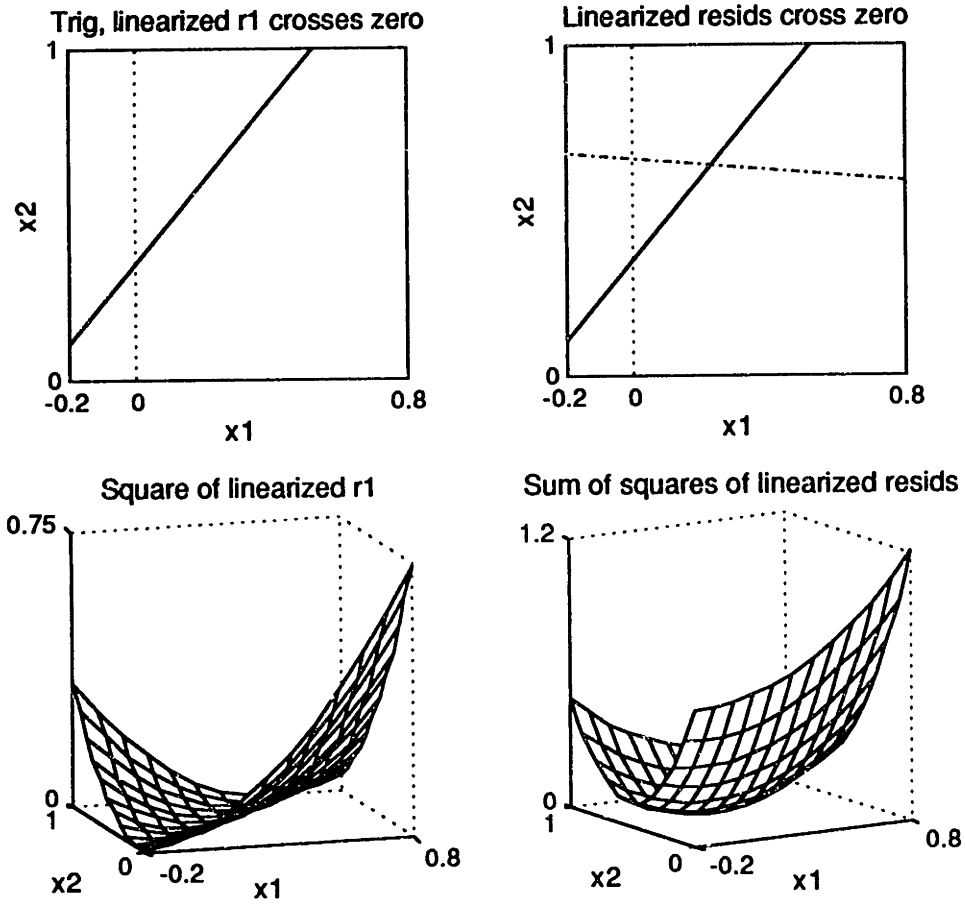

Figure 6.2.1. Newton-Raphson's method and minimization-based equation-solving methods extract different information from the linearized residual models. (a, b) Newton-Raphson zeros each residual model. (c, d) Descent methods model the norm (here, r-square) by substituting the linearized residuals.

Figure 6.2.1 shows how an algebraic equation solver such as the double dogleg method of Section 5.3 extracts information from the linear residual models. The two plots on the left show the information of interest in the first linearized residual function, while the rightmost plots show how the Newton-Raphson and function minimization methods combine the residual models.

Newton-Raphson's method solves the linear system $\hat{r}_{[k]} = 0$. For example, setting $\hat{r}_{[0]} = 0$ in Equation 6.2.1 gives $x_{[1]:NR} = (0.2279, 0.6323)^T$. According to the geometric interpretation of Section 2.2, each row of the linearized system gives the hyperplane which lies tangent to the corresponding residual, and Newton-Raphson's method finds the common intersection of these tangent hyperplanes in the hyperplane $r = 0$. Recall that Figure 6.1.1c shows the plane tangent to r_1 at $x_{[0]}$, cut off at its intersection with the plane $r = 0$. The line where the tangent plane intersects $r = 0$ defines a range of points satisfying $\hat{r}_{1[0]} = 0$. Figure 6.2.1a shows this first line of intersection. Any point along this line sets the first linearized residual to zero, but Newton-Raphson also must set $\hat{r}_{2[0]} = 0$. Figure

6.2.1b adds the line where the plane tangent to r_2 at $x_{[0]}$ crosses $r = 0$. The two lines intersect at $(0.2279, 0.6323)^T$, the Newton-Raphson solution.

To implement the descent requirement, the double dogleg algorithm constructs a model of the residual norm by substituting the linearized residuals into the relation $\emptyset\{r\}$ which defines the norm. For example, the third subplot of Figure 6.2.1 shows $\hat{r}_{1[0]}^2$, which approximates r_1^2 in the neighborhood of $x_{[0]}$. Of course the paraboloid shown in the figure touches the zero plane along the same line $\hat{r}_{1[0]} = 0$ shown in the first subplot. Figure 6.2.1d completes the model of $\emptyset_{(2)}$. It shows

$$f_{(2)[0]} = \hat{r}_{1[0]}^2 + \hat{r}_{2[0]}^2 = \hat{r}_{[0]}^T \hat{r}_{[0]}$$

Substituting $\hat{r}_{[0]}$ from Equation 6.2.1 gives

(6.2.2)
$$f_{(2)[0]} = 0.02251 + (-0.2179 \;\; 0.3420)\begin{pmatrix} x_1 - 0.1 \\ x_2 - 0.7 \end{pmatrix} + (x_1 - 0.1 \;\; x_2 - 0.7)\begin{bmatrix} 0.6425 & -0.3958 \\ -0.3958 & 1.779 \end{bmatrix}\begin{pmatrix} x_1 - 0.1 \\ x_2 - 0.7 \end{pmatrix}$$

Comparing with the second-order Taylor series expansion, the gradient $g_{(2)[0]} = (-0.2179, 0.3420)^T$, and the Hessan $G_{(2)[0]}$ is twice the matrix in the third term.

Norm models built from linearized residuals

Equation 6.2.2 models r-square starting with the linear residual models, and yields a gradient and Hessian for use in the function minimization techniques of Chapter 3. An equation solver can apply any Hessian-based technique of function minimization to this model. For example, to try a steepest descent step it would take $d_{[k]} = g_{(2)[0]}$. From Equation 3.2.11a, $\hat{\mu}_{[0]:SD} = 0.2763$. Then by Equation 3.1.4 the Cauchy point

$$x_{[0]} - 0.2763 \cdot g_{(2)[0]} = (0.1602, 0.6055)^T$$

minimizes the quadratic of Equation 6.2.2 in its steepest descent direction from $x_{[0]}$.

Equations 4.2.8 through 4.2.17 reformulate the important function minimization results, for r-square, directly in terms of the Jacobian matrix and residuals at $x_{[k]}$. For example Equation 4.2.10 gives the Cauchy step as

$$\Delta x_{(2)[0]:SD} = \frac{-\|(J^T r)_{[0]}\|^2}{\|(JJ^T r)_{[0]}\|^2} (J^T r)_{[0]} = \begin{pmatrix} 0.06022 \\ -0.09450 \end{pmatrix}$$

from which $x_{[0]} + \Delta x_{(2)[0]:SD} = (0.1602, 0.6055)^T$, as found above. These expressions are more efficient than actually forming a model of the norm, as in Equation 6.2.2, and applying the expressions of Chapter 3. Similarly, applying Newton's method, Equation 3.5.9, to the quadratic gives

$$x_{[1]:N} = x_{[0]} - G_{(2)[0]}^{-1} g_{(2)[0]} = (0.2279, 0.6323)^T$$

which, by Equation 4.2.7, equals the Newton-Raphson step on the original linearized system.

Even though most equation solvers institute their descent requirement on r-square, substituting $\hat{r}$ in any other norm would yield similar expressions. Section 4.3 treats this procedure for a general norm $\emptyset$. This treatment yields two important results: (1) it identifies the sources of curvature in the model of any norm built up from linear residual

models; and (2) it demonstrates the connection between the Newton direction on the norm, and the Newton-Raphson direction on the original system of equations.

First, consider curvature in the model of the norm. Equation 4.3.11 states that the model Hessian formed from the linearized residual relations expresses only the curvature $G \approx J^T P J$, where P represents the algebraic details of how the norm combines the residuals, and is constant over all x for a given norm. The Hessian approximation neglects the terms $p_i \cdot \nabla^2 r_i$ which represent the contributions from curvature in the actual residuals. Returning to Figure 6.2.1c, note that the paraboloid formed by squaring $\hat{r}_{1[0]}$ has curvature only transverse to the line where the linearized residual cuts the zero plane. This curvature results entirely from squaring the linear residual model, and carries no information about second-order effects in r_1. Similarly, all the second-order information depicted in $f_{(2)[0]}$ in Figure 6.2.1d results from algebraic manipulation of the linearized residuals, and none of it from nonlinearities in the actual residual functions. That is, its curvature results from algebraic combinations of the linearized residuals in J, rather than from the second-order effects in the $\nabla^2 r_i$.

Since Newton's method acts on curvature information, the Newton direction does not always match the Newton-Raphson direction. Equation 4.3.13 states that, for the model of a norm formed from the linearized residuals, the Newton step equals the Newton-Raphson step only for the weighted r-square family of norms. Comparing the right hand plots of Figure 6.2.1, the quadratic model of r-square has a minimum at the same $x = (0.2279, 0.6323)^T$ which zeros $\hat{r}_{[0]}$. While this holds for any residual norm constructed from the linearized residuals, it does not mean a Newton step on the model of any norm finds that minimizer. One iteration of Newton's method exactly minimizes only a pure quadratic, and the only way to develop a pure quadratic when combining linear functions is to sum their weighted squares.

6.3 Search Paths

After extracting the model information (such as $g_{(2)[0]}$ and $G_{(2)[0]}$ identified in Equation 6.2.2), a descent-based equation solver chooses a search path. In the double dogleg algorithm of Section 5.3, the path follows the steepest descent direction of r-square to the Cauchy point, then cuts back to the Newton-Raphson direction, and continues on to the full Newton-Raphson step.

Double dogleg path for r-square

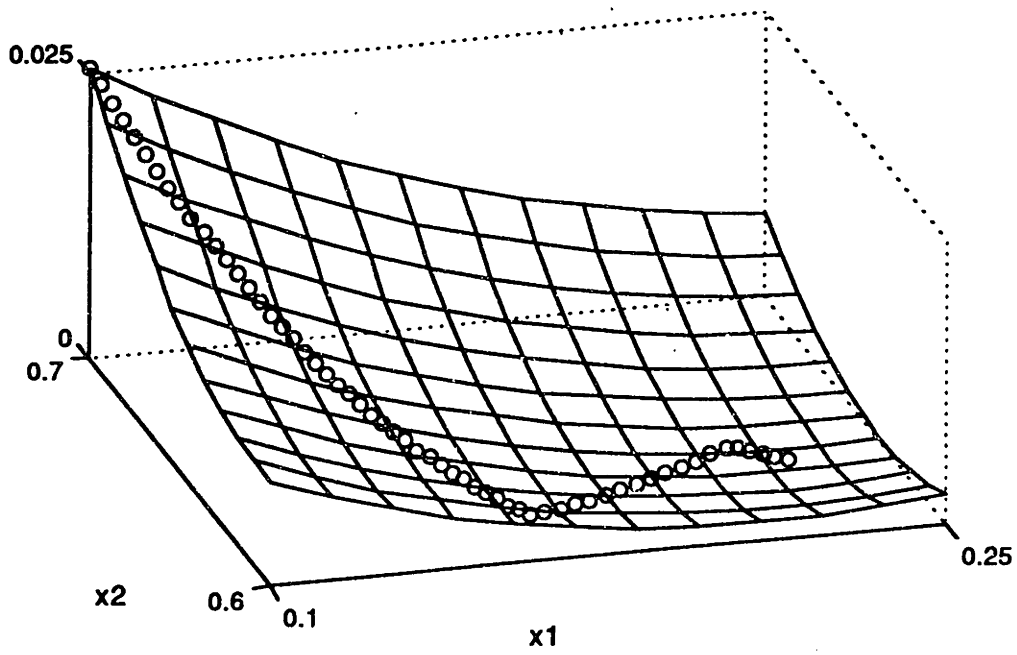

Figure 6.3.1. The predicted residuals along the double dogleg path, and in the region around it. The mesh extends from $x_{[0]}$ in the back left corner, to beyond the minimum of $f_{(2)[0]}$ near the front right corner.

From the work above, the Cauchy point for $f_{(2)[0]}$ is $(0.1602, 0.6055)^T$ and the Newton-Raphson point $x_{[1]:NR} = (0.2279, 0.6323)^T$. The cutback point lies at $x_{[0]} + \eta_{[0]} \cdot \Delta x_{[0]:NR}$. Equation 4.2.13 gives $\eta_{[0]} = 0.9126$, so the cutback point lies at $(0.2167, 0.6382)^T$. Figure 6.3.1 shows the predicted residuals along the double dogleg path, and the surface of the predicted residuals. The residuals may be estimated by Equation 6.2.2, or, more directly, by Equation 4.2.8. Note that the path sets out in the steepest descent direction, and that $f_{(2)[0]}$ decreases continuously along the path all the way to the predicted minimum at the Newton-Raphson point.

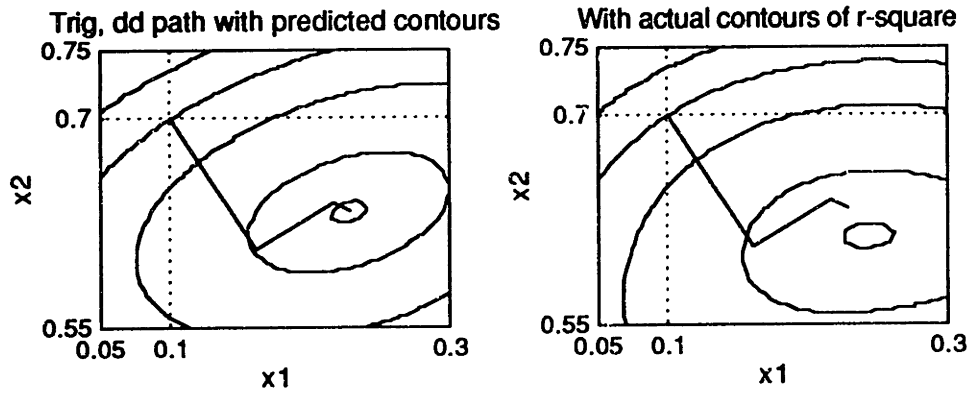

Figure 6.3.2. The double dogleg search path. (a) Superposed on the contours of r-square predicted by Equation 6.2.2, the double dogleg path first minimizes $f_{(2)[0]}$ in its steepest descent direction, and ends at its global minimum. (b) Superposed on the actual contours of r-square, the path misses both minima but defines a reasonable selection of search points over most of its range.

Figure 6.3.2a compresses this information to two dimensions by plotting the double dogleg path along with some level curves of $f_{(2)[0]}$. The double dogleg path begins perpendicular to the level curve of r-square at $x_{[0]}$, because it follows the steepest descent direction (see the discussion of Equation 3.2.3). The first leg of the path ends parallel to a new level curve of $f_{(2)[0]}$, since the Cauchy point minimizes $f_{(2)[0]}$ in the steepest descent direction. As noted for Figure 6.3.1, the path decreases the predicted norm right to its minimum at the full Newton-Raphson step.

The second plot of Figure 6.3.2 shows the same path, but draws the level curves for the actual norm. The path still begins perpendicular to the level curve $\emptyset_{(2)} = \emptyset_{(2)}\{x_{[0]}\}$, because the first-order changes in the residual norm depend only on the first-order residual effects-- linearizing the residuals does not compromise the gradient direction choice. However, the Cauchy point falls short of the actual minimizer of r-square in the $g_{(2)[0]}$ direction, due to nonlinearities in the residual equations. Similarly, the Newton-Raphson point misses the solution to the equations, and hence misses the actual minimum of $\emptyset_{(2)}$.

Figure 6.3.2b suggests how the double dogleg path may actually increase $\emptyset_{(2)}$ over some regions of the search. Somewhere between the Cauchy and cutback points, the double dogleg path falls tangent to a level curve, so that for larger step lengths, up to the cutback point, $\emptyset_{(2)}$ increases. However the norm decreases again for steps beyond the cutback point, all the way to the Newton-Raphson point where the double dogleg curve terminates.

Similarly, if the Cauchy point lies beyond the true minimum of $\emptyset_{(2)}$ in the steepest descent direction, then $\emptyset_{(2)}$ will increase along part of the first leg of the double dogleg curve (though again it may decrease over later sections). In short, nonlinearities may make the norm behave quite differently from its expected performance along the search path. This problem is more severe for equation-based searches than for true function minimization procedures, because function minimization does not compromise the Hessian by dropping its important second-order effects. Thus for function minimization, the Cauchy point reasonably may be expected to lie closer to the true steepest descent

minimizer than does the Newton point to the true function minimum; for a residual norm approximated by the linearized residuals, this may not be the case.

Alternate search directions

As indicated in Chapter 4, $\emptyset_{(2)}$ is not the only possible norm, although Chapter 5 defines only the standard double dogleg algorithm, which employs r-square exclusively. As indicated by Equation 4.3.13, to preserve the property $\Delta x_{[k]:N} = \Delta x_{[k]:NR}$ requires using a weighted r-square norm, formalized by Equation 4.5.1 as

$$\emptyset_{(w)} = \sum_{i=1}^{n} w_i \cdot r_i^2$$

Any selection of the $w_i > 0$ allows the Newton-Raphson step to substitute directly for the Newton step in a function minimization algorithm.

Consider the steepest descent direction of such a norm. The geometric interpretation of Section 4.3 states that the gradient of a residual norm is just a weighted sum of the residual gradients, i.e., of the Jacobian rows. Equation 4.3.6 gives

$$g = \sum_{i=1}^{n} p_i \cdot \nabla r_i$$

and for the weighted r-square norm, Equation 4.5.4b gives

$$p_{i(w)} = 2 \cdot w_i \cdot r_i$$

For example, r-square has $w_i = 1$ for $1 \le i \le n$, and so in the sample problem

$$g_{(2)} = 2 \cdot r_1 \cdot \nabla r_1 + 2 \cdot r_2 \cdot \nabla r_2$$

giving

(6.3.1) $$g_{(2)[0]} = 2 \cdot 0.1453 \begin{pmatrix} -0.7953 \\ 0.6442 \end{pmatrix} + 2 \cdot 0.06625 \begin{pmatrix} 0.09983 \\ 1.168 \end{pmatrix} = \begin{pmatrix} -0.2179 \\ 0.3420 \end{pmatrix}$$

Equation 6.3.1 gives the same $g_{(2)[0]}$ as calculated above, in Equation 6.2.2.

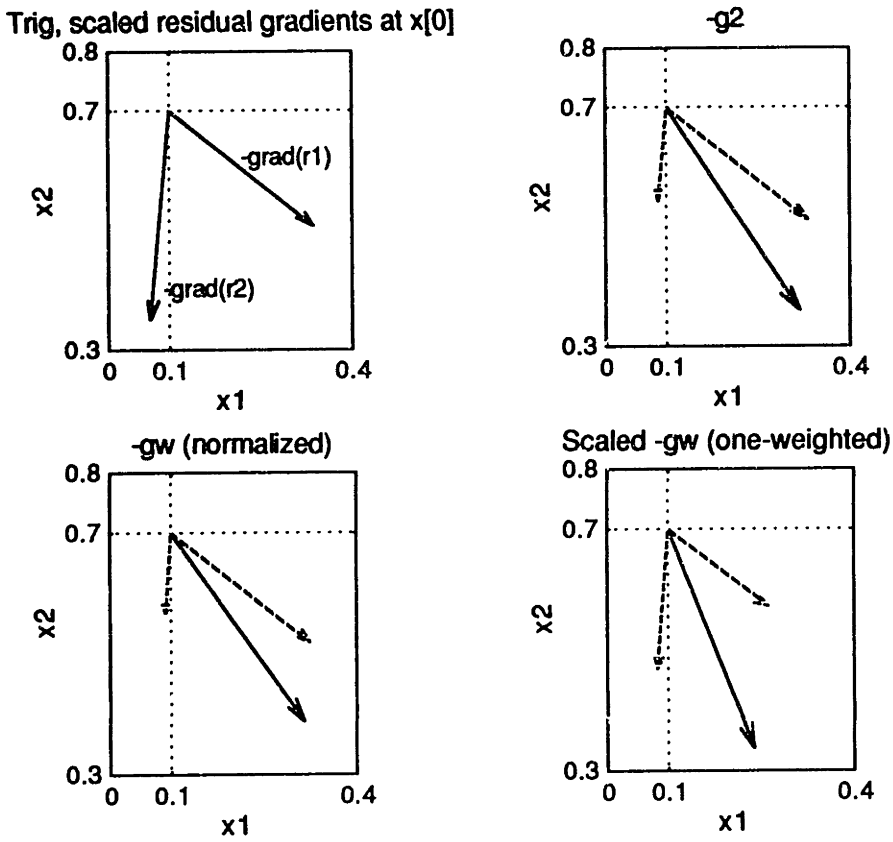

Figure 6.3.3. The negative gradient directions of the residual functions and of some sample norms. (a) Negative gradient directions of r_1 and r_2 at $x_{[0]}$. (b) The graphical sum of Equation 6.3.1 forms the gradient of $\emptyset_{(2)}$ at $x_{[0]}$. (c, d) The graphical sums of Equations 6.3.2 and 6.3.3.

Figure 6.3.3 shows this process graphically. The first subplot, Figure 6.3.3a, shows $-\nabla r_{1[0]}$ and $-\nabla r_{2[0]}$, scaled to fit the frame of the drawing. (All the figures depict the negative gradients since the gradient vectors point towards more positive values of r_1 and r_2, and since a steepest descent search follows the negative gradient direction towards less positive function values.) The second subplot shows $-2 \cdot r_{1[0]} \cdot \nabla r_{1[0]}$ and $-2 \cdot r_{2[0]} \cdot \nabla r_{2[0]}$, and takes their vector sum to find $-g_{(2)[0]}$, the steepest descent direction for $\emptyset_{(2)}$ from $x_{[0]}$. See Equation 6.3.1.

The last two subplots of Figure 6.3.3 anticipate Chapter 8, which develops a weighted double dogleg algorithm, by suggesting two weighting selections besides the $p_i = 2 \cdot r_i$ of $\emptyset_{(2)}$. Figure 6.3.3c uses $p_i = 2 \cdot r_i / \|\nabla r_i\|$. Effectively this normalizes the residual gradients before weighting and summing in Equation 4.3.6:

$$g = 2 \cdot r_1 \cdot \frac{\nabla r_1}{\|\nabla r_1\|} + 2 \cdot r_2 \cdot \frac{\nabla r_2}{\|\nabla r_2\|} \tag{6.3.2}$$

Intuitively this removes scale dependencies introduced by the slopes of the tangent hyperplanes: a steep tangent plane has a large gradient magnitude, which reduces the effective magnitude of the corresponding residual, while a shallow tangent increases the step length needed to zero a residual, increasing its effective magnitude. Chapter 8 develops this idea in a more complete, though still heuristic, argument. Where $\emptyset_{(2)}$ has $w_i = 1$, Equation 6.3.2 corresponds to $w_i = 1/\|\nabla r_i\|$.

Figure 6.3.3d uses $w_i = 1/|r_i|$, as suggested by Equation 4.4.9b. This choice creates a quadratic whose steepest descent direction matches that of the one-norm. From Equation 4.5.4b, the corresponding $p_i = 2 \cdot r_i/|r_i| = 2 \cdot \text{sign}\{r_i\}$ and

$$g = 2 \cdot \text{sign}\{r_1\} \cdot \nabla r_1 + 2 \cdot \text{sign}\{r_2\} \cdot \nabla r_2 \tag{6.3.3}$$

as expected for the one-norm (the factor 2 changes the length, but not the direction, of the gradient). Note that the fourth subplot scales the three vectors to fit the frame of the drawing.

The two-dimensional trigonometric function, with its bounded residual and residual gradient magnitudes, does not show much difference between the three net steepest descent directions calculated in Figure 6.3.3. For a poorly-scaled problem, choosing the w_i by the rules outlined above has greater impact on the level curves of $\emptyset_{(w)}$, and hence on the double dogleg curve.

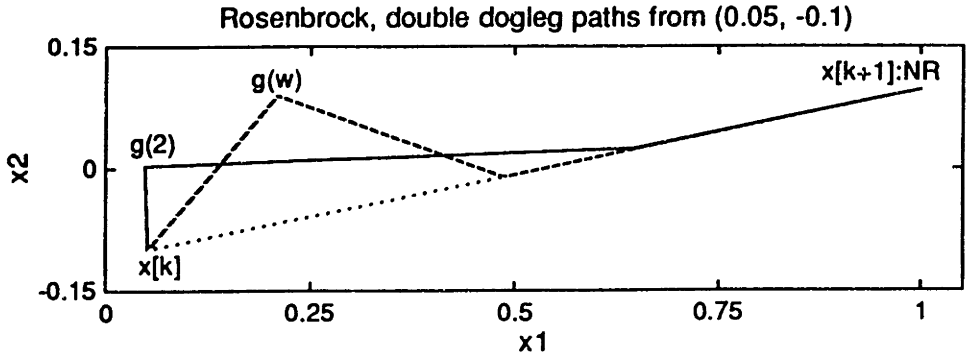

Figure 6.3.4. Double dogleg search paths for the two-dimensional Rosenbrock function. The solid path, for $\emptyset_{(2)}$, defines the standard double dogleg search. The dashed path adapts the double dogleg to $\emptyset_{(w)}$, with the weights chosen to normalize the residual gradients, as in Equation 6.3.2.

Figure 6.3.4 demonstrates the effect of changing the w_i using the two-dimensional Rosenbrock, a poorly-scaled problem, from an initial point $(0.05, -0.1)^T$. The figure shows double dogleg curves for both $\emptyset_{(2)}$ ($w_i = 1$) and the normalized-gradient choice of Equation 6.3.2 ($w_i = 1/\|\nabla r_i\|$). Both search paths terminate at the same point, since the choice of norm does not affect the Newton-Raphson solution, but the initial leg of the paths differ since the first follows the $g_{(2)}$ direction to the Cauchy point of $\emptyset_{(2)}$, while the second follows $g_{(w)}$ for a weighted r-square norm $\emptyset_{(w)}$ with the w_i chosen to normalize the gradients.

6.4 Double Dogleg Solution

Starting from $x_{[0]} = (0.1, 0.7)^T$ on the two-dimensional trigonometric function, the double dogleg algorithm of Section 5.3 (i.e., the double dogleg with r-square as the residual norm) terminates in an uninterrupted sequence of Newton-Raphson steps. Figure 6.4.1 changes the initial point, moving it farther from x** to provide a more illustrative example.

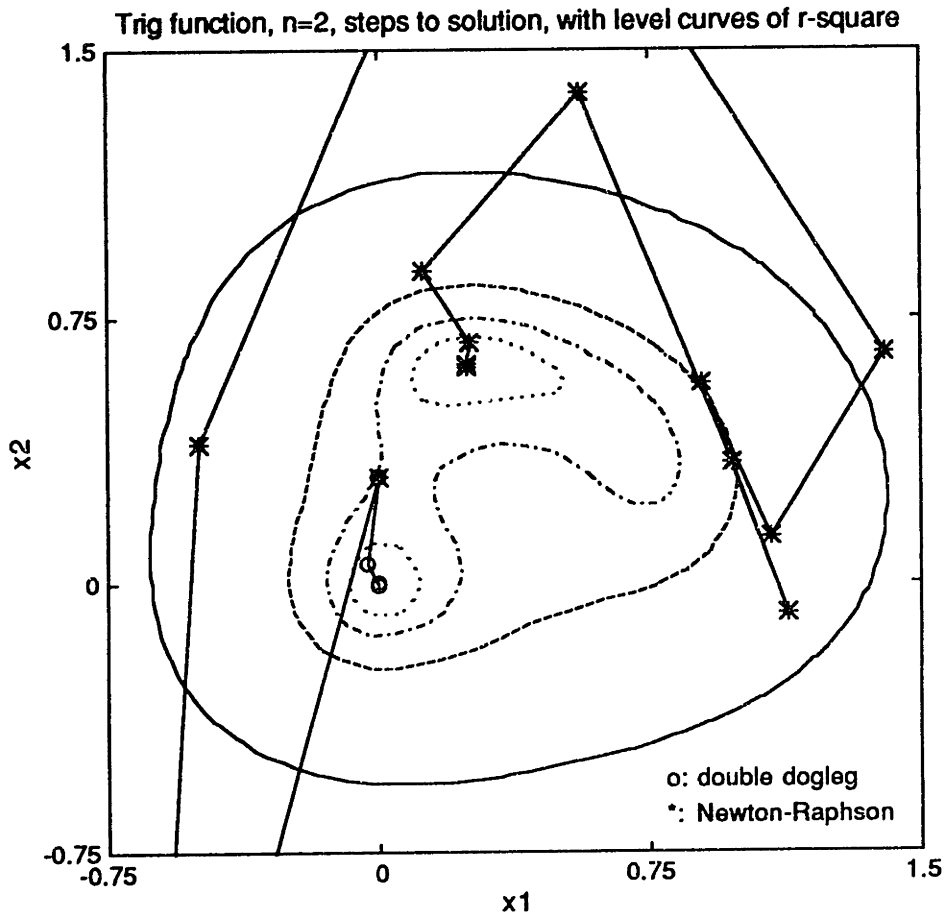

Figure 6.4.1. Newton-Raphson and double dogleg steps for the two-dimensional trigonometric function, showing level curves at $\emptyset_{(2)} = 1$ (solid), 0.1 (dashes), 0.03 (dash-dots), and 0.01 (dots). Note the problem has two solutions on the ranges of x_1 and x_2 depicted.

Figure 6.4.1 compares the Newton-Raphson and double dogleg solutions from $(0, 0.3)^T$, superposing their trajectories over level curves of r-square. From this starting point, Newton-Raphson's method behaves wildly until it falls within the radius of convergence of the root $x^{**} = (0.2431, 0.6127)^T$. The method takes 14 Jacobian evaluations to solve the problem to the default tolerance of the test machine.

The double dogleg algorithm, on the other hand, converges after only 4 Jacobian evaluations. In its first iteration, it pulls back from the over-long Newton-Raphson step, in order to meet the descent requirement. In addition, by mixing the gradient direction into its step choice, the double dogleg bends its search out of the Newton-Raphson line, towards the solution $x^{**} = (0, 0)^T$. The algorithm takes the full Newton-Raphson step in each of its remaining iterations.

Figure 6.4.2 lists the output from the program DBLDOGLG.CPP on this test problem.

```
Reporting from SolveDoubleDogleg():
Zero tol:     6.062734e-06  Step tol:     3.66245e-11
Initial x:
0    0.3

x: iteration:     1
0    0.3
r: SS:    0.028087
0.044664 -0.16153
step lengths: trust:   2.436767 NR:  2.436767
step:     2.436767 cut (poor cost)
step lengths: Cauchy:  0.048862 Cutback: 0.659805
step:     0.243677

x: iteration:     2
-0.034864     0.05883
r: SS:    0.004238
0.037803 -0.052999
step lengths: trust:   0.487353 NR:  0.073082
step:     0.073082

x: iteration:     3
-0.003126     -0.007
r: SS:    6.009352e-05
0.00316  0.007079
step lengths: trust:   0.146165 NR:  0.007583
step:     0.007583

x: iteration:     4
-3.351656e-05     -7.656712e-05
r: SS:    6.987588e-09
3.352061e-05  7.657647e-05
step lengths: trust:   0.015166 NR:  8.35714e-05
step:     8.35714e-05

SolveDoubleDogleg() terminating: Zero residual vector
Final x:
-4.053621e-09     -9.353031e-09
r: SS:    1.03911e-16
4.053621e-09  9.353031e-09
```

Figure 6.4.2. Output from code file DBLDOGLG.CPP, run on the two-dimensional trigonometric problem from $(0, 0.3)^T$.

CHAPTER 7
THE PLANAR HOOK ALGORITHM

This chapter begins a series of observations on, or criticisms of, the standard methods of solving nonlinear algebraic systems. The observations introduce new algorithms intended to address the issues raised.

Section 7.1 remarks on the double dogleg algorithm particularly, pointing out its ineffective use of the underlying mathematics.

Section 7.2 responds to these observations by developing the planar hook algorithm, designed to exploit the double dogleg expressions more fully.

Section 7.3 presents numeric results, for the same test problems used in Chapter 5 to test Newton-Raphson's method and the two double dogleg variations.

7.1 Observations on Double Dogleg Algorithm

Computational inefficiency

The double dogleg algorithm selects each step Δx as a linear combination of the Cauchy and Newton-Raphson steps. As demonstrated in Section 4.2, this choice leads to scalar expressions for $g_{(2)}{}^T\Delta x$ and $\Delta f_{(2)}$ (which depends on $\Delta x^T G_{(2)}\Delta x$). These scalar expressions allow the algorithm to avoid calculating the vector-vector and matrix-vector products for each trial Δx at an iteration.

The published routine [Dennis §A.IV.3] does not use these scalar equivalents, because its intended use with the hook algorithm as well as with the double dogleg algorithm does not permit it to assume Δx lies in the plane of the steepest descent and Newton-Raphson directions.

Specialized for the double dogleg algorithm, the implementation described in Chapter 5 begins to exploit these expressions, as a substitutional convenience within the standard algorithm. Section 5.4 shows that this substitution saves on the order of 5% of the execution time for small problem dimensions. However, the implementation of Chapter 5 fails to exploit the scalar expressions fully, because it retains the piecewise linear search path of the double dogleg curve.

Double dogleg curve not optimal

Section 3.6 presents the double dogleg curve as an approximation to an optimal search path. The optimal, hook, path picks $\Delta x_{[k]}$ to minimize the model $f_{(2)[k]}$ subject to a step length constraint $\|\Delta x_{[k]}\| \leq \delta_{[k]}$. A trust region method monitors the results of each trial step in order to maintain the desired step length $\delta_{[k]}$ at the expected limit of adequacy of the second-order model.

In the hook algorithm, each choice of $\delta_{[k]}$ requires iterations on Equation 3.6.1a,

$$\Delta x_{[k]:opt} = -(G_{[k]} + \nu_{[k]}I)^{-1} g_{[k]}$$

to find the $\nu_{[k]}$ which gives $\|\Delta x_{[k]:opt}\| = \delta_{[k]}$. Moreover the trust region updating rules may attempt several values of $\delta_{[k]}$ before accepting a step. Thus the hook algorithm's computational overhead makes it prohibitively expensive unless the cost of evaluating Ø is high [Dennis §6.4.2].

The double dogleg method approximates the hook path by: (1) setting out in the steepest descent direction, as does the hook curve; (2) ending at the Newton point (or the

Newton-Raphson point in equation solving), as does the hook curve; and (3) choosing points for intermediate trust lengths by a prescribed linear combination of the steepest descent and Newton directions. Thus the double dogleg path always lies in the plane of $g_{[k]}$ and $\Delta x_{[k]:N}$ (or of $g_{(2)[k]}$ and $\Delta x_{[k]:NR}$ for equation solving). The hook curve may select trial points from the entire variable space.

Suppose a function minimization algorithm chooses $\Delta x_{[k]}$ to lie in the plane of the steepest descent and Newton steps. Then, subject to the trust length, it may pick $\Delta x_{[k]}$ to minimize the cost function model in that two-dimensional subspace. [Byrd] describes such a step choice, called an indefinite dogleg step, and shows by numeric testing that the decrease in the quadratic model resulting from this optimization in the two-dimensional subspace typically comes quite close to the optimal step in the entire n-dimensional problem [Byrd §4]. Furthermore, the resulting algorithm typically performs nearly as well as the optimal, hook, algorithm.

Similarly, suppose an equation-solving algorithm chooses $\Delta x_{[k]}$ to lie in the plane of $g_{(2)[k]}$ and $\Delta x_{[k]:NR}$. Then, subject to the trust length, it may pick $\Delta x_{[k]}$ to minimize the model of r-square, $f_{(2)[k]}$. Equation 4.2.17b, giving $\Delta f_{(2)}$ as a function of two scalar parameters ρ_1 and ρ_2, invites such a step selection algorithm.

Model of norm fails expectations

In equation solving, constructing the model of the norm from the linearized residuals $\hat{r}$ compromises the Hessian by neglecting the curvature in the residual functions. Therefore the only second-order effects contained in the model Hessian come from algebraic manipulation of the first-order effects in the residual functions. Equations 4.3.10 and 4.3.11 express this Hessian approximation for a general residual norm, but for $\emptyset_{(2)}$, Equations 4.2.3 and 4.2.4 give the result directly: the double dogleg algorithm replaces the exact Hessian

$$G_{(2)} = 2J^TJ + 2\sum_{m=1}^{n} r_m \nabla^2 r_m$$

with the approximation

$$G_{(2)} \approx 2J^TJ$$

The lower two plots of Figure 6.2.1 show the resulting model for the two-dimensional trigonometric function. The curvature in $\hat{r}_{1[0]}{}^2$, in the lower left plot, represents no meaningful second-order information about r_1, but results from squaring the first-order (slope) information in the residual model. Similarly, the model $f_{(2)[0]}$ in the lower right plot, and more generally any model $f_{(2)[k]}$ formed with the linearized residuals, neglects the curvature in the residuals, and therefore cannot predict the important second-derivative effects in the residual norm.

Consequently the model of the norm violates a key assumption of the trust region algorithms: that the quality of the model decreases with greater step lengths, without regard to the step direction. Rather, the quality of the Hessian information in a given search direction depends on the accident of the relative nonlinearities of the residual functions in that direction. Put another way, with a full quadratic model, short trust lengths throw the burden of step selection more towards the first derivative part of the model (i.e., the gradient direction), on the assumption the second-derivative part of the model loses its validity for longer steps. But when estimating r-square under the linearized residual approximation, the first- and second-order parts of the model are more or less equally

valid, since they merely re-arrange the same Jacobian information about the slopes of the residual functions.

Since the double dogleg method uses the Hessian to calculate the Cauchy point, when applied to equation solving its estimate of the minimizing step length in the steepest descent direction does not account for nonlinearities in the residual functions. Therefore the Cauchy point calculation in an equation-solving algorithm does not carry the same strength of certainty as it does in function minimization. Chapter 9 develops these points more fully, but in terms of the double dogleg path this means that an improved path for equation solving should not follow the steepest descent direction all the way to the Cauchy point before turning towards the Newton-Raphson direction (see for example the discussion of Figure 6.3.2). The planar hook curve accomplishes this, because by minimizing $f_{(2)[k]}$ continuously along the range of possible step lengths it avoids the extremes of the piecewise-linear double dogleg curve.

7.2 The Planar Hook Algorithm

The planar hook algorithm, described here, replaces the double dogleg curve by choosing, for a given trust length $\delta_{[k]} \leq \|\Delta x_{[k]:NR}\|$, that step $\Delta x_{[k]}$ which: (1) has length $\|\Delta x_{[k]}\| = \delta_{[k]}$; and (2) minimizes the model of r-square, $f_{(2)[k]}$, in the plane of the steepest descent direction and the Newton-Raphson step. Otherwise, the algorithm exactly reproduces the step evaluation and trust length updating rules described for the double dogleg algorithm in Section 5.3.

The rest of this section derives the conditions necessary for this step selection, specialized to equation solving, and with the r-square norm modeled as in Section 4.2.

Step selection

Following Equation 4.2.15, constrain each step to the plane formed by the Cauchy and Newton-Raphson steps,

$$\Delta x_{[k]} = \rho 1_{[k]} \cdot \Delta x_{(2)[k]:SD} + \rho 2_{[k]} \cdot \Delta x_{[k]:NR} \tag{7.2.1}$$

Recall the Cauchy point minimizes $f_{(2)}$ in the steepest descent direction. Forming $f_{(2)}$ under the linearized residual approximation, Equation 4.2.10 gives

$$\Delta x_{(2)[k]:SD} = \frac{-\|(J^T r)_{[k]}\|^2}{\|(JJ^T r)_{[k]}\|^2} (J^T r)_{[k]} \tag{7.2.2a}$$

while from Equation 2.2.9a

$$\Delta x_{[k]:NR} = -(J^{-1} r)_{[k]} \tag{7.2.2b}$$

For later convenience note that since $r^T J \cdot J^{-1} r = r^T r = \emptyset_{(2)}$,

$$\Delta x_{(2)[k]:SD}{}^T \Delta x_{[k]:NR} = \frac{\|(J^T r)_{[k]}\|^2}{\|(JJ^T r)_{[k]}\|^2} \emptyset_{(2)[k]} \leq \|\Delta x_{(2)[k]:SD}\| \cdot \|\Delta x_{[k]:NR}\| \tag{7.2.3}$$

where the inequality follows since $a^T b = \|a\| \cdot \|b\| \cdot \cos\theta$ [Strang88 §3.2], and since $0 \leq \cos\theta \leq 1$ since Newton-Raphson defines a descent step by Equation 4.2.2b.

Equation 4.2.17b gives the increase in the quadratic model, $\Delta f_{(2)[k]} = f_{(2)[k]}\{x\} - \emptyset_{(2)[k]}$, as

(7.2.4) $$\Delta f_{(2)[k]} = \rho_{1[k]}\cdot(\rho_{1[k]} + 2\rho_{2[k]} - 2)\frac{\|(J^T r)_{[k]}\|^4}{\|(JJ^T r)_{[k]}\|^2} + (\rho_{2[k]}^2 - 2\rho_{2[k]})\cdot\emptyset_{(2)[k]}$$

Since more negative values of $\Delta f_{(2)[k]}$ indicate greater decreases in the cost function, the planar hook algorithm chooses $\rho_{1[k]}$ and $\rho_{2[k]}$ to minimize Equation 7.2.4 subject to $\|\Delta x_{[k]}\| = \delta_{[k]}$ in Equation 7.2.1.

Step length constraint

Consider the constraint $\|\Delta x_{[k]}\| = \delta_{[k]}$. Setting $\|\Delta x_{[k]}\|^2 = \Delta x_{[k]}^T \Delta x_{[k]} = \delta_{[k]}^2$ in Equation 7.2.1 gives

(7.2.5) $$\rho_2^2\cdot\|\Delta x_{NR}\|^2 + 2\rho_1\rho_2\frac{\|J^T r\|^2}{\|JJ^T r\|^2}\emptyset_{(2)} + \rho_1^2\cdot\|\Delta x_{SD(2)}\|^2 - \delta^2 = 0$$

No ambiguity results from the convenience of dropping the iteration subscript [k], since all quantities apply to the same iteration.

Note that if $\rho_1 = 0$, Equation 7.2.5 gives $\rho_2 = \pm\delta/\|\Delta x_{NR}\|$. To make $\Delta f_{(2)} < 0$ requires ρ_2 positive, and

(7.2.6a,b) $$\rho_2|_{\rho_1=0} = \frac{\delta}{\|\Delta x_{NR}\|} \quad \text{where} \quad \Delta f_{(2)}|_{\rho_1=0} = \left(\frac{\delta^2}{\|\Delta x_{NR}\|^2} - \frac{2\delta}{\|\Delta x_{NR}\|}\right)\emptyset_{(2)}$$

Note Equation 7.2.6b gives $\Delta f_{(2)} < 0$ as long as $\delta \le \|\Delta x_{NR}\|$.

While Equation 7.2.6a places a point at the proper step length, that point does not necessarily minimize $\Delta f_{(2)}$ for that step length. Similarly the point where $\rho_1 = \delta/\|\Delta x_{SD(2)}\|$ and $\rho_2 = 0$ also satisfies the step length constraint (and gives a negative value of $\Delta f_{(2)}$ so long as $\delta < 2\cdot\|\Delta x_{SD(2)}\|$), but does not necessarily give the best selection of ρ_1 and ρ_2 for a given step length $\delta_{[k]}$.

The Chapter 3 discussion of descent directions implies $\rho_1 > 0$ when $\rho_2 = 0$. That is, if Δx lies along the steepest descent direction it must lie in the same direction as the Cauchy point to minimize $\Delta f_{(2)}$. Note however that if $\rho_2 > 0$, it is possible to set $\rho_1 < 0$ and still find $\Delta f_{(2)}$ negative, since the Newton-Raphson choice also is a descent direction. Intuitively, the optimal planar search path should lie between the steepest descent and Newton-Raphson directions (see Figure 3.6.1), making both weights positive, but Equation 7.2.5 does not require this.

Solving Equation 7.2.5 by the quadratic formula,

(7.2.7a) $$\rho_2 = -c_1\rho_1 \pm \sqrt{\rho_1^2(c_1^2 - c_2) + c_3}$$

where

(7.2.7b) $$0 < c_1 = \frac{\|J^T r\|^2\cdot\emptyset_{(2)}}{\|JJ^T r\|^2\cdot\|\Delta x_{NR}\|^2} \le 1$$

(7.2.7c) $$0 < c_2 = \frac{\|\Delta x_{SD(2)}\|^2}{\|\Delta x_{NR}\|^2} < 1$$

and

(7.2.7d) $$0 < c_3 = \frac{\delta^2}{\|\Delta x_{NR}\|^2} < 1$$

The inequality $c_3 < 1$ ignores the case $\delta = \|\Delta x_{NR}\|$, with its trivial solution $\rho_1 = 0$ and $\rho_2 = 1$. The inequality $c_1 \le 1$ follows from Equation 7.2.3 since

$$\frac{\|J^T r\|^2 \cdot \emptyset_{(2)}}{\|JJ^T r\|^2 \cdot \|\Delta x_{NR}\|^2} \le \frac{\|J^T r\|^2 \cdot \emptyset_{(2)}}{\|JJ^T r\|^2 \cdot \|\Delta x_{SD(2)}\| \cdot \|\Delta x_{NR}\|} \le 1$$

where $\|\Delta x_{SD(2)}\| \le \|\Delta x_{NR}\|$ by Equations 3.5.14b and 4.2.7, with equality only for colinear steepest descent and Newton-Raphson directions. Of course for colinear directions both the double dogleg and planar hook search paths reduce to lines.

A different arrangement of Equation 7.2.3 gives

$$\frac{\|J^T r\|^2 \cdot \emptyset_{(2)}}{\|JJ^T r\|^2 \cdot \|\Delta x_{NR}\|^2} \le \frac{\|\Delta x_{SD(2)}\|}{\|\Delta x_{NR}\|}$$

or

(7.2.7e) $$c_1^2 - c_2 \le 0$$

with equality only for colinear steepest descent and Newton-Raphson directions.

Finally, from Equations 4.2.9c and 4.2.11c the greatest decrease in the steepest descent direction is $-\|J^T r\|^4/\|JJ^T r\|^2$, at the Cauchy point. But the best decrease can be no greater than $\emptyset_{(2)}$ itself, so $\|J^T r\|^4/\|JJ^T r\|^2 \le \emptyset_{(2)}$, or, multiplying both sides by $\|J^T r\|^2/\|JJ^T r\|^2$,

(7.2.7f) $$c_2 \le c_1$$

Returning to the quadratic solution, with $c_1^2 - c_2 < 0$ the determinant may be negative for large values of ρ_1^2. To avoid imaginary numbers requires

(7.2.8) $$\rho_1^2 \le \frac{c_3}{c_2 - c_1^2}$$

i.e., a limit on the magnitude of ρ_1. When ρ_1 satisfies Equation 7.2.8, the determinant adds a positive number smaller than one, c_3, to a negative number, $\rho_1^2(c_1^2 - c_2)$, to find a number greater than zero. The result must be smaller than one. Thus

(7.2.9) $$\sqrt{\rho_1^2(c_1^2 - c_2) + c_3} < 1$$

Signs of weights

As noted above, Equation 7.2.5 forces neither $\rho_1 > 0$ nor $\rho_2 > 0$. To demonstrate $\rho_1 > 0$, substitute $\rho_2 = -c_1\rho_1 \pm \sqrt{D}$, with $D = \rho_1^2(c_1^2 - c_2) + c_3$, into Equation 7.2.4. Again dropping the iteration subscript [k], and after some algebraic manipulation,

(7.2.10)
$$\Delta f_{(2)} = \rho_1^2\left(c_1^2\emptyset_{(2)} + (1 - 2c_1)\frac{\|J^T r\|^4}{\|JJ^T r\|^2}\right) + D \cdot \emptyset_{(2)}$$
$$- 2\rho_1\frac{\|J^T r\|^4}{\|JJ^T r\|^2}(1 - \pm\sqrt{D})\left(1 - \frac{\emptyset_{(2)}^2}{\|\Delta x_{NR}\|^2 \cdot \|J^T r\|^2}\right) - \pm 2\sqrt{D} \cdot \emptyset_{(2)}$$

where the sense of the sign of $\sqrt{D}$ corresponds to the sense chosen in Equation 7.2.7a when calculating ρ_2.

The first two terms of Equation 7.2.10 are fixed for a given magnitude of ρ_1, since $D = D\{\rho_1^2\}$. Similarly the fourth term depends on the choice of $\pm\sqrt{D}$ when finding ρ_2, but

not on the sign of ρ_1. Therefore for a given magnitude of ρ_1, its sign affects only the third term of Equation 7.2.10.

Substituting $\|\Delta x_{SD(2)}\| = \|J^T r\|^3/\|JJ^T r\|^2$ from Equation 7.2.2a into Equation 7.2.3, $\emptyset_{(2)} \le \|J^T r\| \cdot \|\Delta x_{NR}\|$, making the last factor of the third term positive. Since Equation 7.2.9 guarantees $\sqrt{D} < 1$, the factor $(1 - \pm\sqrt{D})$ also is positive, regardless of the sign of $\sqrt{D}$. Therefore the third term contributes negatively to $\Delta f_{(2)}$ for $\rho_1 > 0$, and Equation 7.2.10 always gives a more negative number for a positive ρ_1 than for a negative ρ_1 of the same magnitude. Thus to minimize $\Delta f_{(2)}$ subject to Equation 7.2.5 requires

$$\rho_1 \ge 0 \tag{7.2.11}$$

Turning to ρ_2, recall Equation 7.2.6 finds a point where $\rho_1 = 0$ and $\rho_2 = \delta/\|\Delta x_{NR}\|$ (i.e., a point a distance δ along the Newton-Raphson direction) and with $\Delta f_{(2)} < 0$. The point satisfies Equation 7.2.5 but may not minimize $\Delta f_{(2)}$ in the plane at that step length (in fact the point minimizes $\Delta f_{(2)}$ only when $\delta = \|\Delta x_{NR}\|$).

Starting from this point, ρ_1 can only increase, according to Equation 7.2.11. Differentiating Equation 7.2.5 with respect to ρ_1,

$$2\rho_2\frac{d\rho_2}{d\rho_1}\|\Delta x_{NR}\|^2 + 2c_1\rho_2\cdot\|\Delta x_{NR}\|^2 + 2c_1\rho_1\frac{d\rho_2}{d\rho_1}\|\Delta x_{NR}\|^2 + 2\rho_1\cdot\|\Delta x_{SD(2)}\|^2 = 0$$

$$\frac{d\rho_2}{d\rho_1} = \frac{-(c_2\rho_1 + c_1\rho_2)}{c_1\rho_1 + \rho_2} \tag{7.2.12}$$

Equation 7.2.12 governs changes along the arc $\|\Delta x\| = \delta$. Note that as long as $\rho_2 > 0$, $d\rho_2/d\rho_1 < 0$. Therefore starting from the point defined by Equation 7.2.6a, as ρ_1 increases from 0, ρ_2 decreases from $\delta/\|\Delta x_{NR}\|$.

From Equation 7.2.4, the rate of change of $\Delta f_{(2)}$

$$\frac{d(\Delta f_{(2)})}{d\rho_1} = 2(\rho_1 + \rho_2 + \rho_1\frac{d\rho_2}{d\rho_1} - 1)\frac{\|J^T r\|^4}{\|JJ^T r\|^2} + 2(\rho_2 - 1)\frac{d\rho_2}{d\rho_1}\emptyset_{(2)} \tag{7.2.13}$$

At the point where $\rho_1 = 0$,

$$\left.\frac{d(\Delta f_{(2)})}{d\rho_1}\right|_{\rho_1=0} = 2(\rho_2 - 1)\left(\frac{\|J^T r\|^4}{\|JJ^T r\|^2} - c_1\emptyset_{(2)}\right)$$

for the proper step length since $d\rho_2/d\rho_1 = -c_1$ along $\|\Delta x\| = \delta$. Straightforward application of Equations 7.2.2a, 7.2.3, and 7.2.7b shows the result is negative while $\rho_2 < 1$. Therefore unless $\delta = \|\Delta x_{NR}\|$, increasing ρ_1 from zero always decreases $\Delta f_{(2)}$.

As ρ_1 increases from zero, ρ_2 decreases and $\Delta f_{(2)}$ decreases. As ρ_1 continues to increase, $\Delta f_{(2)}$ passes through its minimum, then increases again, as the path sweeps out points at the proper step length. Furthermore ρ_2 decreases with ρ_1 as long as $\rho_2 > 0$.

Now suppose ρ_1 has been increased, and ρ_2 decreased, until $\rho_2 = 0$. Then Equation 7.2.13 gives

$$\left.\frac{d(\Delta f_{(2)})}{d\rho_1}\right|_{\rho_2=0} = 2\rho_1\cdot\left(1 - \frac{c_2}{c_1}\right)\frac{\|J^T r\|^4}{\|JJ^T r\|^2}$$

since $d\rho_2/d\rho_1 = -c_2/c_1 = -\|J^T r\|^4/\|JJ^T r\|^2/\emptyset_{(2)}$ following the proper step length. From Equation 7.2.7f, $c_2/c_1 \le 1$. Therefore if $\rho_2 = 0$, $d(\Delta f_{(2)})/d\rho_1 \ge 0$ and the minimum lies at a less positive value of ρ_1. As shown above, decreasing ρ_1 increases ρ_2 from zero. Therefore

(7.2.14) $$\rho_2 > 0$$

at the minimum of $\Delta f_{(2)}$ on $\|\Delta x\| = \delta$ in the plane of the Cauchy and Newton-Raphson steps.

Returning to Equation 7.2.7a, to make ρ_2 positive with both ρ_1 and c_1 positive requires using the positive square root:

(7.2.15) $$\rho_2 = -c_1\rho_1 + \sqrt{\rho_1^2(c_1^2 - c_2) + c_3}$$

Moreover, $\rho_2 > 0$ requires $\rho_1^2 < c_3/c_2$. This restricts ρ_1^2 more closely than does Equation 7.2.8, with its smaller denominator. Substituting for c_3 and c_2,

(7.2.16) $$0 \le \rho_1 < \frac{\delta}{\|\Delta x_{SD(2)}\|}$$

to minimize $\Delta f_{(2)}$ on $\|\Delta x\| = \delta$ in the plane of points given by Equation 7.2.1. The corresponding value of ρ_2 satisfies $0 < \rho_2 \le \delta/\|\Delta x_{NR}\|$.

Examples: planar hook versus double dogleg curves

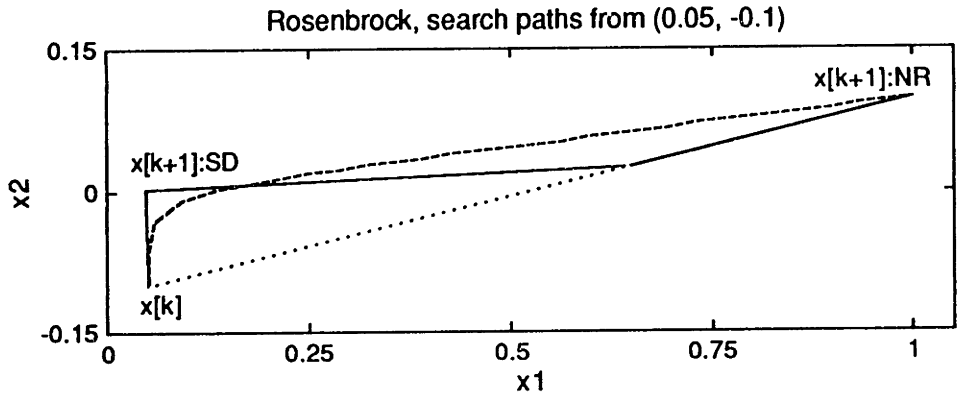

Figure 7.2.1. The Newton-Raphson (dots), double dogleg (solid), and planar hook (dashes) search paths, calculated for the Rosenbrock function from a starting point $(0.05, -0.1)^T$.

Figure 7.2.1 compares the planar hook and double dogleg search paths for the two-dimensional Rosenbrock function, starting from $x_{[0]} = (0.05, -0.1)^T$. Similarly, Figure 7.2.2 shows double dogleg and planar hook search paths for the two-dimensional trigonometric function from $(0.1, 0.7)^T$, the same starting point for which Figures 6.3.1 and 6.3.2 depict a double dogleg search. As proved above, the planar hook path starts out in the steepest descent direction, bends towards the Newton-Raphson line before reaching the Cauchy point, and lies between the steepest descent and Newton-Raphson directions until it reaches the Newton-Raphson point, at the maximum trust length.

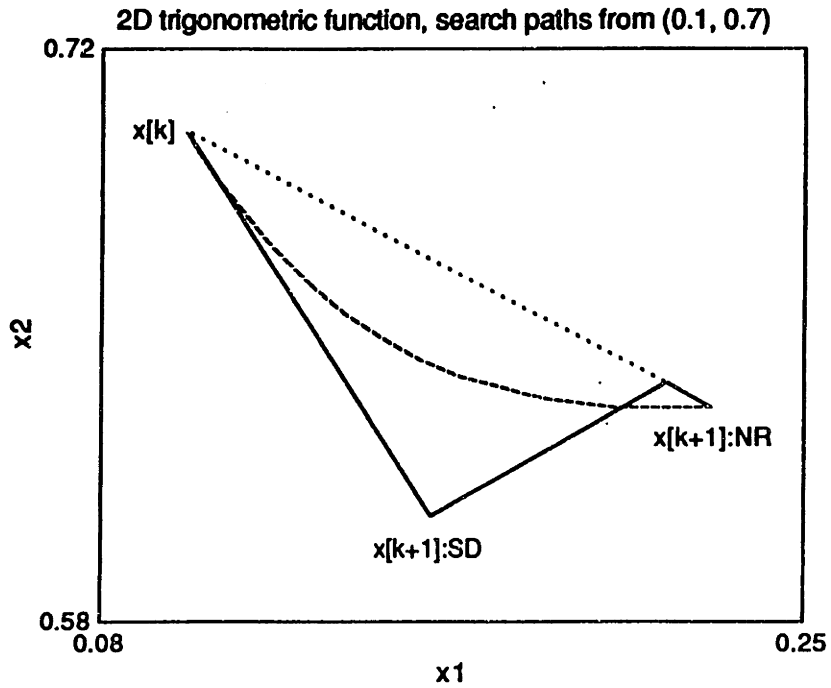

Figure 7.2.2. The planar hook (dashes) versus the double dogleg (solid) search paths for the two-dimensional trigonometric function from $(0.1, 0.7)^T$.

Iterative search for weights

Each point on the planar hook curve minimizes, for some trust length, the model $f_{(2)[k]}$ in the plane of $g_{(2)[k]}$ and $\Delta x_{[k]:NR}$. To choose a point on the curve, an algorithm must pick the $\rho_{1[k]}$ and $\rho_{2[k]}$ which satisfy Equations 7.2.15 and 7.2.16, and which minimize Equation 7.2.4. The following subsections treat this search.

Within each major iteration, i.e. for each iterate $x_{[k]}$ and trust length $\delta_{[k]}$, the planar hook algorithm must choose $\rho_{1[k]}$ and $\rho_{2[k]}$ to minimize $\Delta f_{(2)[k]}$ over the points satisfying $\|\Delta x_{[k]}\| = \delta_{[k]}$ in the plane of the Cauchy and Newton-Raphson steps. Equation 7.2.10 shows that $\Delta f_{(2)}$ is not parabolic in ρ_1-- recall that $D = D\{\rho_1^2\}$--, but that a series of parabolic fits should find the optimal $\rho_{1[k]}$ quickly.

Imagine fitting the parabola $\Delta f_{(2)} = a \cdot \rho_1^2 + b \cdot \rho_1 + c$ to three observations. For each estimate ρ_1, Equation 7.2.15 gives ρ_2, Equation 7.2.4 gives $\Delta f_{(2)}$, and Equations 7.2.12 and 7.2.13 taken together give $d(\Delta f_{(2)})/d\rho_1$. To identify parameters a, b, and c, fit the parabola to two $(\rho_1, \Delta f_{(2)})$ pairs, and a derivative for one of these pairs. Then the extremum of the parabola occurs at $\rho_1 = -b/2a$. (Alternately, fit the parabola to two slopes only, since parameter c does not affect the estimated optimizer.)

Knowing $0 \le \rho_1 < \delta / \|\Delta x_{SD(2)}\|$ establishes left and right bounds on the optimal value of $\rho_{1[k]}$. Evaluating the slope at each new estimate of the optimal $\rho_{1[k]}$ unambiguously indicates the direction towards the minimum point. Therefore as successive parabolic fits select new estimates of $\rho_{1[k]}$, replace the left bound if the new $d(\Delta f_{(2)})/d\rho_1 < 0$, and replace the right bound if the new slope is positive (of course if the new estimate sets the slope to zero, it exactly minimizes $\Delta f_{(2)}$).

Following Equation 5.2.2, terminate the search for $r_{1[k]}$ when the newest estimate has

(7.2.17) $$\left| \frac{d(\Delta f_{(2)})}{d\rho_1} \cdot \frac{\rho_1}{\Delta f_{(2)}} \right| < \text{SLOPE\_TOL}$$

where a typical SLOPE_TOL = (MACH_EPS)$^{1/3}$. Note that Equation 7.2.17 may indicate termination with a large positive $\Delta f_{(2)}$ resulting from a poor initial guess at ρ_1. Therefore it should be coupled with a test for the sign of $\Delta f_{(2)}$, or else tested, without the absolute value, against a positive SLOPE_TOL for the left bound and a negative SLOPE_TOL for the right bound.

Alternately, terminate when the left and right bounds on ρ_1 are sufficiently close together, e.g. when

(7.2.18) $$\rho_{1,\text{right}} - \rho_{1,\text{left}} < \rho_{1,\text{right}} \cdot \text{RHO\_TOL}$$

For example, RHO_TOL = 10^{-5} demands five significant digits in the solution (see Equation 5.2.7). When terminating according to Equation 7.2.18, choose as $\rho_{1[k]}$ the bound with the smallest associated $|d(\Delta f_{(2)})/d\rho_1|$.

The search schemes discussed below fit the parabola to function values and slopes for different trial values of ρ_1. For convenience, label the observations (x_1, y_1), (x_2, y_2), (x_1, y_1'), and (x_2, y_2'). In this notation, x_1 and x_2 represent two guesses at ρ_1 (not ρ_1 and ρ_2), while y represents $\Delta f_{(2)}$.

Search for weights rule 1

The first scheme fits the parabola to two function values and a slope. Solving the system

$$y_1 = a \cdot x_1^2 + b \cdot x_1 + c$$
$$y_2 = a \cdot x_2^2 + b \cdot x_2 + c$$
$$y_1' = 2a \cdot x_1 + b$$

for a, b, and c, and setting $x_{opt} = -b/2a$, gives

(7.2.19) $$x_{opt} = x_1 - \frac{y_1'}{2} \frac{(x_2 - x_1)^2}{(y_2 - y_1) - y_1'(x_2 - x_1)}$$

To insure that the extremum of the parabola is a minimum falling between x_1 and x_2, use the slope observation from the point with the most negative function value. Thus, if the left bound on $\rho_{1[k]}$ gives the more negative $\Delta f_{(2)}$, associate x_1 with the left side; otherwise associate x_1 with the right side observations.

Suppose each iteration of Equation 7.2.19 fits the parabola to the left and right bounds on ρ_1, identifying x_1 by the most negative function value as described above. Call this Rule 1a. Figure 7.2.3 lists the left and right bounds on ρ_1, along with the corresponding $\Delta f_{(2)}$ and $d(\Delta f_{(2)})/d\rho_1$, for several iterations on the duct flow problem, with starting point $x_{[0]} = (0.02, 7, 1)^T$ and with $\delta_{[0]} = \|\Delta x_{SD(2)}\|/2$. The table shows typical behavior for Rule 1a, in which the estimate approaches the optimal point from one side, leaving the other bound unchanged.

iteration	$\rho_{1,left}$	$\rho_{1,right}$	$\Delta f_{(2),left}$	$\Delta f_{(2),right}$	$d(\Delta f_{(2)})/d\rho_{1,l}$	$d(\Delta f_{(2)})/d\rho_{1,r}$
0	0	0.5	-0.1301	-2.4982	-6.6423	3.0922
1	0.4012	0.5	-2.2083	-2.4982	-3.6348	3.0922
2	0.4747	0.5	-2.4468	-2.4982	-2.7343	3.0922
3	0.4924	0.5	-2.4901	-2.4982	-1.9872	3.0922
4	0.4972	0.5	-2.4980	-2.4982	-1.1428	3.0922
5	0.4986	0.5	-2.4992	-2.4982	-0.3663	3.0922
6	0.4988	0.5	-2.4992	-2.4982	-0.1478	3.0922

Figure 7.2.3. Iterations of Rule 1a applied to the duct flow problem at $(0.02, 7, 1)^T$. The trust length $\delta = ||\Delta x_{SD(2)}||/2$. The initial bounds on ρ_1 come from Equation 7.2.16.

To overcome this problem, Rule 1b calculates x_{opt} as in Rule 1a-- it applies Equation 7.2.19 to the current left and right bounds, associating x_1 with the observation with the most negative function value-- but notes when successive iterations consistently replace one bound. If the last three iterations replaced the same bound, Rule 1b applies Aitken extrapolation [Dahlquist §3.2.3] to find

$$x_{atk} = x_{opt} - \frac{(x_{opt} - x_{bnd})^2}{x_{opt} - 2x_{bnd} + x_{lst}} \tag{7.2.20}$$

where: (1) x_{opt} is the estimate calculated using Equation 7.2.19; (2) x_{bnd} is the current left or right bound, on the side whose bounds are consistently replaced; and (3) x_{lst} is the former left or right bound, replaced at the last iteration.

Note that the rule could use Equation 7.2.20 after two iterations replace the same bound, since the extrapolation requires only the current bound and that which the current bound replaced. However, experience indicates that waiting until the third iteration provides better performance.

When Rule 1b tries Aitken extrapolation, it uses x_{atk} rather than x_{opt} as the next trial iterate, provided x_{atk} lies between x_{opt} and the bound which remained unreplaced during the previous three iterations. Otherwise, it uses x_{opt}.

Rule 1c, on the other hand, does not necessarily apply Equation 7.2.19 to the current bounds on $\rho_{1[k]}$. As in Rule 1a, it identifies as x_1 the current bound with the most negative function value. However for x_2 it uses either the opposite bound, or the most recently replaced bound, whichever has the most negative function value. If the resulting x_{opt} lies outside the current bounds, the rule bisects the current interval to find the next iterate.

Figure 7.2.4 compares the three variations on Equation 7.2.19, for the duct flow problem from starting point $x_{[0]} = (0.02, 7, 1)^T$, with a trust length $\delta_{[0]} = ||\Delta x_{SD(2)}||/2$. Blank column entries indicate the rule satisfies termination criterion Equation 7.2.17 with SLOPE_TOL = 10^{-3}.

iteration	Rule 1a		Rule 1b		Rule 1c	
	$\rho_{1,left}$	$\rho_{1,right}$	$\rho_{1,left}$	$\rho_{1,right}$	$\rho_{1,left}$	$\rho_{1,right}$
0	0	0.5	0	0.5	0	0.5
1	0.4012	0.5	0.4012	0.5	0.4012	0.5
2	0.4747	0.5	0.4747	0.5	0.4747	0.5
3	0.4924	0.5	0.4924	0.5	0.4924	0.5
4	0.4972	0.5	0.4989	0.5	0.4972	0.5
5	0.4986	0.5	0.4990	0.5	0.4986	0.5
6	0.4988	0.5	0.4990	0.5	0.4988	0.5
7	0.4989	0.5			0.4988	0.4990
8	0.4989	0.5			0.4990	0.4990
9	0.4990	0.5				
10	0.4990	0.5				

Figure 7.2.4. Iterations of Rules 1 applied to the duct flow problem at $(0.02, 7, 1)^T$. The trust length $\delta = ||\Delta x_{SD(2)}||/2$. The initial bounds on ρ_1 come from Equation 7.2.16.

Search for weights rule 2

The second fitting scheme ignores the function values, and finds parameters a and b using the slopes only. Solving the system

$$y_1' = 2a \cdot x_1 + b$$
$$y_2' = 2a \cdot x_2 + b$$

for a and b, and setting $x_{opt} = -b/2a$, gives

(7.2.21)
$$x_{opt} = x_1 - \frac{y_1'(x_2 - x_1)}{(y_2' - y_1')}$$

Again, x_1 and x_2 denote two guesses for ρ_1, rather than ρ_1 and ρ_2. Since both the slopes imply movement into the region between x_1 and x_2, associating x_1 always with the left side observation, and x_2 with the right, presents no difficulty.

Rule 2a simply uses the left and right observations as x1 and x2 at each iteration.

Rule 2b applies Aitken extrapolation on x_{opt} when the last three iterations replaced the same bound. As above, it keeps x_{atk} as the next iterate provided x_{atk} lies between x_{opt} and the unreplaced bound. Otherwise, it uses x_{opt}.

Finally, Rule 2c counts the imbalance between the number of left and right replacements. When the last three or more iterations replaced the same bound, the rule takes the weighted average of the x_{opt} calculated by Equation 7.2.21, and the unreplaced bound, with the weights determined by the imbalance. Suppose for example that the last three iterations replaced the left bound. Then Rule 2c chooses $(x_{opt} + 2 \cdot x_{right})/3$ as the next trial iterate; if that trial iterate again replaces the left bound, the next iteration again calculates x_{opt} from Equation 7.2.21, but tries $(x_{opt} + 3 \cdot x_{right})/4$; and so on.

Figure 7.2.5 compares the iteration counts required by the various rules to satisfy either Equation 7.2.17 with SLOPE_TOL = 10^{-5}, or Equation 7.2.18 with RHO_TOL = 10^{-5}, for the duct flow problem from a variety of starting points and trust lengths. Note the termination criteria differ from that used for Figure 7.2.4.

Figure 7.2.6 compares iteration counts for the trigonometric function with ten variables.

Based on these results, the code in file PLANARHK.CPP uses Rule 1c to find ρ_1 and ρ_2.

$x_{[0]}$	$\delta_{[0]}$	Iteration count, Rule:					
		1a	1b	1c	2a	2b	2c
$(0.02, 7, 1)^T$	$\|\|\Delta x_{SD}\|\|/20$	6	5	5	7	6	7
	$\|\|\Delta x_{SD}\|\|/2$	16	8	9	29	14	11
	$\|\|\Delta x_{SD}\|\|$	13	10	6	69	28	9
	$(\|\|\Delta x_{SD}\|\| + \|\|\Delta x_{NR}\|\|)/2$	2	2	2	2	2	2
$(90, 90, 90)^T$	$\|\|\Delta x_{SD}\|\|/20$	0	0	0	0	0	0
	$\|\|\Delta x_{SD}\|\|/2$	2	2	2	2	2	2
	$\|\|\Delta x_{SD}\|\|$	3	3	3	3	3	3
	$(\|\|\Delta x_{SD}\|\| + \|\|\Delta x_{NR}\|\|)/2$	2	2	2	3	3	3

Figure 7.2.5. Iterations required to optimize $\Delta f_{(2)}$ for the given starting point and trust length, for the duct flow problem.

$x_{[0]}$	$\delta_{[0]}$	Iteration count, Rule:					
		1a	1b	1c	2a	2b	2c
$(0.1, 0.1, .., 0.1)^T$	$\|\|\Delta x_{SD}\|\|/20$	3	3	3	4	4	5
	$\|\|\Delta x_{SD}\|\|/2$	7	6	5	9	6	10
	$\|\|\Delta x_{SD}\|\|$	5	4	3	8	5	7
	$(\|\|\Delta x_{SD}\|\| + \|\|\Delta x_{NR}\|\|)/2$	4	4	4	5	4	4
$(10, 10, .., 10)^T$	$\|\|\Delta x_{SD}\|\|/20$	2	2	2	2	2	2
	$\|\|\Delta x_{SD}\|\|/2$	2	2	2	2	2	2
	$\|\|\Delta x_{SD}\|\|$	6	4	4	9	5	9
	$(\|\|\Delta x_{SD}\|\| + \|\|\Delta x_{NR}\|\|)/2$	4	4	4	6	4	7

Figure 7.2.6. Iterations required to optimize $\Delta f_{(2)}$ for the given starting point and trust length, for the trigonometric function with n = 10.

7.3 Numeric Results

The figures below report test results from PLANARHK.CPP, as well as reproducing some results from NEWTRAPH.CPP and DBLDOGLG.CPP from Chapter 5.

As before, all results are for the standard termination conditions, as run on a Macintosh 7100/66.

Discussion

Generally, the results fall into three categories, based on the performance of the double dogleg algorithm: (1) where the double dogleg solves the problem purely by Newton-Raphson steps, the planar hook does so as well; (2) where the double dogleg solves the problem using some combination of gradient and Newton-Raphson steps the planar hook does so as well, requiring fewer Jacobian and fewer residual evaluations; and (3) where the double dogleg stagnates or exceeds its iteration limit, the planar hook does so as well.

In one notable exception, the planar hook requires more iterations to solve the duct flow problem from $(90, 90, 90)^T$. Inspecting the iterates, the planar hook slightly outperforms the double dogleg at each iteration-- that is, it reduces $\emptyset_{(2)}$ more aggressively with each iteration. Still, the iterates largely remain comparable in the sense that the values of each component of $x_{[k]}$ remain recognizably consistent between the double dogleg and planar hook solutions. This behavior continues until at one critical iteration the double dogleg, despite starting from a point with a larger value of $\emptyset_{(2)}$, takes a single dramatic step towards the solution-- a step which the planar hook misses.

This sort of behavior demonstrates the degree of chance present in any solution scheme, rather than any inherent difference between the two algorithms.

In another example, the planar hook solves the 5-dimensional trigonometric function, starting from ten times the standard starting point, with only 11 Jacobian evaluations, while Newton-Raphson's method requires 49 iterations and the double dogleg method stagnates after more than 70 Jacobian evaluations. Again, this kind of difference in performance reflects a fortuitous starting point rather than any particular superiority of the planar hook algorithm.

Evaluated by the overall performance on all the test problems, the planar hook is marginally superior to the double dogleg, though it clearly suffers from the same convergence difficulties.

Broyden tridiagonal function

dimension	5			50
starting point	std	10·std	100·std	std
NEWTRAPH	4, 5	7, 8	10, 11	4, 5
STDDOGLG	4, 5	7, 8	10, 11	4, 5
DBLDOGLG	4, 5	7, 8	10, 11	4, 5
PLANARHK	4, 5	7, 8	10, 11	4, 5

Figure 7.3.1. Results for the standard algorithms on the Broyden tridiagonal function. Data pairs give the number of Jacobian and residual evaluations, respectively, to solution.

Discrete boundary value function

dimension	10			100
starting point	std	10·std	100·std	std
NEWTRAPH	2, 3	3, 4	8, 9	1, 2
STDDOGLG	2, 3	3, 4	8, 9	1, 2
DBLDOGLG	2, 3	3, 4	8, 9	1, 2
PLANARHK	2, 3	3, 4	8, 9	1, 2

Figure 7.3.2. Results for the discrete boundary value function.

Discrete integral equation function

dimension	10			100
starting point	std	10·std	100·std	std
NEWTRAPH	2, 3	3, 4	8, 9	2, 3
STDDOGLG	2, 3	3, 4	8, 9	2, 3
DBLDOGLG	2, 3	3, 4	8, 9	2, 3
PLANARHK	2, 3	3, 4	8, 9	2, 3

Figure 7.3.3. Results for the discrete integral equation function.

Duct flow problem

starting point	$(0.02, 7, 1)^T$	$(0.001, 0.0039, 34.06)^T$	$(60, 60, 60)^T$	$(90, 90, 90)^T$
NEWTRAPH	8, 9	14, 64	18, 45	19, 46
STDDOGLG	8, 9	8, 34	18, 54	21, 59
DBLDOGLG	8, 9	8, 34	18, 54	21, 59
PLANARHK	8, 9	8, 34	16, 49	32, 99

Figure 7.3.4. Results for the duct flow problem.

Powell badly scaled function

starting point	std	5·std	10·std	$(-10, -9.9)^T$	$(10, 20)^T$
NEWTRAPH	11, 12	7, 8	4, 5	91, 92	2, 3, nan
STDDOGLG	24, 29	25, 30	4, 5	33, 59, stag	39, 52
DBLDOGLG	24, 29	25, 30	4, 5	34, 58, stag	39, 52
PLANARHK	16, 20	21, 26	4, 5	31, 52, stag	22, 28

Figure 7.3.5. Results for the Powell badly scaled function.

Powell singular function

starting point	std	10·std	100·std
NEWTRAPH	11, 12	14, 15	18, 19
STDDOGLG	11, 12	14, 15	18, 19
DBLDOGLG	11, 12	14, 15	18, 19
PLANARHK	11, 12	14, 15	18, 19

Figure 7.3.6. Results for the Powell singular function.

Rosenbrock function

dimension	2				10
starting point	std	10·std	100·std	$(20, 20)^T$	std
NEWTRAPH	2, 3	2, 3	2, 3	2, 3	2, 3
STDDOGLG	16, 23	3, 5	3, 5	100, 103, itns	16, 23
DBLDOGLG	16, 23	3, 5	3, 5	100, 103, itns	16, 23
PLANARHK	15, 22	3, 5	3, 5	100, 103, itns	15, 22

Figure 7.3.7. Results for the extended Rosenbrock function.

Trigonometric function

dimension	5			10	50
starting point	std	5·std	10·std	std	std
NEWTRAPH	5, 6	84, 85	49, 50	6, 7	8, 9
STDDOGLG	8, 12	71, 100, stag	75, 103, stag	69, 96, stag	100, 115, itns
DBLDOGLG	8, 12	72, 101, stag	77, 106, stag	71, 97, stag	100, 115, itns
PLANARHK	8, 11	14, 18	11, 14	89, 114, stag	100, 119, itns

Figure 7.3.8. Results for the trigonometric function.

CHAPTER 8
THE WEIGHTED DOUBLE DOGLEG ALGORITHM

This chapter develops an equation-solving algorithm following the double dogleg method of Chapter 5, but using a weighted sum of squares of residuals as its cost function.

Section 8.1 examines the case for prefering r-square as a residual norm, concluding that while convenient for use in e.g. Newton's method, r-square has no special properties with regard to solving a nonlinear algebraic problem.

Section 8.2 discusses the role of the residual norm in equation solving. Adding a descent requirement on a residual norm introduces assumptions from minimization theory which do not apply to solving algebraic systems. These assumptions focus attention on minimizing the norm, rather than on solving the equations. The discussion concludes that no single norm inherently applies when formulating a descent-based method of solving nonlinear equations.

Section 8.3 extends the double dogleg algorithm of Section 5.3 to the weighted r-square norm, and proposes a number of rules for choosing the residual weights at each iteration. Section 8.4 presents numeric results for this weighted double dogleg algorithm.

8.1 Observations on R-Square

Chapter 2 introduces the idea of using a residual norm to detect divergence in the Newton-Raphson iterates. So far the algorithms-- the double dogleg and planar hook-- enforce their descent requirement on r-square. This section examines the connections between r-square and equilibrium problems.

R-square as special norm

Sections 4.1 and 4.2 suggest several reasons to prefer r-square in a nonlinear equation solver: (1) experience and custom, deriving from its application to overdetermined systems [Dahlquist §5.7, Press §15.1]; (2) the fact that the Newton-Raphson step always defines a descent direction on $\emptyset_{(2)}$ [Dennis §6.5]; and (3) the fact that the Newton step on a model of $\emptyset_{(2)}$ built up from the linearized residuals exactly matches the Newton-Raphson step on the underlying equilibrium problem.

R-square also is more tractable than other norms, due to: (4) the convenient formulae for projecting its initial slope to, and predicting its function value at, a general step Δx; and (5) its differentiability at $r_i = 0$ [Golub §5.3]. The tractability of r-square has been coupled with the close bounds on the norm magnitudes (see Equation 4.1.6) to suggest that: (6) r-square, if no better than any other norm, at least is no worse for evaluating a step, and therefore should be adopted due to its greater convenience [Dennis §3.1].

Some of these considerations do not apply particularly to r-square when solving nonlinear equations. In particular, r-square has no special properties with regard to indicating progress on an equilibrium problem. On the other hand, its tractability makes it especially convenient computationally. The following subsections treat these topics in greater detail.

R-square and overdetermined systems

First consider the analogy with overdetermined systems. When fitting data, the residuals represent differences between fixed observations, and a fixed model with unknown parameters. Least squares fitting in overdetermined systems seeks those values for the parameters which minimize the sums of squares of the errors. A statistical argument justifies using the sum of squares when the residuals are expected to be independent random errors arranged in a Gaussian distribution [Dahlquist §4.5.5, Fletcher §6.1, Press §15.1].

In equation solving, on the other hand, the residuals-- themselves the observations-- represent the difference between the true solution and a trial solution. Since solving the problem zeros the residuals, their expected distribution is not normal, and the statistical argument does not apply.

R-square and Newton-Raphson

In the second argument, the Newton-Raphson direction always reduces $\emptyset_{(2)}$ in the neighborhood of $x_{[k]}$, making $\emptyset_{(2)}$ an appropriate norm. Section 4.3 demonstrates that the Newton-Raphson step gives a descent direction not only for $\emptyset_{(2)}$, but for any "reasonable" residual norm, i.e., for any norm setting $r_i \frac{\partial \emptyset}{\partial r_i} \geq 0$ for $1 \leq i \leq n$. Informally, since the Newton-Raphson direction zeros the local models of every residual, locally it reduces the magnitude of every residual, and therefore locally it reduces the magnitude of any reasonable residual norm. R-square has no special property in this regard.

Now consider the third fact, that a Newton step on r-square matches the Newton-Raphson step. (Properly, this holds only for the sum of squares of the linearized residuals, $\hat{r}$; see Equation 4.3.13). While convenient for adapting the Levenberg-Marquardt type minimization methods (e.g. the hook and double dogleg algorithms) to equation solving, it does not imply any special tie between r-square and the equilibrium problem. The property $\Delta x_N = \Delta x_{NR}$ applies to a broad class of functions, including r-square, but excluding $\|r\| = \sqrt{\emptyset_{(2)}}$ (because $\|r\|$ has not got constant curvature; see the discussions of Equations 4.2.7 and 4.3.13). Computationally, the only difference between the two-norm and $\emptyset_{(2)}$ is the square root, making $\|r\|$ fully equivalent to, and just as appropriate as, $\emptyset_{(2)}$ for evaluating a step in an equilibrium problem. In fact the gradients of $\emptyset_{(2)}$ and $\|r\|$ line up, since Equation 3.2.1 defines the gradient by

$$\frac{\partial \|r\|}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\Sigma r_i^2\right)^{1/2} = \frac{1}{2}\left(\Sigma r_i^2\right)^{-1/2} \frac{\partial}{\partial x_i}\left(\Sigma r_i^2\right) = \frac{g_{(2)i}}{2 \cdot \|r\|}$$

or

$$\nabla(\|r\|) = \frac{\nabla \emptyset_{(2)}}{2 \cdot \|r\|} \tag{8.1.1}$$

The gradients of $\|r\|$ are just scaled gradients of $\emptyset_{(2)}$. This scale factor, since it depends on $r\{x\}$, gives the two-norm a non-constant Hessian, making its Newton step miss $\hat{r} = 0$.

Now if taking the square root of $\emptyset_{(2)}$ destroys the match between the Newton and Newton-Raphson steps, yet does not affect the validity or appropriateness of the norm for evaluating progress towards the solution of an equilibrium problem, then the property $\Delta x_N = \Delta x_{NR}$ does not indicate any inherent connection between the norm and the equilibrium problem.

Practically this follows because the Newton step matches the Newton-Raphson step only when the model Hessian: (1) is constant; and (2) expresses only curvature due to the way the linearized residuals combine to form the cost function, i.e., excludes curvature due to second-order effects in the residuals (see Equation 4.3.10). These conditions restrict the cost function without making any special assumptions about the nature of the residual equations themselves, and so cannot predict or result in any special properties of the norm with regard to solving the residual equations.

Convenience of r-square

This leaves the convenience of r-square as a motivation for preferring it to other norms. Largely the convenience depends on its differentiability at $r_i = 0$-- for example, this property gives $\emptyset_{(2)}$ well-defined gradients when $r\{x\}$ has a zero component.

Certainly the property $\Delta x_N = \Delta x_{NR}$ allows the easy adaptation of the double dogleg algorithm. Moreover, comparing Equations 4.2.11a and 4.4.4b for $\Delta f_{(2)}$ and $\Delta f_{(1)}$ respectively, in general it is easier to predict the change in $\emptyset_{(2)}$ than in $\emptyset_{(1)}$. (Of course the Newton-Raphson direction, which sets every linearized residual model to zero at a critical stepsize $\hat{\mu} = 1$, makes calculating $\Delta f_{(1)}$ particularly simple.)

The one-norm is less tractable in other ways. The difficulty finding the direction of steepest descent when an $r_i = 0$ represents an even greater obstacle to using the one-norm within the framework of the Levenberg-Marquardt type minimization methods. Recall from Section 4.4 that, with a zero residual: (1) $g_{(1)}$ consists of subgradients, each appropriate for evaluating movement in some but not all search directions; (2) the gradient appropriate for evaluating a search in a subgradient direction is not necessarily the same subgradient; and (3) possibly a subgradient is not a descent direction. Moreover, with no assurance that Δx_N relates to Δx_{NR} (since the one-norm does not satisfy Equation 4.3.13), the double dogleg curve may fail the original criteria of Section 3.6, since: (1) a given trust length may specify more than one point on the curve (the curve may not be well defined); and (2) $f_{(1)}$ may not decrease monotonically along the curve (the curve may not be reasonable).

Equation 4.3.13 defines a general class of residual norm which retains the convenient properties of r-square, at a slight cost in computational and data storage requirements: the weighted r-square, $\emptyset_{(W)}$. Section 4.5 treats this class, without indicating how to choose the weights, by deriving expressions similar to those for r-square. These expressions introduce a diagonal weighting matrix, W; for r-square, W = I. To fully adapt the double dogleg algorithm to $\emptyset_{(W)}$ requires a weighting rule, adding further to the computational burden. Section 8.3 proposes a number of weighting rules.

Relative norm magnitudes

Given the convenience of using r-square, the incentive to use some other norm seems small. Indeed, Equation 4.1.6,

$$\|r\|_\infty \leq \|r\| \leq \|r\|_1 \leq \sqrt{n}\cdot\|r\| \leq n\cdot\|r\|_\infty$$

has been interpreted as indicating that, with the norms more or less bounding one another, the differences between them are relatively unimportant in terms of evaluating the progress of a step, and that therefore the choice should be driven by convenience [Dennis §3.1].

While Equation 4.1.6 restricts the relative values of the norms at a single point-- for instance at the current iterate $x_{[k]}$ or at a proposed iterate $x_{[k+1]}$--, it allows considerable latitude in a particular norm between two points such as $x_{[k]}$ and $x_{[k+1]}$. That is, bounding

the relative magnitudes of $\emptyset_{(1)[k]}$ and $\emptyset_{(2)[k]}$ does little to guarantee any particular relation between $\emptyset_{(1)[k+1]}$ and $\emptyset_{(2)[k+1]}$.

Of course a step which decreases one norm dramatically should decrease the other norms as well. For example, if

(8.1.2a) $$\emptyset_{(2)[k+1]} < \frac{1}{\sqrt{n}} \emptyset_{(2)[k]}$$

then

(8.1.2b) $$\begin{aligned} \emptyset_{(1)[k+1]} \le \sqrt{n} \cdot \emptyset_{(2)[k+1]} < \emptyset_{(2)[k]} \le \emptyset_{(1)[k]} \\ \emptyset_{(1)[k+1]} < \emptyset_{(1)[k]} \end{aligned}$$

and a sufficiently large decrease in $\emptyset_{(2)}$ guarantees a decrease in $\emptyset_{(1)}$ as well.

Unfortunately the bound $\emptyset_{(2)[k+1]} < \emptyset_{(2)[k]}/\sqrt{n}$ describes the sort of progress for which the particular norm clearly ought not to matter. Take for example a problem of small dimension, n = 9. Equation 8.1.2a requires a step to cut $\emptyset_{(2)}$ by two-thirds before Equation 8.1.2b guarantees that the same step decreases $\emptyset_{(1)}$. Larger problem dimensions demand even greater decreases in $\emptyset_{(2)}$-- for example, in a 100-dimensional problem, a step must cut $\emptyset_{(2)}$ to a tenth of its initial value to insure a decrease in $\emptyset_{(1)}$.

The limits on the relative magnitudes of the norms indicate only a rough equivalence between the norms when evaluating a step towards the solution of an equilibrium problem.

Examples

The examples below compare $\emptyset_{(2)}$, $\emptyset_{(1)}$, and the distance ||x - x**|| to the known solution, for line searches on three sample problems and starting points. The examples represent the extremes of behavior, in which: (1) both norms reliably predict the best stepsize (the wall convection problem); (2) one of the norms predicts the stepsize better than the other norm (the duct flow problem); and (3) neither norm adequately predicts the best stepsize (the Rosenbrock function).

All three examples plot $||r|| = \sqrt{\emptyset_{(2)}}$ rather than $\emptyset_{(2)}$, for reasons of scale (again due to Equation 4.1.6). However, stepsizes in the steepest descent directions refer to $g_{(2)}$, rather than to $\nabla(||r||)$. By Equation 8.1.1, the gradients take the same direction, so using stepsizes for $d = \nabla(||r||)$ would affect only the scaling of the independent axis. Thus, the figures merely take the square root of the norm of interest, as the last operation before graphing, and for comparison purposes represent $\emptyset_{(2)}$. Note that $||r||_1 \ge ||r||$ in every curve, as per Equation 4.1.6.

Wall convection problem

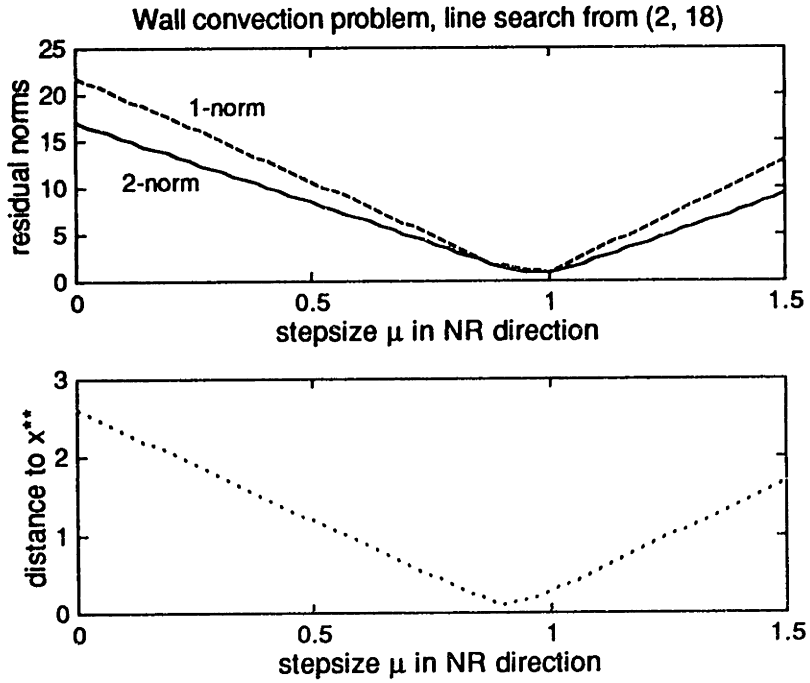

Figure 8.1.1. Line search in the Newton-Raphson direction for the wall convection problem from $(2, 18)^T$. *The upper plot shows* $\sqrt{\emptyset_{(2)}}$ *(solid) and* $\emptyset_{(1)}$ *(dashes); note* $\emptyset_{(1)} \geq \sqrt{\emptyset_{(2)}}$ *by Equation 4.1.6.*

Figure 8.1.1 shows a line search in the Newton-Raphson direction for the wall convection problem. The search starts at $(2, 18)^T$. From the tests in Section 5.4, this starting point presents little difficulty for either the Newton-Raphson or the double dogleg methods; both take three full Newton-Raphson steps to find a zero residual vector, within the default tolerance.

The upper plot shows that both $\emptyset_{(1)}$ and $\emptyset_{(2)}$ predict nearly the same stepsize, and the lower plot shows this predicted stepsize only slightly overestimates the optimal stepsize.

Duct flow problem

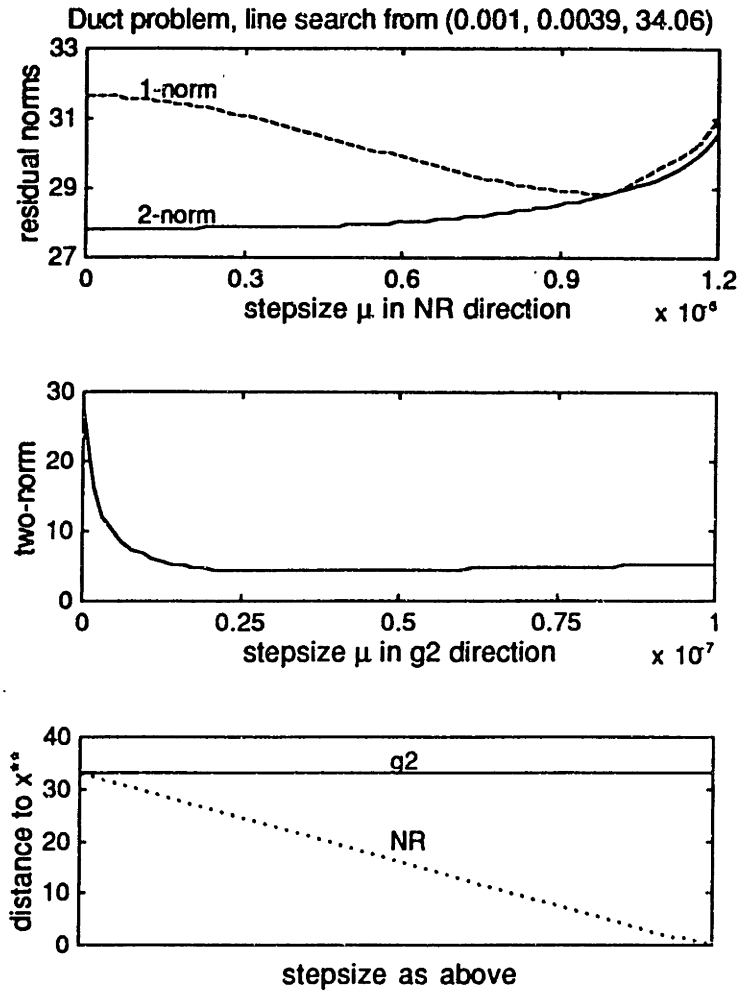

Figure 8.1.2. Line searches for the duct flow problem from $(0.001, 0.0039, 34.06)^T$. The upper plot shows $\emptyset_{(1)}$ (dashes) and $\sqrt{\emptyset_{(2)}}$ (solid) for the Newton-Raphson direction. The middle plot shows $\sqrt{\emptyset_{(2)}}$ in the $g_{(2)}$ direction. The third plot shows the distance to x^{**} for both the Newton-Raphson (dots) and $g_{(2)}$ (solid) search directions.

Figure 8.1.2 gives an example where one norm predicts the optimal stepsize better than another.

The upper plot shows a Newton-Raphson search on the duct flow problem from $(0.001, 0.0039, 34.06)^T$. (This starting point comes from a series of exact line minimizations on $\emptyset_{(2)}$, all in the Newton-Raphson direction, and represents a case where the Newton-Raphson direction becomes orthogonal to the norm of choice [Fletcher §6.1].) In the Newton-Raphson direction, the optimal stepsize $\mu \approx 1.19 \cdot 10^{-6}$ from the lower plot (the figure does not extend significantly beyond the optimal value because stepsizes greater than about $1.24 \cdot 10^{-6}$ induce an error in the first residual, by trying to take the log of a negative number). For this search direction, shown in the upper plot, the minimum of $\emptyset_{(1)}$ at $\mu \approx 9.8 \cdot 10^{-7}$ nearly predicts the optimal stepsize, while the minimum of $\emptyset_{(2)}$ occurs at $\mu \approx 1.2 \cdot 10^{-10}$, a poor prediction.

Switching to the steepest descent direction of $\emptyset_{(2)}$, a line search would find the minimum at $\mu \approx 3.3 \cdot 10^{-8}$ (the Cauchy point is at $\mu \approx 1.9 \cdot 10^{-9}$). Moreover a step in the $g_{(2)}$ direction decreases $\emptyset_{(2)}$ dramatically. From the third plot, however, no step in the steepest descent direction does much to advance the solution-- the best stepsize is $\mu \approx 2.8 \cdot 10^{-8}$-- while the Newton-Raphson direction, if properly exploited, moves much closer to the solution.

The double dogleg algorithm, started from this point, pulls back from the full Newton-Raphson step due to the residual error. After halving the trust length 20 times, it settles on a point very nearly in the steepest descent direction, just beyond the Cauchy point on the double dogleg curve.

Rosenbrock function

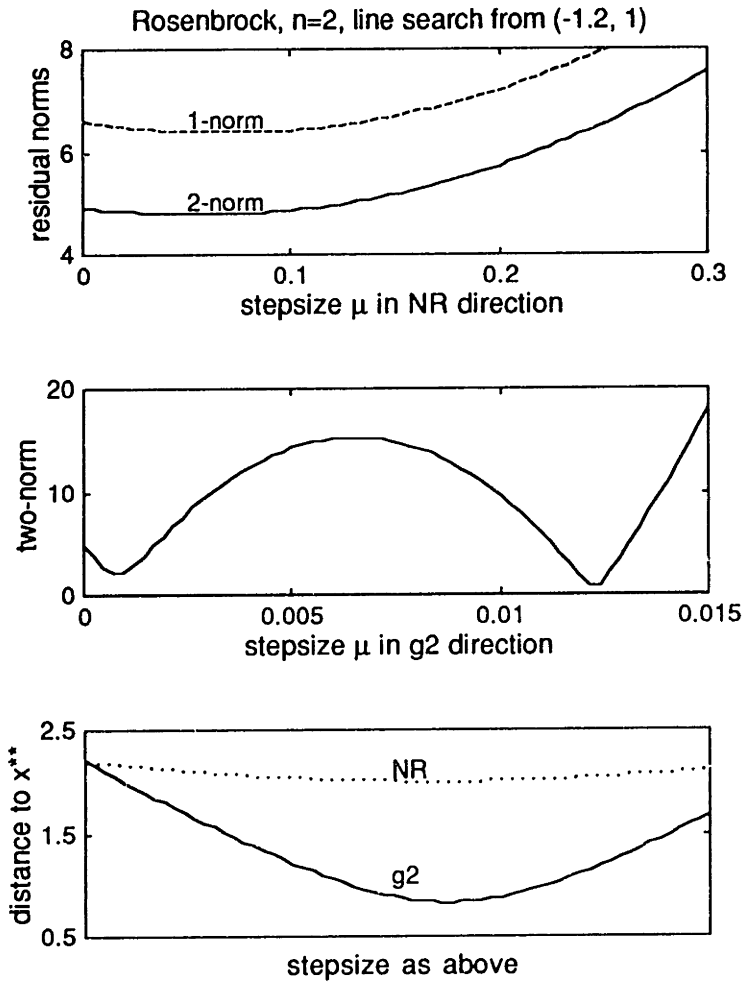

Figure 8.1.3. Line searches for the two-dimensional Rosenbrock function. The searches start at $(-1.2, 1)^T$.

In Figure 8.1.3, neither residual norm predicts the optimal stepsize well, for either the Newton-Raphson or the $g_{(2)}$ directions.

A search in the Newton-Raphson direction, shown in the upper plot, minimizes $\emptyset_{(2)}$ at $\mu \approx 0.056$ and minimizes $\emptyset_{(1)}$ at $\mu \approx 0.068$. However the minimum distance to x**, shown in the bottom plot, occurs at $\mu \approx 0.17$ in the Newton-Raphson direction.

Also from the lowest plot in Figure 8.1.3, the steepest descent direction of $\emptyset_{(2)}$ approaches the solution more closely than does the Newton-Raphson direction. However, from the middle plot, exploiting this gradient direction properly may be difficult-- not only because the $g_{(2)}$ search direction finds two local minima, but also because the optimal stepsize lies between them, at a point where $\emptyset_{(2)}$ obtains about two and a half times its initial value. Thus the best stepsize in terms of reducing ||x - x**|| does not satisfy the descent requirement for a line search in the $g_{(2)}$ direction.

Which of the two minima traps a search in the $g_{(2)}$ direction, depends on the search algorithm. A trust region method could place the next iterate near either minimum, depending on the initial trust length and the rules for increasing or decreasing it. In fact the double dogleg algorithm, started from $(-1.2, 1)^T$, does not choose a point along either of the search directions shown in Figure 8.1.3. After rejecting the full Newton-Raphson step-- as the topmost plot in Figure 8.1.3 implies, $\mu = 1$ in the Newton-Raphson direction fails the descent requirement-- the algorithm cuts the trust length, to one-tenth of the Newton-Raphson length (see Equation 3.6.6). The resulting second trial step lies along the cutback path and so mixes the Newton-Raphson and gradient directions; the algorithm accepts this step. (Figure 8.2.1, below, shows the complete double dogleg search, including this first step. The level curves of $\emptyset_{(2)}$ depicted in the figure explain the hump in the graph of ||r|| in the $g_{(2)}$ direction.)

8.2 Observations on Role of Residual Norm

If r-square enjoys no inherent connection to the underlying structure of an equilibrium problem, neither does any other norm. This section discusses the norm's role in solving nonlinear equations, and shows how treating the norm strictly as a cost function in a descent-based search algorithm introduces inappropriate assumptions about its behavior and meaning.

R-square in double dogleg algorithm

In the double dogleg algorithm, r-square takes on a broader role than simply detecting divergence: it represents a scalar index of the quality of a trial solution, and its gradient suggests an alternate search direction when the full Newton-Raphson step fails. Tacitly the norm serves as a surrogate measure of the distance between a trial iterate and the solution of the equilibrium problem-- presumably a smaller norm indicates a better estimate of x**, and presumably the gradient, by pointing in the direction of greatest decrease in the norm, suggests a likely search direction.

The two examples below show the stabilizing influence a descent requirement exerts on a Newton-Raphson based nonlinear equation solver. They also suggest how this influence may aid or slow its progress. The iteration counts reported in Section 5.4 verify this, and demonstrate that the method of Newton-Raphson may converge quickly despite violating the descent criterion during at least a few iterations.

Example: trigonometric function

Figure 6.4.1, above, compares the Newton-Raphson and double dogleg solutions for the 2-dimensional trigonometric function, superposing their trajectories over the level

curves of r-square on the (x_1, x_2) plane. The figure depicts a successful application of $\emptyset_{(2)}$, both in detecting divergence and in setting the search direction. Starting from $(0, 0.3)^T$, Newton-Raphson's method behaves wildly until it falls within the radius of convergence of the root $x^{**} = (0.2431, 0.6127)^T$. The double dogleg algorithm, on the other hand, converges quickly to $x^{**} = (0, 0)^T$, because its first iteration chooses a step along the cutback path, in order to avoid violating the descent requirement at the full Newton-Raphson step. See Figure 6.4.2 for the complete iterative sequence.

As an indication of Newton-Raphson's instability on this problem, note that small changes in the starting point cause unpredictable changes in the root to which the method finally converges. Ultimately Newton-Raphson's method always finds a root in this two-dimensional version of the problem, where the two fundamental roots shown in Figure 6.4.1 repeat at distances of 2π in either coordinate direction. However for other problems this will not be the case, and Newton-Raphson may diverge without ever entering the radius of convergence of a root.

Example: Rosenbrock function

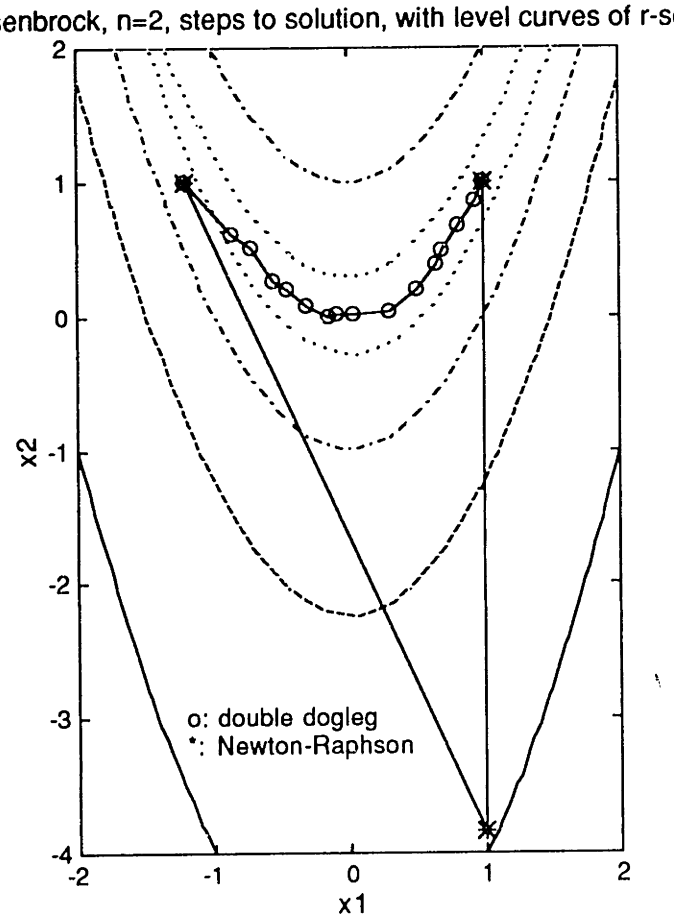

Figure 8.2.1. Newton-Raphson and double dogleg steps for the two-dimensional Rosenbrock function, starting from $(-1.2, 1)^T$. The level curves $\emptyset_{(2)} = 2500$ (solid), 500 (dashes), 100 (dash-dots), and 10 (dots) form elongated curving valleys about the solution $x^{**} = (1, 1)^T$.

Figure 8.2.1 compares the Newton-Raphson and double dogleg trajectories for the 2-dimensional Rosenbrock function. Starting from $(-1.2, 1)^T$, the first iteration of Newton-Raphson's method sets $x_1 = 1$, in order to zero the (linear) second residual equation. From Equation 1.5.8c, this second equation depends only on x_1, so every future Newton-Raphson step, or partial Newton-Raphson step, leaves $x_1 = 1$ constant. Then since the first residual equation is linear in x_2, the next Newton-Raphson step effectively treats a linear system, and solves the problem exactly. (The added equations in the extended Rosenbrock function continue this pair-wise linearity, and Newton-Raphson's method solves it from any starting point in exactly two iterations.)

As shown in the figure, Newton-Raphson's method takes these steps without regard for r-square. The double dogleg algorithm, on the other hand, follows the narrow, banana-shaped valleys formed by the level curves. Each step must end at a lower value of r-square, and hence must fall within the interior of the level curve on which it starts. Thus restricted, the algorithm takes the full Newton-Raphson step only in the last two iterations (however, every other step falls along the cutback path-- in other words, every step exceeds the Cauchy length).

Assumptions borrowed from minimization theory

The success of the descent-based methods of solving nonlinear equations depends on: (1) using the residual norm to detect divergence in the Newton-Raphson step; and (2) developing alternate search directions directly from the norm (i.e., using its gradient) when the Newton-Raphson step fails. The methods borrow this directional choice, and rules for evaluating a step in any direction, from the minimization theory of Chapter 3.

Because the methods of function minimization aim only to reduce the cost function, the cost function itself measures the progress made by each iteration. Carried over to equation solving, the notion that the residual norm measures progress towards the solution tends: (1) to institute the norm as an absolute rule for estimating the quality of a trial iterate; (2) to represent the descent requirement as a convergence mechanism; (3) to imply that the gradient of the residual norm makes the most reasonable alternate to the Newton-Raphson search direction; and (4) to identify the norm as an estimate of the distance from a trial iterate to the solution of the equilibrium problem.

The following subsections treat these assumptions in greater detail.

Assumption: norm has absolute significance

In function minimization, the cost function directly compares two points in an iterative sequence, preferring that with the least positive function value. Used to evaluate every trial point, the cost function applies globally, allowing absolute comparisons of one point to any other across the entire iterative sequence.

Applied to equilibrium problems, taking the residual norm as a global indicator of the quality of a trial solution implies that it meaningfully compares two points even if they do not follow each other in the iterative sequence.

By contrast, the residual norm first appeared in the equation-solving context purely as a local estimater, comparing a new trial point to that immediately preceding it in the iterative sequence. This relative interpretation, emphasizing the norm's role in avoiding overly large steps, originates in the same Newton-Raphson line search which prompted the introduction of the norm. In that search, moving away from $x_{[k]}$, the nonlinear residuals deviate from the behavior predicted by the linearized model, $\hat{r}_{[k]}$, and the norm estimates the deviation.

Under this relative interpretation, the norm determines the acceptable tradeoff between progress in each residual element [Press §9.6]. Different norms achieve different compromises in terms of exchanging an increase in one residual element for a decrease in another. Consider, for example, a two-dimensional problem with $r_{[k]} = (10, 10)^T$. Increasing the first residual to 11, the one-norm stays the same if $r_{[k+1]} = (11, 9)^T$, while $\emptyset_{(2)}$ requires $r_{[k+1]} = (11, 8.89)^T$ to consider the step neutral. In this case the one-norm demands a smaller decrease in r_2 in order to allow the same increase in r_1.

Assumption: descent requirement forces convergence

In function minimization, the descent requirement forces the cost function to decrease at every iteration, and so forces convergence to a stationary point.

Applied to equilibrium problems, the descent requirement forces convergence to a stationary point only if enforced on a single residual norm, which then serves as an absolute measure of progress. If that residual norm has unwanted stationary points $x^* \neq x^{**}$, or is very flat in the region of interest, then the descent requirement causes the method to stagnate, and possibly to discover a singular Jacobian (see below).

Suppose, on the other hand, that an algorithm requires of each step only that it achieve some reasonable tradeoff between progress in the residual components-- for example, it might accept a step that reduces just one of $\emptyset_{(2)}$, $\emptyset_{(1)}$, or $\emptyset_{(\infty)}$, even if the new iterate increases the other norms. Adopting the relative interpretation, the algorithm need not require that any single norm decrease monotonically throughout the iterative sequence. Now the descent requirement only prevents a single step from making an unacceptable tradeoff between the residual elements-- i.e., it prevents divergence of a step, rather than forcing convergence of the overall sequence.

Giving up global convergence to a stationary point may help or hinder a nonlinear equation solver. If it can avoid the unwanted stationary points in a fixed residual norm, the solver may avoid stagnation and false roots. Since close to a solution the full Newton-Raphson step converges quickly, and satisfies a descent requirement on any reasonable norm, changing the norm will not prevent finding the global minimum. However, by changing the norm the solver loses its global test by which to judge the entire iterative sequence, and risks e.g. jumping endlessly between two iterates, each with a smaller value of one norm, but a larger value of another norm, than has the other iterate.

Note that an algorithm which uses a single residual norm, requiring that it decrease at each iteration, effectively adopts the absolute interpretation. Even if it incorporates the norm in order to avoid divergence in a single step, it uses the norm as a global rule for estimating the quality of one trial point relative to another. The original, motivating, local view implicitly embodies the absolute interpretation if it admits only one residual norm, since it then compares every iterate using the same, global, standard.

Assumption: norm dictates gradient direction

In function minimization, the descent requirement leads directly to the steepest descent search direction. Thus the double dogleg algorithm defines its search along the gradient for short steps, in case the Newton direction gives unacceptable changes in the cost function.

Applied to equilibrium problems, taking a norm as a measure of progress towards the solution implies that the gradient of that norm makes a reasonable search direction, for example if the Newton-Raphson direction fails.

By contrast, the geometric interpretation of the residual models permits the choice of a reasonable search direction independently of any residual norm. Indeed, once chosen, the geometric interpretation of a search direction indicates properties of a residual norm appropriate for evaluating progress along it. Equation 4.3.6 gives the gradient of a general residual norm $\emptyset$ as the weighted sum of the gradients of the residual functions, ∇r_i, entered in the Jacobian rows. Thus it suggests choosing a search direction as the linear combination of these residual gradients,

$$d = \sum_{i=1}^{n} p_i \cdot \nabla r_i \tag{8.2.1}$$

with the p_i calculated by some, as yet unspecified, algorithm.

To choose the p_i without reference to a residual norm, first suppose that the Newton-Raphson direction has failed. Since the Newton-Raphson direction reduces the magnitude of every linearized residual, its failure indicates some nonlinear residual relation does not respond to the Newton-Raphson search. A solver determined to reduce just one such residual could reasonably take the corresponding row from the Jacobian as a search direction. As an intermediate strategy, between attempting to reduce all the residuals and attempting to reduce just one, a solver can mix the gradient directions of all the residual functions, taking some linear combination of them, as in Equation 8.2.1.

If the Jacobian models the residuals fairly well-- due either to nearly linear residual relations, or to residual magnitudes small enough that any nonlinearity does not much affect the model over the short step lengths of interest-- then an appropriate choice of the p_i accords greater influence to larger $|r_i|$, since exactly these residuals stand in greatest need of reduction. Giving the corresponding ∇r_i a larger coefficient p_i in the linear summation helps to insure that the resulting search direction aligns significantly with the direction needed to reduce that large residual.

If on the other hand the magnitude of a residual has no bearing on the desirability of travel in its steepest descent direction-- for instance because the residual has a minimum or is highly nonlinear in the region of interest-- then the linear summation of Equation 8.2.1 should not favor the gradient direction for any one residual. Weighting the ∇r_i equally helps to insure that no residual gradient dominates the attempted net search direction.

These arguments, without invoking the notion of a residual norm, imply weighting choices corresponding to $p_i = r_i$ for relatively tractable or predictable residual functions, and to $p_i = \text{sign}\{r_i\}$ for relatively less tractable residuals (the p_i take the sign of the r_i since to zero a negative residual requires a search with $d = -\nabla r_i$). Of course these choices give search directions corresponding to, among other gradients, $g_{(2)}$ and $g_{(1)}$. See Equations 4.3.14 and 4.4.1.

In Equation 4.3.6, the $p_i = \partial\emptyset/\partial r_i$. Given a family of norms, choosing the p_i would select that member of the family whose gradient yields the same search direction. The resulting cost function would be appropriate for evaluating a search in the direction calculated by Equation 8.2.1.

Section 4.3 discusses the problem of constructing a residual norm by choosing the desired p_i and integrating to obtain $\emptyset$. The norms described above satisfy the requirements of Section 4.3 for a "reasonable" selection of the p_i. With the additional requirement that a Newton step on the cost function built up from the linearized residuals should match the Newton-Raphson step (see Equation 4.3.13), the choice $p_i = r_i$ gives $\emptyset_{(2)}/2$ for relatively tractable residuals, and the choice $p_i = \text{sign}\{r_i\}$ for relatively less

tractable residuals describes a version of $\emptyset_{(w)}$, weighted to give the one-norm gradient direction.

Importantly, the discussion above begins with a geometric view of the requirements for a reasonable search direction, and argues towards particular selections of $\emptyset$, rather than starting from a particular residual norm and arguing towards an appropriate search direction. Thus, instead of asserting that $\emptyset_{(1)}$ and $\emptyset_{(2)}$ satisfy the requirements for reasonable norms, so that their gradients make likely search directions, the argument asserts that $g_{(1)}$ and $g_{(2)}$ make reasonable search directions, so that the corresponding norms make natural means of evaluating progress in those directions.

Assumption: norm measures distance to solution

In function minimization, the global minimum remains unknown, so a method may declare any iterate a solution if it finds no point of lesser function value in the immediate neighborhood (see e.g. Equation 5.2.1). Thus the cost function directly measures progress towards a minimum.

Applied to equilibrium problems, taking a norm as the best available measure of progress towards the solution tends to identify it as an indirect measure of the distance from a trial point to the solution of the equilibrium problem. The idea of the radius of convergence-- see for example Equation 2.2.16b-- reinforces this notion by creating a geometric picture that progress in equilibrium problems decreases the distance between the current iterate and the solution, x**.

The examples of Section 8.1 identify the "optimal" stepsize as that which minimizes the distance to the known problem solution along the search path. Similarly, it considers the "best" norm as that whose minimum most nearly predicts the optimal stepsize. Such language presumes that the optimal step along a given search path places the next iterate closest to the (unknown) solution of the problem. This definition, defensible in the context of a single line search in the Newton-Raphson direction, breaks down in the larger context of the iterative sequence. Because a search algorithm introduces considerations beyond the simple line search, other, "nonoptimal," stepsizes and norms may prove more efficacious in terms of solving the equilibrium problem.

Consider a line search in the Newton-Raphson direction. Even supposing that a residual norm predicts ||x - x**|| perfectly, finding the "best" stepsize in the sense used here implies exact line minimizations-- wasted computational effort, since the Newton-Raphson method converges quickly in the neighborhood of the solution, and since exact minimization probably does not significantly advance the search when far from the solution. Too, if an algorithm defines its search path based on the norm's gradient direction-- as do the double dogleg and planar hook-- then comparing two norms along a fixed path does not indicate how the choice of norm affects the search.

In any event, finding the point closest to x** along a search direction may not guarantee the success of the next iteration; possibly a "worse" choice at one iteration could allow better choices in those to follow, by finding an arguably better point from which to start the next iteration. The planar hook algorithm of Chapter 7, started from $(90, 90, 90)^T$ on the duct flow problem, exemplifies this kind of behavior. At each step it slightly outperforms the double dogleg method, until at one critical iteration the double dogleg, despite its "worse" starting point in r-square, takes a much more favorable step towards the solution.

Ultimately, of course, no norm predicts ||x - x**|| very well all the time, and in practice the distance from a trial point to the problem solution remains unknown.

Nevertheless, since the residual norm carries some burden as an absolute predictor of the quality of a trial point, it is reasonable in the examples to compare the norms using ||x - x**||, in the search for insight into the nature of the residual norms.

Failure of minimization assumptions

In function minimization, the cost function defines the problem.

Applied to equilibrium problems, the assumptions discussed above-- of the global significance of the norm, and hence of its absolute abilities both to estimate the quality of a trial solution and to indicate a likely search direction for a better trial solution-- fail because the residual norm enters the problem peripherally, that is, purely as a numeric convenience for controlling the trial step selection.

In fact, the residuals themselves, introduced in Equation 2.1.1, exist only as a notational convenience in the problem specification. Thus Chapter 1 develops the governing equations for the wall convection and duct flow problems without reference to residuals. It follows that no residual norm intrinsically applies to an equilibrium problem, serving as the best estimator of the quality of a trial solution, as the best rule for exchanging divergence in one equation for progress in another, or as the best indicator of likely alternate search directions.

Despite this, r-square remains the single dominant residual norm used in nonlinear equation solvers, and remains the only norm actually applied in the algorithms used in the thesis so far.

Arguments against using a single norm

The arguments applied in Section 8.1 to r-square apply equally to any other norm, and the examples there demonstrate that $\emptyset_{(1)}$ may be no more appropriate for evaluating a step than is $\emptyset_{(2)}$. Similarly, the difficulties with r-square cited in Sections 4.1 and 4.2, all of which relate to the possibility of local minima in $\emptyset_{(2)}$, apply equally to other norms.

Local minima can trap a descent-based equilibrium problem solver away from the solution x**. Equation 4.2.1 expresses the difficulty with such minima in r-square: since $g_{(2)} = 2J^Tr$, finding $g_{(2)} = 0$ with $r \neq 0$ implies a singular Jacobian. (In practice, $g_{(2)} \approx 0$ implies a very flat cost function and possibly a Jacobian singular to machine precision.)

Any residual norm may have minima besides the one at the solution to the equilibrium problem. From Equation 4.3.5, the zero gradient of a general norm, $g = J^Tp = 0$, implies either $p = 0$ or J singular. For a "reasonable" cost function, as defined in Section 4.3, if $r \neq 0$ then $p \neq 0$ and the minimum comes from a singular Jacobian. Since descent-based methods seek out local minima in the cost function of choice, an algorithm using any single residual norm is susceptible either to stagnating or to finding a singular Jacobian.

Arguments for changing the norm

Section 4.3 points out that unless $J_{[k]} = 0$ (that is, unless every residual equation has a local minimum at $x_{[k]}$), some cost function exists for which $J_{[k]}^Tp \neq 0$. A singular Jacobian does not imply a zero gradient, so long as p, the norm's gradient with repect to r, does not lie in the left nullspace of $J_{[k]}$. Thus an algorithm may always escape a local minimum in $\emptyset_{(2)}$, or in any other cost function, by changing to a different cost function.

Now suppose a descent-based nonlinear equation solver, encountering an undesired stationary point in a particular norm, can respond by switching to a different norm. Call such an algorithm, norm-selecting. A norm-selecting solver can do more than just react to

stationary points, it also can choose the norm in order to avoid them. It might do so at the start of each iteration: for example, it might define a relative gradient as in Equation 5.2.2a, and choose from among a selection of norms that whose relative gradient has the greatest magnitude. Alternately a norm-selecting algorithm might change norms during an iteration: for example, it might begin an iteration using a default norm, and change norms (and possibly change its gradient-based search path) depending on the performance of the default norm (or its gradient direction). Chapter 9 suggests other possibilities.

Arguments against changing the norm

In the relative interpretation, changing norms amounts to changing the acceptable tradeoff between progress in the residual elements. Though an alternate norm always exists, it may not always be possible to find an alternate norm which defines a reasonable tradeoff. The Freudenstein and Roth function, discussed in Section 4.1, has local minima in $\emptyset_{(1)}$, $\emptyset_{(2)}$, and $\emptyset_{(\infty)}$ at $(11.41, -0.8968)^T$, precisely because both residuals have the same magnitude there; any residual norm which accords the same weight to residuals of the same magnitude has a local minimum at this point. Clearly the automatic selection of an appropriate replacement norm may be difficult.

Moreover, a norm-selecting algorithm, free to change norms between iterations (and perhaps even during an iteration), can remain a descent method in the relative sense-- so that each step decreases some cost function and thus advances the solution according to some measure of performance-- without meeting any absolute convergence criterion-- so that the iterative sequence does not monotonically decrease a single norm.

Such an algorithm may control divergence, in the sense that it does not allow overly large steps, yet may fail to significantly advance the solution, for example if successive steps systematically cancel each other out. Suppose two iterates x_a and x_b have $\emptyset_{(1)a} < \emptyset_{(1)b}$ and $\emptyset_{(2)a} > \emptyset_{(2)b}$. Presumably $\emptyset_{(1)}$ always evaluates smaller in the neighborhood of x_a than in the neighborhood of x_b, and $\emptyset_{(2)}$ always evaluates larger. A norm-selecting solver could cycle endlessly between iterates in the neighborhood of x_a and others near x_b-- in essence diverging, but without displaying the kind of behavior which originally prompted the introduction of the residual norms.

To avoid this type of divergence, a norm-selecting solver must substitute some other performance criterion intended to insure absolute stability (or to insure against absolute divergence). That is, it must make certain each iterate is appropriate not only in reference to that immediately preceding it, but also with respect to the entire preceding sequence of iterates.

8.3 Weighted R-Square Double Dogleg Algorithms

Section 8.1 argues that of all the properties of r-square, its convenience alone makes a compelling case for using it in a Newton's method type minimization algorithm such as the double dogleg. Section 8.2 argues that no single residual norm captures any essential behavior in the solution of nonlinear equations, and makes a case for changing the norm from iteration to iteration. In addition, Section 8.2 suggests a geometric model for choosing a search direction when the Newton-Raphson direction fails.

This section answers these comments by developing a double dogleg algorithm which minimizes a weighted sum of squares of residuals. This weighted r-square, a class of residual norm to which r-square belongs, retains the convenience of r-square, while allowing alternate valuations of the tradeoff between the residual elements. The weighted

double dogleg algorithm adapts the double dogleg method of function minimization to equation solving using the results of Section 4.5. It establishes weights by a variety of rules, described below.

Rationale

Weighted r-square, because it retains the convenient properties of r-square, makes an attractive alternate residual norm for use when adapting the double dogleg method of function minimization to solve nonlinear equilibrium problems. Moreover it defines a general family of norms which accomodates any selection of p_i in Equation 8.2.1. Thus, using $\emptyset_{(w)}$ an algorithm can identify a search direction, then choose weights to find a cost function with a matching gradient direction.

The algorithm presented here follows the double dogleg of Section 5.3, replacing $\emptyset_{(2)}$ with the weighted sum of squares, $\emptyset_{(w)}$, defined in Section 4.5. From Equations 4.5.4,

$$\emptyset_{(w)} = r^T W r \quad \text{and} \quad p_{(w)} = 2Wr$$

making

$$g_{(w)} = 2J^T W r$$

from Equation 4.5.5a.

The numeric experiments described here try out a number of rules for calculating diagonal entries for the weighting matrix W (note that for convenience these diagonal elements are labelled w_i rather than $W_{i,i}$). Each experiment uses a single weighting rule. Though the weights change from iteration to iteration, so that $W = W_{[k]}$, the rule used to select the $w_{i[k]}$ does not change between iterations. Therefore the weighted double dogleg algorithm does not select norms in the sense defined in Section 8.2. That is, it does not change norms to avoid minima, though the weighting rules do anticipate norm-selecting algorithms by varying the relative tradeoff between progress in each residual element at each iteration.

Confusion between weights

Section 8.2, referring to the p_i in Equation 8.2.1, identifies them as weights in the linear combination $d = \sum p_i \cdot \nabla r_i$. Unfortunately this terminology risks confusion with the w_i in Equation 4.5.1, which weights the squared residuals to find $\emptyset_{(w)} = \sum w_i \cdot r_i^2$.

Equation 4.5.2a relates these weights:

$$p_{i(w)} = 2w_i \cdot r_i$$

but does not eliminate the potential for confusion. Therefore the following discussion refers to the w_i as the residual weights, and to the p_i as the gradient weights.

A second, minor, potential source of confusion lies in the signs of the weights. To define a reasonable norm, the w_i must remain positive, while the p_i must take the sign of the r_i, whether positive or negative. As Equation 4.5.2a shows, satisfying one requirement automatically satisfies the other.

Code file

Appendix 3 lists WTDDOGLG.CPP, a file implementing the weighted double dogleg algorithm. Because the various weighting rules calculate the $w_{i[k]}$ using different data, available at different points in the iterative loop, each places special structural demands on

the flow of logic through the algorithm. The file handles these requirements by: (1) using preprocessor commands to select between relevant patches of code at compile-time; and (2) setting and testing control variables during execution. This program design requires a separate executable to test each weighting rule, with WTDDOGLG.CPP compiled using a distinct preprocessor definition each time. Thus, each weighting rule generates a unique code implementation, as determined by the preprocessor commands.

The resulting code implements the particular weighting rule less directly than could be realized in a dedicated file, due to compromises made in the overall program design. These compromises attempt to minimize the number of code fragments required to implement any given weighting rule, and to impose an overall structure which parallels that of the double dogleg algorithm as closely as possible.

R-square rule 1

Weighting rule 1 sets W = I at every iteration:

$$w_{i[k]\{1\}} = 1 \tag{8.3.1}$$

The rule makes $\emptyset_{(w)} = r^T r$, and checks that the weighted double dogleg algorithm reproduces the double dogleg algorithm of Section 5.3. However its importance extends beyond checking for logical errors in the code file; it also demonstrates that $\emptyset_{(2)}$ represents a rational choice of weights.

Consider the gradient weights corresponding to Equation 8.3.1. From Equation 4.5.2a,

$$p_{i[k]\{1\}} = 2 \cdot r_i$$

Ignoring the factor 2 (since the weighted double dogleg method does not depend on scalings in $W_{[k]}$), rule 1 weights each residual gradient in direct proportion to the size of the corresponding residual-- in other words, it weights a residual's contribution to the overall steepest descent direction roughly according to the need to reduce this residual.

Later weighting rules extend this idea, choosing the gradient weights to achieve a desired overall gradient for the residual norm, and calculating the residual weights w_i based on the selection of the p_i.

Discussion

The figures in Section 8.4 show that weighting rule 1 exactly mimics the double dogleg algorithm for r-square, except in those cases where the solution stagnates. In these cases, slight rounding differences between the implementations affect the exact number of Jacobian and residual evaluations allowed before the algorithm detects stagnation. However, the rounding differences never become significant enough either: (1) to stagnate the weighted algorithm where the original double dogleg found a solution; or (2) to find a solution where the original double dogleg stagnated.

Within the limits of machine precision, weighting rule 1 implements the double dogleg algorithm of Section 5.3.

One-norm weighting

Section 8.2 compares the gradient weights $p_i = r_i$ for $\emptyset_{(2)}/2$ and $p_i = \text{sign}\{r_i\}$ for $\emptyset_{(1)}$, suggesting that in some circumstances the $g_{(1)}$ search direction may be more appropriate than $g_{(2)}$. The one-norm weighting rules mimic the gradient direction of the one-norm, while retaining the convenient computational properties of r-square.

Equations 4.4.9 define a quadratic function $F_{(1)[k]}$ which captures the function value and gradient direction of $\emptyset_{(1)}$ at $x_{[k]}$ by setting

(8.3.2) $$w_{i[k]} = \frac{1}{|r_{i[k]}|}$$

in a weighted r-square norm. Thus from Equation 4.5.1 each nonzero residual element adds

$$w_{i[k]} \cdot r_{i[k]}^2 = \frac{1}{|r_{i[k]}|} \cdot r_{i[k]}^2 = |r_{i[k]}|$$

to $\emptyset_{(w)[k]}$; from Equation 4.1.4 this gives the same contribution as does $\emptyset_{(1)}$. Similarly, from Equations 4.3.6 and 4.5.4b, each nonzero residual element adds

$$2\nabla r_i \cdot w_{i[k]} \cdot r_{i[k]} = 2\nabla r_i \frac{r_{i[k]}}{|r_{i[k]}|} = 2\nabla r_i \cdot \text{sign}\{r_{i[k]}\}$$

to $g_{(w)[k]}$. But from Equation 4.4.1 this is twice the contribution expected for $g_{(1)[k]}$. Therefore with no zero residual element, Equation 8.3.2 defines a weighted r-square norm having the same function value and gradient direction as has $\emptyset_{(1)}$ at $x_{[k]}$.

Using a one-norm weighted $\emptyset_{(w)}$ in place of $\emptyset_{(1)}$ gives access to the $g_{(1)}$ search direction, while retaining the convenient computational properties of r-square. As defined by Equation 8.3.2, the norm still has difficulty at $r_i = 0$. However, unlike the one-norm, with $\emptyset_{(w)}$ the difficulty lies purely in assignment: once selected, the $w_{i[k]}$ unambiguously define a gradient descent direction, even if $r_{[k]}$ has zero elements.

The weighting rules described below differ in how they find $w_{i[k]}$ when an $r_{i[k]} = 0$.

Example: duct flow problem

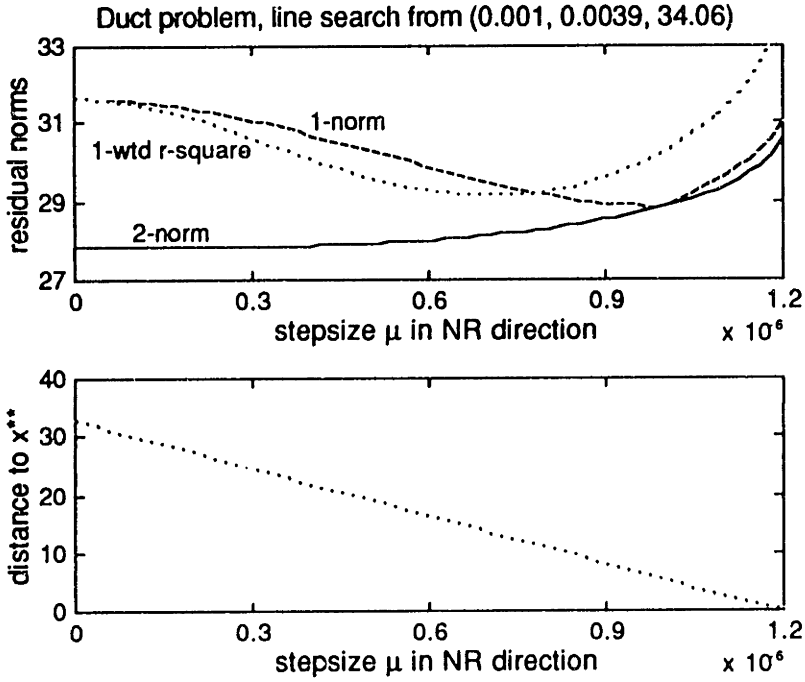

Figure 8.3.1. Line search in the Newton-Raphson direction for the duct flow problem from $(0.001, 0.0039, 34.06)^T$. *The upper plot shows* $\emptyset_{(1)}$ *(dashes),* $\sqrt{\emptyset_{(2)}}$ *(solid), and* $\emptyset_{(w)}$ *weighted according to Equation 8.3.2 (dots). The lower plot shows the distance to* x^{**} *for the same search.*

Consider the duct flow problem, starting from $x_{[0]} = (0.001, 0.0039, 34.06)^T$. At this point, $r_{[0]} = (27.53, -0.001790, 4.102)^T$, so Equation 8.3.2 defines a weighting matrix

$$W_{[0]} = \begin{bmatrix} 0.03633 & 0 & 0 \\ 0 & 558.6 & 0 \\ 0 & 0 & 0.2438 \end{bmatrix}$$

Figure 8.3.1 shows $r^T W_{[0]} r$, along with $\emptyset_{(1)}$ and $\sqrt{\emptyset_{(2)}}$, for a line search in the Newton-Raphson direction. Note the one-norm weighted r-square has the same function value as the one-norm only at the point used to find the weights; the function values differ at other points along the search, as would be expected even for linear residual functions.

One-norm rule 2

Weighting rule 2 chooses the residual weights using Equation 8.3.2, but subject to a maximum weight. This maximum weight corresponds to a small magnitude NEAR_ZERO below which a residual element effectively may be considered zero:

$$w_{i[k]\{2\}} = \begin{cases} |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[k]}|} \\ \text{else}: & \dfrac{1}{\text{NEAR\_ZERO}} \end{cases} \qquad (8.3.3)$$

The termination test of Equation 5.2.6 considers a residual element with magnitude less than CRIT_INF_NORM to be zero, so the weighting rule also treats these elements as zero. Thus it takes NEAR_ZERO = CRIT_INF_NORM, setting the maximum weight directly

from the user input tolerance on the largest residual magnitude which can be considered zero for termination purposes.

Figure 8.3.2 compares the residual weights for rules 1 and 2, showing $w_{i[k]}$ as a function of $r_{i[k]}$. The maximum weight in the figure corresponds to the default zero tolerance for a hypothetical machine with MACH_EPS = 10^{-4}. For such a machine, the default

$$\text{NEAR\_ZERO} = (\text{MACH\_EPS})^{1/3} \approx 0.05$$

and the maximum weight ≈ 20. A real computer, with its smaller fractional resolution of machine numbers, permits a larger maximum weight. For example, a 24 binary-digit floating-point machine has MACH_EPS ≈ 2^{-23} and a default maximum weight of about 200.

Note that the numeric experiments reported in Section 8.4 use only the default tolerance of the test machine (NEAR_ZERO ≈ $6 \cdot 10^{-6}$). None of the trial runs test the effect of changing the tolerance CRIT_INF_NORM.

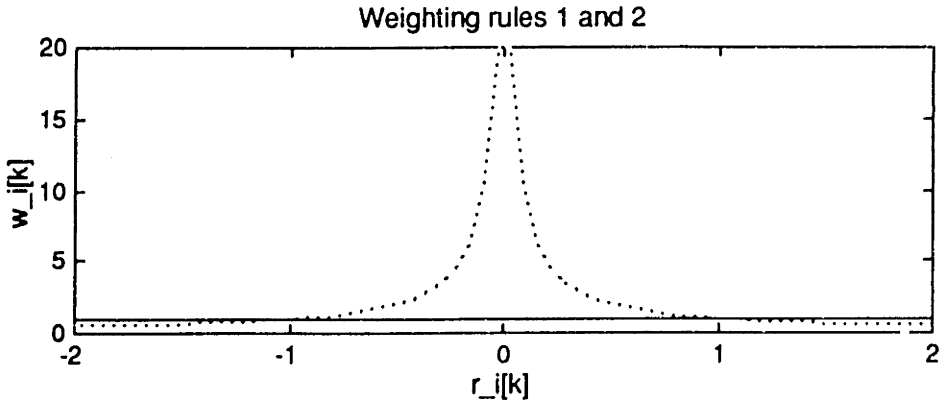

Figure 8.3.2. Residual weight $w_{i[k]}$ as a function of $r_{i[k]}$ for weighting rules 1 (r-square, solid) and 2 (one-norm weighted r-square, dots). The figure uses an artificially large value NEAR_ZERO = 0.05 for rule 2. Smaller values increase the height of the peak, and decrease the width of its plateau, but do not alter the rest of the curve.

One-norm rules 3 and 4

Weighting rules 3 and 4 copy rule 2 for nonzero residual elements, but propose alternate residual weights for elements with magnitude less than NEAR_ZERO. Thus in Figure 8.3.2 they change only the weights assigned in the interval where weighting rule 2 uses the maximum weight.

Rule 3 supposes that a zero, or nearly zero, residual comes from an equation which will not be difficult to zero again, because either: (1) the equation is linear, and so will present no difficulty once the algorithm begins taking full Newton-Raphson steps; or (2) a single step of modest length will not displace the trial solution very far, so that the next linearized model can capture the residual behavior well enough to zero it easily. Thus, rule 3 assumes that a nearly zero residual element represents a relatively tractable residual equation, and gives the residual no weight:

$$w_{i[k]\{3\}} = \begin{cases} |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[k]}|} \\ \text{else}: & 0 \end{cases} \tag{8.3.4}$$

Rule 4 follows similar logic, giving such an element a small but nonzero weight. It assigns the smallest weight found for the nonzero residuals at that iteration (there must be such a residual, or the method would terminate by Equation 5.2.6).

Discussion

In general, the one-norm weighting rules 2 through 4 all require the same number of Jacobian and residual evaluations to solve a given problem from a given starting point. The only notable exceptions-- the Powell badly scaled function from five times the standard starting point, and the trigonometric function with n = 50-- favor weighting rule 3.

Section 8.4 presents the results.

Since the one-norm weighting rules differ only in their treatment of zero residual elements at $x_{[k]}$, and since the rules run the gamut from assigning such elements the maximum weight, 1/NEAR_ZERO (rule 2), to assigning them the minimum weight, 0 (rule 3), apparently the exact treatment given the zero residual elements makes little difference to the overall performance of the algorithm. Detailed analysis of individual runs reveals that in large part this is because the algorithms rarely zero any residual element until the final iterations, when taking the full Newton-Raphson steps (i.e., until the linearized models approximate each residual extremely well).

One-norm rule 5

The one-norm weighted r-square rules described above calculate their residual weights using only the residual magnitudes at $x_{[k]}$. Rule 3, the most successful in the tests reported in Section 8.4, gives a zero residual element no weight, on the premise that such an element represents a linear equation, or an equation whose linearized model performs well in the neighborhood of the current iterate.

Weighting rule 5 attempts to determine whether a zero residual follows from a nearly linear residual equation, or merely by chance in the placement of the current iterate. It does so by placing the first trial step of each iteration in the Newton-Raphson direction, where a zero residual following from a linear residual equation would remain zero.

Specifically, if there are any nearly zero residuals at $x_{[k]}$, rule 5 defers calculating their weights until after taking its first trial point $x_{[k+1]'}$, which it places in the Newton-Raphson direction. Then for the zero residual elements, it takes the weight from an average residual magnitude at the initial point and at the first trial iterate.

If there are no zero residuals, rule 5 finds the residual weights using Equation 8.3.2, places its first step on the double dogleg curve, evaluates the step, and updates the trust region, all exactly as defined for a general weighted r-square double dogleg step.

If on the other hand some $r_{i[k]} <$ NEAR_ZERO, rule 5 puts the first step in the Newton-Raphson direction, using the appropriate trust length. It changes the trust length only in response to a residual error, when it halves the trust length and tries again, repeating until it finds a good point in the Newton-Raphson direction. Call this first iterate $x_{[k+1]'}$.

After establishing $x_{[k+1]'}$, rule 5 weights the zero residuals using approximately the average residual magnitude $(|r_{i[k]}| + |r_{i[k+1]'}|)/2$. Calling $r_{i[k]} \approx 0$ in these instances, the overall weighting rule chooses

$$(8.3.5) \qquad w_{i[k]\{5\}} = \begin{cases} |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[k]}|} \\ \dfrac{|r_{i[k+1]'}|}{2} \geq \text{NEAR\_ZERO}: & \dfrac{2}{|r_{i[k+1]'}|} \\ \text{else}: & 0 \end{cases}$$

When some $r_{i[k]} < \text{NEAR\_ZERO}$, rule 5 evaluates the step in the Newton-Raphson direction only after finding the weights according to Equation 8.3.5.

After finding the weights, rule 5 evaluates the step in the Newton-Raphson direction, by finding $\emptyset_{(w)}\{r_{[k+1]'}\}$. If the step does not meet the standard double dogleg acceptance criterion, and if the trust length is less than the cutback length, then the method does not reduce the trust length after this first trial point. Rather it places the next trial point on the cutback or steepest descent parts of the curve, using the unchanged trust length. If on the other hand this first Newton-Raphson direction step makes good progress, rule 5 doubles the trust length according to the usual double dogleg trust length update procedure. Thus if the increased trust length is less than the cutback length, rule 5 places the new trial point on the double dogleg curve, i.e., it does not keep to the Newton-Raphson direction after $x_{[k+1]'}$.

Discussion

Rule 5 repeats the performance of one-norm weighting rules 2 through 4, generally taking the same number of Jacobian and residual evaluations as the other one-norm weighted rules.

In the same two cases where rule 3 seems superior to the other one-norm weighting rules-- the Powell badly scaled function from five times the standard starting point, and the trigonometric function with n = 50-- rule 5 performs similarly well. However the greater compuational complexity of rule 5 makes it much less attractive than rule 3. Moreover, finding only two cases does not provide sufficient evidence that either rules 3 or 5 inherently outperform the other one-norm weighting rules.

One-norm rule 6

Rule 6 extends the notion from rule 5 that an initial step in the Newton-Raphson direction may indicate an appropriate residual magnitude on which to base a one-norm weighting selection. Instead of using an average magnitude only for the zero residuals, rule 6 weights every residual element by the average residual magnitude for every residual at every iteration:

$$(8.3.6) \qquad w_{i[k]\{6\}} = \begin{cases} \dfrac{|r_{i[k]}|+|r_{i[k+1]'}|}{2} \geq \text{NEAR\_ZERO}: & \dfrac{2}{|r_{i[k]}|+|r_{i[k+1]'}|} \\ \text{else}: & 0 \end{cases}$$

Thus it takes an initial step $x_{[k+1]'}$ in the Newton-Raphson direction at every iteration, even if no residual magnitude falls below NEAR_ZERO at $x_{[k]}$.

After taking this initial Newton-Raphson direction step, rule 6 updates the trust region as described for rule 5. That is, it finds the residual weights, evaluates the step, and applies the trust region updating rules to this initial step. If the step shows insufficient decrease in the cost function, but the step length is less than the cutback length, then instead of reducing the trust length the rule places the next trial iterate on the double dogleg curve at the same trust length. Moreover if the Newton-Raphson direction step

shows sufficient decrease in the cost function to justify increasing the trust length during the current iteration, the algorithm places the next trial step on the double dogleg curve, even if that means departing from the Newton-Raphson direction.

Discussion

The numeric experiments tabulated in Section 8.4 show rule 6 on average about as effective as the other one-norm weighting rules, though at times taking more residual evaluations. These excessive residual evaluations occur most notably for those problems which resist solution by any of the methods (for example the Powell badly scaled function from $(-10, -9.9)^T$, and the trigonometric function). The large residual evaluation counts arise when $x_{[k+1]'}$, the initial Newton-Raphson direction step, consistently fails its acceptance test. This forces the evaluation of a new trial point on the double dogleg curve, often at the same step length.

In general, rule 6 does not evaluate its initial Newton-Raphson direction choice $x_{[k+1]'}$ favorably, in the sense that while rule 6 may not reject $x_{[k+1]'}$ for insufficient decrease in $\emptyset_{(w)}$, only rarely does it attempt to double the trust length during the current iteration.

Examining the sequences generated by rule 6, the rule most often simply accepts $x_{[k+1]'}$ or rejects it for insufficient decrease. In fact it is far more common that the algorithm rejects $x_{[k+1]'}$, places a new point on the double dogleg curve at the same trust length, and evaluates this new point so favorably that it doubles the trust length, than that it simply doubles the trust length on strength of the evaluation of $x_{[k+1]'}$ in the first place.

R-square versus one-norm weighted r-square

The evaluation counts compiled in Section 8.4 show that for a given problem, usually either $\emptyset_{(2)}$ or a one-norm weighted $\emptyset_{(w)}$ better evaluates progress towards the solution for every starting point, but that over all the problems, neither norm does better than the other. Backing the contention of Section 8.2 that no single norm successfully indicates progress for all problems or search directions, note that on some occasions the one-norm weighted algorithms perform better (e.g. the trigonometric function), and sometimes worse (e.g. the Powell badly scaled function from the standard starting point).

From the figures of Section 8.4, in general:

(1) If an uninterrupted sequence of full Newton-Raphson steps solves a problem directly, then any weighting rule, r-square or one-norm, allows the sequence to proceed. See for example the Broyden tridiagonal function and the discrete boundary value function. The Powell badly scaled function is a key exception.

(2) If the double dogleg solves a problem using some combination of the Newton-Raphson and steepest descent directions, then so do the one-norm weighted r-square algorithms. See for example the duct flow problem, where the one-norm weighted algorithms require a comparable number of Jacobian and residual evaluations; and the Powell badly scaled function and the Rosenbrock function, where the one-norm weights perform sometimes better and sometimes worse than the r-square weights.

(3) If the r-square weights and one-norm weights require dramatically different evaluation counts to solve a problem, the results tend to favor r-square. See for example the Powell badly scaled function and the Rosenbrock function.

(4) When the r-square weights dramatically outperform the one-norm weights, the best one-norm weights tend to come from rule 6. Examining the sequences each rule generates, the difficulty in these cases lies with the inability of the one-norm weights to

exploit the Newton-Raphson direction. Rule 6, which takes its $w_{i[k]}$ in part from a trial point in the Newton-Raphson direction, evaluates the Newton-Raphson step most favorably of the one-norm weights, but not as successfuly as does r-square.

(5) The one-norm weighted r-square resists stagnation better than does the standard, r-square weighted, double dogleg. See for example the Powell badly scaled function and the trigonometric function.

Mixing r-square and the one-norm

In the results so far, typically the double dogleg algorithm can solve any of the sample equilibrium problems using either $\emptyset_{(2)}$ or a one-norm weighted $\emptyset_{(w)}$, or both. (The notable exceptions are the Powell badly scaled function from $(-10, -9.9)^T$, and the trigonometric function generally.)

The discussion of Equation 8.2.1 suggests that the difference between $\emptyset_{(2)}$ and $\emptyset_{(1)}$ resides, not primarily in their ability to evaluate a step, but in the way they combine the residual gradients to form an overall steepest descent direction

In that analysis, $\emptyset_{(2)}$ succeeds where a residual's gradient direction remains valid over long enough distances to credit it with the possibility of zeroing the residual. This possibility makes it appropriate to account for the residual magnitude when summing the residual gradients. Moreover, $\emptyset_{(1)}$-- or more accurately a one-norm weighted r-square norm which reproduces the gradient direction of $\emptyset_{(1)}$-- succeeds where the residual gradients change rapidly enough with Δx that a residual's magnitude has little to do with the relative importance of travel in its gradient direction.

On the strength of this analysis, the weighting rules developed below attempt to mix the $\emptyset_{(2)}$ and $\emptyset_{(1)}$ gradient choices. If in fact the significant difference between the norms lies in the alternate search directions represented by their gradients, rather than in their ability to evaluate a particular step, then setting the gradient weights p_i by some combination of the $\emptyset_{(2)}$ and $\emptyset_{(1)}$ rules may combine the strengths of both methods.

Mix rules 7 and 8

Since $\emptyset_{(2)}$ sets $p_i = 2 \cdot r_i$, and $\emptyset_{(1)}$ sets $p_i = 2 \cdot \text{sign}\{r_i\}$, one way to mix the two rules takes $p_i = 2 \cdot \sqrt{|r_i|} \cdot \text{sign}\{r_i\}$. At $|r_i| = 1$, where the two-norm and one-norm rules both give $p_i = \pm 1$, the square root gives the same value. At larger residual magnitudes, the square root weights the residual gradient less heavily than does $\emptyset_{(2)}$, while for smaller residual manitudes, it assigns a larger gradient weight. In both cases it brings p_i closer to the value used by the one-norm.

From Equation 4.5.2a, to set $p_i = 2 \cdot \sqrt{|r_i|} \cdot \text{sign}\{r_i\}$ requires $w_i = 1/\sqrt{|r_i|}$ in the weighted r-square norm. As with the one-norm weighting rules, the difficulty at $r_i = 0$ invites a number of alternate assignments. Following rules 2 and 3, let

$$w_{i[k]\{7\}} = \begin{cases} |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{\sqrt{|r_{i[k]}|}} \\ \text{else}: & \dfrac{1}{\sqrt{\text{NEAR\_ZERO}}} \end{cases} \tag{8.3.7}$$

and

$$w_{i[k]\{8\}} = \begin{cases} |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{\sqrt{|r_{i[k]}|}} \\ \text{else}: & 0 \end{cases} \tag{8.3.8}$$

Discussion

The square root weighting rules 7 and 8 appear to strike a balance between the r-square and the one-norm weighted r-square norms. Somewhat less prone to excessive evaluation counts than the one-norm weighting rules, still the square root weighting rules stagnate on the same problems that cause r-square difficulty (in the only striking exception, the 5-dimensional trigonometric function from five times its standard starting point, the planar hook also avoids stagnation, suggesting that the exception arises from some circumstance particular to this starting point).

However, the rules do not improve on r-square, since they stagnate on the same problems, and generally require more Jacobian and residual evaluations on those problems they do solve. Finally when the square root rules outperform the one-norm weights, so does r-square. In sum, there is no reason to prefer either of rules 7 or 8 to rule 1 (or to DBLDOGLG.CPP).

Mixing rules need step length estimates

The square root weighting rules 7 and 8 attempt to mix the gradient choices of $\emptyset_{(2)}$ and $\emptyset_{(1)}$ by averaging their gradient weights $p_{(2)}$ and $p_{(1)}$. Specifically, they take the geometric average, since for example

$$w_{i[k]\{7\}} = \sqrt{w_{i[k]\{1\}} \cdot w_{i[k]\{2\}}}$$

This strategy captures the merits of neither $g_{(2)}$ nor $g_{(1)}$, because it does not diagnose the conditions under which those gradients seem to work best. The square root rules do not so much choose the appropriate search direction as split the difference between them.

In the analysis of Equation 8.2.1, the choice between $g_{(2)}$ and $g_{(1)}$ depends on the expected performance of the linearized model. If the model performs well over the step lengths needed to zero the residuals, then the gradient weights should account for the residual magnitude (the $\emptyset_{(2)}$ rule); otherwise, the residual magnitude does not matter (the $\emptyset_{(1)}$ rule). Weighting rules 7 and 8 mix the strategies by accounting for the magnitude only partially when the $\emptyset_{(2)}$ rule should apply, and allowing it to count to some extent when the $\emptyset_{(1)}$ rule should apply.

Suppose instead that a weighting rule adopts by default the one-norm weights of Equation 8.3.2. However for zero residuals, and for residuals which can be expected to be zeroed by a step of reasonable length, suppose the rule uses the weights appropriate for $\emptyset_{(2)}$. Such a rule mixes the weighting strategies on a residual-by-residual basis.

Such a rule requires estimates of: (1) the step length required to zero each linearized residual; and (2) the step length over which the linearized residual models remain valid.

In the trust region methods, the trust length makes a natural metric for estimating the distance over which the linearized models remain valid. To estimate the step length required to zero a residual requires an estimate of the slope in the linearized residual equation.

Slope in linearized residual

Consider a one-dimensional problem, for example $r = \arctan\{x\}$ from Equation 2.2.5a. From Equation 2.2.5b, at $x_{[k]}$ the slope $(\partial r/\partial x)|_{[k]} = (dr/dx)|_{[k]} = (1 + x_{[k]}^2)^{-1}$. Following the gradient direction, the linearized model reaches zero after a distance

$$\|\Delta x\| \simeq \left| \frac{r_{[k]}}{(dr/dx)|_{[k]}} \right|$$

Figure 2.2.1 shows the arctangent curve and its linearized model at $x_{[k]} = 1.2$, and suggests how the slope of the gradient as well as the magnitude of the residual at the point of linearization affects the expected step length.

In multiple dimensions, the slope in each linearized residual function depends on the search direction. From Equation 3.2.2, for a search direction $d_{[k]}$,

$$\frac{dr_{i[k]}}{d\mu} = -\nabla r_{i[k]}{}^T d_{[k]}$$

where μ determines the extent of the search in Equation 3.1.3, $x = x_{[k]} - \mu \cdot d_{[k]}$. During the search,

$$\|x - x_{[k]}\| = |\mu| \cdot \|d_{[k]}\|$$

$$d\|\Delta x\| = \|d_{[k]}\| \cdot |d\mu|$$

so the slope in a search direction $d_{[k]}$ has magnitude

$$\left| \frac{dr_{i[k]}}{d\|\Delta x\|} \right| = \frac{|\nabla r_{i[k]}{}^T d_{[k]}|}{\|d_{[k]}\|} = \left| \nabla r_{i[k]}{}^T \left(\frac{d_{[k]}}{\|d_{[k]}\|} \right) \right| \tag{8.3.9}$$

The derivative $dr_{i[k]}/d\|\Delta x\|$ reads more easily as the change in r_i for a change in the step length, than as the change in r_i for a change in the length of the change in x.

Since $d_{[k]}/\|d_{[k]}\|$ has length one, Equation 8.3.9 makes explicit the caution given in the discussion of Equation 3.2.2, that to compare slopes in different search directions requires normalizing the search direction lengths.

Following the one-dimensional example, the linearized model predicts that to zero $r_{i[k]}$ requires a step of length

$$\|\Delta x\|_{i[k]} = \left| \frac{r_{i[k]}}{\left(\frac{dr_{i[k]}}{d\|\Delta x\|} \right)} \right| = \left| \frac{r_{i[k]}}{\nabla r_{i[k]}{}^T \left(\frac{d_{[k]}}{\|d_{[k]}\|} \right)} \right| \tag{8.3.10}$$

Slope and gradient weights

Return to the notion that, for relatively tractable residuals, the gradient weights p_i should reflect the residual magnitude, since a larger residual requires greater travel in its gradient direction. Within this framework, consider two equal residuals $r_1 = r_2$, whose gradients differ greatly in length. Say $\|\nabla r_1\| >> \|\nabla r_2\|$. Summing the ∇r_i by Equation 8.2.1, $d = 2r_1 \cdot (\nabla r_1 + \nabla r_2)$. Clearly ∇r_1 dominates the net search direction, contrary to the stated intention of choosing the p_i to reflect the distance of travel anticipated to be required in a given direction.

In the example, the larger $\|\nabla r_1\|$ creates relatively more change in r_1 for a fixed change $\|\Delta x\|$ in its steepest descent direction. Thus choosing $p_i = r_i$ does not weight ∇r_i by the expected distance required to zero r_i; from Equation 8.3.10 the rule should set

(8.3.11) $$p_{i[k]} = \frac{r_{i[k]}}{\left| \nabla r_{i[k]}^T \left(\frac{d_{[k]}}{\|d_{[k]}\|} \right) \right|}$$

Note Equation 8.3.11 modifies Equation 8.3.10 to give $p_{i[k]}$ the same sign as has $r_{i[k]}$.

Now for a steepest descent direction search, $d = g = \sum p_i \cdot \nabla r_i$, making it difficult to use Equation 8.3.11 to identify self-consistent gradient weights: to set the $p_{i[k]}$ accounting for the slopes requires knowing the search direction $d_{[k]}$, which itself depends on the $p_{i[k]}$.

Normalized gradient rules 9 to 11

In lieu of attempting to find self-consistent gradient weights, set $d_{[k]} = \nabla r_{i[k]}$ in Equation 8.3.11. Then the denominator has $|\nabla r_{i[k]}^T \nabla r_{i[k]}| / \|\nabla r_{i[k]}\| = \|\nabla r_{i[k]}\|$, so

$$p_{i[k]} = \frac{r_{i[k]}}{\|\nabla r_{i[k]}\|}$$

Intuitively, this gradient weighting rule constructs the overall gradient of the residual norm by summing the residual gradients: (1) in proportion to r_i, which indicates the need to make progress in that residual; and (2) in inverse proportion to the length of the residual gradient, which indicates the ability of a change in x to change that residual.

To achieve this with the weighted r-square norm requires setting $w_{i[k]} = (2 \cdot \|\nabla r_{i[k]}\|)^{-1}$. Dropping the scale factor two, since it affects all the weights equally, weighting rule 9 sets

(8.3.12) $$w_{i[k]\{9\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ \text{else:} & \dfrac{1}{\|\nabla r_{i[k]}\|} \end{cases}$$

where the safeguard against division by zero assigns a zero residual weight when a residual has a stationary point at $x_{[k]}$.

Equation 8.3.12 defines a version of $\emptyset_{(2)}$ adjusted for the gradient lengths, i.e., one which accounts for the lengths of the Jacobian rows (recall $\emptyset_{(2)}$ sets $w_{i[k]} = 1$).

Note that in the double dogleg algorithm, the test $\|\nabla r_{i[k]}\| = 0$ in Equation 8.3.12 avoids an evaluation error only. Since this condition implies $\nabla r_{i[k]} = 0$, it indicates a singular Jacobian, and signals a catastrophic failure beyond the scope of recovery that any assignment to $w_{i[k]}$ may provide. A more robust algorithm could simply abandon the Newton-Raphson direction altogether, searching only in the gradient direction.

Finally note that the weighted r-square norm $\emptyset_{(w)\{9\}}$ of Equation 8.3.12 has a gradient

$$g_{(w)\{9\}} = 2 \sum_{i=1}^{n} r_i \frac{\nabla r_i}{\|\nabla r_i\|}$$

Therefore an alternate interpretation of rule 9 has it summing the normalized residual gradients to find its net gradient, rather than summing the residual gradients with the p_i adjusted to normalize the residual gradients.

This alternate interpretation suggests similar variations on the one-norm weighting rules, variations in which the gradient weights multiply normalized residual gradients. Thus, weighting rules 10 and 11 copy one-norm weighting rules 2 and 3, except that they adjust for the residual gradient magnitudes:

$$(8.3.13)\qquad w_{i[k]\{10\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[k]}| \cdot \|\nabla r_{i[k]}\|} \\ \text{else}: & \dfrac{1}{\text{NEAR\_ZERO} \cdot \|\nabla r_{i[k]}\|} \end{cases}$$

and

$$(8.3.14)\qquad w_{i[k]\{11\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[k]}| \cdot \|\nabla r_{i[k]}\|} \\ \text{else}: & 0 \end{cases}$$

Discussion

Comparing rule 9 to rule 1, that is, comparing the normalized gradient r-square to r-square: (1) rule 9 solves every problem that rule 1 solves; (2) rule 9 usually solves those problems in fewer Jacobian and fewer residual evaluations than rule 1, with the duct flow problem providing the only exceptions; and (3) rule 9 solves some problems that rule 1 does not, for example the Powell badly scaled function from $(-10, -9.9)^T$ and the 10-dimensional trigonometric function from its standard starting point.

Rule 9 performs best relative to rule 1 in the Powell badly scaled function and the Rosenbrock function, where it takes typically half the number of evaluations of the standard double dogleg algorithm. In particular, for two starting points on the Powell badly scaled function, normalizing the residual gradients allows the double dogleg to converge with a complete sequence of full Newton-Raphson steps.

Comparing rules 10 and 11 to the other one-norm weighting rules, normalizing the residual gradients as part of the one-norm weighting rules occasionally improves the method-- for example on the Powell badly scaled and Rosenbrock functions-- and occasionally slows the method down-- for example on the trigonometric function and on the duct flow problem.

In general, taking all the rules together, if both r-square and one-norm weighted r-square methods solve a problem, then rule 9 with its normalized gradient r-square weights performs best, and if only one-norm weighting rules solve a problem, it is best not to normalize the gradient lengths, and to use e.g. weighting rule 3.

Mix rule 12

Based on the results from the normalized gradient rules, and cumulative experience with the r-square and one-norm weights generally, consider a rule which: (1) identifies, based on the trust length and the step length required to zero the linearized model of each residual, whether to weight that residual by a $\emptyset_{(2)}$ rule or by a $\emptyset_{(1)}$ rule; (2) uses the normalized gradient weighting rule 9 for r-square weights; and (3) uses Equation 8.3.2 for one-norm weights.

Note it is not necessary to treat the problem of a zero residual when using the one-norm weighting rule, since a zero residual clearly should receive the normalized gradient r-square weighting of rule 9.

The distinguishing test compares the trust length, $\delta_{[k]}$, to the step length needed to zero the linearized model of $r_{i[k]}$, given by Equation 8.3.10 with $d_{[k]} = \nabla r_{i[k]}$. If $\delta_{[k]} > \|\Delta x\|_{i[k]}$, that is, if

(8.3.15) $$\delta_{[k]} > \frac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}$$

then after setting the weights the algorithm will take a large step compared to that needed to zero $\hat{r}_{i[k]}$. In this case the residual weight should reflect the residual magnitude. Thus,

(8.3.16) $$w_{i[k]\{12\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{1}{\|\nabla r_{i[k]}\|} \\ \text{else:} & \dfrac{1}{|r_{i[k]}|} \end{cases}$$

Rule 12 uses the normalized gradient r-square rule for every residual in the first iteration, k=0, because the trust length, arbitrarily initialized either to the Cauchy length (in the literature) or to the Newton-Raphson length (in the algorithm used here), does not meaningfully measure the expected step length until after the first step. Furthermore, setting the $w_{i[0]}$ by a special rule at the first iteration allows the program to choose the weights before finding the Newton-Raphson step, thus avoiding the less efficient multiplications by a factored Jacobian matrix when finding J^TW and JJ^TW in the expressions of Section 4.5.

Discussion

Rule 12 solves every problem except the 50-dimensional trigonometric function. Furthermore its iteration counts are comparable to, and sometimes better than, those of every other method, except on the duct flow problem from $(60, 60, 60)^T$.

The rule appears to mix the normalized gradient r-square and one-norm weighted r-square rules as intended. For example, in the Powell badly scaled function it turns in the same iteration counts as weighting rule 9, the $\emptyset_{(2)}$ rule with normalized gradients; while for the 5-dimensional trigonometric function it avoids the stagnation problems which beset rule 9.

Turning to the two cases on which rule 12 does not do well:

(1) The rule takes an excessive number of Jacobian evaluations compared to the other rules, and an excessive number of residual evaluations for each Jacobian evaluation, on the duct flow problem from $(60, 60, 60)^T$. The difficulty lies with repeated residual evaluation errors. The method encounters a surface beyond which the first residual equation gives an error, and the method follows this surface, at one iteration increasing the trust length as it parallels the surface, then at the next iteration halving the trust length repeatedly after turning to cross the surface. Apparently this behavior depends less on the weighting rule than on an unfortunate choice of starting points. Note that on the duct flow problem from $(90, 90, 90)^T$, rule 12 shows the best performance of any rule so far.

(2) The rule cannot solve the trigonometric problem with n=50 in the allowed number of Jacobian iterations. The difficulty appears to lie squarely with the weighting rule. Over a long sequence of iterations, the residual weights for certain elements alternate between ≈10 and ≈500, changing at each iteration. By contrast, in the better-behaved solutions, and in most of the elements for the n=50 trigonometric solution, rule 12 settles on a consistent order of magnitude for each weight fairly quickly. Though it allows steady trends, for example where a weight increases at each iteration, in the successful cases it

rarely undoes a large jump in a residual weight at one iteration with another jump, in the opposite direction, at the next iteration.

Tracing the program execution step by step, the alternating large then small residual weights occur for residual elements which satisfy Equation 8.3.15 nearly as an equality. At successive iterations, these elements alternate between the normalized gradient r-square rule, $w_{i[k]} = 1/\|\nabla r_{i[k]}\|$, and the one-norm weighted r-square rule, $w_{i[k]} = 1/|r_{i[k]}|$, with the one-norm part of the rule giving the large $w_{i[k]}$.

Though this behavior suggests that the difficulty may lie with switching between the gradient normalization rules, in fact the majority of the weighting rules also fail to solve the 50-dimensional trigonometric function. This implies that the explanation for the varying residual weights observed for rule 12 in this case lies, not with the weighting rule's mixing strategy, but with the inherent difficulty of the trigonometric function.

Mix rules 13 to 16

Weighting rules 13 to 16 assume that the difficulty with the 50-dimensional trigonometric function in rule 12 comes from the discontinuity between the $w_{i[k]\{12\}}$ generated for trust lengths larger than that needed to zero the linearized model, and those generated for smaller trust lengths.

Weighting rules 13 and 14 partially address this discontinuity by adopting the same gradient normalization practice towards residuals of both large and small relative magnitude. However, neither provides a continuous selection of $w_{i[k]}$ across the break point where $\delta_{[k]} = |r_{i[k]}|/\|\nabla r_{i[k]}\|$.

From the results of rules 9 to 12, note: (1) gradient normalization appears on average to have a greater positive effect on r-square rule 9 than its negative effect on rules 10 and 11; and (2) where gradient normalization has a strong negative effect on rules 10 and 11, rule 12 tends to follow rule 9 anyway. Therefore rule 13 chooses $w_{i[k]}$ to normalize the gradients in both cases:

(8.3.17)
$$w_{i[k]\{13\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{1}{\|\nabla r_{i[k]}\|} \\ \text{else:} & \dfrac{1}{|r_{i[k]}| \cdot \|\nabla r_{i[k]}\|} \end{cases}$$

Rule 14, on the other hand, does not normalize the gradients:

(8.3.18)
$$w_{i[k]\{14\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & 1 \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & 1 \\ \text{else:} & \dfrac{1}{|r_{i[k]}|} \end{cases}$$

Note that neither rules 13 nor 14 define $w_{i[k]}$ continuously across the break point in rule 12, since neither involves the trust length. Rules 15 and 16 force continuity: rule 15 by bridging the gap between the normalized gradient $\emptyset_{(2)}$ weights and the one-norm weights; rule 16 by redefining the one-norm weights.

Consider the fraction

(8.3.19a) $$f_{i[k]} = \exp\left(1 - \frac{|r_{i[k]}|}{\delta_{[k]} \cdot \|\nabla r_{i[k]}\|}\right)$$

At the break point $\delta_{[k]} = |r_{i[k]}|/\|\nabla r_{i[k]}\|$ in rule 12, $f_{i[k]} = 1$. For smaller trust lengths, $f_{i[k]}$ diminishes rapidly. For example, when $\delta_{[k]}$ is half the step length needed to zero $\hat{r}_{i[k]}$, $f_{i[k]} = \varepsilon^{-2} \approx 0.38$, and when $\delta_{[k]}$ is a quarter the required length, $f_{i[k]} \approx 0.05$.

If $\delta_{[k]} > |r_{i[k]}|/\|\nabla r_{i[k]}\|$, rule 15 follows rule 12, using the normalized gradient $\emptyset_{(2)}$ rule, since the expected step length exceeds that needed to zero the linearized model. For smaller trust lengths, rule 15 mixes the normalized r-square rule with the one-norm rule in the ratio $f_{i[k]}:(1-f_{i[k]})$. Thus for very small trust lengths, rule 15 gives $w_{i[k]}$ very nearly the same as that of the original one-norm rule:

(8.3.19b) $$w_{i[k]\{15\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{1}{\|\nabla r_{i[k]}\|} \\ \text{else:} & \dfrac{f_{i[k]}}{\|\nabla r_{i[k]}\|} + \dfrac{1-f_{i[k]}}{|r_{i[k]}|} \end{cases}$$

Where rule 15 bridges the discontinuity between the previously defined residual weights $1/\|\nabla r_{i[k]}\|$ and $1/|r_{i[k]}|$, rule 16 scales the one-norm weights by $\delta_{[k]}$:

(8.3.20) $$w_{i[k]\{16\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{1}{\|\nabla r_{i[k]}\|} \\ \text{else:} & \dfrac{\delta_{[k]}}{|r_{i[k]}|} \end{cases}$$

This makes $w_{i[k]\{16\}} = 1/\|\nabla r_{i[k]}\|$ on either side of the break point.

Unlike rule 15, rule 16 admits a geometric interpretation. First note that scaling any of the one-norm weighting rules by $\delta_{[k]}$ merely scales the weighting matrix $W_{[k]}$, without changing the relative magnitudes of its diagonal elements. Second, recall the geometric motivation behind Equation 8.3.11, which weights each residual gradient by the distance of travel needed to zero the corresponding linearized residual model. Now suppose that distance exceeds the trust length. Then choosing $p_{i[k]} = \delta_{[k]} \cdot \text{sign}\{r_{i[k]}\}$ weights the residual gradient by the distance the step is expected to take. Converting to $w_{i[k]}$ by Equation 4.5.2a, and dropping the factor 2 as was done for rule 9 in Equation 8.3.12, the resulting $w_{i[k]} = \delta_{[k]}/|r_{i[k]}|$, as shown in Equation 8.3.20.

Rule 16 weights each residual gradient either by the distance needed to zero the linearized residual model, or by the trust length, whichever is less.

Discussion

As expected, rule 13 betters rule 14, since it adopts the overall more successful gradient normalizing strategy. While rule 13 does improve on rule 12 in the duct flow problem from $(60, 60, 60)^T$, it does not avoid the high iteration count on the 50-dimensional trigonometric function. In addition, it exceeds the allowed iteration count on another trigonometric function starting point, one which rule 12 solves directly.

Rule 15 shows no significant difference from rule 12. Still unable to solve the 50-dimensional trigonometric function within the maximum number of iterations, rule 15 slightly affects the iteration counts on those problems which both solve. In some cases, rule 15 solves these problems with fewer iteration counts. However the small differences between the rules do not justify the added complexity of rule 15.

Nor does rule 16 improve on rule 12. Again, it performs about the same on all the problems, and still exceeds the maximum iteration count on the 50-dimensional trigonometric function. Additionally, rule 16 exceeds its allowed iterations on another of the trigonometric function starting points, and on two of the duct flow problem starting points.

Overall, rules 13 to 16 fail to improve on rule 12. Apparently the attempt to select $w_{i[k]}$ continuously across the break point $\delta_{[k]} = |r_{i[k]}|/\|\nabla r_{i[k]}\|$ introduces new convergence difficulties.

Persistence of norm

To take a more symptomatic approach to the changing residual weights in the rule 12 solution of the 50-dimensional trigonometric function, recall the Section 8.1 discussion of the possible difficulty with changing the norm from iteration to iteration. Giving up an absolute convergence criterion, an algorithm may select from among a range of norms to prevent divergence in a single step, yet may fail to converge if a step substantially undoes the progress made by those preceding it.

Since the weighted double dogleg algorithm chooses a new set of weights at each iteration, it may fail to converge even though it controls each step according to some selection of $W_{[k]}$ in $\emptyset_{(w)}$. Thus, in its successful applications rule 12 establishes approximate magnitudes for the residual weights fairly early on. While it changes the weights from iteration to iteration, in these cases either it changes each w_i in a consistent direction, increasing or decreasing it each time, or it changes the w_i by relatively small amounts. The consistent, or substantially consistent, residual weights create something resembling an absolute convergence criterion.

From this perspective, the continuity of $w_{i[k]}$, say as a function of $r_{i[k]}$, matters less than the continuity of w_i from iteration to iteration. In this respect, weighting rule 12 is not unique. Though other rules choose their residual weights less robustly across all the problems and starting points, in general a successful application of any rule finds it choosing weights more consistently than not. This holds even for those rules which define $w_{i[k]}$ as a continuous function of $r_{i[k]}$.

These observations suggest that the high iteration counts reflect the lack of an absolute convergence criterion. Set aside the question whether maintaining a consistent magnitude of the w_i causes convergence by enforcing an absolute performance criterion, or reflects the appropriateness of the given weighting rule to the problem at hand. Clearly a rule which generates inconsistent or wildly varying w_i from iteration to iteration risks slow convergence, and hence high iteration counts. Conversely, when a weighting rule generates w_i of consistent magnitude, it often converges without difficulty. Therefore a weighting rule which prevents its w_i from changing too rapidly may be less prone to high iteration counts, although it risks stagnating due to convergence to a stationary point $x^* \neq x^{**}$.

Significantly, with all the variable weighting rules considered so far, only rarely does the method stagnate. Typically if the method fails, it runs through its maximum allowed number of iterations without stagnating. Merely changing the residual weights slightly

from iteration to iteration seems to provide some measure of protection against converging to a point which is not a solution.

The weighting rules developed below attempt to achieve a persistent norm, that is, one which does not change significantly from one iteration to the next, though it may change significantly over the course of many iterations.

Persistent rules 17 to 22

Weighting rules 17 to 22 estimate weights using one of the rules already discussed above, then average them, element by element, with the weights used for the previous iteration. This averaging reduces large changes and smooths out small fluctuations in the w_i, while allowing broad trends from the original weighting rule to affect the weights actually used.

The rules use both the arithmetic and geometric means. The geometric mean favors smaller residual weights, since $\sqrt{a \cdot b} \leq (a+b)/2$. Therefore when a residual weight changes dramatically from one iteration to the next, the geometric mean weights it less heavily than does the arithmetic mean. On the other hand, some machines may calculate the arithmetic mean more quickly.

Averaging the residual weights from iteration to iteration to create a persistent norm from the one-norm weighting rules, let

$$w_{i[k]\{17\}} = \begin{cases} k = 0 \text{ and } |r_{i[0]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[0]}|} \\ k = 0 \text{ and } |r_{i[0]}| < \text{NEAR\_ZERO}: & 0 \\ |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \dfrac{w_{i[k-1]} + \dfrac{1}{|r_{i[k]}|}}{2} \\ \text{else}: & \dfrac{w_{i[k-1]}}{2} \end{cases} \tag{8.3.21}$$

for arithmetic averaging, and

$$w_{i[k]\{18\}} = \begin{cases} k = 0 \text{ and } |r_{i[0]}| \geq \text{NEAR\_ZERO}: & \dfrac{1}{|r_{i[0]}|} \\ k = 0 \text{ and } |r_{i[0]}| < \text{NEAR\_ZERO}: & 1 \\ |r_{i[k]}| \geq \text{NEAR\_ZERO}: & \sqrt{\dfrac{w_{i[k-1]}}{|r_{i[k]}|}} \\ \text{else}: & \sqrt{w_{i[k-1]}} \end{cases} \tag{8.3.22}$$

for the geometric mean. Note the treatment given to a zero residual at the first iteration gives a residual weight corresponding to repeated application of the treatment accorded a zero residual in later iterations: in Equation 8.3.21, repeatedly halving a weight reduces it to zero; in Equation 8.3.22, repeatedly taking the square root of a weight takes it to one.

Averaging to create a persistent norm from the normalized gradient r-square rule,

$$w_{i[k]\{19\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \text{else}: & \dfrac{w_{i[k-1]} + \dfrac{1}{\|\nabla r_{i[k]}\|}}{2} \end{cases} \tag{8.3.23}$$

and

$$(8.3.24)\qquad w_{i[k]\{20\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \text{else:} & \sqrt{\dfrac{w_{i[k-1]}}{\|\nabla r_{i[k]}\|}} \end{cases}$$

Note that since $\|\nabla r_{i[k]}\| = 0$ signals a singular $J_{[k]}$, rules 19 and 20 do not use $w_{i[k-1]}$ to define $w_{i[k]}$ in this case. An algorithm which, unlike the weighted dogleg considered here, reacts to a singular Jacobian by redefining its search path, could e.g. use $w_{i[k]} = w_{i[k-1]}$.

Finally, averaging to create a persistent norm from the mix weighting rules,

$$(8.3.25)\qquad w_{i[k]\{21\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{w_{i[k-1]} + \dfrac{1}{\|\nabla r_{i[k]}\|}}{2} \\ \text{else:} & \dfrac{w_{i[k-1]} + \dfrac{1}{|r_{i[k]}|}}{2} \end{cases}$$

for the arithmetic mean, and

$$(8.3.26)\qquad w_{i[k]\{22\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{|r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \sqrt{\dfrac{w_{i[k-1]}}{\|\nabla r_{i[k]}\|}} \\ \text{else:} & \sqrt{\dfrac{w_{i[k-1]}}{|r_{i[k]}|}} \end{cases}$$

for the geometric mean.

Discussion

First, compare the arithmetic and the geometric means. While the experimental results do not indicate a strong preference, the more noticeable cases tend to favor the geometric mean, possibly because in a given situation it chooses residual weights of smaller magnitude than does the arithmetic average.

Comparing the one-norm weighting rules, i.e. rules 17 and 18 against rules 2 to 4, averaging to create a persistent norm either improves the rule noticeably (e.g. the Powell badly scaled function and the Rosenbrock function from their standard starting points), or has only a minor effect on the iteration count.

Averaging has a less pronounced effect on the normalized gradient r-square rule. Comparing rules 19 and 20 to rule 9, averaging the w_i from iteration to iteration often decreases the evaluation counts, but never as dramatically as for the one-norm rules.

Finally, comparing the mix weighting rule 12 to its averaged counterparts in rules 21 and 22, in every case but one-- the duct flow problem from $(90, 90, 90)^T$-- averaging either

improves the evaluation counts, or does not affect them at all. However, averaging does not solve the problem of excessive evaluations in the 50-dimensional trigonometric function.

Mix weighting rules revisited

In the results so far, the mix weighting rules tend not so much to improve the convergence possible on a particular problem, as to allow convergence where an unmixed weighting rule failed to converge. Thus, the mix weighting rules achieve success by identifying which of the r-square, normalized gradient r-square, or one-norm weighted r-square rules to apply to a given residual at a given iteration.

Attempting to improve on rule 12, subsequent mix weighting rules: (1) try using different Jacobian row normalization treatments on either side of the break point $\delta_{[k]} = |r_{i[k]}|/\|\nabla r_{i[k]}\|$ (rules 13 and 14); (2) try matching the weights assigned on either side of the break point, to achieve a continuous definition of $w_{i[k]}$ (rules 15 and 16); and (3) try adding a memory of the weight used at the last iteration, to achieve a persistent norm (rules 21 and 22).

None of these new mix weighting rules alter the last fundamental parameter which defines rule 12: the location of the break point where the r-square or normalized gradient r-square rule gives way to the one-norm weighted r-square rule.

Mix rules 23 and 24

Rules 23 and 24 try moving the break point, by altering the test expression of Equation 8.3.15. Since the mix weighting rules presented above tend to fail on the trigonometric function on starting points for which the pure one-norm weighted rules succeed, the new break point places a more stringent requirement on the trust length before it uses the normalized gradient r-square weights. Thus, if $\delta_{[k]} < 2 \cdot \|\Delta x\|_{i[k]}$, rules 23 and 24 use the one-norm style residual weights.

Because this new break point heightens the discontinuity in $w_{i[k]}$, increasing the likelihood that w_i may shift dramatically from one iteration to the next, rules 23 and 24 also average $w_{i[k-1]}$ into the current residual weight selection:

$$w_{i[k]\{23\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{2 \cdot |r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \dfrac{w_{i[k-1]} + \dfrac{1}{\|\nabla r_{i[k]}\|}}{2} \\ \text{else:} & \dfrac{w_{i[k-1]} + \dfrac{1}{|r_{i[k]}|}}{2} \end{cases} \tag{8.3.27}$$

for the arithmetic mean, and

$$w_{i[k]\{24\}} = \begin{cases} \|\nabla r_{i[k]}\| = 0: & 0 \\ k = 0: & \dfrac{1}{\|\nabla r_{i[0]}\|} \\ \delta_{[k]} > \dfrac{2 \cdot |r_{i[k]}|}{\|\nabla r_{i[k]}\|}: & \sqrt{\dfrac{w_{i[k-1]}}{\|\nabla r_{i[k]}\|}} \\ \text{else:} & \sqrt{\dfrac{w_{i[k-1]}}{|r_{i[k]}|}} \end{cases} \tag{8.3.28}$$

Discussion

In the trigonometric function, rules 23 and 24 do not show the same consistency between results for the arithmetic and geometric average, as do the pairs of rules 17 and 18, 19 and 20, and 21 and 22. Where rule 23 fares significantly worse than does rule 12, taking the geometric mean with the tighter break point requirement on the trust length solves the trigonometric function from every starting point, and in two cases solves it with smaller evaluation counts than any other method.

Note, however, that rule 24 cannot solve every version of the trigonometric function within the allotted iterations. For instance, from the standard starting point with n=60, it terminates after 100 Jacobian and 151 residual evaluations. Therefore the fact that rule 24 solves every test problem shown in the figures of Section 8.4 does not necessarily indicate any superiority over, for example, rule 12. As with other tests, the results must be taken on average, so that a fortuitous selection of starting points does not unduly influence the evaluation of the algorithm.

Comparing results for the other problems, apparently the exact location of the break point between the normalized gradient and one-norm weighted r-square rules does not greatly affect the mix weighting algorithm. Essentially rule 24 matches rule 12, though, on average, rule 24 turns in slightly lower evaluation counts.

Summary of mixing rules

Any of the mix weighting rules, 12 to 16 and 21 to 24, probably is superior to an r-square, normalized gradient r-square, or one-norm weighted r-square rule along, for solving a general nonlinear equilibrium problem using a double dogleg type minimization algorithm. The numeric experiments documented in Section 8.4 give little support for claiming any clear choice from among these mix weighting rules, though possibly rules 12, 16, 23, and 24 are the most reasonable, and rules 12 and 24 the most robust.

For a particular problem, or class of problem, regardless of the starting point, one of the unmixed rules may converge more quickly or reliably. In general the normalized gradient r-square, rule 9, converges in fewer iterations than the standard r-square rule 1, i.e. in fewer iterations than the double dogleg algorithm of Section 5.3. If both the r-square versions stagnate, then a one-norm weighted r-square is likely to solve the problem.

8.4 Numeric Results

The figures below report the performance of the methods introduced in this chapter, reproducing for comparison some results from the algorithms of the preceding chapters.

As before, all results are for the standard termination conditions, as run on a Macintosh 7100/66.

Because WTDDOGLG.CPP implements only one weighting scheme per executable, the test results compare runs made using different builds. The various builds, utilizing the system resources differently, show slight variations in their execution times, even for the unchanged solution schemes. Rather than scaling the running time of each new weighting scheme using the running time of a standard algorithm from the same build, the figures do not report running times (the number of iterations required do not vary from build to build since these depend only on basic machine characteristics such as MACH_EPS).

Untabulated results

Every algorithm and every weighting rule solves the following test functions from each trial starting point using a pure Newton-Raphson sequence: (1) the Broyden trigdiagonal function; (2) the discrete boundary value function; (3) the discrete integral equation function; and (4) the Powell singular function.

Since the evaluation counts do not vary from one algorithm to another, the figures of Section 7.3 show the numeric results for these test functions.

Summary of weighting rules

For convenience of reference, Figure 8.4.1 summarizes the weighting rules by: (1) the type of weighting, e.g. r-square or one-norm weighted r-square, with or without gradient normalization; (2) the break point used when mixing the weights, if applicable; and (3) other variations, such as persistent norms, described in Section 8.3.

Weighting rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
R-square	x													x										
One-norm		x	x	x	x	x						x		x	x	x	x	x			x	x	x	x
R-square with normalized gradient									x			x	x		x	x			x	x	x	x	x	x
One-norm with normalized gradient										x	x		x											
Mix, break point at $\|r_{i[k]}\|/\|\|\nabla r_{i[k]}\|\|$												x	x	x	x	x					x	x		
Mix, break point at $2\cdot\|r_{i[k]}\|/\|\|\nabla r_{i[k]}\|\|$																							x	x
Norm persists by averaging																	x	x	x	x	x	x	x	x
Continuity between mixed rules															x	x								
Initial step in NR direction					x	x																		
Mix by square root							x	x																
Weighting rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Figure 8.4.1. Weighting rules identified by category.

Duct flow problem

starting point	$(0.02, 7, 1)^T$	$(0.001, 0.0039, 34.06)^T$	$(60, 60, 60)^T$	$(90, 90, 90)^T$
NEWTRAPH	8, 9	14, 64	18, 45	19, 46
DBLDOGLG	8, 9	8, 34	18, 54	21, 59
PLANARHK	8, 9	8, 34	16, 49	32, 99
WTDDOGLG-1	8, 9	8, 34	18, 54	21, 59
WTDDOGLG-2	8, 9	8, 35	15, 47	21, 64
WTDDOGLG-3	8, 9	8, 35	15, 47	21, 64
WTDDOGLG-4	8, 9	8, 35	15, 47	21, 64
WTDDOGLG-5	8, 9	8, 35	15, 47	21, 64
WTDDOGLG-6	8, 9	9, 39	20, 35	18, 31
WTDDOGLG-7	8, 9	8, 34	19, 53	21, 63
WTDDOGLG-8	8, 9	8, 34	19, 53	21, 63
WTDDOGLG-9	8, 9	9, 37	17, 46	30, 72
WTDDOGLG-10	8, 9	9, 43	22, 44	33, 68
WTDDOGLG-11	8, 9	9, 43	22, 44	33, 68
WTDDOGLG-12	8, 9	9, 37	50, 162	16, 51
WTDDOGLG-13	8, 9	9, 38	34, 75	14, 36
WTDDOGLG-14	8, 9	8, 34	100, 131, itns	100, 126, itns
WTDDOGLG-15	8, 9	9, 37	25, 69	17, 71
WTDDOGLG-16	8, 9	9, 37	100, 136, itns	100, 146, itns
WTDDOGLG-17	8, 9	8, 35	16, 50	16, 42
WTDDOGLG-18	8, 9	8, 35	21, 66	18, 51
WTDDOGLG-19	8, 9	9, 39	25, 80	16, 55
WTDDOGLG-20	8, 9	9, 38	24, 65	24, 71
WTDDOGLG-21	8, 9	9, 41	23, 68	17, 54
WTDDOGLG-22	8, 9	8, 36	14, 39	20, 58
WTDDOGLG-23	8, 9	11, 45	15, 45	17, 54
WTDDOGLG-24	8, 9	8, 36	30, 85	22, 66

Figure 8.4.2. Results for the duct flow problem. Data pairs give the number of Jacobian and residual evaluations, respectively, to solution.

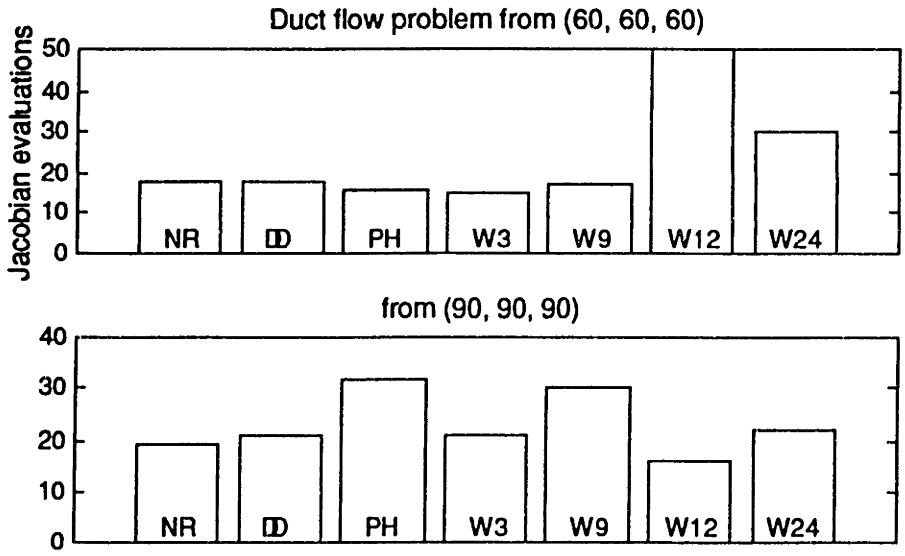

Figure 8.4.3. Jacobian evaluations, taken from Figure 8.4.2, for selected algorithms and starting points on the duct flow problem.

Powell badly scaled function

starting point	std	5·std	10·std	$(-10, -9.9)^T$	$(10, 20)^T$
NEWTRAPH	11, 12	7, 8	4, 5	91, 92	2, 3, nan
DBLDOGLG	24, 29	25, 30	4, 5	34, 58, stag	39, 52
PLANARHK	16, 20	21, 26	4, 5	31, 52, stag	22, 28
WTDDOGLG-1	24, 29	25, 30	4, 5	35, 60, stag	39, 52
WTDDOGLG-2	77, 83	65, 97	4, 5	100, 185, itns	23, 32
WTDDOGLG-3	77, 83	19, 42	4, 5	100, 185, itns	23, 32
WTDDOGLG-4	77, 83	24, 29	4, 5	100, 185, itns	23, 32
WTDDOGLG-5	77, 83	13, 27	4, 5	100, 185, itns	23, 32
WTDDOGLG-6	40, 74	26, 63	4, 5	100, 278, itns	2, 3, nan
WTDDOGLG-7	28, 33	33, 38	4, 5	58, 90, stag	34, 44
WTDDOGLG-8	28, 33	26, 35	4, 5	58, 90, stag	34, 44
WTDDOGLG-9	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-10	11, 12	7, 8	4, 6	28, 44	16, 33
WTDDOGLG-11	11, 12	7, 8	4, 6	28, 44	13, 30
WTDDOGLG-12	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-13	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-14	24, 29	25, 30	4, 5	100, 268, itns	39, 52
WTDDOGLG-15	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-16	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-17	16, 19	24, 29	4, 5	100, 129, itns	23, 32
WTDDOGLG-18	16, 19	25, 29	4, 5	100, 135, itns	17, 26
WTDDOGLG-19	11, 12	7, 8	4, 5	25, 27	14, 22
WTDDOGLG-20	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-21	11, 12	7, 8	4, 5	25, 27	14, 22
WTDDOGLG-22	11, 12	7, 8	4, 5	25, 27	15, 23
WTDDOGLG-23	11, 12	7, 8	4, 5	25, 27	12, 20
WTDDOGLG-24	11, 12	7, 8	4, 5	25, 27	15, 23

Figure 8.4.4. Results for the Powell badly scaled function.

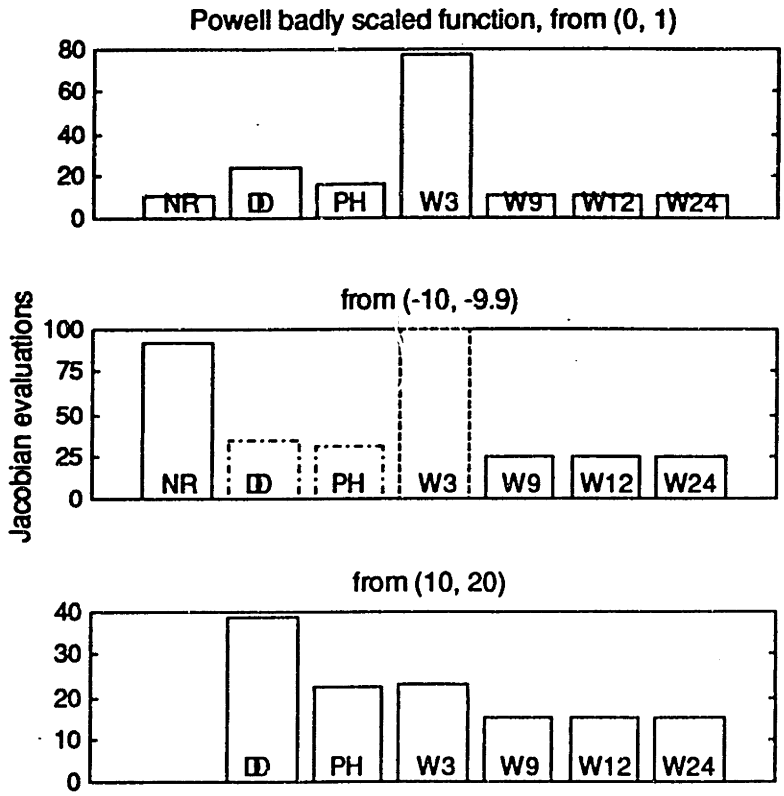

Figure 8.4.5. Jacobian evaluations for selected algorithms and starting points on the Powell badly scaled function. Solid lines show the method found a solution; dashed lines that the method reached its 100-iteration limit; dash-dot lines that the method stagnated.

Rosenbrock function

dimension	2				10
starting point	std	10·std	100·std	$(20, 20)^T$	std
NEWTRAPH	2, 3	2, 3	2, 3	2, 3	2, 3
DBLDOGLG	16, 23	3, 5	3, 5	100, 103, itns	16, 23
PLANARHK	15, 22	3, 5	3, 5	100, 103, itns	15, 22
WTDDOGLG-1	16, 23	3, 5	3, 5	100, 103, itns	16, 23
WTDDOGLG-2	52, 85	3, 5	3, 5	2, 3	52, 85
WTDDOGLG-3	52, 85	3, 5	3, 5	2, 3	52, 85
WTDDOGLG-4	52, 85	3, 5	3, 5	2, 3	52, 85
WTDDOGLG-5	52, 85	3, 5	3, 5	2, 3	52, 85
WTDDOGLG-6	22, 29	3, 5	3, 5	2, 3	22, 29
WTDDOGLG-7	17, 21	3, 5	3, 5	2, 4	17, 21
WTDDOGLG-8	17, 21	3, 5	3, 5	100, 103, itns	17, 21
WTDDOGLG-9	9, 13	3, 5	3, 5	2, 3	9, 13
WTDDOGLG-10	6, 10	2, 3	2, 3	2, 3	6, 10
WTDDOGLG-11	6, 10	2, 3	2, 3	2, 3	6, 10
WTDDOGLG-12	9, 15	3, 5	3, 5	2, 3	9, 13
WTDDOGLG-13	9, 15	3, 5	3, 5	2, 3	9, 13
WTDDOGLG-14	17, 24	3, 5	3, 5	100, 103, itns	16, 22
WTDDOGLG-15	9, 13	3, 5	3, 5	2, 3	9, 13
WTDDOGLG-16	9, 12	3, 5	3, 5	2, 3	9, 14
WTDDOGLG-17	31, 38	3, 5	3, 5	2, 3	31, 38
WTDDOGLG-18	22, 27	3, 5	3, 5	2, 3	22, 27
WTDDOGLG-19	8, 12	3, 5	3, 5	2, 3	8, 12
WTDDOGLG-20	8, 12	3, 5	3, 5	2, 3	8, 12
WTDDOGLG-21	9, 13	3, 5	3, 5	2, 3	8, 11
WTDDOGLG-22	9, 13	3, 5	3, 5	2, 3	8, 11
WTDDOGLG-23	9, 13	3, 5	3, 5	2, 3	9, 13
WTDDOGLG-24	9, 13	3, 5	3, 5	2, 3	9, 13

Figure 8.4.6. Results for the extended Rosenbrock function.

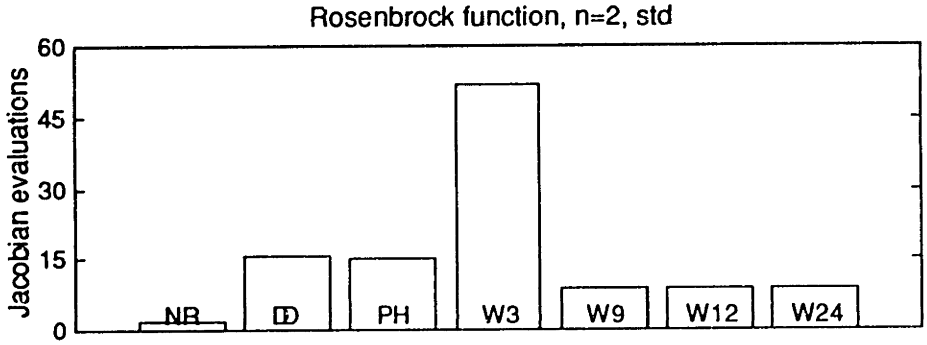

Figure 8.4.7. Jacobian evaluations for the 2-dimensional Rosenbrock function from its standard starting point.

Trigonometric function

dimension	5			10	50
starting point	std	5·std	10·std	std	std
NEWTRAPH	5, 6	84, 85	49, 50	6, 7	8, 9
DBLDOGLG	8, 12	72, 101, stag	77, 106, stag	71, 97, stag	100, 115, itns
PLANARHK	8, 11	14, 18	11, 14	89, 114, stag	100, 119, itns
WTDDOGLG-1	8, 12	73, 103, stag	72, 100, stag	72, 98, stag	100, 116, itns
WTDDOGLG-2	9, 14	100, 127, itns	12, 15	11, 19	100, 131, itns
WTDDOGLG-3	9, 14	100, 156, itns	12, 15	11, 19	12, 18
WTDDOGLG-4	9, 14	100, 155, itns	12, 15	11, 19	100, 133, itns
WTDDOGLG-5	9, 14	100, 174, itns	12, 15	11, 19	12, 20
WTDDOGLG-6	8, 11	100, 245, itns	100, 232, itns	12, 22	100, 171, itns
WTDDOGLG-7	8, 10	13, 18	100, 132, itns	100, 130, itns	100, 116, itns
WTDDOGLG-8	8, 10	13, 18	100, 132, itns	100, 131, itns	100, 191, itns
WTDDOGLG-9	7, 9	47, 72, stag	71, 134, stag	8, 13	100, 126, itns
WTDDOGLG-10	11, 17	100, 116, itns	13, 16	16, 25	24, 37
WTDDOGLG-11	11, 17	100, 158, itns	13, 16	16, 25	14, 20
WTDDOGLG-12	7, 9	14, 21	18, 28	8, 13	100, 199, itns
WTDDOGLG-13	7, 9	100, 197, itns	21, 33	8, 13	100, 180, itns
WTDDOGLG-14	8, 12	16, 21	100, 188, itns	100, 211, itns	100, 196, itns
WTDDOGLG-15	7, 9	14, 22	16, 23	8, 13	100, 182, itns
WTDDOGLG-16	7, 9	13, 20	100, 137, itns	8, 13	100, 150, itns
WTDDOGLG-17	11, 19	100, 135, itns	12, 16	11, 18	15, 26
WTDDOGLG-18	8, 13	13, 19	100, 140, itns	9, 16	11, 16
WTDDOGLG-19	7, 9	52, 84, stag	45, 73, stag	8, 12	100, 121, itns
WTDDOGLG-20	7, 9	15, 22	73, 144, stag	8, 13	100, 131, itns
WTDDOGLG-21	7, 9	14, 21	12, 15	8, 12	100, 175, itns
WTDDOGLG-22	7, 9	14, 23	12, 15	8, 13	100, 186, itns
WTDDOGLG-23	8, 12	100, 169, itns	100, 144, itns	7, 12	100, 153, itns
WTDDOGLG-24	8, 13	13, 19	12, 15	7, 10	8, 13

Figure 8.4.8. Results for the trigonometric function.

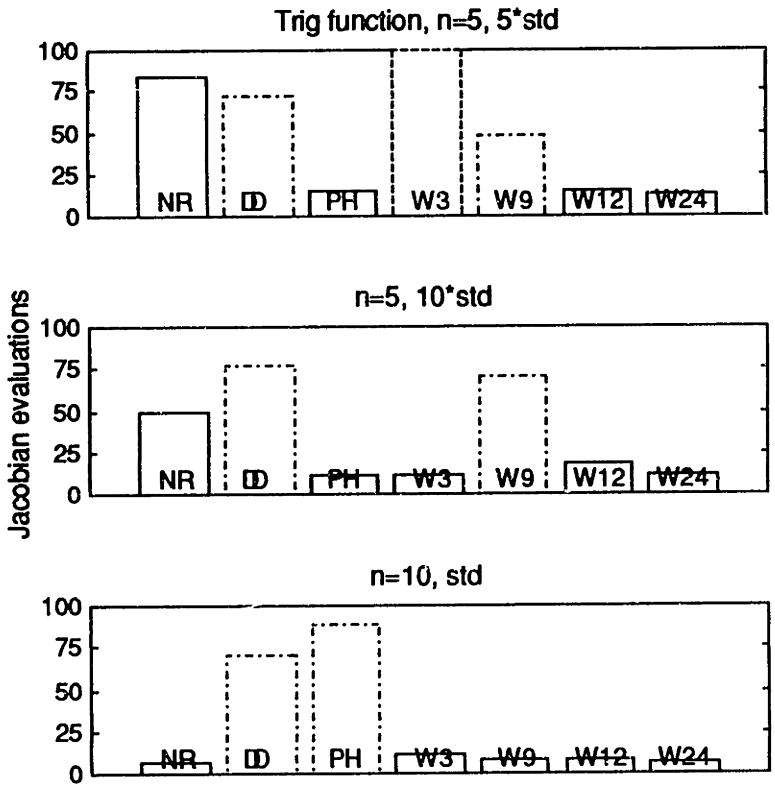

Figure 8.4.9. Jacobian evaluations for the trigonometric function. Solid lines show the method found a solution; dashed lines that the method reached its 100-iteration limit; dash-dot lines that the method stagnated.

CHAPTER 9
DIVERGENCE MEASURES

This chapter develops alternate means for detecting divergence in a Newton-Raphson-based equation solver, as a first step towards overcoming problems with the descent-based divergence tests used in the double dogleg, planar hook, and weighted double dogleg methods. Much of this material speculates on the behavior of proposed, but untested, numeric methods, based on the observed behavior of the methods tested in Chapters 5, 7, and 8 of the thesis.

Section 9.1 discusses the Newton-Raphson direction, concluding that the descent methods often fail fully to exploit this computationally-expensive search direction. The discussion continues the critique of the standard methods of adapting function minimization techniques to the solution of equilibrium problems, and demonstrates that no Jacobian-based model of a residual norm can adequately predict the performance of the norm in the Newton-Raphson search direction. Then since Newton-Raphson's method acts on local models of the residuals, it defines an appropriate search direction for short trust lengths.

Section 9.2 develops individual and lumped measures of divergence in the residual models during a single step, and discusses how these could partially replace the residual norm as a means of updating the trust region in a double dogleg type search method. The norm retains its value in deciding between two trial points which both meet some divergence criterion, and for defining reasonable alternate search directions when the full Newton-Raphson step fails.

Section 9.3 shows how the convergence results for Newton-Raphson's method suggest a divergence test to be applied between successive steps, rather than during a single step.

9.1 Observations on the Newton-Raphson Direction

The analyses of the double dogleg, planar hook, and weighted double dogleg algorithms in the preceding chapters contain a common theme that these descent methods fail to exploit the Newton-Raphson direction fully. In perhaps the most striking example, Newton-Raphson's method solves the Rosenbrock function in two Jacobian evaluations, regardless of the starting point or problem dimension, while a descent-based method rarely does so-- from the standard starting point, the double dogleg method takes 16 Jacobian and 23 residual evaluations.

Need to exploit Newton-Raphson direction

Practically, an algorithm should extract as much use as possible from the Newton-Raphson direction, due to the computational cost of finding it. For an n-dimensional problem, finding a steepest descent direction J^Tp from Equation 4.3.5 requires $2n^2$ floating point operations, or flops [Golub §1.2.4]. That is, it takes n multiplications and n additions for each row of the Jacobian. To predict the stepsize in this steepest descent direction takes at least another matrix-vector multiply (see for example Equations 4.2.9c and 4.5.7c), for another $2n^2$ flops; the entire operation is still order n^2 flops.

By contrast, the Newton-Raphson step requires $2n^3/3$ flops for LU factorization [Golub §3.2.6], plus another $2n^2$ flops for the substitution steps needed to find $\Delta x_{[k]:NR}$

[Golub §3.1.1, §3.1.2]. As the problem dimension increases, the expense of finding the Newton-Raphson step dominates the problem.

The following subsections summarize observations which indicate the failure of the descent methods to exploit the Newton-Raphson direction fully: (1) the evaluation counts of the Newton-Raphson versus double dogleg methods; (2) the specific experience with rule 6 of the weighted double dogleg algorithm; and (3) the path selection of the planar hook algorithm.

Evaluation counts

Figure 8.2.1 suggests the difficulty facing any descent-based method when solving the two-dimensional Rosenbrock function: the contours of any reasonable residual norm form narrow, curving valleys which constrain a descent-based solver. The discussion points out that Newton-Raphson's remarkable success on the Rosenbrock function comes from the particular nature of the defining equations. However the experimental results from Section 8.4 indicate that the unchecked Newton-Raphson sequence solves many of the test problems more effectively than the double dogleg.

For four of the eight test problems, the descent methods succeed by allowing the full Newton-Raphson step at each iteration. Furthermore, Newton-Raphson's method requires fewer Jacobian evaluations than any other method for three of the five starting points on the trigonometric function (see Figure 8.4.8), while for the Powell badly scaled function, it ties with other methods for the best counts in three of the five starting points (Figure 8.4.4).

On the other hand, Newton-Raphson's method performs poorly on the other starting points for these two problems. For the trigonometric function, it does converge, but only after a large number of iterations compared to those descent based methods which do converge. For the Powell badly scaled function, Newton-Raphson fails to converge from the starting pont $(10, 20)^T$, from which its second iteration gives an infinite residual to machine precision.

Overall, the evaluation counts reported in Section 8.4 show that Newton-Raphson's method often converges quickly, despite increasing any number of reasonable residual norms during at least one iteration. The descent requirement, or the evaluation rules associated with it, therefore often prevent a complete Newton-Raphson sequence from successfully solving the problem in a very few number of iterations. However, Newton-Raphson's method does require some check against divergence in a significant number of cases, and so cannot stand alone.

Rule 6 experience

Recall weighting rule 6 from Section 8.3. The rule places its initial step of each iteration at the trust length in the Newton-Raphson direction, then calculates its residual weights based on this initial step. In general, the rule does about as well as the other one-norm weighting rules on those problems it can solve. On poorly scaled problems it does better than the other one-norm rules, but on problems with excessive Jacobian evaluations it takes significantly more residual evaluations than the other one-norm rules.

From the discussion of rule 6, it rarely evaluates its initial Newton-Raphson direction choice favorably enough to double the trust length during the current iteration. That is, the one-norm weighting selection of rule 6 tends not to create norms whose models the Newton-Raphson direction can satisfy easily. In fact, the rule more often rejects its initial point for insufficient decrease in the norm, then places a new trial iterate at

the same trust length but along the double dogleg curve, and then doubles the trust length because of a strongly favorable decrease in the norm. This suggests that the double dogleg evaluation rules, stressing a descent in the norm of choice, bias the selection of iterates against the Newton-Raphson direction-- at least for the one-norm weighting rules.

Planar hook path

Comparing the planar hook to the double dogleg search path, for short step lengths (i.e. for steps less than the Cauchy length) the planar hook departs from the double dogleg path, biasing towards the Newton-Raphson direction; while for relatively long steps (i.e. those greater than the cutback length, where the double dogleg path rejoins the Newton-Raphson direction) the planar hook still mixes some of the steepest descent direction into its step choice. See Figures 7.2.1 and 7.2.2.

Therefore, possible explanations for the superiority of the planar hook focus on steps of different lengths:

(1) The planar hook may handle shorter steps better by turning toward the Newton-Raphson direction sooner. A short trust length indicates that the model $f_{(2)}$ does not approximate $\emptyset_{(2)}$ well. If not, then the Cauchy point does not minimize $\emptyset_{(2)}$ in the $g_{(2)}$ direction, and turning toward the Newton-Raphson direction helps to insure that a short step gives a sufficient decrease in the norm.

(2) The planar hook may handle longer steps better by tempering the Newton-Raphson step choice. This may either improve the quality of the step selection, or increase the likelihood that the evaluation rules assess the step favorably. Since Newton-Raphson makes its step choice based on the local slopes in the residual functions, extrapolating each linearized model to its intersection with the plane $r = 0$, the method depends on every residual model more or less holding over the entire step length. If that fails-- as it will for nonlinear systems far from x**-- then retaining a component of the steepest descent direction may help by preferentially reducing residuals of the greatest magnitude. Either this improves the actual step (from the point of view of the minimization algorithm) by providing a greater decrease in the actual norm, or it improves the step evaluation, by allowing a better match between the norm and its model.

Recall that the descent-based evaluation rules increase the trust length not only when a step gives large decreases in the residual norm, but also when the model closely approximates the actual behavior of the norm. (Conversely, they cut the step length when the model performs poorly.) In this context, a more favorable evaluation of a step on the planar hook curve may result from a smaller disparity between the predicted and actual norms, rather than from an inherently better step selection. In either case, the difference between the planar hook and double dogleg curves at long steps highlights the difficulty of favorably evaluating a step in the Newton-Raphson direction.

All these effects may contribute to giving the planar hook its slight edge over the double dogleg search path. Note, however, that the explanations posit essentially different phenomena-- arguably better step selections for the shorter steps, versus better evaluations of longer steps (even if the planar hook succeeds at longer steps by providing greater decreases in the residual norm, the difference between the planar hook curve and the Newton-Raphson direction at these step lengths is unlikely to be significant from the point of view of solving the algebraic system). In either case, the contribution of the planar hook search path may be seen as one of extracting as much use from the Newton-Raphson direction as possible.

Failure of model

These arguments suggest, without proving, that two main obstacles prevent the double dogleg algorithm from taking full advantage of the Newton-Raphson direction: (1) the apparent difficulty, in some circumstances, of favorably evaluating a Newton-Raphson step using the descent criteria; and (2) the double dogleg curve's adherence to the steepest descent direction all the way to the Cauchy point. Both possibilities indicate that $f_{(2)[k]}$, the model built using the function and derivative information at $x_{[k]}$, fails adequately to predict the essential behavior of the norm, $\emptyset_{(2)}$.

A mismatch between $\emptyset_{(2)}$ and $f_{(2)[k]}$ indicates a disparity between the actual and modeled behavior of one or more of the residuals from which $\emptyset_{(2)}$ is constructed, i.e. a mismatch between an r_i and its model, $\hat{r}_{i[k]}$. From Equation 2.2.4, the first-order Taylor series approximation

$$\hat{r}_{i[k]}\{x\} = r_{i[k]} + \nabla r_{i[k]}^T(x - x_{[k]})$$

captures the function value and slope of r_i at $x_{[k]}$, but omits its curvature and higher derivatives. Therefore it matches a linear equation over all x, but approximates a nonlinear residual only for a limited range about $x_{[k]}$. See Figure 6.1.1, which shows r_1 from the two-dimensional trigonometric function, and its linear and quadratic models about the point $(0.1, 0.7)^T$.

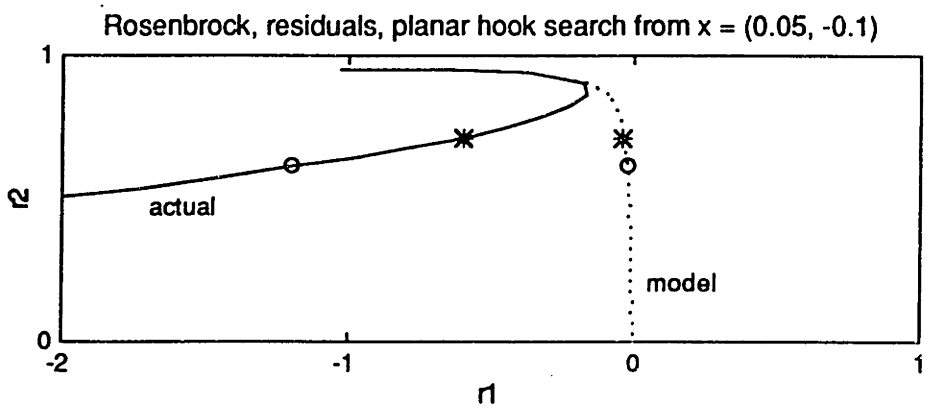

Figure 9.1.1. Residual trajectory for a planar hook search from $x_{[0]} = (0.05, -0.1)^T$ on the two-dimensional Rosenbrock function. The actual and modeled residuals from two trial points on the search path show that the model diverges due to nonlinearities in the first residual equation.

Figure 9.1.1 demonstrates the effect of linearizing r_1 on the path through residual space taken by a planar hook search on the Rosenbrock function from $x_{[0]} = (0.05, -0.1)^T$. (Figure 7.2.1 shows the search path itself, in the independent variable space, x; Figure 9.1.1 shows the resulting trajectory in the dependent variables, r.)

Because it ends at the full Newton-Raphson step, the planar hook path zeros the models of r_1 and r_2, as shown in the figure. For short steps, the path follows $g_{(2)}$ and so the modeled residual trajectory does not take a straight line from $r_{[0]} = (-1.025, 0.95)^T$ to r = 0, as it would if the search followed the Newton-Raphson direction.

For short steps, $\hat{r}_1$ represents r_1 well enough that the actual and predicted paths coincide, but they diverge as the step length increases. The Jacobian model represents the second, linear, residual equation exactly, so the deviation between the actual and modeled residual trajectories comes from r_1 only. Thus the residual pair marked by stars in Figure

9.1.1, which have the same value of r_2, both come from the same trial point $x_{t1} = (0.2878, 0.0228)^T$ on the planar hook curve. Similarly, the trial point $x_{t2} = (0.3926, 0.0346)^T$ gives the residual pair marked by circles. Note that the greater step length to x_{t2} induces a greater error in the linearized residual model $\hat{r}_1$.

Including second-order information in a model $\hat{r}_{i[k]}$ would extend the range over which it represents a nonlinear r_i; however, to do so would require an extra n count of matrices of n^2 elements each-- each matrix containing the second derivatives $\nabla^2 r_{i[k]}$ of a single residual-- and would increase the computational cost of forming the residual models at the beginning of each iteration. Section 2.2 mentions other second-order modeling options.

First derivatives in a second-order method

Lacking these second derivatives, the double dogleg method of equation solving models $\emptyset_{(2)}$ using only the linearized $\hat{r}_{i[k]}$, and imbeds the resulting $f_{(2)[k]}$ into a second-derivative-based algorithm, the double dogleg method of function minimization. Similarly, the weighted double dogleg method of Section 8.3 constructs $f_{(w)[k]}$ from the same first-order residual models, then attempts to minimize $\emptyset_{(w)}$ using this model in the second-order search technique from Section 3.6.

Section 8.2 lists several assumptions about the residual norm which carry over from a too-literal application of minimization theory to a descent-based solver. Adding to the list, interpreting $f_{(2)[k]}$ and $f_{(w)[k]}$ as Hessian-based cost function models tends to imply that they contain meaningful second-order information.

Since $\emptyset_{(2)}$ is a special case of $\emptyset_{(w)}$, the following discussion treats only the weighted sum of squares. Section 7.1 treats $\emptyset_{(2)}$.

Assumption: double dogleg a second-order method

In function minimization, the Newton step minimizes the second-order model of the cost function, and the shorter step to the Cauchy point follows the search direction suggested by the first-order part of that model (although to calculate the Cauchy point requires the second derivatives as well). When a search path mixes some of the steepest descent direction into the step choice for shortened trust lengths, it relies less on the curvature information, and places greater emphasis on the slope information, contained in the cost function model. For example, $\nu_{[k]}$ in Equation 3.6.1a controls the mix of the steepest descent and Newton directions for the hook search path. The trust region, measuring how far the quadratic model accurately predicts the actual cost function, also dictates the extent to which the second-derivative information in the Hessian influences the step selection.

Applied to equilibrium problems, interpreting $f_{(w)}$ as a second-order cost function model implies that the failure of the full Newton-Raphson step represents a failure in some second-order part of the model, which can be compensated by biasing the step selection to favor the first-derivative part of the model.

However, in the weighted double dogleg method the second-derivative part of the model $f_{(w)}$ depends on the same information which sets its first-derivative characteristics-- that is, on the first-order residual models, and on their algebraic combination. The second derivatives which cause the actual norm to diverge from its model reside in the unmodeled second derivatives of the residuals themselves.

Explicitly, suppose a descent-based equation solver uses the approximate Hessian of Equation 4.3.11,

$$G \approx J^T P J$$

Then its model curvature includes only the first derivatives of the residuals, expressed in the Jacobian J, and the algebraic form of the residual norm, expressed in the matrix P, a constant. From Equation 4.5.4c, $P_{(w)[k]} = 2W_{[k]}$ for the weighted r-square norm; $P_{(w)[k]}$ is constant for each iteration of the weighted double dogleg algorithm.

To make such a solver a true minimization procedure would require using the exact Hessian of Equation 4.3.10,

$$G = J^T P J + \sum_{i=1}^{n} p_i \nabla^2 r_i$$

Since $p_{i(w)} = 2w_i r_i$ from Equation 4.5.4b, the model $f_{(w)}$ in the weighted double dogleg algorithm neglects the curvature $2 \cdot \sum w_i r_i \nabla^2 r_i$ due to the curvature $\nabla^2 r_i$ in each residual equation.

Recall from Section 4.3 that the Newton step on a weighted sum of squares of residuals matches the Newton-Raphson step on the underlying equilibrium problem only when the weighted sum of squares neglects the second-order effects in the residuals. Thus, the double dogleg method models its cost function by the weighted sum of squares of the linearized residuals. This essential condition for deriving Equation 4.3.13-- and thus for identifying $\emptyset_{(w)}$ as appropriate for use in the double dogleg algorithm-- means that the only curvature effects which influence the double dogleg step selection come from the same first-order models which set the Newton-Raphson direction.

Similar comments apply to any residual norm model constructed from the linearized residual models. Geometrically, each residual model replaces the surface $r_i\{x\}$ by its tangent hyperplane at the current iterate $x_{[k]}$ (see for example Figure 2.2.4). The model of a residual norm combines these tangent hyperplanes-- not the residuals themselves-- and so cannot predict important second-order effects in the norm. Figure 6.2.1 shows a graphical example.

Observations on model composition

Recognizing $f_{(w)[k]}$ as the algebraic composition of n linear residual models, which therefore neglects the second-order (and higher) effects that cause $\emptyset_{(w)}$ to behave unpredictably, implies that: (1) to apply the trust region idea to equilibrium problems properly requires not one, but n, estimates of the greatest step length for which a model holds; (2) the model $f_{(w)}$ itself cannot be used to compare possible weighting selections for their probable success at evaluating longer steps; and (3) for short steps the difference between the Newton-Raphson direction and a steepest descent direction represents a strategic choice rather than an improper use of the model.

The following subsections address these observations in greater detail.

Model requires n trust regions

Since $f_{(w)[k]}$ models $\emptyset_{(w)}$ by combining n linear residual models, the equilibrium problem can be described by n, not 1, trust regions: each giving the extent to which one of the residual models captures the performance of the corresponding residual. The overall trust region, which gives the maximum step length for which $f_{(w)[k]}$ accurately approximates $\emptyset_{(w)}$, depends on the trust length of each residual, and on the relative weight the residual carries in setting the norm.

In the double dogleg method of Section 5.3, the ability of $f_{(2)[k]}$ to estimate $\emptyset_{(2)}$ varies at each iteration because the method forms new residual models, with different abilities to represent their respective residual function than at the previous iteration. The overall trust length depends on the extent to which the residual models hold. (Of course the individual trust lengths also depend on the search direction, a distinction beyond the scope of this discussion.)

In the weighted double dogleg of Section 8.3, choosing the $w_{i[k]}$ by various weighting rules affects the ability of $f_{(w)[k]}$ to predict $\emptyset_{(w)}$, even if the linearized residual models remain unchanged. Compare, for example, the alternate residual weightings $W_{[k]} = \text{diag}\{0, 1\}$ and $W_{[k]} = \text{diag}\{1, 0\}$ at some $x_{[k]}$ in the two-dimensional Rosenbrock function. For the first set of residual weights, $f_{(w)[k]}$ matches $\emptyset_{(w)}$ exactly for all x, because it depends only on the second, linear, residual. The trust length is infinite. For the second set of weights, on the other hand, $f_{(w)[k]}$ represents $\emptyset_{(w)}$ only as well as $\hat{r}_{1[k]}$ represents r_1-- in other words, only for small excursions about $x_{[k]}$. Therefore the overall trust length depends on the relative weights given to the residuals, as well as on the trust lengths of the residual models.

Based on this observation, an equilibrium problem solver might attempt to update a trust region for each residual model, possibly using rules similar to those given in Section 3.6 for the double dogleg method of function minimization. Since such a method must update the trust regions for all n models based on the same step, it may only be able to indicate when the trust length for a particular model extends beyond (or falls short of) the length of the latest step. Moreover, to determine the length of the next trial step, it must have some rule for combining these n trust lengths into an overall trust length for the selected norm.

This added complexity, while satisfying the formal basis of the residual norm's model, may not have much value in practice. Results for the weighted double dogleg algorithm, given in Section 8.4, do not support the notion that the calculated trust length $\delta_{[k]}$ estimates some actual trust region better when the residual weights vary less between iterations.

Suppose changing the residual weights caused significant variations in the actual extent to which $f_{(w)[k]}$ captures $\emptyset_{(w)}$. Then those weighting selections which allow large changes in the residual weights ought, on average, to require more residual evaluations per iteration, since on average they would be more inclined to force the algorithm to increase or to decrease the inherited trust length in response to the new weights.

However, the average number of residual evaluations per iteration for each run of the Powell badly scaled function, the Rosenbrock function, and the trigonometric function, do not show any consistent differences between the double dogleg, planar hook, and weighted dogleg methods. Calculate the ratio as $(n_r - 1)/n_j$, decrementing the number of residual evaluations, n_r, by 1 since one residual evaluation is required to start the first of the n_j Jacobian iterations. Note that the duct flow problem was not included, since the possibility of residual evaluation errors obscures the question of how many residual evaluations were required to establish a new trust region.

The ratios range from a low of 1 for Newton-Raphson's method, to typical values of 1.1-1.5 for the other methods, to occasional high values in the range of 2.5, especially for weighting rule 6 which always makes a preliminary step in the Newton-Raphson direction. The weighted double dogleg, despite changing the residual weights at each iteration, does not in general require more residual evaluations per iteration than the double dogleg or planar hook methods. Nor do comparisons between the averaged, "persistent," norms and

their unaveraged counterparts within the weighted dogleg method reveal any preference in terms of residual evaluations per iteration (though the averaged norms can require fewer iterations overall).

First-order models cannot predict success

Because the Jacobian model contains only first order information about the residuals, no norm selection criterion based on the Jacobian model can anticipate second order effects in the norm. In particular this applies to $\emptyset_{(w)}$. Thus, for example, a weighted double dogleg type algorithm cannot use the model $f_{(w)}$ itself to choose weights $W_{[k]}$, or to choose between two or more possible weighting selections, such that $\emptyset_{(w)}$ evaluates the Newton-Raphson direction as favorably as possible.

As shown above, the only second-derivative part of $f_{(w)}$ comes from the linearized residuals and their algebraic combination. The approximate Hessian $G_{(w)} \approx 2J^TWJ$ does not carry any information about curvature in the residuals, and so cannot estimate second-order effects in the norm.

At short steps, where the linearized models hold and where first-order effects dominate the change in $\emptyset_{(w)}$, $f_{(w)}$ approximates $\emptyset_{(w)}$ well. However for longer steps the ability of $f_{(w)}$ to predict $\emptyset_{(w)}$ depends on the second-order behavior of the residuals--behavior about which the solver has no information. Furthermore these second-order effects influence the actual behavior of the norms to varying extents in different search directions. Thus it is impossible to predict, using $f_{(w)[k]}$, whether the residual models better approximate the actual residuals in the Newton-Raphson direction than along the double dogleg or planar hook paths. Similarly it is impossible, based on first-derivative models, to choose weights $W_{[k]}$ in order to guarantee that $\emptyset_{(w)}$ favorably evaluates a step in the Newton-Raphson direction.

Consider two possible indicators of the extent to which a norm matches the Newton-Raphson direction: (1) the initial slope in the norm, relative to its magnitude; and (2) the angle between the norm's steepest descent direction and the Newton-Raphson direction. Both appeal to intuitive notions of the meaning of a descent direction for a residual norm. However, neither can predict the utility of the norm for use in evaluating the Newton-Raphson direction.

From Equation 4.5.6b, any weighted r-square has initial slope $-2 \cdot \emptyset_{(w)}$ in the Newton-Raphson direction. Just as the relative gradient in Equation 5.2.2a considers the magnitude of the cost function when evaluating whether its rate of change is small or large, to compare initial slopes from one set of weights to another requires dividing the slope by the magnitude of the norm, $\emptyset_{(w)}$. Thus, take the initial slope in the Newton-Raphson direction relative to the weighted norm itself:

$$\frac{1}{\emptyset_{(w)}} \frac{d\emptyset_{(w)}}{d\mu}\bigg|_{x_{[k]:NR}} = -2$$

a constant, regardless of the weights selected. This applies whether the norm's steepest descent direction aligns with the Newton-Raphson direction exactly, or lies nearly perpendicular to it. Therefore the initial slope in the Newton-Raphson direction does not distinguish between one set of weights and another.

Turning to the angle between the likely search directions, consider choosing residual weights to make $g_{(w)}$ colinear with the Newton-Raphson direction. Such a selection must be possible because if the Newton-Raphson direction exists then $J_{[k]}$ has full rank, and so $J_{[k]}^T$ has full rank, and so a $W_{[k]}$ exists for which $g_{(w)[k]} = -\Delta x_{[k]:NR}$:

$$g_{(w)[k]} = 2J_{[k]}^T W_{[k]} r_{[k]} = -\Delta x_{[k]:NR} = J_{[k]}^{-1} r_{[k]}$$

(9.1.1) $$W_{[k]} r_{[k]} = \frac{1}{2} J_{[k]}^{-T} J_{[k]}^{-1} r_{[k]}$$

With $W_{[k]}$ diagonal, Equation 9.1.1 may be solved for the weights by inspection, given a right hand side. Any scalar multiple of the resulting $W_{[k]}$ makes $g_{(w)[k]}$ colinear with $\Delta x_{[k]:NR}$. Unfortunately Equation 9.1.1 can give negative residual weights, and so fail to define a reasonable norm. That is, although the Newton-Raphson step defines a descent direction for every reasonable weighted r-square, it may not match the steepest descent direction of any of them.

Now suppose Equation 9.1.1 gives positive weights. It does not follow that the corresponding norm has any special properties for evaluating the Newton-Raphson step: Equation 9.1.1 selects the weights without regard for second-derivative effects in the residuals, and so cannot anticipate second-derivative effects in the resulting norm. Yet these effects may be strongest for precisely the residual with the largest weight determined by the first-order information in Equation 9.1.1.

More generally, the angle between $g_{(w)[k]}$ and $-\Delta x_{[k]:NR}$, depending as it does on first derivatives only, cannot indicate whether second-order effects in the Newton-Raphson direction will favor an evaluation by the descent criteria. (As a further objection to choosing $W_{[k]}$ to satisfy Equation 9.1.1, note that the resulting norm does not offer an alternate search direction in the event the Newton-Raphson direction fails.)

No selection criterion based on the first-order models can insure that $f_{(w)[k]}$ models $\emptyset_{(w)}$ well (e.g., in the Newton-Raphson direction), or that $\emptyset_{(w)}$ evaluates a step favorably using the double dogleg method's descent criteria.

Newton-Raphson valid for short steps

Since it is impossible, using only first-derivative information about the residuals, to predict the relative success of the model $f_{(w)[k]}$ at estimating $\emptyset_{(w)}$ in any of several possible search directions, the solver has no a priori reason to prefer a steepest descent direction to the Newton-Raphson direction for short steps. In other words, since Newton-Raphson's method uses residual models which capture only local behavior, it defines an appropriate local search direction, i.e., an appropriate direction for short steps. (Conversely, for steps longer than the Cauchy length a descent based solver must prefer the Newton-Raphson to the steepest descent direction, in order to satisfy its descent requirement.)

Note that in function minimization, a method seeking to decrease a cost function as much as possible with a short step takes the steepest descent direction because the Newton direction anticipates curvature effects which dominate only for longer steps. Following the Newton direction for short step lengths would constitute an incomplete use of the model. This does not hold for equation solving, since the curvature expressed in $f_{(w)}$ applies equally at short steps; again, the curvature affecting long steps resides in the neglected $\nabla^2 r_i$.

Geometrically, the lack of a clear preference between search directions for short steps follows from the view of each residual model as a hyperplane tangent to the residual surface. Starting from the point of tangency, $x_{[k]}$, a step in the steepest descent direction of some norm may increase the residual magnitude on some hyperplanes, while decreasing the magnitude of one or more others. Following the Newton-Raphson direction, on the other hand, decreases the magnitude on every tangent hyperplane (except

those where $r_{i[k]} = 0$; for these the Newton-Raphson direction follows the intersection of that hyperplane with the plane $r = 0$).

For a step short enough that the slopes in the residual models dominate the change from $r_{[k]}$, the choice between the search directions represents a choice between two strategies for seeking the solution of a nonlinear equilibrium problem, i.e. between: (1) strongly decreasing some residuals, at the possible expense of increasing others; or (2) decreasing every residual, though possibly by relatively small amounts. If a solver has no reason, based only on the first-order residual models, to prefer a steepest descent to the Newton-Raphson direction, it may search in the Newton-Raphson direction for short trust lengths.

Objections to Newton-Raphson for short steps

The experience with the double dogleg, planar hook, and weighted dogleg algorithms indicates two likely objections to following the Newton-Raphson direction for short steps: (1) the problem of favorably evaluating the step by a descent criterion; and (2) the lack of alternate search directions.

At face value, evaluating short steps in the Newton-Raphson direction should present less difficulty than long steps, because $f_{(w)[k]}$ models $\emptyset_{(w)}$ better. However the trust length, set by the history of previous iterations and trial steps, should by definition push the model to the limits of its predictive range. Thus a short trust length should indicate that a nonlinearity in at least one residual prevents $f_{(w)[k]}$ from capturing $\emptyset_{(w)}$ over longer steps. (In practice, a short trust length also can indicate that a previous step had difficulties, and that although models formed in the intervening steps have performed reasonably well, they have not performed well enough to justify increasing the trust length again.)

The steepest descent direction compensates for nonlinearities in the residual functions by placing its trial iterates in an attempt to decrease $\emptyset_{(w)}$ as much as possible with the short step. This increases the chance of significantly decreasing $\emptyset_{(w)}$, by making the decrease more robust to deviations between a residual and its model. Of course, the steepest descent direction does not minimize those deviations, it only minimizes the effect that a fixed deviation in a residual can have on the overall decrease in the norm. Thus, while the steepest descent direction helps guarantee a decrease in $\emptyset_{(w)}$, it does not guarantee that the actual decrease matches that predicted by $f_{(w)[k]}$.

Compared to the steepest descent direction, a short step in the Newton-Raphson direction attempts a modest decrease in every residual. Probably this makes $\emptyset_{(w)}$ more sensitive to deviations between a single residual and its model. That is, because the Newton-Raphson direction does not anticipate large decreases in a residual norm whose steepest descent direction favors decreases in some residuals over decreases in others, a fixed deviation in one residual from its model probably has greater impact on the resulting norm. Moreover the resulting step has no greater chance of matching the actual decrease to the predicted decrease, than does a step of equal length in the steepest descent direction. Thus the Newton-Raphson direction choice is at least as likely to cause the standard evaluation rules to decrease the trust length at the next iteration.

Besides the problem of evaluation, the Newton-Raphson direction, dominated by residual functions with shallow slopes, becomes unstable near local extrema in the residuals. From the geometric interpretation of Section 2.2, the full Newton-Raphson step finds the common intersection between the tangent hyperplanes, and the hyperplane $r = 0$. Therefore an extremely flat tangent hyperplane creates long Newton-Raphson steps whose direction changes rapidly with small changes in the angle of the hyperplane-- and near an

extremum the tangent hyperplanes have both small slopes and variable angles over small steps.

Coupled with a descent requirement, a local minimum in an $|r_i|$ can trap a search, and as the iterative sequence approaches the minimum, successive Newton-Raphson steps can elongate and can shift directions wildly for small changes between the iterates. See Figure 2.2.5.

The steepest descent direction may overcome this difficulty if it weights the offending residual lightly enough to ignore increases along r_i as the search escapes the local extremum. Note that the mixed weighting rules of Section 8.3 more or less attempt to diagnose this condition: if the anticipated step length cannot zero the residual given its current magnitude relative to the slope of its tangent hyperplane, then the rules do not consider that residual's relative magnitude when assigning it a weight in the norm.

Alternative to descent-based criteria

If the descent criterion can trap a Newton-Raphson search in local extrema of individual residual functions, and if the descent-based trust length evaluation rules may not be robust for steps in the Newton-Raphson direction, consider alternate means of evaluating a step in the Newton-Raphson direction.

Since the Newton-Raphson direction decreases the linearized model of every residual, and since these models hold for limited step lengths about $x_{[k]}$, a solver might follow the Newton-Raphson direction as long as the linearized models adequately predict the changes in the actual residuals. Imagine an extreme version of this strategy, one which steps only in the Newton-Raphson direction, and which accepts a step only if it decreases every residual magnitude (or leaves the magnitude of a zero residual unchanged). This method avoids divergence by restricting the step length roughly to the range over which the Jacobian model holds; once outside this range, it requires a new Jacobian model. Since it limits the trust length to the shortest distance allowed by any of its residual models, such a method possibly requires an excessive number of Jacobian evaluations. Furthermore, by allowing only the Newton-Raphson direction, it risks finding a local extremum in one residual, which would either stall the method with an unreasonably short step, or end it by finding a singular Jacobian with a row $\nabla r_i^T = 0$.

To avoid these problems, a method seeking to follow the Newton-Raphson direction should: (1) allow some divergence between the predicted and actual residuals, possibly exchanging good performance on the part of some models for poor performance on the part of others; (2) provide some means of avoiding or escaping from local extrema in individual residuals; and (3) possibly introduce a test to guard against divergence of the method as a whole, after the manner discussed in Section 2.2.

Note these conditions on a useful evaluation rule do not apply uniquely to the Newton-Raphson direction. The next section develops an evaluation criterion which serves equally well in any step direction.

9.2 Detecting Divergence in the Residual Models

A nonlinear equation solver which chooses and evaluates its steps by any kind of model-based criteria depends, generally, on the ability of the residual models to estimate the residuals. Therefore any measure of the divergence of an actual step from its predicted result should begin with a measure of the deviation the step induces in an individual residual model-- for example, with the error

$$|r_i - \hat{r}_{i[k]}|$$

The model works well when it keeps this error small compared to the distance from the point where it was formed-- in other words, compared to the step length $\|\Delta x_{[k]}\|$.

This section develops several measures of the deviation in the residual models, suitable for detecting divergence either of a single model $\hat{r}_{i[k]}$, or of the Jacobian model $\hat{r}_{[k]}$ as a whole.

Bounding residual model errors

To evaluate the error $|r_i - \hat{r}_{i[k]}|$ in a single residual model requires comparing it to some predetermined acceptable deviation. Since the model predicts the change in the residual from its value at the point where the model was formed, consider restricting the model error to a fraction of this predicted change:

(9.2.1) $$|r_i - \hat{r}_{i[k]}| \le \alpha_i \cdot |r_{i[k]} - \hat{r}_{i[k]}|$$

Nominally $0 < \alpha_i < 1$. Recall $r_{i[k]}$ gives the value $r_i\{x_{[k]}\}$ of the i^{th} residual at $x_{[k]}$, as opposed to the model $\hat{r}_{i[k]}$ formed there, which estimates r_i over all x. The test accepts equality since the initial point $x_{[k]}$, where the model was formed, sets both left and right hand sides to zero.

This test places an unrealistic burden on the predictive capability of the model. Suppose a search in the Newton-Raphson direction starts with $r_i\{x_{[k]}\} = 0$. The Newton-Raphson direction keeps $\hat{r}_{i[k]} = 0$ throughout the search, setting the right hand side of Equation 9.2.1 to zero. Yet second-order effects make $r_i \neq \hat{r}_{i[k]}$ for a step of any length. Unless $r_i\{x\}$ is linear, no such Newton-Raphson direction step could satisfy Equation 9.2.1.

The same problem affects other step choices. Consider a search path that for short steps increases $|\hat{r}_{i[k]}|$, in order to decrease the predicted magnitude of some other residual. Perhaps the search follows the steepest descent direction of some norm in which r_i has little weight. Such a path must end at the Newton-Raphson point, in order to zero every residual model, so some intermediate step gives $\hat{r}_{i[k]}$ the same value it had at $x_{[k]}$. Again, $r_{i[k]} - \hat{r}_{i[k]} = 0$. In the neighborhood of this point, Equation 9.2.1 allows little deviation in the residual, despite a possibly considerable step length.

Equation 9.2.1 fails because the model's ability to predict r_i depends on the step $\Delta x_{[k]}$, not on the change in r_i predicted by the model. The step length of interest, $\|\Delta x_{[k]}\|$, does not produce proportionate changes in $|\hat{r}_{i[k]}|$, or in $|r_{i[k]} - \hat{r}_{i[k]}|$.

To express the right hand side of Equation 9.2.1 in terms of the step $\Delta x_{[k]}$ itself, substitute $r_{i[k]} - \hat{r}_{i[k]} = -\nabla r_{i[k]}^T(x - x_{[k]})$ from Equation 2.2.12:

$$|r_i - \hat{r}_{i[k]}| \le \alpha_i \cdot |\nabla r_{i[k]}^T(x - x_{[k]})|$$

Using $a^Tb = \|a\| \cdot \|b\| \cdot \cos\theta$ [Strang88 §3.2],

$$|r_i - \hat{r}_{i[k]}| \le \alpha_i \cdot \|\nabla r_{i[k]}\| \cdot \|x - x_{[k]}\| \cdot |\cos\theta_{i[k]}|$$

where $\theta_{i[k]}$ is the angle between the step and the residual gradient. The counterexamples above set the right hand side to zero by choosing steps $\Delta x_{[k]}$ perpendicular to the residual gradient ∇r_i.

Since $0 \le |\cos\theta| \le 1$, the second form suggests bounding the divergence by

(9.2.2) $$|r_i - \hat{r}_{i[k]}| \leq \alpha_i \cdot \|\nabla r_{i[k]}\| \cdot \|\Delta x_{[k]}\|$$

Equation 9.2.2 requires a small error compared to the distance from the point where the model was formed, and uses the magnitude of the vector of the $\partial r_i/\partial x_j$ to scale changes in x onto changes in r_i.

Another reasonable scale factor uses the largest magnitude of the $\partial r_i/\partial x_j$ to relate changes in x to changes in r_i:

(9.2.3) $$|r_i - \hat{r}_{i[k]}| \leq \alpha_i \cdot \|\nabla r_{i[k]}\|_\infty \cdot \|\Delta x_{[k]}\|$$

By Equation 4.1.6, this expression allows less deviation in the model $\hat{r}_{i[k]}$ than does Equation 9.2.2. The discussion below considers only the length of $\nabla r_{i[k]}$ as a scale factor.

Bounding Jacobian model error

Alternately, to avoid problems of scale, lump all the changes $|r_{i[k]} - \hat{r}_{i[k]}|$ into a single distance in the residual space, $\|r_{[k]} - \hat{r}_{[k]}\|$. For example, the test

(9.2.4) $$\|r - \hat{r}_{[k]}\| \leq \alpha \cdot \|r_{[k]} - \hat{r}_{[k]}\|$$

requires a small overall deviation compared to the overall predicted change in the residuals. Because the step $\Delta x_{[k]}$ changes some residual value in the linearized model, Equation 9.2.4 does not share with Equation 9.2.1 the difficulty of a zero expected change in the model.

Where Equation 9.2.2 focuses on the predictive capability of an individual residual model, producing n distinct error bounds, Equation 9.2.4 defines one bound on an overall error, the length of the vector difference between r and $\hat{r}_{[k]}$. Clearly an algorithm may require individual divergence estimates, for example if it changes residual weights or otherwise seeks to modify its search direction based on the performance of specific residual models. On the other hand, an algorithm seeking only to control an overall trust region might use a single test on Equation 9.2.4 instead of using multiple tests on Equation 9.2.2. The test could replace descent-based evaluation rules by deciding whether to extend, accept, or reject a trial step, based on the divergence of the model used to generate it.

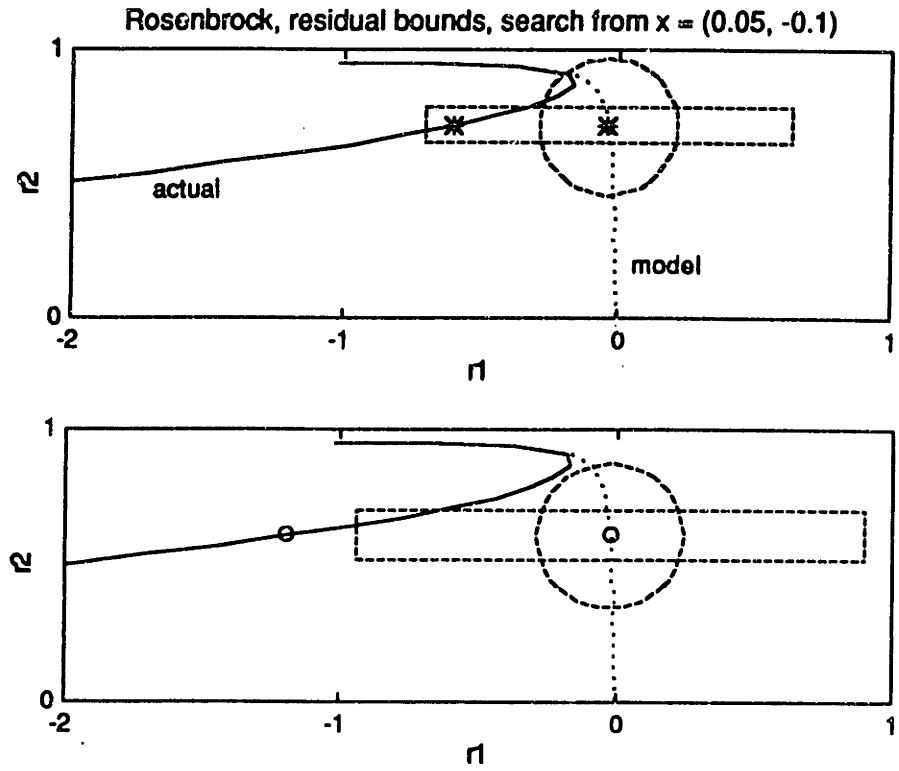

Figure 9.2.1. Bounds on the actual residual values, calculated with $\alpha = 0.25$ in Equations 9.2.2 and 9.2.4, for two points on the planar hook search path of Figure 7.2.1. The search starts from $x_{[0]} = (0.05, -0.1)^T$ on the two-dimensional Rosenbrock function.

Figure 9.2.1 applies both bounds, with $\alpha = 0.25$, to the planar hook search starting at $x_{[0]} = (0.05, -0.1)^T$ on the two-dimensional Rosenbrock function. (Figure 7.2.1 shows the search path itself.) Figure 9.2.1 plots the residual trajectories, as in Figure 9.1.1, and adds the bounds for the two trial points, x_{t1} and x_{t2}.

The residual observations in the upper plot correspond to $x_{t1} = (0.2878, 0.0228)^T$. At this point the Jacobian model predicts $\hat{r}_{[0]}\{x_{t1}\} = (-0.0344, 0.7122)^T$, as shown on the model curve. In fact the residuals give $r\{x_{t1}\} = (-0.6, 0.7122)^T$. The deviation in the model comes entirely from the first residual equation; $\hat{r}_{2[k]}$ captures the linear second residual equation exactly.

This first trial point passes the 25% divergence test imposed by Equation 9.2.2, but fails that of Equation 9.2.4. The bounds they impose differ in shape and extent because: (1) Equation 9.2.2 limits each residual independently, defining a rectangle of acceptable points about the predicted value, while Equation 9.2.4 bounds the overall deviation, defining a circle; and (2) Equation 9.2.2 scales the range of acceptable values independently in each residual dimension, while Equation 9.2.4 avoids scale factors.

Note the bounds hold only for the starting point $x_{[0]}$. Other starting points would generate different models and different steps to x_{t1}, changing the bounds (in the language of Section 8.2, the bounds define an acceptance criterion with relative, rather than absolute, significance). Thus, strictly speaking the trial point x_{t1} does not satisfy Equation 9.2.2, rather the residual models formed at $x_{[0]}$ satisfy the error bound when evaluated at x_{t1}.

Also note that although the figure interprets the limits as if centered about the model prediction $\hat{r}_{[0]}\{x_{t1}\}$, it might as well center them at the actual residual, $r\{x_{t1}\}$. In either

case, the bounds limit the distance allowed between the two points, for the model to be considered accurate.

The lower plot of Figure 9.2.1 shows bounds on the residual model evaluated at a trial point $x_{t2} = (0.3926, 0.0346)^T$. At this slightly greater step length along the planar hook curve, both Equations 9.2.2 and 9.2.4 allow greater errors in the residual models than at x_{t1}. However $\hat{r}_{1[0]}$ performs so poorly that the actual residuals do not meet the bounds established by either equation.

Scaling Jacobian model error bounds

In Figure 9.2.1, Equation 9.2.4 gives a circular bound on the difference between the actual and modeled residual vectors. Of course in problems of dimension greater than n = 2, Equation 9.2.4 establishes spheroids about the model point.

A spheroidal limit does not scale the residuals differently, but distributes the entire allowed deviation $\alpha \cdot \|r_{[k]} - \hat{r}_{[k]}\|$ equally among all the residual directions. To introduce scaling to Equation 9.2.4, first write the left hand side

$$\|r - \hat{r}_{[k]}\| = \sqrt{(r - \hat{r}_{[k]})^T (r - \hat{r}_{[k]})}$$

and note that this term gives the distance from $\hat{r}_{[k]}\{x_t\}$ to $r\{x_t\}$. Next, define a scaling matrix

$$S_{[k]} = \text{diag}\{s_{1[k]}, s_{2[k]}, \ldots, s_{n[k]}\} \tag{9.2.5}$$

with positive scale factors on the diagonal. Like $W_{[k]}$ in the weighted r-square norm, $S_{[k]}$ is symmetric, and its square root may be found by taking the root of its diagonal elements individually. Finally, take the weighted distance

$$\sqrt{(r - \hat{r}_{[k]})^T S_{[k]} (r - \hat{r}_{[k]})} = \|\sqrt{S_{[k]}} \cdot (r - \hat{r}_{[k]})\|$$

between the actual and predicted residuals.

The matrix $\sqrt{S_{[k]}}$ scales the i^{th} residual axis by a factor $\sqrt{s_{i[k]}}$ [Strang §2.6], and so scales the model error as well. To scale the errors in Equation 9.2.4, creating ellipsoidal bounds, introduce $S_{[k]}$ to its left hand side:

$$(r - \hat{r}_{[k]})^T S_{[k]} (r - \hat{r}_{[k]}) \le \alpha^2 \cdot \|r_{[k]} - \hat{r}_{[k]}\|^2 \tag{9.2.6}$$

Increasing $s_{i[k]}$ permits a smaller error $|r_i - \hat{r}_{i[k]}|$ for a given right hand side, and so contracts the bound in the i^{th} residual direction.

Unfortunately, multiplying $S_{[k]}$ by a constant expands or shrinks the ellipsoid without altering its shape-- the limits along each residual axis expand or shrink in proportion to one another. Thus, the bound established by Equation 9.2.6 depends on the overall scaling of $S_{[k]}$ as well as on the relative magnitudes of its diagonal elements.

To avoid this scale dependency, either: (1) introduce $S_{[k]}$ on the right hand side as well; or (2) choose the diagonal elements of $S_{[k]}$ in order to make the ellipsoid extend to predetermined boundaries.

The first option, introducing $S_{[k]}$ on the right hand side of Equation 9.2.6, produces a bound invariant to the scaling of $S_{[k]}$:

$$(r - \hat{r}_{[k]})^T S_{[k]} (r - \hat{r}_{[k]}) \le \alpha^2 \cdot (r_{[k]} - \hat{r}_{[k]})^T S_{[k]} (r_{[k]} - \hat{r}_{[k]}) \tag{9.2.7}$$

However, changing a single $s_{i[k]}$ not only changes the proportions of the ellipsoid along the i^{th} axis, it may also affect its absolute extent in every residual direction.

Where on the left hand side of Equation 9.2.7 increasing an $s_{i[k]}$ contracts the ellipsoid along the i^{th} axis, on the right it increases the size of the ellipsoid by an amount proportional to the distance between $r_{i[k]}$ and $\hat{r}_{i[k]}$. These changes do not necessarily cancel each other: if the residual model predicts no change in the i^{th} residual, increasing $s_{i[k]}$ does not increase the overall divergence allowed by the right hand side-- though the ellipsoidal bound still contracts along the i^{th} axis (in the other dimensions, the ellipsoid remains unchanged).

Conversely, if the search path expects a large change in a residual, then increasing $s_{i[k]}$ contributes an increased $s_{i[k]} \cdot |r_{i[k]} - \hat{r}_{i[k]}|^2$ to the overall deviation allowed by the right hand side. Thus increasing $s_{i[k]}$ expands the ellipsoid in directions other than along the i^{th} residual axis, since it allows greater divergence overall. Moreover, if $\hat{r}_{i[k]}$ performs fairly well, then the small deviation $|r_i - \hat{r}_{i[k]}|$ takes up very little of the overall divergence limit, and increasing $s_{i[k]}$ allows greater deviations in some other residuals before Equation 9.2.7 flags the model as diverging.

The second option, choosing the elements of $S_{[k]}$ in Equation 9.2.6 to establish a desired extension in each residual direction, solves the problem of scale by adjusting $S_{[k]}$ to the scale of the right hand side. Therefore the particular value of the right hand side no longer matters. Anticipating that Equation 9.2.2 will set the desired extent of the ellipsoid along each residual axis, change the right hand side of Equation 9.2.6 to bound the overall error by

$$(r - \hat{r}_{[k]})^T S_{[k]} (r - \hat{r}_{[k]}) \le \alpha^2 \cdot \|\Delta x_{[k]}\|^2 \tag{9.2.8a}$$

Equation 9.2.8a retains the ellipsoidal limits defined by the left hand side of Equation 9.2.6, and the direct dependence on $\|\Delta x_{[k]}\|$ from the right hand side of Equation 9.2.2. Choosing $s_{i[k]}$ to make $\sqrt{s_{i[k]}} \cdot |r_i - \hat{r}_{i[k]}| = \alpha \cdot \|\Delta x_{[k]}\|$ when $|r_i - \hat{r}_{i[k]}| = \alpha_i \cdot \|\nabla r_{i[k]}\| \cdot \|\Delta x_{[k]}\|$, the ellipsoid's axes extend to the limits set by Equation 9.2.2 when

$$s_{i[k]} = \frac{1}{\|\nabla r_{i[k]}\|^2} \tag{9.2.8b}$$

assuming $\alpha_i = \alpha$ for each evaluation of Equation 9.2.2.

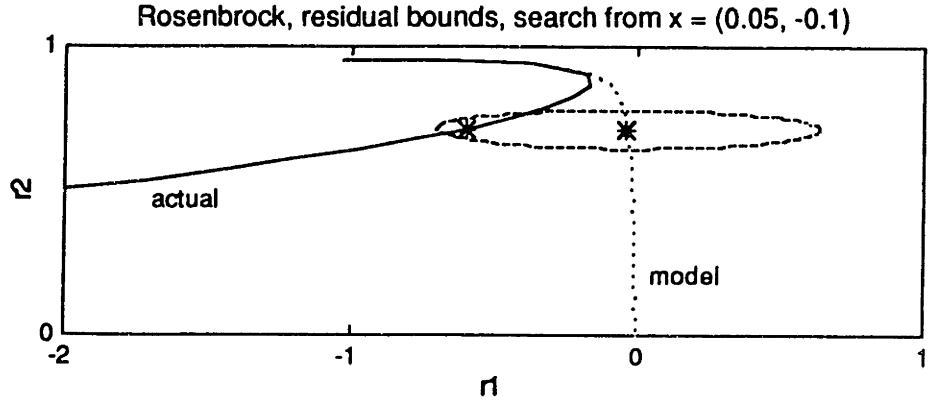

Figure 9.2.2. Bounds on the actual residual values, calculated with $\alpha = 0.25$ in Equation 9.2.8a, for a point on the planar hook search path of Figure 7.2.1. Equation 9.2.8b gives the scaling matrix $S_{[k]}$.

Figure 9.2.2 shows the bound calculated by Equation 9.2.8, with $\alpha = 0.25$, for the same test point x_{t1} used in the upper plot of Figure 9.2.1. The scaling matrix flattens the circle bound of Equation 9.2.4 into an ellipse, and this ellipse, like the rectangular bounds from Equation 9.2.2, encloses the actual residual. At the point x_{t1}, the residual models formed at $x_{[0]}$ satisfy the 25% deviation limit defined by Equation 9.2.8.

Comparing the two forms of ellipsoidal bound, the scale invariance of $S_{[k]}$ in Equation 9.2.7 would make it useful for testing divergence when the scale factors are set independently of the bound itself. For example, the weighted double dogleg algorithm might replace the descent-based evaluation criteria with Equation 9.2.7, using $S_{[k]} = W_{[k]}$, where $W_{[k]}$ comes from one of the weighting rules of Section 8.3. On the other hand, a Newton-Raphson direction step implies no particular choice of $W_{[k]}$, and Equation 9.2.7 introduces considerable ambiguity to the choice of appropriate scale factors. In this case either Equation 9.2.2 or its lumped counterpart, Equation 9.2.8, probably is more appropriate.

Tradeoffs in residual model performance

To choose between the individual and lumped bounds, recall that only along the residual axes do the ellipsoidal limits of Equation 9.2.8 extend as far as the individual limits of Equation 9.2.2. Off the axes, the limits of Equation 9.2.2 remain unchanged; the error allowed for one residual model does not depend on the errors expressed by others.

Equation 9.2.8, on the other hand, decreases the error allowed in one residual model as the divergence of others increases. Thus Equation 9.2.8 enforces a kind of tradeoff between divergences, by disallowing trial points which push too many residual models to the far reaches of their predictive capability.

The lumped bounds-- i.e., Equations 9.2.7 and 9.2.8-- achieve this tradeoff by treating the right hand side as a reservoir of allowed divergence, on which the scaled error terms of the left hand side may draw as needed. For instance, the right hand side of Equation 9.2.7 accumulates divergence contributions from the scaled distance each residual model steps from its starting point.

Since their right hand sides lump together the changes in the model as a result of the step, while their left hand sides distribute the allowed divergence among the various residual models, these evaluation rules may be seen as exchanging good performance in one residual model for poor performance in another. Consequently, the cuboid limits of Equation 9.2.2 enforce a less stringent divergence bound than the ellipsoidal limits of comparable extent given by Equation 9.2.8. It does not follow therefore that Equation 9.2.2 represents a less meaningful test. While a single estimate of the divergence may be more convenient, the residual models operate independently of each other. The fact that a step pushes one residual model to the limit of its predictive capability (as defined by Equation 9.2.2) does not necessarily preclude similarly pushing another model with the same step.

Lumped bounds in Newton-Raphson direction

Figures 9.2.1 and 9.2.2 depict the error bounds as regions of acceptable values, centered at the predicted residual, which the actual residual may take. The figures show predicted and actual residuals for a trial point on the planar hook curve, but the same geometric interpretation holds for trial points placed by any means.

A trial point in the Newton-Raphson direction puts the modeled residual on the line between $r_{[k]}$ and $r = 0$. From Equation 2.2.9a, a step $\mu \cdot \Delta x_{[k]:NR} = -\mu \cdot J_{[k]}^{-1} r_{[k]}$, so the model $\hat{r}_{[k]}$ of Equation 2.2.8 predicts $\hat{r}_{[k]:NR} = r_{[k]} - \mu \cdot J_{[k]} \cdot J_{[k]}^{-1} r_{[k]}$ or

$$\hat{r}_{[k]:NR} = (1 - \mu) r_{[k]}$$

in the Newton-Raphson direction. As μ increases from 0 to 1, the predicted residual changes from $r_{[k]}$ to 0. This simple residual model leads to a straightforward geometric interpretation of the lumped bounds. The scale invariance of Equation 9.2.7 gives a more intuitive result, but Equation 9.2.8, with its identical left hand side, follows the same logic.

Evaluate Equation 9.2.7 in the Newton-Raphson direction. Since $r_{[k]} - \hat{r}_{[k]:NR} = \mu \cdot r_{[k]}$, the bound requires

$$(r - (1-\mu) r_{[k]})^T S_{[k]} (r - (1-\mu) r_{[k]}) \leq \alpha^2 \mu^2 \cdot r_{[k]}^T S_{[k]} r_{[k]}$$

or

$$\frac{r^T S_{[k]} r}{r_{[k]}^T S_{[k]} r_{[k]}} - 2(1 - \mu) \frac{r_{[k]}^T S_{[k]} r}{r_{[k]}^T S_{[k]} r_{[k]}} + (1 - \mu)^2 \leq \alpha^2 \mu^2$$

in the Newton-Raphson direction. Clearly $S_{[k]}$ in the first term acts like $W_{[k]}$ in the weighted r-square norm, even if a method uses $W_{[k]}$ to choose a steepest descent search direction and uses an $S_{[k]} \neq W_{[k]}$ to evaluate a trial step in that direction. For convenience, call $\emptyset_{(s)}$ the scaled r-square norm $r^T S r$, just as in Equation 4.5.4a. Then the first term gives $\emptyset_{(s)} / \emptyset_{(s)[k]}$, the value of the scaled r-square at the trial point, relative to its value at the start of the search.

Finding this term on the left hand side implies that the divergence estimate includes a descent requirement; as shown below, this is not strictly correct.

In the second term, use $S = (\sqrt{S})^T (\sqrt{S})$, and $a^T b = \|a\| \cdot \|b\| \cdot \cos\theta$, to express

$$\frac{r_{[k]}^T S_{[k]} r}{r_{[k]}^T S_{[k]} r_{[k]}} = \frac{(\sqrt{S_{[k]}} \cdot r_{[k]})^T (\sqrt{S_{[k]}} \cdot r)}{\|\sqrt{S_{[k]}} \cdot r_{[k]}\|^2} = \frac{\|\sqrt{S_{[k]}} \cdot r\|}{\|\sqrt{S_{[k]}} \cdot r_{[k]}\|} \cos\theta$$

where θ is the angle between $\sqrt{S_{[k]}} \cdot r_{[k]}$ and $\sqrt{S_{[k]}} \cdot r$. Here, $S_{[k]}$ transforms the residual space, stretching or compressing the i^{th} residual axis by $\sqrt{s_{i[k]}}$ [Strang §2.6]. Thus, θ measures the angle between $r_{[k]}$ and r in the transformed residual space. For this Newton-Raphson direction step, $\hat{r}_{[k]}$ lies in the same direction as $r_{[k]}$, so θ also measures the angular deviation between the actual and predicted residuals, in the transformed space. (The same linear transformation interprets the first term as the squared length of r in the transformed residual space, divided by the squared length of the stretched and compressed $r_{[k]}$.)

Substituting, the bound becomes

$$\frac{\emptyset_{(s)}}{\emptyset_{(s)[k]}} - 2(1 - \mu) \sqrt{\frac{\emptyset_{(s)}}{\emptyset_{(s)[k]}}} \cos\theta \leq \alpha^2 \mu^2 - (1 - \mu)^2 \qquad (9.2.9a)$$

since $\|\sqrt{S_{[k]}} \cdot r\| = \sqrt{r^T S_{[k]} r} = \sqrt{\emptyset_{(s)}}$. To complete the square on the left hand side, add $(1-\mu)^2 \cdot \cos^2\theta$ to both sides. Then since $1 - \cos^2\theta = \sin^2\theta$, the bound requires

$$\left(\sqrt{\frac{\emptyset_{(s)}}{\emptyset_{(s)[k]}}} - (1 - \mu) \cdot \cos\theta \right)^2 \leq \alpha^2 \mu^2 - (1 - \mu)^2 \cdot \sin^2\theta \qquad (9.2.9b)$$

to accept a step $\mu \cdot \Delta x_{[k]:NR}$ in the Newton-Raphson direction.

From Equations 9.2.9, note:

(1) At $\mu = 0$, where $\emptyset_{(s)} = \emptyset_{(s)[k]}$ and $\theta = 0°$, the bound reduces to $0 \leq 0$, as expected for the start of a search.

(2) At $\mu = 1$, were the model predicts $\hat{r}_{[k]} = 0$, the bound reduces to a pure descent requirement, $\emptyset_{(s)} \leq \alpha^2 \cdot \emptyset_{(s)[k]}$. For instance, $\alpha = 0.25$ requires the step to reduce $\emptyset_{(s)}$ by 94%. To enforce a 90% reduction, use $\alpha \approx 0.32$; $\alpha = 0.5$ requires a 75% reduction.

(3) At intermediate steps in the Newton-Raphson direction, $0 < \mu < 1$, the bound penalizes large angular deviations between r and $\hat{r}_{[k]}$.

(4) At intermediate steps, the bound does not reward reductions in $\emptyset_{(s)}$; in fact it penalizes a step for decreasing the magnitude of some residuals by a greater amount than predicted.

The following subsections address these last two points in greater detail.

Penalty for angular deviation

The linearized residual models predict that a step in the Newton-Raphson direction places the actual residuals on the line between r = 0 and $r_{[k]}$. As the Newton-Raphson search progresses from $\mu = 0$, nonlinearities in the residual relations move r off of this line, increasing the angle between r and $r_{[k]}$ as seen from r = 0. The angle θ, measured in the transformed residual space $\sqrt{S_{[k]}} \cdot r$, increases also, and its cosine decreases from 1.

At a given μ and a fixed value of $\emptyset_{(s)}$, large angles θ decrease the likelihood of satisfying Equation 9.2.7, especially as θ increases past 90°, giving a negative cosine and a positive contribution from the second term on the left hand side of Equation 9.2.9a. Geometrically, $\theta > 90°$ means the actual residual overshoots r = 0 in the scaled residual space.

However, as μ increases towards 1, the effect of a fixed angular deviation on the bound becomes smaller. That is, the likelihood that a fixed angular deviation in the actual residuals will cause a trial point to fail the divergence test, decreases as the step length to the trial point increases. Thus, overshooting r = 0 exacts no performance penalty at the full Newton-Raphson step, $\mu = 1$, but a large penalty for short steps.

Penalty for length deviation

The linearized residual models predict that a step in the Newton-Raphson direction moves the actual residuals closer to r = 0 in direct proportion to the step length. Since Equation 9.2.7 checks for divergence in the models, it penalizes reductions in the length of r beyond that predicted by the model (of course, it also penalizes larger than expected residual lengths).

In the transformed residual space, $\sqrt{\emptyset_{(s)}}$ gives the length of the residual vector. Therefore the divergence test penalizes excessive deviations in this length. For a given μ and a fixed angular deviation θ, the best value of $\emptyset_{(s)}$ sets the left hand side of Equation 9.2.9b to zero by

$$\sqrt{\emptyset_{(s)}} = (1 - \mu) \cdot \cos\theta \cdot \sqrt{\emptyset_{(s)[k]}}$$

Note that the best length varies with the angular deviation; with no angular deviation, θ = 0°, the best length comes from $\sqrt{\emptyset_{(s)}} = (1 - \mu) \cdot \sqrt{\emptyset_{(s)[k]}}$, or $r = (1 - \mu) \cdot r_{[k]}$, as expected. As θ increases, the length which minimizes the left hand side of Equation 9.2.9b decreases

(however, since the right hand side also decreases, increasing θ does not compensate for bad predictions of the length).

Values of $\emptyset_{(s)}$ significantly larger or smaller than expected violate the divergence limit. In an extreme case, suppose a trial point finds $r = 0$, and $\emptyset_{(s)} = 0$, before reaching the full Newton-Raphson step. Then Equation 9.2.9b requires $(1 - \mu)^2 \cdot \sin^2\theta + (1 - \mu)^2 \cdot \cos^2\theta \leq \alpha^2\mu^2$, or $(1 - \mu) \leq \alpha \cdot \mu$, or $\mu \geq 1/(1+\alpha)$. Thus if $\alpha = 0.5$ then a search in the Newton-Raphson direction which finds $r = 0$ would fail the divergence test if $\mu < 2/3$.

Descent exemption from divergence test

Equation 9.2.7 tests for divergence, and if for example a step $\Delta x_{[k]} = 0.1 \cdot \Delta x_{[k]:NR}$ expects to set $\hat{r}_{[k]} = 0.9 \cdot r_{[k]}$, but in reality finds $r = 0$, then the residuals do diverge from the model. A divergence test such as Equation 9.2.7 checks whether a trial point exceeds the trust length of the model, but does not necessarily indicate that the trial point ought to be rejected.

In practice, the divergence tests might determine whether to double the trust length (e.g. if the trial point satisfies a relatively tight divergence criterion such as $\alpha = 0.25$), and whether to accept a step (e.g. if the trial point satisfies a relatively loose criterion such as $\alpha = 0.5$, or even greater). However, to reject a step ought also to require that it fail some absolute measure of performance, such as a descent-based test on the appropriate residual norm.

Suppose, for instance, that a trial step satisfies Equation 9.2.7 with a divergence of 10%. Based on this result, an algorithm might double the trust length. If the new trial step gives a large divergence, say 90%, but reduces the scaled r-square norm associated with $S_{[k]}$ from its value at the first trial point, then the algorithm should consider accepting the second trial point (though probably it should return the trust region to its original length).

Measuring divergence

Equations 9.2.2, 9.2.7, and 9.2.8 not only bound the error allowed in a trial step, they also define α as a measure of the divergence induced by a step. For example, rearranging Equation 9.2.2 defines

$$\alpha_{i[k]} = \frac{|r_{i[k+1]} - \hat{r}_{i[k]}\{x_{[k+1]}\}|}{\|\nabla r_{i[k]}\| \cdot \|\Delta x_{[k]}\|} \tag{9.2.10}$$

as a measure of the divergence in the k^{th} model of each residual, resulting from the step from $x_{[k]}$ to $x_{[k+1]}$. Similarly, Equation 9.2.8 defines an overall divergence

$$\alpha_{[k]} = \frac{1}{\|\Delta x_{[k]}\|} \cdot \sqrt{\sum_{i=1}^{n} \left(\frac{|r_{i[k+1]} - \hat{r}_{i[k]}\{x_{[k+1]}\}|}{\|\nabla r_{i[k]}\|} \right)^2} \tag{9.2.11}$$

The overall divergence is just the two-norm of a vector of the residual divergences measured by Equation 9.2.10:

$$\alpha_{[k]} = \sqrt{\sum_{i=1}^{n} \alpha_{i[k]}^2} \tag{9.2.12}$$

Examples

Figures 9.2.3 to 9.2.5 below compare the overall divergence associated with the double dogleg algorithm, against the divergence from rule 9 in the weighted double dogleg algorithm. Each figure takes one test problem from Section 8.4, and shows the overall divergence at the end of each iteration, calculated by Equation 9.2.11. (For convenience, the iteration index in the figures runs from 1 to the number of Jacobian evalutions, rather than from 0 to n_j - 1.)

The figures compare rule 9 to the double dogleg since in general rule 9, which introduces the normalized gradients, is the simplest of the weighting rules which consistently outperform the double dogleg algorithm across all test problems and starting points.

The iterative sequence from each method divides roughly into two stages. At first the method takes steps less than the Newton-Raphson length, as it adjusts the trust region according to the updating rules from Section 3.6. During this stage, α measures how well the trust region pushes the residual models to the limits of their predictive capabilities. The updating rules may err too conservatively or too liberally. If conservative, they leave the trust region too short; presumably then a small value of α indicates the iteration did not extract as much use from the Jacobian model as possible. If liberal, the rules overextend the trust region, creating consistently large values of α. In this analysis, a single large value of α probably indicates that the iteration decreased the residual norm by an unexpectedly large amount, rather than indicating a systematic failure of the updating rules.

In the second stage, the method converges on the solution, taking the full Newton-Raphson step at each iteration. During this stage, α should decrease rapidly towards zero as the linearized models predict the result of a full Newton-Raphson step with greater and greater accuracy. Thus, a small α no longer indicates the incomplete use of the Jacobian model-- rather, it validates the descent method's decision to take the full Newton-Raphson step.

Rosenbrock function

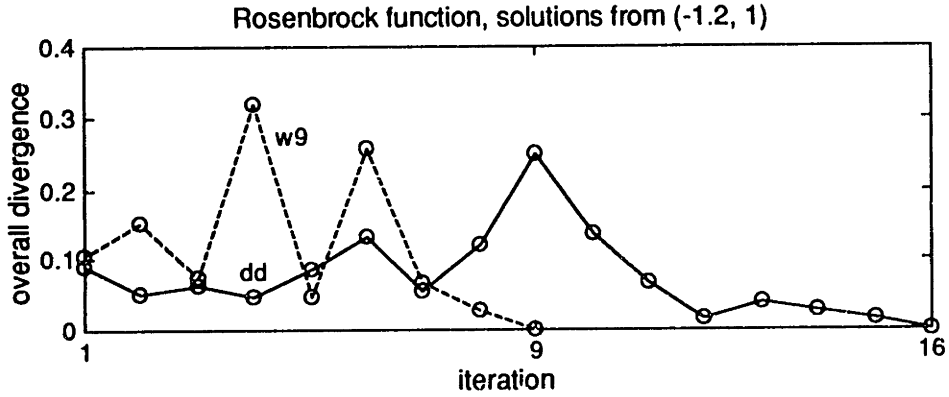

Figure 9.2.3. Overall divergence at each iteration for two solutions of the Rosenbrock function from (-1.2, 1)^T. The linear second residual equation gives zero divergence in Equation 9.2.10, so the overall divergence also gives the divergence of the first residual model.

Figure 9.2.3 shows data from two solutions of the two-dimensional Rosenbrock function from (-1.2, 1)^T. From Figure 8.4.6, the double dogleg solution requires 16

Jacobian evaluations to solve the problem, while rule 9 of the weighted double dogleg requires 9.

In the double dogleg solution, iterations 1 through 14 take less than the full Newton-Raphson step; only the last two iterations take the full Newton-Raphson step. Similarly, rule 9 of the weighted dogleg solution uses the full step only for the eighth and ninth iterations. For both solutions, the divergence of successive iterates begins to decrease even before they enter the second stage of the solution.

The fact that rule 9 allows a greater average divergence during its first stage than does the double dogleg, suggests that it converges more quickly because it allows greater step lengths, and hence uses the Newton-Raphson step more completely (recall two full Newton-Raphson steps can solve the Rosenbrock function from any starting point). However the weighted double dogleg also proposes different steepest descent directions, so possibly the difference resides in the step directions as well as in their lengths.

The Powell badly scaled function and trigonometric function examples, below, make the case for both possible explanations. In the Powell badly scaled function example, rule 9 outperforms the double dogleg by allowing longer steps. In the trigonometric function example, the differing step direction choice probably best explains the performance difference between the two algorithms. Whichever explanation holds for the Rosenbrock function example, to judge by the small divergence it forces in the residual models, the double dogleg algorithm seems to restrict the step length unnecessarily compared to the rule 9 weighting.

Powell badly scaled function

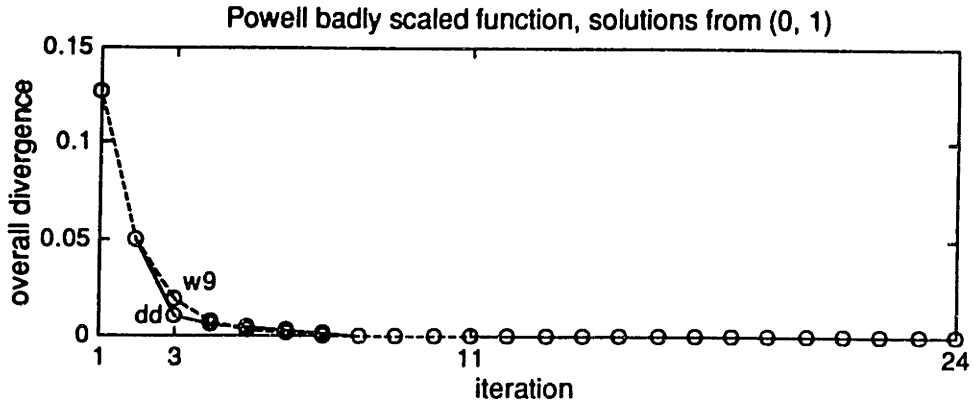

Figure 9.2.4. Overall divergence for a double dogleg and a rule 9 weighted dogleg solution of the Powell badly scaled function from $(0, 1)^T$.

Solving the Powell badly scaled function from $(0, 1)^T$, the double dogleg requires 24 Jacobian evaluations, as opposed to 11 for rule 9 in the weighted double dogleg algorithm. In Figure 9.2.4, the divergence associated with the steps for each method are virtually indistinguishable. However, a critical difference separates the two series of divergence measures.

Using rule 9, the weighted double dogleg solves the problem using the full Newton-Raphson step at every iteration. The divergence measurements depicted in Figure 9.2.4 show that the full Newton-Raphson step-- the optimal step from the point of view of the residual models-- creates only small errors in the residual models.

On the other hand, the double dogleg method allows only a few of the iterations to take the full Newton-Raphson step. The bulk of the divergence measurements shown in Figure 9.2.4 come from steps in the first stage of the iterative solution. Therefore these measurements show that the double dogleg algorithm fails to exploit the models, by not taking longer steps despite the high quality of the predictions they give.

Compare the two sequences during the first few iterations. At iterations one and two, both algorithms take the full Newton-Raphson step. At the end of the second iteration, the double dogleg method halves the trust region for the next iteration. From Equation 3.6.9, this means the model $f_{(2)}$ overestimated the actual decrease in $\emptyset_{(2)}$ by more than a factor of ten. Since the weighted dogleg method does not halve the trust region as a result of the same step, the model $f_{(w)}$ predicts the actual decrease in $\emptyset_{(w)}$ within the factor of ten. Thus an overall deviation of 5% in the residual models (in the second iteration from Figure 9.2.4) can sometimes trigger the trust length decrease, and sometimes not, depending on the weights assigned the residuals.

Continuing on to the third iteration, the weighted double dogleg allows the full Newton-Raphson step. Meanwhile the double dogleg evaluates $\emptyset_{(2)}$ at its shorter trust length, and based on the result places a second trial point at the full Newton-Raphson step. However it rejects the Newton-Raphson point for giving a greater value of $\emptyset_{(2)}$ than at the shorter step. Therefore at the third iteration, despite starting from the same point as the weighted method, the double dogleg takes a shorter step and induces a smaller overall deviation.

Note that this third iteration provides a specific example where the descent criterion itself favors a shorter step on the double dogleg curve (in fact, on the cutback portion of the curve) to the full Newton-Raphson step.

From Figure 9.2.4, if the double dogleg method tested its first trial step of the third iteration using Equation 9.2.8, it would conclude, for any reasonable value of α, that the residual models predicted the actual residuals sufficiently well to double the trust length. In addition, the resulting Newton-Raphson step also satisfies any reasonable divergence limit, as shown by the rule 9 curve in the figure. A step acceptance mechanism based on Equation 9.2.8 (or Equation 9.2.11) would allow the same step as the rule 9 weighted algorithm-- even though the different weights of the double dogleg method define a different search path for arriving at that full Newton-Raphson step.

Following the double dogleg sequence to the end of the solution, every subsequent step suffers from the shortened trust length. Some of these steps, like that of the third iteration, attempt to double the trust length, but then reject the new trial step for having a greater norm than at the shorter step. Other iterations do not even attempt to change the trust length. Finally at iteration 23 the Newton-Raphson length becomes short enough to fall within the unchanged trust length, and the method finishes in two full Newton-Raphson steps.

Trigonometric function

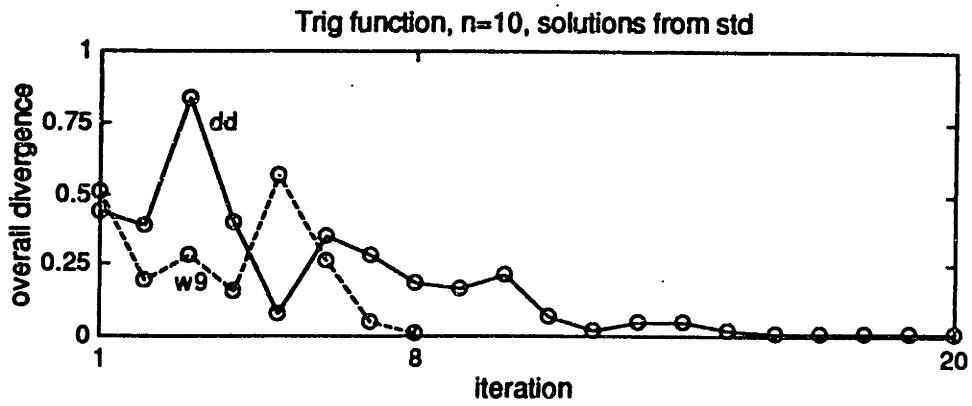

Figure 9.2.5. Overall divergence for two attempted solutions of the ten-dimensional trigonometric function from its standard starting point. The double dogleg solution does not terminate at $n_j = 20$, but continues for 71 Jacobian iterations before stagnating.

Figure 9.2.5 shows the divergence calculated by Equation 9.2.11 for each iteration of a double dogleg and of a weighting rule 9 solution of the ten-dimensional trigonometric function. From the standard starting point, the double dogleg method stagnates after 71 Jacobian evaluations. The weighted dogleg algorithm converges after 8 Jacobian evaluations.

Unlike the Powell badly scaled function in Figure 9.2.4, the trust length alone does not explain the difference between the two solutions, since the weighted dogleg does not take the full Newton-Raphson step until its last two iterations. Unlike the Rosenbrock function in Figure 9.2.3, the difference cannot be attributed to the divergence each step creates, since the double dogleg solution allows on average the same or greater divergence during its first ten iterations as does the weighted dogleg in its first six.

If anything, the divergences reported for the trigonometric function are unreasonably high. Compared to the data from Figures 9.2.3 and 9.2.4, both solution methods use the residual models at much further extremes of their predictive ability than they do in the Rosenbrock and Powell badly scaled functions. In no case would $\alpha = 1$, a 100% divergence, appear reasonable, yet the double dogleg accepts a descent step which causes a divergence of 84%, in its third iteration. Furthermore both solution methods create divergences between 25% and 50% in their first iterations.

(Interestingly, iterations 9 and 10 of the double dogleg search, with divergences of 16% and 21% respectively, both take a full Newton-Raphson step; these are the only double dogleg iterations which allow the full step, since the sequence stagnates.)

These observations, along with the fact that the double dogleg solution stagnates, imply that weighting rule 9 succeeds on this problem because it defines a superior search path. That is, the essential difference between the methods does not come from some varying ability to maintain a reasonable level of divergence when not taking the full Newton-Raphson step, rather it comes from the differing properties of the $g_{(2)}$ and the $g_{(w)\{9\}}$ search directions.

Summary of divergence measure experience

Much of the work on divergence measures above represents conjecture rather than accepted fact or tested theory. Nevertheless, the analysis and examples begin to indicate

how a trust region method might replace the descent based evaluation rules from Section 3.6 with a divergence based set of rules:

(1) Define a lower bound on the divergence in the overall model. Increase the trust length, up to that of the full Newton-Raphson step, to exploit the residual model more fully when the overall divergence falls below this bound.

(2) Define a maximum acceptable divergence. Reduce the trust length during an iteration, if necessary, to meet this upper bound.

(3) Supplement the upper divergence bound with a descent test on a residual norm, possibly a weighted norm with $W_{[k]} = S_{[k]}$. Use this test to avoid rejecting a step at an increased trust length if it violates the maximum divergence bound, but delivers a smaller residual norm than at the original step length.

(4) Reformulate the rules for updating the trust length at the end of a step, basing them on the overall divergence $\alpha_{[k]}$ rather than on the predictive capacity of $f_{(w)[k]}$.

(5) Retain a search path which allows rational alternate search directions when the Newton-Raphson direction fails. Probably the same paths which best avoid stagnation in the weighted double dogleg algorithm make likely choices.

(6) Consider using the Newton-Raphson direction for short steps, but define clear criteria for identifying the failure of these steps, and for switching back to the alternate search path.

9.3 Detecting Divergence Between Steps

The divergence measures of Section 9.2 apply to the Jacobian model formed during a single iteration of an equation solving algorithm. This section identifies a possibility for detecting the divergence of the method as a whole, by comparing the Jacobian models from successive iterations.

Need for divergence tests between steps

Divergence measures such as Equations 9.2.10 and 9.2.11 quantify the errors between the residual functions, and their behavior as predicted by models formed at the beginning of a step. The discussion above speculates that setting trust lengths by a divergence measure can help insure the step makes full use of the model without exceeding its predictive capability.

Unfortunately, limiting the divergence within a step does not guarantee the convergence of the method.

Consider a search strategy which follows the Newton-Raphson direction only. Section 2.2 describes how an extremum in a single residual function may dominate a descent-based search in the Newton-Raphson direction. Replacing the descent requirement with a divergence limit does not avoid the problem, which arises from the shallow slope of the tangent hyperplane near the extremum. Thus the method must provide alternate search directions in the event the Newton-Raphson direction fails.

Any of the double dogleg, planar hook, or weighted double dogleg algorithms suggest alternate step choices. The weighted dogleg in particular defines search paths which avoid stagnation at local minima. Therefore the new algorithm has only to identify the failure of the Newton-Raphson direction.

The convergence results of Section 2.2 relate the steps made by successive iterates of a successful Newton-Raphson sequence. This suggests identifying the failure of a Newton-Raphson sequence by the changes between steps. For instance, an algorithm

might use the divergence measures of Section 9.2 to exercise a Jacobian model during a step, and choose its search path for each iteration based on changes between successive Jacobian models.

Newton-Raphson convergence result

Equation 2.2.22 states that sufficiently close to x**, at worst each Newton-Raphson step halves the distance to the solution:

$$\|x_{[k+1]} - x^{**}\| \leq \frac{1}{2} \|x_{[k]} - x^{**}\|$$

This implies that if $\Delta x_{[k+1]:NR}$, the Newton-Raphson step at $x_{[k+1]}$, predicts a solution more than half as far from $x_{[k+1]}$ as it is from $x_{[k]}$, then the Newton-Raphson sequence is not yet sufficiently close to x**.

Since the Newton-Raphson step predicts $x^{**} = x_{[k+1]} + \Delta x_{[k+1]:NR}$, direct substitution gives

$$\|\Delta x_{[k+1]:NR}\| \leq \frac{1}{2} \|x_{[k]} - x_{[k+1]} - \Delta x_{[k+1]:NR}\|$$

or, from Equation 2.2.9b,

$$\|\Delta x_{[k+1]:NR}\| \leq \frac{1}{2} \|\Delta x_{[k]:NR} + \Delta x_{[k+1]:NR}\| \tag{9.3.1}$$

for a presumed bound on a converging Newton-Raphson sequence. If the Newton-Raphson step at $x_{[k+1]}$ doubles back on the step taken from $x_{[k]}$, then some cancellation occurs in $\Delta x_{[k]:NR} + \Delta x_{[k+1]:NR}$, reducing the bounding step length. On the other hand, if both Newton-Raphson steps line up, they give an increased bound. Thus, the step length allowed by a converging sequence depends on the angle between the current step and the previous step.

Bounding step length of converging sequence

Call this angle ß. Explicitly, $ß_{[k+1]}$ gives the angle between $\Delta x_{[k+1]:NR}$ and $\Delta x_{[k]:NR}$. Consecutive Newton-Raphson steps in the same direction have $ß_{[k+1]} = 0°$, while a Newton-Raphson step directly opposed to that preceding it defines $ß_{[k+1]} = 180°$.

Using ß, write the distance $\|\Delta x_{[k]:NR} + \Delta x_{[k+1]:NR}\|$ from Equation 9.3.1 as

$$\sqrt{(\|\Delta x_{[k]:NR}\| + \|\Delta x_{[k+1]:NR}\| \cdot \cos ß_{[k+1]})^2 + \|\Delta x_{[k+1]:NR}\|^2 \cdot \sin^2 ß_{[k+1]}}$$

or

$$\sqrt{\|\Delta x_{[k]:NR}\|^2 + 2 \cdot \|\Delta x_{[k]:NR}\| \cdot \|\Delta x_{[k+1]:NR}\| \cdot \cos ß_{[k+1]} + \|\Delta x_{[k+1]:NR}\|^2}$$

Then Equation 9.3.1 gives

$$3 \cdot \|\Delta x_{[k+1]:NR}\|^2 - 2 \cdot \|\Delta x_{[k]:NR}\| \cdot \|\Delta x_{[k+1]:NR}\| \cdot \cos ß_{[k+1]} - \|\Delta x_{[k]:NR}\|^2 \leq 0$$

or, using the quadratic formula,

$$\|\Delta x_{[k+1]:NR}\| \leq \|\Delta x_{[k]:NR}\| \cdot \frac{\cos ß_{[k+1]} + \sqrt{3 + \cos^2 ß_{[k+1]}}}{3} \tag{9.3.2}$$

for a converging sequence.

From Equation 9.3.2, if $ß_{[k+1]} = 0°$, i.e. if the Newton-Raphson steps line up, then $\|\Delta x_{[k+1]:NR}\| \leq \|\Delta x_{[k]:NR}\|$ in a converging sequence. On the other hand, if the next

Newton-Raphson step takes the exact opposite direction from that just completed, then $\beta_{[k+1]} = 180°$, and $\|\Delta x_{[k+1]:NR}\| \le \|\Delta x_{[k]:NR}\|/3$. Thus, a converging Newton-Raphson sequence can undo no more than a third of the previous step.

To find $\cos\beta_{[k+1]}$, use

$$\cos \beta_{[k+1]} = \frac{\Delta x_{[k]:NR}{}^T \Delta x_{[k+1]:NR}}{\|\Delta x_{[k]:NR}\| \cdot \|\Delta x_{[k+1]:NR}\|} \tag{9.3.3}$$

[Strang88 §3.2]. Equations 9.3.2 and 9.3.3, taken together, require about the same computational overhead as Equation 9.3.1. An algorithm which has later use for $\Delta x_{[k]:NR}$ requires an extra n-vector of storage to find the sum on the right hand side of Equation 9.3.1.

Bounding step length of diverging sequence

Equation 9.3.1, though derived for a sequence of points generated by Newton-Raphson's method, does not apply particularly to full Newton-Raphson steps. However generated, any two successive steps satisfying Equation 9.3.1 could come from an iterative sequence that at each step halves the distance to the solution (note this is not the same as halving the step length at each iteration). Thus, the expression

$$\|\Delta x_{[k+1]}\| \le \frac{1}{2} \|\Delta x_{[k]} + \Delta x_{[k+1]}\|$$

qualifies any two steps as possibly belonging to a converging sequence.

Conversely, the test may disqualify two steps from possibly belonging to a converging sequence. To convert the expression to a test for nonconvergence, it should allow a greater margin of uncertainty, for example by a factor of two increase in the bound. Then finding

$$\|\Delta x_{[k+1]}\| > \|\Delta x_{[k]} + \Delta x_{[k+1]}\| \tag{9.3.4a}$$

or

$$\|\Delta x_{[k+1]}\| > \|\Delta x_{[k]}\| \cdot \left(\frac{2}{3}\right)\left(\cos\beta_{[k+1]} + \sqrt{3 + \cos^2\beta_{[k+1]}}\right) \tag{9.3.4b}$$

would indicate that $x_{[k]}$, $x_{[k+1]}$, and $x_{[k+2]}$ do not come from a converging sequence.

Alternately, the bound might allow a factor of two margin of uncertainty when the second step points back to $x_{[k]}$, and $\cos\beta_{[k+1]} = -1$, and a factor of 4 margin when the second step keeps the same direction as the first, $\cos\beta_{[k+1]} = 1$. Scaling linearly between the two limits, it could test

$$\|\Delta x_{[k+1]}\| > \|\Delta x_{[k]}\| \cdot \left(\frac{\cos\beta_{[k+1]} + \sqrt{3 + \cos^2\beta_{[k+1]}}}{3}\right)(3 + \cos\beta_{[k+1]}) \tag{9.3.5}$$

Again, two steps satisfying Equation 9.3.5 would be assumed to come from a nonconverging sequence.

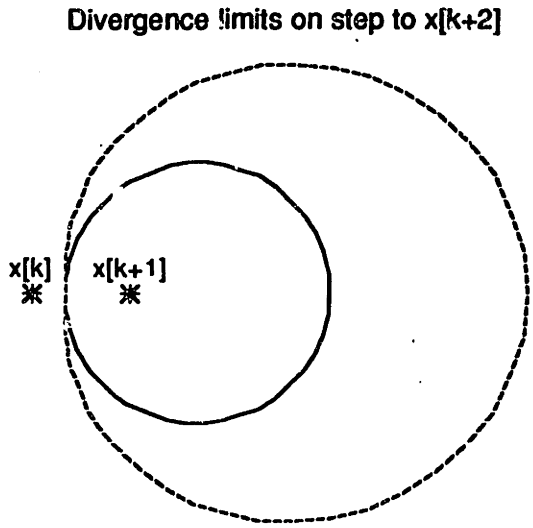

Figure 9.3.1. Limits on a step $\Delta x_{[k+1]}$, calculated by Equation 9.3.4 (solid) and Equation 9.3.5 (dashes), for which the iterates $x_{[k]}$, $x_{[k+1]}$, and $x_{[k+2]}$ may be considered to come from a converging method.

Neither Equations 9.3.4 or 9.3.5 can be justified formally; both place somewhat arbitrarily limits on the step choices which may be considered to come from a converging sequence. Figure 9.3.1 shows the bounds they place on $x_{[k+2]}$ for fixed iterates $x_{[k]}$ and $x_{[k+1]}$. Points falling within one of the bounds depicted would be accepted as possibly belonging to a converging sequence.

In practice, an algorithm would not use either expression to restrict the length of $\Delta x_{[k+1]}$. Instead, it would place $x_{[k+2]}$ by some other means, for example limiting the divergence during the step using a residual norm and the descent rules of Section 3.6, or using an overall divergence bound such as Equation 9.2.8. After accepting $x_{[k+2]}$, it might apply Equation 9.3.4 or Equation 9.3.5 to determine whether to change the step selection strategy in the next iteration. In particular, an algorithm might initially follow the Newton-Raphson direction at each iteration, and use one of these bounds to determine when to switch to a steepest descent-based search path such as that of the weighted double dogleg algorithm.

CHAPTER 10
FUTURE WORK

This chapter summarizes the thesis, indicating the most likely avenues for extending the ideas, and for further testing the methods, it develops.

Section 10.1 surveys the background material, highlighting the critical arguments leading up to the new algorithms. The section concludes by indicating the possibilities for future work, unrelated to equation solving, suggested by the background chapters.

Section 10.2 discusses the new algorithms themselves, the planar hook and the weighted double dogleg methods of solving nonlinear equations. A review of their motivation and development leads to a list of work relating directly to extending and refining these methods.

Section 10.3 reviews the work towards replacing the residual norm as a measure of divergence. The conjectural nature of much of that material makes the list of future work more general and broadly-defined than the specific suggestions of the preceding sections.

10.1 Review of Background Material

Because most of the insights and new methods presented in the thesis depend on a strict separation between the theories of equation solving and of function minimization, the early, background, chapters represent a significant contribution to the overall argument.

Standard theories

Chapter 2 presents Newton-Raphson's method of solving systems of nonlinear equations. Newton-Raphson's method linearizes the equations into the Jacobian matrix and solves the resulting linear system to find a new trial point. The Newton-Raphson sequence can diverge by taking wild steps at subsequent iterations; a standard method of detecting and controlling this type of divergence enforces a descent requirement on some residual norm, usually r-square.

Chapter 3 presents the theory and methods of minimizing a general cost function. It develops the two basic search strategies: the steepest descent, which follows the gradient of the cost function; and Newton's method, which minimizes a quadratic model of the cost function. The chapter ends with the double dogleg method, which selects trial iterates along a piecewise-linear search path defined by both the steepest descent and Newton search directions.

Chapters 2 and 3 develop the disciplines of equation solving and function minimization along parallel but distinct lines. Reviewing the standard mathematics of their subject areas, they establish notational conventions which express the strong correspondence between both theories, without conflating them. For instance, where Chapter 2 describes Newton-Raphson's method for solving a system of nonlinear equations, Chapter 3 discusses Newton's method for minimizing a scalar cost function. Later chapters maintain the distinction, separating Newton-Raphson's method from Newton's method, residual norms from cost functions, and equilibrium problems from minimization problems, by direct and unambiguous reference to the conventions established in these early chapters.

Besides establishing a clear terminology and notation, Chapters 2 and 3 interpret their methods geometrically whenever possible. Later chapters return to these physical

models, not only to explain their mathematical results on an intuitive level, but also to motivate new equation-solving methods. In addition, Chapter 3 contributes new search directions for function minimization: the net direction resulting from successive steepest descent searches; and, by extension, the projected net direction, which uses the Hessian information without the computational expense associated with Newton's method.

Conflation of standard theories

Many sources, failing adequately to distinguish between the theories of equation solving and of function minimization, formulate an equilibrium problem solver wholly as a function minimization routine. Designed to require that each step decrease a single residual norm, the resulting solver becomes a function minimization routine whose only connection to the underlying algebraic system consists of implementation details-- such as using the Newton-Raphson solution of the residual equations as a robust and efficient shortcut to finding the Newton minimizer of the residual norm.

Ultimately this approach exposes the solution technique to the same difficulties associated with any function minimization routine: (1) stagnation due to numerical effects in a flat cost function (even though a flat residual norm has no particular meaning for the underlying equations); and (2) convergence to local minima in the residual norm (i.e., convergence to minima which do not correspond to the solution of the equations). Section 4.1 discusses these difficulties.

Integration of standard theories

Chapter 4 makes the connection between the methods of Chapters 2 and 3 explicit. This paves the way for expressing the distinction between the two bodies of theory at the algorithmic, rather than at the compuational, level.

Section 4.3 defines a vector, p, whose elements, $p_i = \partial \emptyset / \partial r_i$, relate first-order changes in a general residual norm, $\emptyset$, to first-order changes in the residuals. Importantly, the entries of p depend only on the way the residual norm combines the residuals, without regard to the algebraic form of the residual equations themselves. Since the Jacobian matrix of Chapter 2 models the residual relations, this approach expresses the results of function minimization directly in terms of the vector p, which depends on the selected norm, and in terms of the Jacobian matrix and residual vector, which define the equilibrium problem.

Using these expressions, Section 4.3: (1) ties together the geometric models from the preceding chapters, showing how the direction of steepest descent for any residual norm follows from the geometry of the Jacobian matrix; (2) establishes conditions on reasonable residual norms; and (3) demonstrates that r-square belongs to a broad class of norms, the weighted sum of squares of residuals, whose Newton's method minimizer equals the Newton-Raphson step.

In addition, Chapter 4 discusses r-square, the one-norm, and the weighted sum of squares, expressing the function minimization results for these norms in terms of the linearized models of the residual relations. The one-norm does not fit the general function minimization framework well, because the linearized residuals give it zero curvature, and because its steepest descent direction may not be defined when a residual has zero magnitude.

Implementation

Chapter 5 brings together the background material on r-square, adapting the standard double dogleg algorithm to the solution of nonlinear systems. The implementation differs from that given in the literature, because it employs more direct mathematical expressions than does the published algorithm. Chapter 5 also shows how discretization effects limit the ability of a numeric method to identify the exact minimum in a line search. The result does not apply to equation solving, and so Chapter 5 does not pursue it any farther than to show that it estimates these discretization effects more accurately than a published, heuristic, bound.

Chapter 6 demonstrates, in an extended example problem, the major geometric and algorithmic steps of the preceding chapters.

Future work

Further research suggested by the background work could:

(1) Explore the use of the net steepest descent direction in a minimization method. The method would follow the net direction taken by two successive line minimizations, or near-minimizations, in the steepest descent direction, and would not require a Hessian matrix of second derivatives.

(2) Explore the use of a modified net steepest descent direction in a minimization method. The method would follow a direction defined jointly by the current gradient and by the net direction from the previous two successive line minimizations, for example replacing the part of the current gradient parallel to the net direction by the scaled net direction itself, while retaining the orthogonal part of the current gradient. Again, the method would not require a Hessian matrix.

(3) Explore the use of the estimated net steepest descent direction in a minimization method. The method would at each iteration find the net direction resulting from two steepest descent minimizations on the Hesssian-based model formed at that iterate, and then search the estimated net direction.

(4) Test a new double-break-point search path in a minimization method. The estimated net steepest descent direction, like the steepest descent direction, has an estimated minimizer in the Hessian-based cost function model. This gives a point corresponding to the Cauchy point in the steepest descent direction, but at a greater step length. The method would replace the cutback point of the double dogleg curve with the estimated minimizer in the estimated net steepest descent direction. Two possible advantages would accrue: (a) the path, still ending at the Newton point, would add a third spatial dimension to the double dogleg path, by searching outside the plane of the steepest descent and Newton directions; and (b) for short trust lengths the method could avoid finding the computationally-expensive Newton direction.

(5) Test a method similar to (4) above, but which sets its second break point by the conjugate gradient method.

(6) Test the discretization result of Chapter 5 more extensively. In particular, use true scalar cost functions, rather than the r-square norm from an equilibrium problem, to insure against giving unfair advantage to the discretization result, derived from the quadratic model.

10.2 Review of New Algorithms

This section summarizes the motivation, development, and numeric experience of the planar hook and weighted double dogleg algorithms, developed in Chapters 7 and 8.

Planar hook algorithm

The planar hook algorithm, discussed in Chapter 7, modifies the double dogleg method of function minimization to exploit more fully the underlying structure imposed by the system of nonlinear equations. Its only change-- placing its trial points to minimize the simplified model of r-square derived in Chapter 4-- specializes a minimization technique to the approximate Hessian for r-square derived in Chapter 4. The resulting planar hook curve improves on the double dogleg search path in an otherwise identical trust region algorithm.

In general, the method has the same convergence properties as the double dogleg, solving the same problems from the same starting points, and stagnating or exceed its iteration limit where the double dogleg also does so. However the planar hook search path often allows the solution to proceed with fewer Jacobian and residual evaluations, and occasionally solves a problem for which the double dogleg path stagnates.

As reported in the literature, and verified by the results from the double dogleg algorithm, a nonlinear equation solver based on minimizing r-square can fail by stagnating, or may simply fail to converge in the specified number of iterations. From the numeric experiments reported in Chapter 7, placing test points on the planar hook curve can avoid stagnation and speed convergence in some cases, but ultimately cannot compensate for the inherent shortcomings of using r-square as the sole residual norm.

Criticisms of single norms

Chapter 8 criticizes the use of a single residual norm for evaluating a trial step in an equilibrium problem solution. In partial response to these criticisms, it develops a weighted double dogleg algorithm. This modification to the double dogleg does not change the search path definition, but changes the residual norm. This in turn affects the search path, since its first break point lies in the direction of steepest descent for the residual norm of choice.

Section 8.1 takes up the question of r-square and its seeming natural application to nonlinear equation solving. It argues that of the reasons usually cited for using r-square-- including some of the special properties that suit it to the double dogleg algorithm-- only its convenience and tractability justify the preference. The discussion concludes that r-square has no inherent connection to systems of nonlinear equations.

Section 8.2 centers on the theme that no single residual norm best measures the tradeoff between the residual elements during a single step of a nonlinear equilibrium problem solution. The section begins by examining the role of the norm. It argues that recasting the problem of solving an equilibrium problem as one of minimizing a particular residual norm: (1) redefines the role of the norm, from that of a local check against divergence in a step, to that of a global estimator of the quality of a trial iterate; (2) forces convergence to local minima or near-minima in that residual norm, even if the stagnation point has no particular attractive properties in the underlying algebraic system; and (3) implies that the steepest descent direction for the norm of choice defines a reasonable serach direction for the equilibrium problem.

Section 8.2 rejects these positions as holdovers from function minimization theory, arguing that since the residual norm appears peripherally to the equilibrium problem,

instead of integrally in its definition, a nonlinear equation solver should not restrict itself to evaluating steps or to generating alternate search directions using a single residual norm.

Of these points, perhaps rejecting the third-- that the gradient of a residual norm necessarily gives an appropriate search direction when the Newton-Raphson step fails-- provides the most striking alternate viewpoint. The mathematics of function minimization lead immediately to the gradient (or steepest descent) direction when seeking to decrease a known cost function. In equation solving, on the other hand, the same mathematics apply to the residual functions individually. Therefore an algorithm can construct a reasonable search direction without reference to a residual norm. For instance, Section 8.2 appeals to the geometric models of the preceding chapters to suggest two alternate search directions: one for use when the expected step length is sufficient to zero the linearized residual models, and the other for shorter steps.

Given a search direction constructed by summing the gradients of the residual functions, the background work of Section 4.3 suggests an appropriate residual norm for evaluating that search direction. Section 8.2 shows that the two geometrically-derived search directions correspond to, among other norms, r-square and the one-norm, respectively.

Section 8.2 concludes with a discussion of the possible benefits and difficulties associated with changing the residual norm from iteration to iteration.

Weighted double dogleg algorithm

The weighted double dogleg algorithm proceeds naturally from the observations: (1) that no particular norm best measures progress towards the solution of an equilibrium problem; (2) that the tractability of r-square does make it particularly attractive for use in the double dogleg algorithm of function minimization; and (3) that r-square belongs to a broad class of residual norms sharing these same tractable properties.

Section 8.3 proposes using this broader class-- the weighted sum of squares of residuals-- as the cost function in the double dogleg algorithm. Chapter 4 derives all the necessary background material, in terms of a general weighting matrix W. Thus the bulk of Section 8.3 defines rules for choosing the residual weights in W.

Weighting rules

Section 8.3 develops weighting rules for use in the weighted double dogleg algorithm, both by applying the background material from previous sections, and by studying the results of numeric experiments using previous weighting rules. In general, this iterative process concludes:

(1) The standard r-square norm represents an incomplete application of a rational weighting strategy: choosing weights such that the steepest descent direction from each residual contributes to the overall gradient direction of the norm according to that residual's relative magnitude.

(2) Choosing weights to duplicate the steepest descent choice of the one-norm provides considerable resistance to stagnation. Since the weighted r-square norm can incorporate the steepest descent choice of the one-norm, without incurring the one-norm's problems with a zero Hessian or with nondifferentiability when a residual element is zero, its steepest descent direction is well-defined.

(3) The particular treatment accorded a zero residual in the one-norm weighting rule makes little difference in practice, probably because the algorithm rarely zeros any residual

until its last iterations, when the full Newton-Raphson steps ensure convergence for any reasonable choice of weights.

(4) Comparing the one-norm weighted r-square to the standard r-square norm, one of these two basic variations can solve any of the test problems, though neither can solve all the test problems. Based on this observation, many later weighting schemes attempt to mix the two basic methods.

(5) The most successful of these mixing strategies take up the geometric picture from Section 8.2, which proposes the gradient direction from r-square as a search direction when the expected step length matches or exceeds the distance required to zero the linearized models of the residuals, and which proposes the gradient direction from the one-norm otherwise. However, instead of switching between the norms, say using $\emptyset_{(2)}$ at one iteration and a $\emptyset_{(1)}$ weighting at the next, the rules mix them element-by-element. Thus, at each iteration the weighting rules compare the distance required to zero each linearized residual model to the length of the expected step, and choose that residual weight accordingly.

(6) If the standard double dogleg algorithm solves a test problem, then weighting the residuals to normalize the lengths of the Jacobian rows solves the problem in the same or fewer iterations. This result is consistent with the geometric interpretation developed in Section 8.2, because normalizing the Jacobian row lengths in the residual weights corresponds to finding the residual's relative magnitude when summing residual gradients.

(7) Normalizing the Jacobian row lengths does not improve the performance of the one-norm weighted r-square algorithms.

(8) The proper model for mixing the r-square and one-norm weighted r-square rules appears to involve the trust length, that is, the expected length of the next step the method will take. This is consistent with the geometric interpretation of Section 8.2. However, there appears to be considerable latitude in choosing where to make the transition between assigning weights by the r-square rule, and assigning them by the one-norm weighted r-square rule.

(9) The greatest difficulty with any rule lies in fully exploiting the Newton-Raphson direction. Generally the pure Newton-Raphson sequence does not diverge in any way obviously signalled by the number of Jacobian matrix evaluations it requires. In fact the double dogleg methods which require the fewest Jacobian evaluations solve the problem only about as well as does Newton-Raphson, and often not as well. The descent requirement, instituted to guard against divergence in the Newton-Raphson sequence, more often prevents a full Newton-Raphson step which would otherwise form part of a successful sequence.

(10) For a given problem, one of the weighting rules provides consistently better performance than the other rules, across all the starting points. The strength of the weighting rules which mix weights lies not in superior convergence, but in more consistent convergence to the problem solution.

(11) Merely changing the residual weights from iteration to iteration provides considerable resistance to stagnation. For example, the one-norm weighted r-square resists stagnation better than does the standard r-square-based double dogleg. Presumably, changing norms helps to avoid local minima in other, competing, norms (except for that minimum common to all the norms, at x** where r = 0).

(12) If the algorithm changes the weights too dramatically between iterations, it may allow an iteration to undo the steps made in those preceding it. When this happens, though

the method usually solves the problem, it tends to require an excessively high evaluation count to do so.

(13) Discontinuities in the relationship defining the residual weights at each iteration do not hurt the convergence properties of a method so much as do large changes in the residual weights assigned from one iteration to the next. Moreover, defining the residual weight by a continuous rule at each iteration does not necessarily provide stability in the weight from one iteration to the next.

Future work

Further research intended to extend or to further explore the planar hook and weighted double dogleg algorithms could:

(1) Explore the effect of using different termination conditions on the one-norm weighted r-square algorithms. Since these algorithms differ in how they treat a zero residual, their behavior depends on a user input, CRIT_INF_NORM. Section 8.4 reports results only for the default user input.

(2) Test the weighting rules on more sample problems. Of the eight problems used, only four-- the duct flow problem, the Powell badly scaled function, the Rosenbrock function, and the trigonometric function-- differentiate between the methods.

(3) Test the weighting rules on more sample problems which give residual errors. Of the eight problems used, only one-- the duct flow problem-- generates residual errors for trial points outside some well-defined region; the other problems define the residuals for all x. From the results reported for the duct flow problem, the possibility of residual errors clearly affects the success of the solution method.

(4) Test the weighting rules with a larger maximum iteration count. The successful weighting rules all avoid stagnation, but may terminate without finding a solution after 100 Jacobian evaluations, making it difficult to compare results.

(5) Investigate the weighting rules at the iteration level. Chapter 8 develops new weighting rules by identifying broad trends in the performance of earlier weighting rules, usually without analyzing the output from the trial runs on an iteration-by-iteration basis. A detailed investigation comparing the weights chosen to the resulting search path, and to the method's evaluation of trial points along that path, could yield further insight into the nature of the weighted sum of squares.

(6) Test a new double-break-point search path in an equation-solving method. The method would replace the cutback point of the double dogleg search path with the predicted minimizer in the estimated net direction from two steepest descent steps on $f_{(w)[k]}$, or with the result of two iterations of a conjugate gradient method. Because it can determine the new break point using only matrix-vector multiplications, the method could avoid calculating the Newton-Raphson direction when short trust lengths do not demand it. In large, sparse problems the time savings could be significant.

(7) Adapt the weighted double dogleg algorithm to use the planar hook search path. Loosely, the planar hook gives better trial points than the double dogleg path when minimizing r-square, while the weighted double dogleg algorithm chooses better residual norms than r-square. Given a weighted sum of squares chosen by one of the residual weighting rules of Chapter 8, find the optimal linear combination of the Newton-Raphson direction and the steepest descent direction for that weighted norm. The resulting weighted planar hook curve may give better trial points than the double dogleg path.

(8) Make the weighted double dogleg algorithm robust to a singular Jacobian. The code in Appendix 3 terminates for a singular Jacobian, but, as indicated in Chapter 8, this

condition does not prevent finding and searching the steepest descent direction of an appropriate norm. In the standard double dogleg algorithm, which seeks out local minima in r-square, finding a singular Jacobian probably indicates a zero gradient in r-square as well, so a search in the gradient direction probably would not provide an escape. The weighted r-square, on the other hand, always allows a selection of residual weights which give a nonzero gradient, unless $J_{[k]} = 0$.

(9) Test the various weighting rules with a secant update of the Jacobian [Dennis §8.1-8.4, Dahlquist §6.9.3]. The tests of Section 8.4 all use exact Jacobian evaluations at each iteration, but in practice, equation-solving algorithms may estimate changes to the Jacobian using the step changes $\Delta x_{[k]}$ and $\Delta r_{[k]}$.

(10) Re-examine the iterative search method for finding the parameters $\rho_{1[k]}$ and $\rho_{2[k]}$ in the planar hook path. Chapter 7 develops an iteration which chooses a trial ρ_1, and calculates ρ_2 accordingly. An algorithm based on choosing ρ_2 may place the trial iterate more quickly, since $\partial f_{(2)}/\partial \rho_2$ is likely to be greater than $\partial f_{(2)}/\partial \rho_1$. See Figure 7.2.1.

(11) Investigate the possibility of scaling the residual weighting matrix $W_{[k]}$ during computation, to avoid rounding errors. Though scaling $W_{[k]}$ does not change the value given by the test expressions for the double dogleg algorithm, numeric effects may cause computational problems for very large or very small $w_{i[k]}$.

(12) Implement more sophisticated trend-tracking rules for creating persistent weights in the weighted double dogleg algorithm. For instance, if the residual weights for a particular residual tend to decrease monotonically from iteration to iteration, then there is less incentive to stabilize the weights by averaging than if they change randomly.

10.3 Review of New Divergence Measures

This section summarizes the Chapter 9 work on divergence measures. These measures, constructed primarily to replace the descent requirement for avoiding divergence in a step, also promise to replace the descent-based trust region updating rules.

Newton-Raphson search direction

The primary motivation for developing an alternate means of detecting divergence in a step comes from the apparent problems the descent-based rules have in assessing the Newton-Raphson direction accurately. Section 9.1 discusses several indications of this difficulty.

Any evaluation rule attempting to constrain the divergence between the predicted and actual behavior of a residual norm ultimately tries to restrict the divergence between the individual residual equations and their models. The models used in the double dogleg, planar hook, and weighted double dogleg algorithms, $f_{(2)}$ and $f_{(w)}$, though apparently second-order, carry only function and slope information about the residuals. Therefore they do not model the meaningful second-order effects-- the curvature in the residual equations-- which cause the actual norms $\emptyset_{(2)}$ and $\emptyset_{(w)}$ to deviate from their idealized behavior.

Section 9.1 speculates that to exploit the Newton-Raphson direction to the greatest extent possible may require more than improving the rules for evaluating a long step in the Newton-Raphson direction. Newton-Raphson's method, based on local models of the residuals, forms a rational search direction for short steps. Therefore a nonlinear equation-solving algorithm should consider following the Newton-Raphson direction for short as well as long trust lengths. To do so may require some measure of the divergence of the

method between successive iterations, as well as a measure of the divergence of the residual models during an iteration.

Divergence measures

Section 9.2 develops measures of the divergence in the linearized residual models. Starting with the error $|r_i - \hat{r}_{i[k]}|$ between a residual function and its model, these measures directly assess the effect of neglecting second-order (and higher) residual terms.

To bound the error in a residual model by an acceptable value requires some sense of the extent to which a step $\Delta x_{[k]}$ perturbs the model. The predicted change in r_i does not adequately represent this perturbation, essentially because a model may predict zero change in the residual for a large step from the point where the model was formed. Section 9.2 uses the magnitude of the i^{th} row of the Jacobian, $\|\nabla r_{i[k]}\|$, to relate a step of length $\|\Delta x_{[k]}\|$ to a representative change in r_i. The magnitude of this i^{th} residual gradient aggregates the effect of each partial derivative $\partial r_i/\partial x_j$ on the residual.

Using the same ideas, Section 9.2 develops a limit on the overall error in the residual vector, replacing the vector of n bounds on the residual functions with a scalar bound on the Jacobian model as a whole. This new bound places a tighter restriction on the residual models, because for a given deviation limit a large error in one model precludes an equally large error in another.

The bounds express the error allowed in a model as a fraction of the change from the point where the model was formed. Therefore the bounds also define measures of divergence. For instance, the scalar bound gives a measure of the overall divergence a step induces in the Jacobian model, as a fraction of the step length (or, properly, as a fraction of the step length scaled by its relative ability to change the residuals). This overall measure is just the length of a vector of measured divergences in the n residual models.

Experience

Initial results using the overall divergence measure suggest:

(1) For a step in the Newton-Raphson direction, the measure assesses both: (a) deviations between predicted and actual values of an appropriate residual norm; and (b) deviations in the angle of the residual vector from its predicted angle. In other words, the error bound not only penalizes length deviations in the residual vector (as do the descent-based trust region updating rules), it also penalizes angular deviations.

(2) The measure can replace the descent-based trust length evaluation rules, but only if supplemented by a limited version of the descent requirement.

(3) Applying the measure to individual iterations of the algorithms in Chapters 5 and 8 indicates strengths and weaknesses of the step length and step direction choices of the various algorithms.

Future work

Further research on the divergence measures could:

(1) Continue comparing the overall divergence for successful versus unsuccessful solutions of a test problem. The comparisons should provide further insight into the differences between the nonlinear equation solvers.

(2) Compare the divergence measures for individual residual equations in the same successful and unsuccessful solutions.

(3) Test a particular rule in the weighted double dogleg algorithm. In general, the successful weighting rules choose gradient weights according to the relative magnitude of

the residual, i.e., gradient weights proportional to the residual magnitude and inversely proportional to the magnitude of its gradient. The steepest descent direction of the resulting norm sums the residual gradients roughly in proportion to the need to reduce that residual, assuming the models hold over the entire step length. Extending this idea, consider making the gradient weights also proportional to the relative difficulty the model experienced in predicting the results of the last step. Then if the method must take a short step, it tries to reduce the magnitude of the least tractable residuals. In other words, choose weights proportional to a residual's divergence in the last step. Then the current iteration places greater emphasis on reducing the magnitudes of residuals with less capable models. Conversely, if the residual model did well at the last iteration, the current iteration has no incentive to expend a short step minimizing that residual, since taking a full Newton-Raphson step at a later iteration will zero it. Note that this strategy may encounter difficulties with local extrema in individual residual equations, since near these extrema the residual model will fare poorly, and hence acquire greater weight in setting the steepest descent direction for the next iteration.

(4) Investigate strategies for replacing the descent-based evalution rules in a trust region algorithm. For example, identify divergence bounds appropriate for accepting, rejecting, and doubling the trust length, both during an iteration and in preparation for the next iteration.

(5) Seek rules for using the Newton-Raphson direction preferentially at short trust lengths, and for detecting when to switch to the steepest descent direction of an appropriate norm.

(6) Investigate using the individual divergence bounds to determine whether to reformulate the search path, using the same Jacobian, at a successful trial iterate. The double dogleg method, for example, doubles the trust length when the model $f_{(2)[k]}$ predicts $\emptyset_{(2)}$ well. If a solver finds that the Jacobian model predicts every residual with a small divergence, then instead of extending the step based on the same model, the solver could re-form the model using function values from the current residuals, but slopes from the old Jacobian (which presumably remain very nearly correct). The method essentially would start a new iteration, only without forming a new Jacobian, every time it places a trial step and finds very small divergences in every residual model.

(7) Consider using the individual divergence bounds to determine whether to update each row of the Jacobian on a case-by-case basis. A modular or component-based modeling environment such as IDA [Sahlin] or HVACSIM+ [Clark] groups the residual functions defining a component into a single computational routine. Such a solver can evaluate the residuals, and their gradients, semi-independently of one another. Therefore if the residuals grouped together in one component routine all have small divergences-- perhaps the component represents a linear process-- then the solver may avoid re-evaluating the corresponding rows of the Jacobian at the start of the next iteration.

(8) Consider using differences between the Jacobian models at successive iterations to detect divergence. Geometrically, the test would aim to identify a very small slope of changing sign in one or more residuals, which would indicate instability in the Newton-Raphson direction due to a local extremum. For example, consider comparing the Newton-Raphson direction implied by solving the current residuals with the previous Jacobian, against the Newton-Raphson direction for the current residuals and the current Jacobian. Wildly different step directions and step lengths, as defined for example by the convergence results for Newton-Raphson's method, could identify instabilities resulting from a local extremum in a single residual equation.

APPENDIX 1

REFERENCES

A1.1 Numeric Methods

Byrd. Richard H. Byrd, Robert B. Schnabel, and Gerald A. Schultz, "Approximate Solution of the Trust Region Problem by Minimization Over Two-Dimensional Subspaces." In *Mathematical Programming*, Series A, v. 40, n. 3, pp. 247-263. Mathematical Programming Society, Elsevier Science Publishers B.V., Amsterdam. April 1988.

Dahlquist. Germund Dahlquist and Åke Björck, *Numerical Methods*. Prentice-Hall, Englewood Cliffs NJ. 1974.

Dennis. J.E. Dennis Jr. and Robert B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs NJ. 1983. Also Society for Industrial and Applied Mathematics, Philadelphia PA. 1996.

Ellis. Robert Ellis and Denny Gulick, *Calculus with Analytic Geometry*. Harcourt Brace Jovanovich, San Diego CA. 1978.

Feng. Dan Feng, Paul D. Frank, and Robert B. Schnabel, "Local Convergence Analysis of Tensor Methods for Nonlinear Equations." In *Mathematical Programming*, Series B, v. 62, n. 2, pp. 427-459. Mathematical Programming Society, Elsevier Science Publishers B.V., Amsterdam. November 1993.

Fletcher. Roger Fletcher, *Practical Methods of Optimization. Volume 1: Unconstrained Optimization*. John Wiley and Sons Ltd., Chichester. 1980.

Golub. Gene Golub and Charles Van Loan, *Matrix Computations*, second edition. Johns Hopkins University Press, Balitmore MD. 1989.

Press. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, second edition. Cambridge University Press, Cambridge GB. 1992.

Schnabel. Robert B. Schnabel, "Unconstrained Optimization in 1981." In *Nonlinear Optimization 1981*, Proceedings of the NATO Advanced Research Institute, edited by M.J.D. Powell, pp. 3-16. Academic Press Inc., New York NY. 1982.

Shor. Naum Zuselevich Shor, *Minimization Methods for Non-Differentiable Functions*. Springer-Verlag, Heidelberg. 1985.

Strang86. Gilbert Strang, *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley MA. 1986.

Strang88. Gilbert Strang, *Linear Algebra and Its Applications*, third edition. Harcourt Brace Jovanovich, San Diego CA. 1988.

A1.2 Modeling

ASHRAE. *ASHRAE Handbook-- 1993 Fundamentals*. American Society of Heating, Refrigerating and Air-Conditioning Engineers Inc., Atlanta GA. 1993.

Axley. J. Axley and R. Grot, "The Coupled Airflow and Thermal Analysis Problem in Building Airflow System Simulation." In *ASHRAE Transactions*, v. 95, pt. 2, pp. 621-628. American Society of Heating, Refrigerating and Air-Conditioning Engineers Inc., Atlanta GA. 1989.

Bourdouxhe. Jean-Pascal H. Bourdouxhe, Marc Grodent, Jean Lebrun, and Claudio Saavedra, "A Toolkit for Primary HVAC System Energy Calculation-- Part 1: Boiler Model." In *ASHRAE Transactions*, v. 100, pt. 2, pp. 759-73. American Society of Heating, Refrigerating and Air-Conditioning Engineers Inc., Atlanta GA. 1994.

Braun87a. J.E. Braun, J.W. Mitchell, S.A. Klein, and W.A. Beckman, "Models for Variable-Speed Centrifugal Chillers." In *ASHRAE Transactions*, v. 93, pt. 1, pp. 1794-1813. American Society of Heating, Refrigerating and Air-Conditioning Engineers Inc., Atlanta GA. 1987.

Braun87b. J.E. Braun, J.W. Mitchell, S.A. Klein, and W.A. Beckman, "Performance and Control Characteristics of a Large Cooling System." In *ASHRAE Transactions*, v. 93, pt. 1, pp. 1830-1852. American Society of Heating, Refrigerating and Air-Conditioning Engineers Inc., Atlanta GA. 1987.

Clarke. J.A. Clarke, "Assessing Building Performance by Simulation." In *Building and Environment*, v. 28, n. 4, pp. 419-427. Pergamon Press, Oxford. October 1993.

Feustel. Helmut E. Feustel and Juergen Dieris, "A Survey of Airflow Models for Multizone Structures." In *Energy and Buildings*, v. 18, n. 2, pp. 79-100. 1992.

Gerhart. Philip M. Gerhart and Richard J. Gross, *Fundamentals of Fluid Mechanics*. Addison-Wesley, Reading MA. 1985.

Hensen. J.L. Hensen, "Towards an Integral Approach of Building and HVAC System." In *Energy and Buildings*, v. 19, pp. 297-302. 1993.

Incropera. Frank P. Incropera and David P. DeWitt, *Fundamentals of Heat and Mass Transfer*, third edition. John Wiley and Sons, New York NY. 1990.

Lorenzetti. David M. Lorenzetti, "Modeling Adjustable Speed Drive Fans to Predict Energy Savings in VAV Systems." In *Energy Impact of Ventilation and Air Infiltration*, Proceedings of the 14th AIVC Conference, pp. 269-77. Air Infiltration and Ventilation Centre, Coventry Great Britain. 1993.

Mills. Anthony F. Mills, *Heat Transfer*. Richard D. Irwin, Inc., Homewood IL. 1992.

Rodríquez. E.A. Rodríquez and F. Allard, "Coupling COMIS Airflow Model with Other Transfer Phenomena." In *Energy and Buildings*, v. 18, pp. 147-157. 1992.

Sauer. Harry J. Sauer Jr. and Ronald H. Howell, "Estimating the Indoor Air Quality and Energy Performance of VAV Systems." In *ASHRAE Journal*, v. 34, n. 7, pp. 43-50. July 1992.

A1.3 Software

Boisvert. Ronald F. Boisvert, "The Architecture of an Intelligent Virtual Mathematical Software Repository System." In *Mathematics and Computers in Simulation*, v. 36, n. 4-6, pp. 269-279. Elsevier Science Publishers B.V., Amsterdam. 1994.

Buzzi-Ferraris. Guido Buzzi-Ferraris, *Scientific C++: Building Numerical Libraries the Object-Oriented Way*. Addison-Wesley Publishing Company, Wokingham England. 1993.

Clark. D.R. Clark, *HVACSIM+ Building Simulation and Equipment Simulation Program Reference Manual and User's Guide*. NBSIR 84-2996. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg. 1985.

Hiebert81. K.L. Hiebert, "An Evaluation of Mathematical Software That Solves Nonlinear Least Squares Problems." In *ACM Transactions on Mathematical Software*, v. 7, n. 1, pp. 1-16. Association for Computing Machinery, New York NY. March 1981.

Hiebert82. K.L. Hiebert, "An Evaluation of Mathematical Software That Solves Systems of Nonlinear Equations." In *ACM Transactions on Mathematical Software*, v. 8, n. 1, pp. 5-20. Association for Computing Machinery, New York NY. March 1982.

Klein. Sanford A. Klein, William A. Beckman, and John A. Duffie, "TRNSYS-- A Transient Simulation Program." In *ASHRAE Transactions*, v. 82, pt. 1, pp. 623-33. American Society of Heating, Refrigeration and Air-Conditions Engineers Inc., Atlanta GA. 1976.

Lippman. Stanley B. Lippman, *C++ Primer*, second edition. Addison-Wesley Publishing Company, Reading MA. 1993.

Moré81. Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom, "Testing Unconstrained Optimization Software." In *ACM Transactions on Mathematical Software*, v. 7, n. 1, pp. 17-41. Association for Computing Machinery, New York NY. March 1981.

Moré82. Jorge J. Moré, "Notes on Optimization Software." In *Nonlinear Optimization 1981*, Proceedings of the NATO Advanced Research Institute, edited by M.J.D. Powell, pp. 339-352. Academic Press Inc., New York NY. 1982.

Sahlin. Per Sahlin and Axel Bring, "IDA Solver-- A Tool for Building and Energy Systems Simulation." In *Proceedings of the IBPSA Building Simulation '91, (Nice)*, pp. 339-348. International Building Performance Association. 1991.

Walton. George N. Walton, *AIRNET-- A Computer Program for Building Airflow Network Modeling*, NISTIR 89-4072. U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg MD. 1989.

APPENDIX 2
NOTATION

Symbols	
d	Search direction
e	Coordinate direction
f	Model of cost function or residual norm
g	Gradient of a scalar cost function or residual norm
G	Hessian of a scalar cost function or residual norm
i, j	Typically, components of a vector or matrix
J	Jacobian matrix of derivatives of residual equations
k	Typically, iteration
n	Typically, problem dimension
p	Gradient of a residual norm with respect to residuals
P	Hessian of a residual norm with respect to residuals
q	Vector of length 1, from orthogonal system
Q	Orthonormal matrix of vectors q
r	Residual vector in algebraic problem
s	Eigenvectors, or scale factor in divergence measure
S	Matrix of eigenvectors, or diagonal matrix of scaling factors
w	Residual weights in a weighted r-square norm
W	Diagonal matrix of residual weights
x	Independent variables
y	Independent variables in orthonormal eigenvector coordinates
α	Divergence measure, or scale factor in trust region method
δ	Trust length
ε	Machine precision, or a small value
η	Scales Newton point to cutback point in double dogleg method
λ	Eigenvalue of a matrix, or scale factor in double dogleg method
Λ	Diagonal matrix of eigenvalues
ρ	Scale factor in planar hook method
$\emptyset$	Scalar cost function or residual norm
μ	Stepsize in a line search

Operators	
d	Exact derivative
∂	Partial derivative
Δ	Change in quantity, especially a small variation
{ }	Function of arguments in brackets
$\|x\|$	Absolute value of scalar x
$\|\|x\|\|_i$	i-Norm of vector x
$\|\|x\|\|$	2-Norm of vector x
$\approx$	Approximately equal to
∇	Gradient operator
∇_r	Gradient operator with respect to dependent variables r

Subscripts	
$x_{[k]}$	Value of x at iteration k
x_i	Component i of vector x
$x_{i[k]}$	Component i of vector x at iteration k
$x_{(j)}$	Value of x association with residual norm j; see below
$x_{[k]:NR}$	Value of x specified by Newton-Raphson's method at iteration k
$x_{[k]:N}$	Value of x specified by Newton's method
$x_{[k]:SD}$	Value of x in steepest descent direction or at Cauchy point
$x_{[k]:CB}$	Value of x at cutback point on double dogleg curve
(1)	One-norm
(2)	R-square, square of the two-norm
(∞)	Infinity-norm
(w)	Weighted r-square norm
s	Symmetric part of a matrix

Overscripts	
$\hat{x}$	Estimate of, model of behavior of x

Superscripts	
x^T	Transpose of x
x**	Solution of algebraic system
x*	Local minimum in function minimization problem

APPENDIX 3
CODE FOR NUMERIC METHODS

This appendix lists code for the methods described in the thesis.

A3.1 Program Design

Figure A3.1.1 shows an approximate file hierarchy for the code implementing the numeric methods discussed in the thesis. The major rows of the figure trace an approximate order of file definition: code from the lower groups tends to depend on information or functions defined by files entered in higher groups.

	File name	Purpose
Main program	SOLVDRIV.CPP	Choose solution method and initial conditions.
Meta-information	PROBDEFN.HPP	Communicate with driver code.
	UTLDEPND.HPP	Localize implementation dependencies.
Utility	UTLINPUT	User input.
	UTLREPRT	Output.
	UTLTIMER	Time and count to evaluate performance.
Linear algebra	ARRAYOPS	Vector operations on arrays of real numbers.
	FULLJCBN	Unfactored square Jacobian matrix.
	PLUJCBN	PLU factorize a Jacobian matrix.
Numeric method	DBLDOGLG	Double dogleg with scalar equivalents.
	NEWTRAPH	Classic Newton-Raphson iteration.
	PLANARHK	Planar hook algorithm.
	STDDOGLG	Standard double dogleg implementation.
	TERMCOND	Check for termination conditions.
	WTDDOGLG	Double dogleg for weighted r-square.
Problem definition	PRBROYTR.CPP	Broyden tridiagonal function.
	PRDSCTBV.CPP	Discrete boundary value function.
	PRDSCTIN.CPP	Discrete integral equation function.
	PRDUCTFL.CPP	Duct flow problem.
	PRGENRIC.HPP	Generic header file for problem definitions.
	PRPOWSCL.CPP	Powell badly scaled function.
	PRPOWSNG.CPP	Powell singular function.
	PRROSNBK.CPP	Rosenbrock function.
	PRTRIGNO.CPP	Trigonometric function.
	PRWLCONV.CPP	Wall convection problem.

Figure A3.1.1 File hierarchy for the numeric methods. Major row divisions indicate levels of interdependence among the files. If the table does not list a three-character extension for a file, then two files exist, with both the .cpp and .hpp extensions.

File hierarchy

The main program file initiates the problem solution, passing user inputs and the problem definition functions to the solver.

Meta-information files establish a number of high-level structural conventions which govern the code files: (1) the type, e.g. float or double, of the variable and Jacobian data; (2) the return values of functions used to evaluate the residual and Jacobian data; (3) compile-time information for optimizing from one implementation to another; and (4) system-dependent run-time functions.

Utility files support generic tasks such as input and output.

Linear algebra files form the basis of the numeric methods, implementing array and Jacobian operations, such as vector addition and matrix-vector products.

Numeric method files implement particular equation-solving algorithms.

Problem definition files evaluate the residual and Jacobian functions. Each file defines a particular nonlinear algebraic problem.

User inputs

Broadly, the user affects the program at two stages: (1) at link time, by selecting one of the problem definition files; and (2) at run time, by supplying the starting point, termination conditions, and output options under which the program will operate.

The main program SOLVDRIV.CPP, a driver for the numeric methods, passes in as arguments to the numeric method function not only the starting point, termination conditions, and so on; but also the problem definition. It does this using pointers to the residual and Jacobian evaluation functions. This design makes possible a "reverse communication" interface in which the driver mediates between the solution method and the evaluation routines if necessary [Moré82 §1.5].

While SOLVDRIV.CPP initializes a problem and allows the user to select one of the numeric methods, each executable file references only one problem, specified at link time by including one of the problem definition files, e.g. PRBROYTR.CPP. All the problem definition files use the same function names, declared in the common header file PRGENRIC.HPP.

Under this program design, a user codes residual and Jacobian functions for each new problem, following the function prototypes given in PRGENRIC.HPP (the functions must also follow the rules for using the Jacobian matrix defined in FULLJCBN.CPP). Compiling the problem file and linking with object code from the driver and solution methods forms a complete executable file.

A more sophisticated driver might: (1) evaluate the residuals by assembling modular components, e.g. according to a user's run time specification; (2) support multigrid methods in problems whose defining equations do not change across a spatial discretization of varying resolution; (3) calculate the Jacobian by finite differences in cases where an analytic function is not available; (4) solve independent problems successively; or (5) solve an equilibrium problem as the initial-value calculation in an evolution problem.

Separation of problem definition and solution method

Except when a file name carries only one extension, i.e. either ".cpp" or ".hpp," Figure A3.1.1 does not list its extension. That is, the figure does not independently list the header files associated with each code file.

A header file declares the primary functions defined by the code, identifying their arguments and return values. Using the header information for functions defined elsewhere allows a code fragment to be compiled independently of the code for every function it calls.

Code for the numeric methods exists independently of the problem definition in the sense that its compilation does not require header files from the problem definition functions. Instead, the argument list of a numeric method specifies prototypes for the residual and Jacobian evaluation functions, thereby specifying the form these functions must take, if not their names. All other communication between the problem definition and the solution methods is specified by the meta-information files; see above.

Reusability of core code

The vector (array) and Jacobian code, intended for iterative calculation routines, rather than for a general matrix-vector environment such as MATLAB®, sacrifices a degree of error checking in the interest of run-time efficiency. Thus, responsibility for their correct use rests with the programmer of a numeric method or problem definition.

For example, in a general linear algebra function library, every vector and matrix would store its own dimension, and every vector-vector and vector-matrix operation would check that the dimensions agreed, every time the operation was called (see for example [Buzzi-Ferraris]). However, within a loop that adds the same two vectors at every iteration, verifying that the dimensions match each time merely adds to the computational overhead. Here, since every vector and matrix supports a single nonlinear problem, there is no reason to store the problem dimension separately for each one.

The linear algebra functions implemented here tend to check conditions which affect the way an operation is performed, and which might change from one iteration of a numeric routine to another-- for example, the matrix-vector multiplication routines check whether the matrix is factored. By contrast, they do not tend to check for the types of errors resulting from a programmer mistake, and which would persist from one iteration to the next. Thus a matrix-vector product routine does not insure that its two array arguments do not overlap.

In general, the user is responsible for: (1) ensuring that matrix and vector dimensions agree; (2) using unique arrays in function arguments where required; and (3) modifying the Jacobian properly. The header files for the linear algebra code files specify the rules governing their use.

Procedural versus object-oriented programming

Because the numeric method routines are highly iterative, much of the code is procedural. Thus, vector operations are handled by passing array addresses, and the common dimension, in the argument list of an array function:

```
double x[dim]; ...
norm = ArraySumSquares(dim, x);
ArraySum(dim, x, y, z);
```

In constrast, a Vector class might support the operations using member functions and operators:

```
Vector x(dim); ...
norm = x.SumSquares();
z = x + y;
```

The code presented here mixes procedural programming and abstract data types [Lippman §10.3]. It defines the Jacobian as a matrix object for two reasons: (1) to hide its internal representational details from the rest of the program; and (2) to allow derivation of factored classes from an unfactored class. In both cases, the actual program design is better suited to the nature of the problems at hand than to a general matrix environment: (1) The FullJacobian class compromises information-hiding by granting the user direct access to its matrix elements. This avoids unnecessary copying of user-generated data, but means that the user can violate the Jacobian integrity; and (2) Derivation allows the Jacobian evaluation function to take the base class FullJacobian as an argument, regardless of the derived class factorization used by the solution method. This makes the Jacobian evaluation code independent of the particular handling of the Jacobian (e.g. PLU or QR factorization), but is less efficient than a direct implementation with an established factorization.

The header files discuss the program design more completely.

Portability

Except for UTLDEPEND.CPP, all the code complies with the ANSI C++ standard [Lippman]. The header file UTLDEPND.HPP identifies some system and compiler dependencies, which for the numeric methods largely affect only speed of execution (for more information, see the header file). Moreover the code file allows nonstandard C++ code, for example to establish environmental dependencies such as the screen size.

Besides these system dependencies, all code from this appendix is, in theory, portable. It has been compiled using Symantec C++ v. 8.0.5 on a Macintosh 7100/66. Most of the code has also been compiled and run under Watcom C++ on a Dell 450/L.

A3.2 Code Files

The complete source code appears below, with files arranged alphabetically, except that the header file precedes the corresponding code file.

```
/* ====================< File ARRAYOPS.HPP >==========================
======================< rev 06/22/96 >================================

     Declares standard functions for working with arrays.

     Functions implemented:
(1) Swap array pointers.
(2) Copy and compare arrays.
(3) Add, subtract, and scalar-multiply arrays.
(4) Find maximum and minimum magnitudes stored in arrays.
(5) Find norms of vectors stored in arrays.
(6) Take inner products of vectors stored in arrays.
(7) Project one vector array onto another.
(8) Print an array to a stream.

     User preferences:
(1) Array copies are either called as functions, or expanded
iniine. The choice is made in the files which #include the header,
on a case-by case basis. The choice is made by calling either
the function or its _INLINE macro.

     User preferences set in other files:
(1) Arrays are of either floats or doubles. The choice is
made in "probdefn.hpp".
(2) Arrays are copied using either the string command memcpy,
or an element-by-element version. The choice is made in
"utldepnd.hpp".
(3) Arrays are compared using either the string command memcmp,
or an element-by-element version. The choice is made in
"utldepnd.hpp".
(4) Array functions can check for overflow or not. The choice
is made in "utldepnd.hpp".
(5) Vector-vector inner products are calculated either
directly or using a scaled version. The choice is made in
"utldepnd.hpp".

     Flat versus dimensional array functions:
     Some array functions can treat the array data purely
as a stream of numbers, without regard to any conceptual
divisions which may group data within the array. Functions
such as array addition, scalar multiplication, and finding
maximum values can treat the array according to a "flat"
model, in which the elements of the array(s) involved always
can be accessed in the same order in which they are stored,
that is, in a continuous sweep from the beginning of the
array to its end.
     The vector inner product, on the other hand, is
"dimensional" in the sense that the arrays involved are
intended to represent vectors. When this is the case, the
vector elements may not be stored next to each other in
the arrays. In particular this happens when one vector is
a column of a matrix which is stored by rows.
     A jump access mode, described below, is used to allow
the vector inner product to process noncontiguous array
elements.

     Vector access modes:
     Vectors can be stored in one of two modes: contiguous
or jump. A third common access mode, described here as harrow,
is not implemented.
     Contiguous array elements are accessed simply by moving
element-by-element through the array.
     In the jump mode, the vector elements are separated by
a constant number of array elements-- a jump distance or
offset. A jump offset of one fetches contiguous array elements.
     In the harrow mode, an array of pointers gives the
location of each element. There is one pointer for each
```

vector element. Unlike the jump mode, with harrow access the offset between array elements is arbitrary.

Tests (using Symantec C++ on Mac 680x0 and PPC processors) indicate that for a constant offset, harrow access is only marginally faster for small problems (vector lengths on the order of 3 to 5). For medium-size problems (vector lengths of 20), jump access is faster, even for small jumps (i.e. offsets of 3). As the offset becomes larger, the superiority of jump access becomes more pronounced.

Accordingly, harrow access is not implemented here; for non-square matrices the defining program should implement the vector-vector operations independently.

Array arguments in function calls:

The flat array operations take the name of the array pointer as the function argument. Thus if double x[] (or double *x) defines x as a pointer to an array of doubles, then the function call uses x (not x[] or &x). Since x is passed by value, the array function can modify x, making it point to a new location, without changing the array pointer in the environment of the calling function. (Of course when the array function changes *x, the contents of x, then the change still holds in the environment of the calling function).

When an array function should access but not change the values held in the array, the function prototype is written e.g. const double x[], to indicate that x is a pointer to an array of constant doubles. The compiler can coerce an array of non-constant doubles to match.

Jump access uses the same array calling rules. In addition, an integer offset is passed (by value) to the array function.

Sources:

Guido Buzzi-Ferraris, "Scientific C++: Building Numerical Libraries the Object-Oriented Way," Addison-Wesley, 1993.

Stanely Lippman, "C++ Primer," second edition, Addison-Wesley, 1991. C++ reference.

Gilbert Strang, "Linear Algebra and its Applications," third edition, Harcourt Brace Jovanovich, 1988. Basic linear algebra.

J.W. Daniel et al, "Reorthogonalization and Stable Algorithms for Updating the Gram-Schmidt QR Factorization," in Mathematics of Computation, v. 30, n. 136, October 1976, pp. 772-795. Gives a scaled vector length algorithm, on which the scaled inner product is based. Note Daniel's scaling uniformly scales all array elements based on the maxabs element of the array; the scaling used here only rescales when the term to be summed into the inner product is much larger than the inner product accumulated so far.

Setup:

```
VariableType x[10];
     //   VariableType is double or float, depending on typedef
     // declared in "probdefn.hpp". Recall double x[10]
     // makes x a pointer to double, pointing to the first
     // element in an array x[0], x[1], ..., x[9].
x[0] = 1.2; x[1] = -2.56; ...; x[9] = 89;
VariableType y[] = {0.2, 1.1, ..., 9.9};
int arrayCount = (sizeof x)/sizeof(double);
     // Indirect way to establish number of array elements.
VariableType z[10];
```

Array Management:

```
ARRAY_SWAP_INLINE(x,y); // Inline expansion swaps array pointers.
ArrayCopy(10,x,y); // Copy x to y.
ARRAY_COPY_INLINE(10,x,y);
ArrayCompare(10,x,y); // Return 1 if arrays are same, else 0.
```

ArrayOps.hpp

```
      Array addition and subtraction:
ArraySum(10,x,y,z); // z = x + y
ArraySum(10,x,y,x); // x = x + y
ArrayDifference(10,x,y,z); // z = x - y
ArrayDifference(10,x,y,y); // y = x - y.

      Array scalar multiples:
ArrayMinus(10,x,z); // z = -x;
ArrayMinus(10,x,x); // x = -x
double c = 4.6;
ArrayScalarProduct(10,c,x,z); // z = c*x
ArraySaxpy(10,c,x,y,z); // z = c*x + y

      Array magnitudes:
VariableType mag;
double dub;
mag = ArrayMax(10,x);
mag = ArrayMaxAbs(10,x);
mag = ArrayMin(10,x);
mag = ArrayMinAbs(10,x);
dub = ArrayNormOne(10,x);
dub = ArrayNormTwo(10,x);

      Vector-vector inner products:
double dub;
dub = ArraySumSquares(10,x);
dub = ArrayInnerProduct(10, x,1, y,10);

      Array stream print:
ArrayStreamPrint(10,x);
      //   Prints to cout.
      //   Print format controlled by variables declared in
      // utlreprt.hpp. If utlReportColsPerLine < 1, or > arrayCount,
      // the entire array is printed on one line. Elements are
      // separated by utlReportDelimiter.
      //   Printing unaffected by value of utlReportLevel.

=======================< File ARRAYOPS.HPP >======================== */

#ifndef ARRAYOPS_HPP
#define ARRAYOPS_HPP


#include "probdefn.hpp"
      // For typedef of VariableType.
#include "utldepnd.hpp"
      // For macros governing user preferences.


// =====< Array management: Swap >=====
#define ARRAY_SWAP_INLINE(ax,ay) \
      {VariableType *temp = ax; ax = ay; ay = temp;}

// =====< Array management: Copy elements >=====
void ArrayCopy(int arrayCount,
      const VariableType cpyFrom[], VariableType cpyTo[]);
#if (COPY_ARRAYS_USING_MEMCPY == 1)
#define ARRAY_COPY_INLINE(ct,from,to) \
      memcpy(to, from, (ct)*VARIABLE_TYPE_SIZE)
#else
#define ARRAY_COPY_INLINE(ct,from,to) \
      {VariableType *ff=from; VariableType *tt=to; \
      *tt = *ff; \
      for (int i=1; i<(ct); i++) *++tt = *++ff;}
#endif
```

ArrayOps.hpp

```
// ======< Array management: Compare arrays >======
int ArrayCompare(int arrayCount,
      const VariableType x[], const VariableType y[]);

// ======< Array addition and subtraction >======
void ArraySum(int arrayCount,
      const VariableType lval[], const VariableType rval[],
      VariableType result[]);
void ArrayDifference(int arrayCount,
      const VariableType lval[], const VariableType rval[],
      VariableType result[]);

// ======< Array scalar multiples >======
void ArrayMinus(int arrayCount,
      const VariableType rval[], VariableType result[]);
void ArrayScalarProduct(int arrayCount,
      const double scale, const VariableType rval[],
      VariableType result[]);
void ArraySaxpy(int arrayCount,
      const double scale, const VariableType lval[],
      const VariableType rval[],
      VariableType result[]);

// ======< Array magnitudes >======
VariableType ArrayMax(int arrayCount,
      const VariableType x[]);
VariableType ArrayMaxAbs(int arrayCount,
      const VariableType x[]);
VariableType ArrayMin(int arrayCount,
      const VariableType x[]);
VariableType ArrayMinAbs(int arrayCount,
      const VariableType x[]);
double ArrayNormOne(int arrayCount,
      const VariableType x[]);
double ArrayNormTwo(int arrayCount,
      const VariableType x[]);

// ======< Vector-vector inner products >======
double ArraySumSquares(int arrayCount,
      const VariableType x[]);
double ArrayInnerProduct(int arrayCount,
      const VariableType rowVec[], const int rowVecOffset,
      const VariableType colVec[], const int colVecOffset);

// ======< Print to stream >======
void ArrayStreamPrint(const int arrayCount,
      const VariableType x[]);

#endif


/* ====================< End File ARRAYOPS.HPP >================== */
```

ArrayOps.cpp

```
/* ====================< File ARRAYOPS.CPP >=========================
=======================< rev 06/28/96 >================================

     Defines standard functions for working with arrays.
     Documentation is in the header file.

=======================< File ARRAYOPS.CPP >======================= */

#include <float.h>
     // For DBL_MAX, FLT_MAX.
#include <math.h>

#include "utlreprt.hpp"
     // For ostream, standard enumerations, and report control.


#include "arrayops.hpp"


/* ==========< User preferences >============================
     Check user preferences.
=============< User preferences >======================== */
// =====< Variable Type >=====
#ifndef VARIABLES_ARE_DOUBLES
#error VARIABLES_ARE_DOUBLES is not defined
#endif

// =====< Copy and compare arrays >=====
#ifndef ARRAY_COPIES_USE_MEMCPY
#error ARRAY_COPIES_USE_MEMCPY is not defined
#endif

#ifndef ARRAY_COMPARES_USE_MEMCMP
#error ARRAY_COMPARES_USE_MEMCMP is not defined
#endif

#if (ARRAY_COPIES_USE_MEMCPY == 1 \
     || ARRAY_COMPARES_USE_MEMCMP == 1)
const int VARIABLE_TYPE_SIZE = sizeof(VariableType);
#endif

// =====< Control overflows >=====
#ifndef ARRAY_OPS_CONTROL_OVERFLOWS
#error ARRAY_OPS_CONTROL_OVERFLOWS is not defined
#endif

#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
#if (VARIABLES_ARE_DOUBLES == 1)
const double VARIABLE_TYPE_MAX = DBL_MAX;
#else
const float VARIABLE_TYPE_MAX = FLT_MAX;
#endif
#endif

// =====< Scale inner products >=====
// Note LOG_BASE and SCALE_THRESHOLD are declared extern.
#ifndef INNER_PRODUCTS_USE_SCALE_FACTOR
#error INNER_PRODUCTS_USE_SCALE_FACTOR is not defined
#endif

/* ==========< End user preferences >================== */


/* ==========< Array management >=========================
     Swap, copy, and compare arrays.
     Inline versions of some of the management functions are
defined as macros in the header file. Repeated definitions
```

```
are resolved by preprocessor #if statements:
#define ARRAY_SWAP_INLINE(ax,ay) \
     {VariableType *temp = ax; ax = ay; ay = temp;}
#define ARRAY_COPY_INLINE(ct,from,to) \
     memcpy(to, from, (ct)*sizeof(VariableType))
#define ARRAY_COPY_INLINE(ct,from,to) \
     {VariableType *ff=from; VariableType *tt=to; \
     for (int i=0; i<(ct); i++) *tt++ = *ff++;}
============< Array management >====================== */
// =====< Array management: Copy elements >=====
void ArrayCopy(int arrayCount,
     const VariableType cpyFrom[], VariableType cpyTo[])
     {
#if (ARRAY_COPIES_USE_MEMCPY == 1)
     memcpy(cpyTo, cpyFrom, arrayCount*VARIABLE_TYPE_SIZE);
#else
     *cpyTo = *cpyFrom;
     for (; arrayCount>1; arrayCount--)
          *++cpyTo = *++cpyFrom;
#endif
     return;
     }

// =====< Array management: Compare arrays >=====
int ArrayCompare(int arrayCount,
     const VariableType x[], const VariableType y[])
     {
     // Return 1 if data of x is same as data of y.
     if (x == y) return 1;
#if (ARRAY_COMPARES_USE_MEMCMP == 1)
     if (memcmp(x, y, arrayCount*VARIABLE_TYPE_SIZE)==0)
          return 1;
     return 0;
#else
     for (; arrayCount>0; arrayCount--)
          if (*x++ != *y++) return 0;
     return 1;
#endif
     }

/* ==========< End array management >=================== */


/* ==========< Array addition and subtraction >=============
     Add and subtract array elements.
============< Array addition and subtraction >========== */
void ArraySum(int arrayCount,
     const VariableType lval[], const VariableType rval[],
     VariableType result[])
     {
     //   Sum two arrays into third. Handles repeated arrays
     // in arguments, e.g. Sum(cnt,x,y,x);.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
     for (; arrayCount>0; arrayCount--)
          {
          long double sum = (*lval++) + (*rval++);
          if (sum > VARIABLE_TYPE_MAX)
               sum = VARIABLE_TYPE_MAX;
          else if (sum < -VARIABLE_TYPE_MAX)
               sum = -VARIABLE_TYPE_MAX;
          *result++ = sum;
          }
#else
     *result = *lval + *rval;
     for (; arrayCount>1; arrayCount--)
          *++result = (*++lval) + (*++rval);
#endif
```

```
      return;
      }

void ArrayDifference(int arrayCount,
      const VariableType lval[], const VariableType rval[],
      VariableType result[])
      {
      //   Subtract second array from first, place in third.
      // Handles repeated array arguments.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      for (; arrayCount>0; arrayCount--)
            {
            long double diff = (*lval++) - (*rval++);
            if (diff > VARIABLE_TYPE_MAX)
                  diff = VARIABLE_TYPE_MAX;
            else if (diff < -VARIABLE_TYPE_MAX)
                  diff = -VARIABLE_TYPE_MAX;
            *result++ = diff;
            }
#else
      *result = *lval - *rval;
      for (; arrayCount>1; arrayCount--)
            *++result = (*++lval) - (*++rval);
#endif
      return;
      }

/* ==========< End array addition and subtraction >====== */


/* ==========< Array scalar multiples >=====================
      Array scalar multiples including unary minus.
=============< Array scalar multiples >================== */
void ArrayMinus(int arrayCount,
      const VariableType rval[], VariableType result[])
      {
      *result = -*rval;
      for (; arrayCount>1; arrayCount--)
            *++result = -(*++rval);
      return;
      }

void ArrayScalarProduct(int arrayCount,
      const double scale, const VariableType rval[],
      VariableType result[])
      {
      // Multiply an array by a scalar, result in second array.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      for (; arrayCount>0; arrayCount--)
            {
            long double product = scale * (*rval++);
            if (product > VARIABLE_TYPE_MAX)
                  product = VARIABLE_TYPE_MAX;
            else if (product < -VARIABLE_TYPE_MAX)
                  product = -VARIABLE_TYPE_MAX;
            *result++ = product;
            }
#else
      *result = scale * (*rval);
      for (; arrayCount>1; arrayCount--)
            *++result = scale * (*++rval);
#endif
      return;
      }

void ArraySaxpy(int arrayCount,
      const double scale, const VariableType lval[],
```

```
      const VariableType rval[],
      VariableType result[])
      {
      // Saxpy(cnt,c,x,y,z) gives z = cx + y.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      for (; arrayCount>0; arrayCount--)
            {
            long double saxpy = scale * (*lval++) + (*rval++);
            if (saxpy > VARIABLE_TYPE_MAX)
                  saxpy = VARIABLE_TYPE_MAX;
            else if (saxpy < -VARIABLE_TYPE_MAX)
                  saxpy = -VARIABLE_TYPE_MAX;
            *result++ = saxpy;
            }
#else
      *result = scale * (*lval) + (*rval);
      for (; arrayCount>1; arrayCount--)
            *++result = scale * (*++lval) + (*++rval);
#endif
      return;
      }

/* ==========< End array scalar multiples >============== */


/* ==========< Array magnitudes >=========================
      Maximum and minimum elements of an array, straight or
with absolute values. The 1-norm (sum of absolute values of
the array elements), 2-norm (the square root of the sum of
squares) and the sum of squares (sum of squares of the
array elements).
==============< Array magnitudes >======================= */
VariableType ArrayMax(int arrayCount,
      const VariableType x[])
      {
      // Find maximum value in array.
      VariableType max = *x;
      for (; arrayCount>1; arrayCount--)
            {
            VariableType temp = *++x;
            if (max<temp) max = temp;
            }
      return max;
      }

VariableType ArrayMaxAbs(int arrayCount,
      const VariableType x[])
      {
      // Find maximum absolute value in array.
      VariableType maxAbs = fabs(*x);
      for (; arrayCount>1; arrayCount--)
            {
            VariableType temp = fabs(*++x);
            if (maxAbs<temp) maxAbs = temp;
            }
      return maxAbs;
      }

VariableType ArrayMin(int arrayCount,
      const VariableType x[])
      {
      // Find minimum value in array.
      VariableType min = *x;
      for (; arrayCount>1; arrayCount--)
            {
            VariableType temp = *++x;
            if (min>temp) min = temp;
```

```
            }
      return min;
      }

VariableType ArrayMinAbs(int arrayCount,
      const VariableType x[])
      {
      // Find minimum absolute value in array.
      VariableType minAbs = fabs(*x);
      for (; arrayCount>1; arrayCount--)
            {
            VariableType temp = fabs(*++x);
            if (minAbs>temp) minAbs = temp;
            }
      return minAbs;
      }

double ArrayNormOne(int arrayCount,
      const VariableType x[])
      {
      // Sum of absolute values of array elements.
      long double normOne = fabs(*x);
      for (; arrayCount>1; arrayCount--)
            normOne += fabs(*++x);
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      if (normOne > DBL_MAX) return DBL_MAX;
      if (normOne < -DBL_MAX) return -DBL_MAX;
#endif
      return normOne;
      }

double ArrayNormTwo(int arrayCount,
      const VariableType x[])
      {
      // Euclidean length of vector stored in array.
      return(sqrt(ArraySumSquares(arrayCount, x)));
      }

/* ==========< End array magnitudes >==================== */


/* ==========< Vector-vector inner products >===============
      Inner product (TProduct) xTy of two vectors stored in
arrays.
      The array elements are separated by a constant distance
or offset. A jump offset of one fetches contiguous array
elements.
      Inner product of a vector with itself is implemented as
a special function, ArraySumSquares.
==============< Vector-vector inner products >============ */
double ArraySumSquares(int arrayCount,
      const VariableType x[])
      {
      //    Inner product xTx of an array with itself. Gives
      // sum of squares of array elements.
      //
#if (INNER_PRODUCTS_USE_SCALE_FACTOR == 0)
      // Find sum of squares directly.
      VariableType temp = *x;
      long double sumSquares = temp*temp;
      for (; arrayCount>1; arrayCount--)
            {
            temp = *++x;
            sumSquares += temp*temp;
            }
      //
#else
```

```
      // Find sum of squares using dynamic scaling.
      long double scaleFactor = 1;
      VariableType temp = *x;
      long double sumSquares = temp*temp;
      for (; arrayCount>1; arrayCount--)
            {
            temp = *++x;
            temp = temp*temp/scaleFactor;
            if (temp > SCALE_THRESHOLD*sumSquares)
                  {
                  long double moreScaleFactor =
                        pow(FLT_RADIX, floor(log(sumSquares)/LOG_BASE));
                  sumSquares /= moreScaleFactor;
                  temp /= moreScaleFactor;
                  scaleFactor *= moreScaleFactor;
                  }
            sumSquares += temp;
            }
      //
      sumSquares *= scaleFactor;
#endif
      //
      // Return sum of squares.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      if (sumSquares > DBL_MAX) return DBL_MAX;
#endif
      return sumSquares;
      } // End ArraySumSquares().

double ArrayInnerProduct(int arrayCount,
      const VariableType rowVec[], const int rowVecOffset,
      const VariableType colVec[], const int colVecOffset)
      {
      // InnerProduct = xTy.
      //
#if (INNER_PRODUCTS_USE_SCALE_FACTOR == 0)
      // Find inner product directly.
      long double innerProduct = (*rowVec) * (*colVec);
      for (; arrayCount>1; arrayCount--)
            {
            rowVec += rowVecOffset;
            colVec += colVecOffset;
            innerProduct += (*rowVec) * (*colVec);
            }
      //
#else
      // Find inner product using dynamic scaling.
      long double scaleFactor = 1;
      long double innerProduct = (*rowVec) * (*colVec);
      for (; arrayCount>1; arrayCount--)
            {
            rowVec += rowVecOffset;
            colVec += colVecOffset;
            VariableType temp = ((*rowVec) * (*colVec))/scaleFactor;
            if (fabs(temp/innerProduct) > SCALE_THRESHOLD)
                  {
                  long double moreScaleFactor = pow(FLT_RADIX,
                        floor(log(fabs(innerProduct))/LOG_BASE));
                  innerProduct /= moreScaleFactor;
                  temp /= moreScaleFactor;
                  scaleFactor *= moreScaleFactor;
                  }
            innerProduct += temp;
            }
      //
      innerProduct *= scaleFactor;
#endif
```

```
      //
      // Return inner product.
#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
      if (innerProduct > DBL_MAX) return DBL_MAX;
      if (innerProduct < -DBL_MAX) return -DBL_MAX;
#endif
      return innerProduct;
      } // End ArrayInnerProduct().

/* ==========< End vector-vector inner products >======== */


/* ==========< Print to stream >============================
=============< Print to stream >======================== */
void ArrayStreamPrint(const int arrayCount,
      const VariableType x[])
      {
      int printColsPerLine = utlReportColsPerLine;
      if (printColsPerLine < 1
            || printColsPerLine > arrayCount)
            printColsPerLine = arrayCount;
      int currentCol = 1;
      int endCol = printColsPerLine;
      OnOffList printBlockHeadingMode;
      if (endCol == arrayCount
            || endCol == 1)
            printBlockHeadingMode = OFF;
      else printBlockHeadingMode = ON;
      const VariableType *elementP = x;
      //
      while (currentCol <= arrayCount)
            {
            if (endCol > arrayCount) endCol = arrayCount;
            if (printBlockHeadingMode == ON)
                  {
                  cout << "\nelements " << currentCol
                        << " to " << endCol;
                  cout.put(':');
                  }
            cout.put('\n') << *elementP++;
            for (; currentCol<endCol; currentCol++)
                  cout.put('\t') << *elementP++;
            currentCol++;
            endCol += printColsPerLine;
            }
      //
      return;
      } // End rayStreamPrint().

/* ==========< End print to stream >==================== */


/* ====================< End File ARRAYOPS.CPP >=================== */
```

DblDoglg.hpp

```
/* ====================< File DBLDOGLG.HPP >==========================
=======================< rev 07/03/96 >================================

     Headers for DBLDOGLG.CPP.
     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis.
     Modify the computational details, but not the logical structure,
of the algorithm, to exploit the special structure imposed by the
equilibrium problem.

     Source:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983; Society for Industrial and Applied Mathematics, 1996.

=======================< File DBLDOGLG.HPP >======================== */

#ifndef DBLDOGLG_HPP
#define DBLDOGLG_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For form of residual and Jacobian functions.


/* ==========< Function SolveDoubleDogleg >=================
=============< Function SolveDoubleDogleg >============== */
SolveNonLinearReturnList SolveDoubleDogleg
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance);

/* ==========< End Function SolveDoubleDogleg >========== */

#endif


/* ====================< End File DBLDOGLG.HPP >=================== */
```

```
/* ====================< File DBLDOGLG.CPP >=========================
=======================< rev 08/27/96 >==============================

     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis.
     Documentation is in the header file.

     Programming notes for double dogleg method:
     This file implements the pseudocode from Appendix A of the
text by Dennis and Schnabel. Major exceptions:
(1) It does not scale the residuals.
(2) It handles termination criteria by separate functions in
"termcond.cpp".
(3) It does not perturb a Jacobian identified as singular.
(4) It subtracts steps from, rather than adding steps to, xK when
finding xKp1. Thus, xKp1 = xK - dx where Dennis uses xK + dx.
(5) It reacts to residual errors, by reducing the trust region.
(6) It tests for zero residuals.
(7) It initializes the trust region to the Newton-Raphson length,
rather than to the Cauchy length.
(8) It performs the multiplication required to find J*(JT*r)
before, rather than after, factoring the Jacobian.
(9) It uses the sum of squares of residuals, not half the sum
of squares, as the cost function.
(10) Where possible, it replaces matrix-vector multiplications,
vector-vector inner products, and vector sums of squares with
equivalent scalar expressions. (The scalar operations are derived
by applying the linear algebra operations to the equilibrium
problem structure and chosen cost function.)
(11) Where required to support the new computational details, it
changes the computational sequence, but not the logic, of the
algorithm.

=======================< File DBLDOGLG.CPP >======================== */

#include <math.h>


#include "arrayops.hpp"
#include "plujcbn.hpp"
#include "termcond.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "dbldoglg.hpp"


/* ==========< Global constants >===============================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
=============< Global constants >============================ */
const int MAX_LOOPS = 100;
const double ALPHA = 1E-4;

/* ==========< End global constants >=================== */


/* ==========< Iteration status >=============================
=============< Iteration status >============================ */
enum TrustRegionStatusList
     {
     TRS_ZERO_RESIDUALS,
     TRS_STAGNANT_SOLUTION,
     TRS_ACCEPTED,
     TRS_ACCEPTED_AFTER_REJECT,
     TRS_INHERITED,
```

```
      TRS_REDUCED,
      TRS_INCREASED
      };
      //   Dennis: retcode.
      //   Indicates if attempt to find the dogleg step is the
      // first try with an inherited trust region, a subsequent
      // try with a reduced trust region (retcode = 2), or a
      // subsequent try with an increased trust region
      // (retcode = 3).
      //   May also indicate that step selection is complete,
      // either because current step accepted (retcode = 0),
      // because current step is so small that the termination
      // test for a stagnant estimated solution is satisfied
      // (retcode = 1), or because found a zero residual vector.
      //   May also indicate that an increased step should be
      // rejected, and the original step, i.e. the one that
      // caused the increase, should be accepted.

/* ==========< End iteration status >=================== */


/* ==========< Global variables >============================
     "Static" keyword limits variables to file scope.
==============< Global variables >======================== */
// =====< Global versions of function arguments >=====
static int probDim;
static VariableType relStepTol;

// =====< Global scalar variables >=====
static double stepLenNR, stepLenCauchy, stepLenCutback;
      //   Distances, respectively, to: the Newton-Raphson
      // solution of the linearized equations; the predicted
      // minimizer of r-square in the steepest descent direction;
      // and the point at which the double dogleg curve rejoins
      // the Newton-Raphson direction.
static double stepLenTrust, stepLenTrustPrev;
      //   Dennis del: trust region length. Would like to
      // take a step of this length. If increase trust length
      // during the current iteration, store its previous
      // value (just prior to increasing).
static TrustRegionStatusList trustRegionStatus;
static double rSquareK, rSquareKpl, rSquareKplPrev;
      //   The cost function at beginning of iteration; at trial
      // point during iteration; and, if the trust length was
      // increased during the current iteration, at the last
      // trial point for the current iteration.
static double lenSqJTr, lenFrthJTrOnLenSqJJTr;
static double scaleJTrToCauchy;
      //   Multiply vector JTr by this to find step to Cauchy point,
      // the minimizer of rSquare in the steepest descent direction
      // when residuals linearized.
static double eta;
      // Ratio of stepLenCutback/stepLenNR.
static double g2Tdx;
      //   Dennis calls this the initial slope. Not exactly
      // initial slope, but gTdx where g is the gradient of
      // rSquare at xK.
      //   The value should be negative, since going downhill
      // in values of rSquare. Calculate as -gTdx, using
      // negative sign since gTdx is positive for a descent
      // direction when dx subtracted. Dennis does not use
      // negative, but he adds dx.
static double deltaRSquare, deltaRSquarePredict;
static YesNoList firstDog;
      //   Indicates whether have calculated points on double
      // dogleg curve for the current iteration.
static SolveNonLinearReturnList exitReason;
```

```
// =====< Global dimensioned variables >=====
static PLUJacobian *J;
static VariableType *xK;
static VariableType *rCurrent;
     // Sometimes holds rK, sometimes rKpl.
static VariableType *dxNR, *dx;
     //   The Newton-Raphson step, and actual step taken,
     // respectively.
     //   Newton-Raphson step goes to solution of linearized
     // residual equations, which also minimizes the sum of
     // squares of the linearized residuals.
static VariableType *xKpl;
     // xKpl = xK - dx. Note Dennis uses x + dx.
static VariableType *xHold, *rHold;
     //   Store intermediate value of xKpl, rKpl for convenience
     // during one iteration. Also used for intermediate tasks.
static VariableType *vectorJTr;
     // Note gradient of rSquare = 2*JT*r.

/* ==========< End global variables >==================== */


/* ==========< Declare helper functions >===================
=============< Declare helper functions >================ */
static void FindStep(void);
static TrustRegionStatusList EvaluateStep(void);

/* ==========< End declare helper functions >============ */


/* ==========< Function SolveDoubleDogleg >=================
=============< Function SolveDoubleDogleg >============== */
SolveNonLinearReturnList SolveDoubleDogleg
     (const int problemDim,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance)
     {
     // Assume r has been evaluated for the given x.
     //
     // Allocate memory.
     J = new PLUJacobian(problemDim);
     xK = x;
     rCurrent = r;
     dxNR = new VariableType[problemDim];
     dx = new VariableType[problemDim];
     xKpl = new VariableType[problemDim];
     xHold = new VariableType[problemDim];
     rHold = new VariableType[problemDim];
     vectorJTr = new VariableType[problemDim];
     if (!J
          || !dxNR || !dx
          || !xKpl || !xHold || !rHold
          || !vectorJTr)
          FATAL(ERR_FREE_STORE, 1190404);
     //
     // Initialize.
     probDim = problemDim;
     exitReason = SNLR_ITERATION_COUNT;
```

```
      CheckAbsoluteZeroTolerance(absoluteZeroTolerance);
      CheckRelativeStepTolerance(relativeStepTolerance);
      relStepTol = *relativeStepTolerance;
      rSquareK = ArraySumSquares(problemDim, rCurrent);
      //
      // Report if necessary.
      if (utlReportLevel > RL_MUTE)
           {
           cout << "\nReporting from SolveDoubleDogleg():\nZero tol:\t"
                << *absoluteZeroTolerance
                << "\tStep tol:\t" << *relativeStepTolerance
                << "\nInitial x:";
           ArrayStreamPrint(problemDim, xK);
           }
      //
      for (int loopCount=1; loopCount<=MAX_LOOPS; loopCount++)
           {
           //   Known: xK, rK (in rCurrent), rSquareK. Unknown:
           // J, grad.
           //   Vectors free for initializing calculations: dxNR,
           // dx, vectorJTr, xKp1, xHold, rHold.
           //
           //   Initialize loop variables.
           //   Find vectors requiring Jacobian product, before
           // factor Jacobian.
           EvaluateJacobian(xK, rCurrent, J);
           J->TProduct(rCurrent, vectorJTr);
           J->Product(vectorJTr, rHold);
                //   rHold free until try to increase a step. Here,
                // holds JJTr in case needed to find Cauchy point.
           firstDog = YES;
           //
           //   Find Newton-Raphson step, the solution to linearized
           // residual equations, J(xK - xKp1) = J(dxNR) = rK. Will
           // have xKp1 = xK - dxNR.
           if (J->Solve(dxNR, rCurrent) == FAILURE)
                {
                if (utlReportLevel == RL_VERBOSE)
                     {
                     cout << "\n\nJacobian singular at x:";
                     ArrayStreamPrint(problemDim, xK);
                     }
                exitReason = SNLR_FACTORIZATION_ERROR;
                break;
                }
           stepLenNR = sqrt(ArraySumSquares(problemDim, dxNR));
           //
           // Set trust length for first iteration to NR length.
           if (loopCount == 1)
                stepLenTrust = stepLenNR;
           //
           // Report if necessary.
           if (utlReportLevel == RL_VERBOSE)
                {
                cout << "\n\nx: iteration:\t" << loopCount;
                ArrayStreamPrint(problemDim, xK);
                cout << "\nr: SS:\t" << rSquareK;
                ArrayStreamPrint(problemDim, rCurrent);
                cout << "\nstep lengths: trust:\t" << stepLenTrust
                     << "\tNR:\t" << stepLenNR;
                }
           //
           //   Inner loop: Find test point on double dogleg curve,
           // at step length = trust length. Evaluate residuals and
           // cost function there. Update the current trust region,
           // and repeat until find an acceptable point.
           trustRegionStatus = TRS_INHERITED;
```

```
while (trustRegionStatus >= TRS_INHERITED)
     {
     FindStep();
     // Take the step and evaluate residuals.
     ArrayDifference(problemDim, xK, dx, xKpl);
     if (EvaluateResiduals(xKpl, rCurrent) == RER_ERROR)
          {
          if (trustRegionStatus == TRS_INCREASED)
               {
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\trejected (residual error)";
               trustRegionStatus = TRS_ACCEPTED_AFTER_REJECT;
               }
          else
               {
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\thalved (residual error)";
               stepLenTrust /= 2;
               trustRegionStatus = TRS_REDUCED;
               }
          } // end residual error processing
     else
          {
          // Here, have good residuals.
          rSquareKpl = ArraySumSquares(problemDim, rCurrent);
          if (CheckZeroVector(problemDim,
               rCurrent, *absoluteZeroTolerance) == TC_TERMINATE)
               {
               // Found a solution.
               trustRegionStatus = TRS_ZERO_RESIDUALS;
               }
          else
               trustRegionStatus = EvaluateStep();
          } // end good residual processing
     } // end search for acceptable dx
//
//   Here, know an acceptable dx, or met a termination
// criterion.
if (trustRegionStatus == TRS_ACCEPTED)
     {
     // Update trust region and prepare for next iteration.
     if (deltaRSquare > 0.1*deltaRSquarePredict)
          {
          //   Recall deltas are negative, so if actual
          // reduction < 0.1*predicted reduction, i.e. if not
          // doing well, reduce trust region for next step.
          stepLenTrust /= 2;
          }
     else if (deltaRSquare <= 0.75*deltaRSquarePredict)
          {
          //   If actual reduction >= 0.75*predicted, doing
          // well. Increase trust region for next step.
          stepLenTrust *= 2;
          }
     VariableType *temp = xK; xK = xKpl; xKpl = temp;
     rSquareK = rSquareKpl;
     }
else if (trustRegionStatus == TRS_ACCEPTED_AFTER_REJECT)
     {
     //   Reject an increased trust region. Accept the
     // step which induced the increase, and take its
     // length as the trust length.
     stepLenTrust = stepLenTrustPrev;
     VariableType *temp = xK; xK = xHold; xHold = temp;
     temp = rCurrent; rCurrent = rHold; rHold = temp;
     rSquareK = rSquareKplPrev;
     }
```

```
          else
               {
               //   Termination. Set exit reason and break out of
               // for-loop of MAX_LOOPS iterations.
               if (trustRegionStatus == TRS_ZERO_RESIDUALS)
                    exitReason = SNLR_ZERO_RESIDUAL_VECTOR;
               else
                    exitReason = SNLR_STAGNANT_ESTIMATED_SOLUTION;
               break;
               }
          } // end sequence of double dogleg steps
     //
     //   Copy current values to arrays passed as function
     // arguments, if necessary.
     if (xKpl != x)
          ArrayCopy(problemDim, xKpl, x);
     if (rCurrent != r)
          ArrayCopy(problemDim, rCurrent, r);
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\n\nSolveDoubleDogleg() terminating: "
               << SolveNonLinearReturnExplanation[exitReason]
               << "\nFinal x:";
          ArrayStreamPrint(problemDim, x);
          cout << "\nr: SS:\t" << rSquareKpl;
          ArrayStreamPrint(problemDim, r);
          }
     //
     //   Release memory allocated from free store. Since have
     // been trading arrays around between pointers, make sure
     // don't try to delete the arrays passed as arguments to
     // this function.
     delete J;
     if (xK != x)
          delete [] xK;
     if (rCurrent != r)
          delete [] rCurrent;
     delete [] dxNR;
     delete [] dx;
     if (xKpl != x)
          delete [] xKpl;
     if (xHold != x)
          delete [] xHold;
     if (rHold != r)
          delete [] rHold;
     delete [] vectorJTr;
     return exitReason;
     } // End SolveDoubleDogleg().

/* ==========< End Function SolveDoubleDogleg >========== */


/* ==========< Helper functions >==========================
============< Helper functions >======================== */
static void FindStep(void)
     {
     //   Find test point on double dogleg curve, at step
     // length = trust length. Also find g2Tdx and predicted
     // change in cost function for this step.
     //
     if (stepLenTrust >= stepLenNR)
          {
          // Take the full Newton-Raphson step.
          stepLenTrust = stepLenNR;
          ArrayCopy(probDim, dxNR, dx);
```

```
        g2Tdx = -2*rSquareK;
        deltaRSquarePredict = -rSquareK;
        }
    else
        {
        // Take a step smaller than the full NR step.
        if (firstDog == YES)
            {
            // Calculate points on the double dogleg curve.
            firstDog = NO;
            lenSqJTr = ArraySumSquares(probDim, vectorJTr);
            //   Find Cauchy point, the minimizer of rSquare in
            // steepest descent direction if residuals were linear.
            // Recall rHold temporarily stores JJTr.
            scaleJTrToCauchy =
                lenSqJTr/ArraySumSquares(probDim, rHold);
            stepLenCauchy = scaleJTrToCauchy*sqrt(lenSqJTr);
            //   Find cutback point, where double dogleg
            // search path rejoins NR direction.
            lenFrthJTrOnLenSqJJTr = scaleJTrToCauchy*lenSqJTr;
            eta = 0.2 + 0.8*lenFrthJTrOnLenSqJJTr/rSquareK;
            stepLenCutback = eta*stepLenNR;
            if (utlReportLevel == RL_VERBOSE)
                {
                cout << "\nstep lengths: Cauchy:\t" << stepLenCauchy
                    << "\tCutback:\t" << stepLenCutback;
                }
            } // end firstDog == YES
        //
        // Here, have dogleg points and need dx.
        double rho1, rho2;
            // Scales for dx = rho1*dxCauchy + rho2*dxNR.
        if (stepLenTrust >= stepLenCutback)
            {
            // Take partial step in NR direction.
            rho1 = 0;
            rho2 = stepLenTrust/stepLenNR;
            }
        else if (stepLenTrust <= stepLenCauchy)
            {
            // Take partial step in SD direction.
            rho1 = stepLenTrust/stepLenCauchy;
            rho2 = 0;
            }
        else
            {
            // Find convex combination of dxCauchy and dxNR.
            double b = 2*eta*rSquareK/lenFrthJTrOnLenSqJJTr
                - 2;
            double c = stepLenCauchy*stepLenCauchy;
            double a = stepLenCutback*stepLenCutback/c
                - b - 1;
            c = 1 - stepLenTrust*stepLenTrust/c;
            double lambda = (sqrt(b*b-4*a*c) - b)/2/a;
            rho1 = 1 - lambda;
            rho2 = lambda*eta;
            }
        //
        // Here, have rho1 and rho2.
        if (rho1 == 0)
            {
            // Entirely in NR direction.
            ArrayScalarProduct(probDim, rho2, dxNR, dx);
            g2Tdx = -2*rho2*rSquareK;
            deltaRSquarePredict = g2Tdx + rho2*rho2*rSquareK;
            }
        else
```

```
                {
                // A component in SD direction...
                ArrayScalarProduct(probDim,
                     rho1*scaleJTrToCauchy, vectorJTr, dx);
                g2Tdx = -2*rho1*lenFrthJTrOnLenSqJJTr;
                deltaRSquarePredict = rho1*(rho1 + 2*rho2)*
                     lenFrthJTrOnLenSqJJTr;
                if (rho2 != 0)
                     {
                     // ...and a component in NR direction.
                     ArraySaxpy(probDim, rho2, dxNR, dx, dx);
                     g2Tdx -= 2*rho2*rSquareK;
                     deltaRSquarePredict += (rho2*rho2*rSquareK);
                     }
                deltaRSquarePredict += g2Tdx;
                }
           } // end finding step with lenTrust < lenNR
      //
      // Here, have step dx with length stepLenTrust.
      if (utlReportLevel == RL_VERBOSE)
           cout << "\nstep:\t" << stepLenTrust;
      return;
      } // End FindStep().

static TrustRegionStatusList EvaluateStep(void)
      {
      //   Update the trust region, to seek a different dx in
      // current iteration, or to use during next iteration.
      //
      // Check if increased trust region failed.
      if (trustRegionStatus == TRS_INCREASED
           && rSquareKp1 > rSquareKp1Prev)
           {
           //   Trust region was increased into a worse cost function.
           // Restore previous values and accept.
           //   Note that previous values of actual and estimated
           // change in cost function are still valid.
           if (utlReportLevel == RL_VERBOSE)
                cout << "\trejected (worse cost)";
           return TRS_ACCEPTED_AFTER_REJECT;
           }
      //
      //   Prepare to evaluate step's progress.
      //   Both g2Tdx ("initial slope") and deltaRSquare are negative
      // for smaller rSquare. Good progress made when delta is more
      // negative than g2Tdx, or when delta < g2Tdx.
      //   Acceptable progress when delta <= ALPHA*g2Tdx.
      //   g2Tdx and deltaRSquarePredict calculated earlier.
      deltaRSquare = rSquareKp1 - rSquareK;
      //
      // Check that cost function making progress.
      if (deltaRSquare > ALPHA*g2Tdx)
           {
           // Actual rSquare too large.
           if (CheckStagnatedVector(probDim, dx, xKp1, relStepTol)
                == TC_TERMINATE)
                {
                return TRS_STAGNANT_SOLUTION;
                }
           else if (trustRegionStatus == TRS_INCREASED)
                {
                //   Last trial was step good enough that increased
                // trust length. If here, the increased length had
                // better cost than previously, so don't reject.
                return TRS_ACCEPTED;
                }
           else
```

```
                {
                // Decrease trust length.
                if (utlReportLevel == RL_VERBOSE)
                     cout << "\tcut (poor cost)";
                double lambda = g2Tdx/2/(g2Tdx - deltaRSquare);
                if (lambda < 0.1) lambda = 0.1;
                else if (lambda > 0.5) lambda = 0.5;
                stepLenTrust *= lambda;
                return TRS_REDUCED;
                }
          } // end check that making sufficient progess
     //
     //   Cost function sufficiently small. May increase or accept
     // it for current iteration, and may also change it for next
     // iteration, all depending on success of model at predicting
     // values.
     //   Make decision about current, acceptable, trust region.
     if (trustRegionStatus == TRS_REDUCED
          || stepLenTrust == stepLenNR)
          {
          //   If just reduced, or if using NR step, then don't
          // consider increasing trust region during this iteration.
          return TRS_ACCEPTED;
          }
     else if (fabs(deltaRSquarePredict - deltaRSquare)
          <= -0.1*deltaRSquare
          ||
          deltaRSquare <= g2Tdx)
          {
          //   Increase trust region if prediction was very good,
          // or if actual decrease very large. Store current values
          // so can recover later if needed.
          if (utlReportLevel == RL_VERBOSE)
               cout << "\tdoubled (good cost)";
          stepLenTrustPrev = stepLenTrust;
          stepLenTrust *= 2;
          VariableType *temp = xHold; xHold = xKpl; xKpl = temp;
          temp = rHold; rHold = rCurrent; rCurrent = temp;
          rSquareKplPrev = rSquareKpl;
          return TRS_INCREASED;
          }
     // Here, no reason to increase the acceptable trust region.
     return TRS_ACCEPTED;
     } // End EvaluateStep().

/* ==========< End helper functions >==================== */


/* ====================< End File DBLDOGLG.CPP >================== */
```

FullJcbn.hpp

```
/* ====================< File FULLJCBN.HPP >==========================
=======================< rev 07/01/96 >================================

     Headers for FULLJCBN.CPP.
     Declares a class FullJacobian.

     An object of class FullJacobian implements a square matrix
of real numbers (either floating-point or double-precision;
see notes below). The matrix operations support simple operations
on a full Jacobian matrix, such as assignment and printing.
     The class is intended as a base class for derivations which
support factorization and indirect solution techniques.
     In addition to its matrix elements, a Jacobian object carries
its own dimension, and routines for using the matrix in linear
and nonlinear problems, e.g. matrix-vector products, rank one
updates, and handing over its storage space for direct changes to
the elements. The user is responsible for calculating the elements,
by analytic evaluation, finite difference approximation, or by
performing Broyden updates.

     User preferences set in other files:
(1) Jacobian elements are of either floats or doubles. The choice
is made in "probdefn.hpp".
(2) Variables (vector elements) are of either floats or doubles.
The choice is made in "probdefn.hpp".
(3) Arrays are copied using either the string command memcpy,
or an element-by-element version. The choice is made in
"utldepnd.hpp".
(4) Array functions can check for overflow or not. The choice
is made in "utldepnd.hpp". UNIMPLEMENTED.
(5) Vector-vector inner products are calculated either
directly or using a scaled version. The choice is made in
"utldepnd.hpp". UNIMPLEMENTED.

     Streamlined standard operations:
     Several decisions were made to speed up standard operations
at the expense of complete control over the class variables
and over error checking. Usually this was done by dropping some
tests. This means that the user is responsible for using the
class properly or in the most efficient way possible, to an
extent which would not be true for a more tightly-controlled
class design.
     (1) The FullJacobian class is designed to be used on one
problem at a time, using a single factorization technique.
Therefore the problem dimension and factorization are set
when a class instant is created. This streamlines the
class operations by reducing the number of tests a class
member function has to perform to insure the environment
is properly set for a requested operation (e.g. to insure
the arrays have been allocated and the problem dimension
is nonzero).
     A function is provided for changing the problem size of an
existing class instance. No information survives a change of
problem size.
     (2) Array privacy. See below. In particular the array
element assignment functions do not check to make sure that
an inappropriate array is being used to assign to the
matrix.
     (3) The Product and TProduct functions do not check
that their array arguments are unique (i.e. they do not check
for unique array addresses). If the same array is used for
both vectors, the functions return bad results.

     Array privacy:
     In a tightly controlled class implementation, the user
would calculate or estimate the Jacobian data in a memory
array independent of the Jacobian object's memory, and copy
```

```
it in on command. The 'Set' functions implement this type
of use, and indeed require it when the user provides Jacobian
data by column instead of by row.
     However, to avoid a large number of array element copies,
and to reduce the amount of storage required for large problems,
the user can insert row data directly into the Jacobian's
allocated memory. The 'Get' functions return pointers to the
object storage space so that the user can write directly
to the Jacobian arrays. This is nonstandard since ordinarily
a class would not allow direct modification of its data.
     The greatest danger in direct access is that a factored
matrix can be replaced by an unfactored one, without making
the requisite changes to the object's controlling variables.
Then the unfactored data are used as if they represent a
factored matrix, giving incorrect results. Some attempt is
made to safeguard against problems by setting the matrix status
to NONE when the 'Get' and 'Set' functions are used. The status
is changed to UNFACTORED by the 'Accept' and 'SetAccept'
functions.
     This program design creates several ways to specify data
for an unfactored Jacobian.
     To provide the Jacobian data row by row, either:
(1) Store the row data directly using 'get' to access the row
pointers, and 'accept' the matrix as unfactored (without using
'set' for each row);
(2) Store the row data directly in a contiguous array using
'get' to access the matrix pointer, and either 'SetAccept'
or 'Accept' the matrix;
(3) Develop the row data in independent memory, 'set'
each one, and 'Accept' the matrix; or
(4) Develop the row data in independent contiguous memory
and 'SetAccept' the matrix.
     To provide the Jacobian data column by column, either:
(1) Develop the column data in independent arrays and call
the 'set' functions for each column, then 'Accept' the array;
(2) Develop the column data in a single contiguous independent
array, and call the 'SetAcceptMatrixTranspose' function; or
(3) Store the column data directly in a contiguous array
using 'get' to access the matrix pointer, then call the
'SetAcceptMatrixTranspose' function.

     Array indexing:
     For assignment operations, the rows and columns are
numbered from 0 to numVars-1, rather than from 1 to numVars.
This is consistent with array indexing conventions in C++.
Since the user is expected to be tied to the same conventions,
indexing the rows from 0 should be more efficient than
indexing them from 1.

     Sources:
     Guido Buzzi-Ferraris, "Scientific C++: Building Numerical
Libraries the Object-Oriented Way," Addison-Wesley, 1993. Basic
ideas of C++ class design, and design for linear algebra.
     Stanley Lippman, "C++ Primer," second edition, Addison-
Wesley, 1991. C++ reference.
     Gilbert Strang, "Linear Algebra and its Applications,"
third edition, Harcourt Brace Jovanovich, 1988. Basic linear
algebra.

     Preparation:
JacobianType j;
JacobianType *jp;
VariableType *r;

     Constructors:
FullJacobian J;
     // Default constructor. Should not be used.
```

FullJcbn.hpp

```
int problemDim=5;
FullJacobian J(problemDim);
      // Sizes matrix. No matrix element initialization.
FullJacobian B = J;
      //   Copy-initializer. Also used to pass a Jacobian by
      // value as a function argument (a bad idea-- pass a pointer
      // to the Jacobian instead). Also used to pass a Jacobian
      // as a return from a function (a bad idea).

      Memory management:
problemDim = 16;
J.NewProblemDimension(problemDim);
      // All matrix element information is lost.
J.SwapWith(&B);
      // Trade data contents of two Jacobians.

      Assign to unfactored matrix:
jp = J.GetRowPointer(2);
jp[0] = 1.1; ...
J.AcceptMatrix();
JacobianType indepJp[problemDim];
indepJp[0] = -5; ...
J.SetRow(2,indepJp);
indepJp[0] = 14.2; ...
J.SetColumn(4,indepJp);
J.AcceptMatrix();
jp = GetMatrixPointer();
jp[0] = 19.2; ...
J.AcceptMatrix();
JacobianType indepJpLong[problemDim*problemDim];
indepJpLong[0] = ...;
J.SetAcceptMatrix(indepJpLong);
indepJpLong[0] = ...;
J.SetAcceptMatrixTranspose(indepJpLong);

      Assign from other matrix:
J.CopyJacobian(B);

      Linear algebra:
J.Transpose();
VariableType r[problemDim];
r[0] = ...;
VariableType rUnique[problemDim];
J.Product(r,rUnique);
      // rUnique = J*r
J.TProduct(r,rUnique);
      // rUnique = JT*r
VariableType u[problemDim];
u[0] = ...;
VariableType vT[problemDim];
vT[0] = ...;
J.UpdateRankOne(u,vT);
      // J = J + uvT

      Reporting:
int oldDim = J.GetProblemDimension();
J.GetRowLengths(u);
      // The ith element of u = length of ith row of J.
J.StreamPrint();
      //   Prints to cout.
      //   Print format controlled by variables declared in
      // utlreprt.hpp. If utlReportColsPerLine < 1, or > numVars,
      // each row is printed on one line. Elements are separated
      // by utlReportDelimiter.
      //   Printing unaffected by value of utlReportLevel.

======================< File FULLJCBN.HPP >====================== */
```

FullJcbn.hpp

```
#ifndef FULLJCBN_HPP
#define FULLJCBN_HPP


#include "probdefn.hpp"
      // For typedef of VariableType and JacobianType.


/* ==========< Class FullJacobian >=========================
=============< Class FullJacobian >====================== */
class FullJacobian
      {
public:
      // =====< Constructors >=====
      FullJacobian(void);
            //   Default constructor: FullJacobian J;
            // Provided for consistency only. Allocates for
            // a one-dimensional problem.
      FullJacobian(const int problemDimension);
            // Standard constructor: FullJacobian J(20);
      FullJacobian(const FullJacobian &const rval);
            //   Copy-initializer. Preferred: FullJacobian B(J);
            // Alternate form: FullJacobian B = J;
            // This constructor implicitly used e.g. to pass
            // Jacobian as return from functions.

      // =====< Destructor >=====
      ~FullJacobian(void);

      // =====< Memory management >======
      void NewProblemDimension(const int newProblemDimension);
      void SwapWith(FullJacobian *other);

      // =====< Assign to unfactored matrix >======
      JacobianType *GetRowPointer(const int rowToGet);
      void SetRow
            (const int rowToSet, const JacobianType newRow[]);
            // e.g. newRow is array of constant double
      void SetColumn
            (const int colToSet, const JacobianType newCol[]);
      JacobianType *GetMatrixPointer(void)
            {matrixStatus = MS_NONE; return squareMatrix;}
      void SetAcceptMatrix(const JacobianType newMat[]);
      void SetAcceptMatrixTranspose(const JacobianType newMatT[]);
      void AcceptMatrix(void)
            {matrixStatus = MS_UNFACTORED;}

      // =====< Assign from other matrix >=====
      void CopyJacobian(const FullJacobian &const copyFrom);

      // =====< Linear algebra >=====
      void Transpose(void);
      void Product
            (const VariableType rval[], VariableType result[]) const;
      void TProduct
            (const VariableType rval[], VariableType result[]) const;
      void UpdateRankOne
            (const VariableType u[], const VariableType vT[]);

      // =====< Reporting >=====
      int GetProblemDimension(void) const
            {return numVars;}
      void GetRowLengths(VariableType rowLengths[]) const;
      void StreamPrint(void) const;

protected:
```

FullJcbn.hpp

```
      // =====< Matrix variables >=====
      int numVars;
      JacobianType *squareMatrix;
      enum MatrixStatusList
            {
            MS_NONE,
            MS_UNFACTORED,
            MS_MODIFIED
            } matrixStatus;

      // =====< Memory management >=====
      void InitializeFull(const int problemDimension);

      };

/* ==========< End class FullJacobian >================== */

#endif


/* ====================< End File FULLJCBN.HPP >=================== */
```

```
/* ====================< File FULLJCBN.CPP >==========================
=======================< rev 07/01/96 >================================

     Defines a class FullJacobian.
     Documentation is in the header file.

     Programming notes for matrix allocation:
     The squareMatrix stores the elements of J sequentially
in row order.
     Internally, rows are numbered (0..numVars-1) to match array
indexing conventions. However rows and columns may be referred
to using (1..numVars) to aid output labelling.

     Programming notes for matrix status:
     The class is designed primarily as a base class for matrix
factorizations. However it also stands alone as a class for a
square unfactored Jacobian.
     The function implementations assume the matrix status can
be NONE or UNFACTORED. A third matrix status, MODIFIED, is provided
but is never used explicitly here. A derived class supporting a
particular factorization should set the matrix status to MODIFIED
whenever the it changes the representational details of the matrix.
Then the derived class functions which replicate those already
defined here for the base class should check the matrix status
and call the base class version when the status is != MODIFIED.

=======================< File FULLJCBN.CPP >======================== */

#include <float.h>
#include <stdlib.h>
#include <math.h>

#include "utldepnd.hpp"
     // For macros governing user preferences.
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "fulljcbn.hpp"


/* ============< User preferences >============================
     Check user preferences.
================< User preferences >======================== */
// =====< Variable Type >=====
#ifndef VARIABLES_ARE_DOUBLES
#error VARIABLES_ARE_DOUBLES is not defined
#endif

// =====< Jacobian element type >=====
#ifndef JACOBIAN_ELEMENTS_ARE_DOUBLES
#error JACOBIAN_ELEMENTS_ARE_DOUBLES is not defined
#endif

// =====< Copy arrays >=====
#ifndef ARRAY_COPIES_USE_MEMCPY
#error ARRAY_COPIES_USE_MEMCPY is not defined
#endif

#if (ARRAY_COPIES_USE_MEMCPY == 1)
const int JACOBIAN_TYPE_SIZE = sizeof(JacobianType);
#endif

/*
Control overflows is unimplemented.
// =====< Control overflows >=====
#ifndef ARRAY_OPS_CONTROL_OVERFLOWS
```

```
#error ARRAY_OPS_CONTROL_OVERFLOWS is not defined
#endif

#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
#if (JACOBIAN_ELEMENTS_ARE_DOUBLES == 1)
const double JACOBIAN_TYPE_MAX = DBL_MAX;
#else
const float JACOBIAN_TYPE_MAX = FLT_MAX;
#endif
#endif
*/

/*
Scale inner products is unimplemented.
// ======< Scale inner products >=====
// Note LOG_BASE and SCALE_THRESHOLD are declared extern.
#ifndef INNER_PRODUCTS_USE_SCALE_FACTOR
#error INNER_PRODUCTS_USE_SCALE_FACTOR is not defined
#endif
*/

/* ==========< End user preferences >=================== */


/* ==========< Constructors >================================
============< Constructors >============================== */
FullJacobian::FullJacobian(void)
      {
      // Default constructor.
      InitializeFull(1);
      }

FullJacobian::FullJacobian
      (const int problemDimension)
      {
      // Preferred constructor.
      InitializeFull(problemDimension);
      }

FullJacobian::FullJacobian
      (const FullJacobian &const rval)
      {
      // Copy-initializer.
      InitializeFull(rval.numVars);
      CopyJacobian(rval);
      }

/* ==========< End constructors >======================== */


/* ==========< Destructor >================================
      Called when the Jacobian object leaves scope.
============< Destructor >============================== */
FullJacobian::~FullJacobian(void)
      {
      // Release dynamically-allocated memory to free store.
      delete [] squareMatrix;
      }

/* ==========< End destructor >========================== */


/* ==========< Memory management >=======================
      Control matrix object memory.
      Inline (defined in header file):
None.
============< Memory management >====================== */
```

```
// =====< Memory management, public >=====
void FullJacobian::NewProblemDimension
      (const int newProblemDimension)
      {
      delete [] squareMatrix;
      InitializeFull(newProblemDimension);
      return;
      }

void FullJacobian::SwapWith(FullJacobian *other)
      {
      // Swap pointers to data.
      JacobianType *tempSquare = squareMatrix;
      squareMatrix = other->squareMatrix;
      other->squareMatrix = tempSquare;
      //
      // Swap bookkeeping variables.
      int tempInt = numVars;
      numVars = other->numVars;
      other->numVars = tempInt;
      MatrixStatusList tempStatus = matrixStatus;
      matrixStatus = other->matrixStatus;
      other->matrixStatus = tempStatus;
      return;
      }

// =====< Memory management, protected >=====
void FullJacobian::InitializeFull
      (const int problemDimension)
      {
      // Allocate square matrix.
      if ((numVars = problemDimension) <= 0)
            FATAL(ERR_INPUT_RANGE, 1091913);
      squareMatrix = new JacobianType [numVars*numVars];
      if (!squareMatrix)
            FATAL(ERR_FREE_STORE, 2091913);
      matrixStatus = MS_NONE;
      return;
      } // End InitializeFull().

/* ==========< End memory management >================== */


/* ==========< Assign to unfactored matrix >================
      Set matrix elements and control factorization status.
      Inline (defined in header file):
JacobianType *GetMatrixPointer(void)
      {matrixStatus = MS_NONE; return squareMatrix}
void AcceptMatrix(void)
      {matrixStatus = UNFACTORED;}
=============< Assign to unfactored matrix >============= */
// =====< Assign to unfactored matrix, public >=====
JacobianType *FullJacobian::GetRowPointer
      (const int rowToGet)
      {
      if (rowToGet<0 || rowToGet>=numVars)
            FATAL(ERR_INPUT_RANGE, 1071816);
      matrixStatus = MS_NONE;
      return squareMatrix + rowToGet*numVars;
      }

void FullJacobian::SetRow
      (const int rowToSet, const JacobianType newRow[])
      {
      if (rowToSet<0 || rowToSet>=numVars)
            FATAL(ERR_INPUT_RANGE, 11918);
      matrixStatus = MS_NONE;
```

```
      JacobianType *rowP = squareMatrix + rowToSet*numVars;
      if (newRow != rowP)
            {
#if (ARRAY_COPIES_USE_MEMCPY == 1)
            memcpy(rowP, newRow, numVars*JACOBIAN_TYPE_SIZE);
#else
            *rowP = *newRow;
            for (int col=1; col<numVars; col++)
                  *++rowP = *++newRow;
#endif
            }
      return;
      }

void FullJacobian::SetColumn
      (const int colToSet, const JacobianType newCol[])
      {
      if (colToSet<0 || colToSet>=numVars)
            FATAL(ERR_INPUT_RANGE, 11903);
      matrixStatus = MS_NONE;
      JacobianType *colP = squareMatrix + colToSet;
      *colP = *newCol;
      for (int row=1; row<numVars; row++)
            {
            colP += numVars;
            *colP = *++newCol;
            }
      return;
      }

void FullJacobian::SetAcceptMatrix
      (const JacobianType newMat[])
      {
      matrixStatus = MS_UNFACTORED;
      if (newMat != squareMatrix)
            {
#if (ARRAY_COPIES_USE_MEMCPY == 1)
            memcpy(squareMatrix, newMat,
                  numVars*numVars*JACOBIAN_TYPE_SIZE);
#else
            JacobianType *matP = squareMatrix;
            *matP = *newMat;
            for (int matEl=1; matEl<numVars*numVars; matEl++)
                  *++matP = *++newMat;
#endif
            }
      return;
      }

void FullJacobian::SetAcceptMatrixTranspose
      (const JacobianType newMatT[])
      {
      matrixStatus = MS_UNFACTORED;
      if (newMatT == squareMatrix)
            Transpose();
      else
            {
            JacobianType *colP;
            for (int col=0; col<numVars; col++)
                  {
                  colP = squareMatrix + col;
                  *colP = *newMatT++;
                  for (int row=1; row<numVars; row++)
                        {
                        colP += numVars;
                        *colP = *newMatT++;
                        }
```

```
                }
            }
        return;
        }

/* ==========< End assign to unfactored matrix >========= */


/* ==========< Assign from other matrix >==================
     Assign to Jacobian object. Note assignment is not the
same as initialization, which occurs when the variable is
defined. Initialization: float x = 1; Assignment: x = 2;
     Inline (defined in header file):
None.
==============< Assign from other matrix >================ */
// ======< Assign from other matrix, public >=====
void FullJacobian::CopyJacobian
     (const FullJacobian &const copyFrom)
     {
     // Set size.
     if (numVars != copyFrom.numVars)
          NewProblemDimension(copyFrom.numVars);
     //
     // Copy square matrix; don't worry if none defined.
     matrixStatus = copyFrom.matrixStatus;
#if (ARRAY_COPIES_USE_MEMCPY == 1)
     memcpy(squareMatrix, copyFrom.squareMatrix,
          numVars*numVars*JACOBIAN_TYPE_SIZE);
#else
     JacobianType *myArray = squareMatrix;
     const JacobianType *copyFromArray = copyFrom.squareMatrix;
     for (int arrayEl=0; arrayEl<numVars*numVars; arrayEl++)
          *myArray++ = *copyFromArray++;
#endif
     return;
     } // End CopyJacobian().

/* ==========< End assign from other matrix >========== */


/* ==========< Linear algebra >============================
     Inline (defined in header file):
None.
==============< Linear algebra >========================== */
// ======< Linear algebra, public >=====
void FullJacobian::Transpose(void)
     {
     if (matrixStatus == MS_UNFACTORED)
          {
          JacobianType temp;
          JacobianType *rowP, *colP;
          colP = rowP = squareMatrix;
          for (int rowCount=1; rowCount<numVars; rowCount++)
               {
               //   For each row, assume rowP and colP point to
               // row's diagonal element (which exchanges with
               // itself). Then increment both-- rowP by one to
               // move along row, and colP by numVars to move
               // down a row. Trade the elements and repeat until
               // exhange the last element in the row.
               //   Note no need to do last row since all
               // non-diagonal elements have been traded as
               // column elements in previous steps.
               for (int col=rowCount; col<numVars; col++)
                    {
                    colP += numVars;
                    temp = *(++rowP);
```

```
                    *rowP = *colP;
                    *colP = temp;
                    }
               //   Move rowP and colP to next diagonal element.
               // Adding one to rowP moves to start of next row.
               // Then add rowCount to move to diagonal element,
               // since rowCount is one greater than rows indexed
               // from 0.
               colP = rowP = rowP + rowCount + 1;
               }
          }
     else FATAL(ERR_NEED_UNFACTORED, 1201801);
     return;
     } // End Transpose().

void FullJacobian::Product
     (const VariableType rval[], VariableType result[]) const
     {
     //   Find result = J*rval.
     //   Note no test for rval == result even though this
     // makes results wrong.
     //
     if (matrixStatus == MS_UNFACTORED)
          {
          const JacobianType *matP = squareMatrix;
          for (int row=0; row<numVars; row++)
               {
               const VariableType *vecP = rval;
               long double innerProduct = (*matP++) * (*vecP);
               for (int col=1; col<numVars; col++)
                    innerProduct += (*matP++) * (*++vecP);
               *result++ = innerProduct;
               }
          }
     else FATAL(ERR_NEED_UNFACTORED, 1161815);
     return;
     } // End Product().

void FullJacobian::TProduct
     (const VariableType rval[], VariableType result[]) const
     {
     //   Find result = JT*rval.
     //   Note no test for rval == result even though this
     // makes result wrong.
     //   Note the row of the transposed J is accessed as the
     // column of the stored J.
     if (matrixStatus == MS_UNFACTORED)
          {
          for (int row=0; row<numVars; row++)
               {
               const JacobianType *matP = squareMatrix + row;
               const VariableType *vecP = rval;
               long double innerProduct = (*matP) * (*vecP);
               for (int col=1; col<numVars; col++)
                    {
                    matP += numVars;
                    innerProduct += (*matP) * (*++vecP);
                    }
               *result++ = innerProduct;
               }
          }
     else FATAL(ERR_NEED_UNFACTORED, 1201618);
     return;
     } // End TProduct().

void FullJacobian::UpdateRankOne
     (const VariableType u[], const VariableType vT[])
```

```
      {
      // Update J = J + u*vT.
      if (matrixStatus == MS_UNFACTORED)
            {
            JacobianType *matP = squareMatrix;
            for (int row=0; row<numVars; row++)
                  {
                  //   To each (row, col) of unfactored matrix, add
                  // corresponding row of u times corresponding
                  // column of vT.
                  const VariableType *colP = vT;
                  for (int col=0; col<numVars; col++)
                        *matP++ += ((*u) * (*colP++));
                  u++;
                  }
            }
      else FATAL(ERR_NEED_UNFACTORED, 1211815);
      return;
      } // End UpdateRankOne().

/* ==========< End linear algebra >====================== */


/* ==========< Reporting >==================================
      Inline (defined in header file):
int GetProblemDimension(void) const
      {return numVars;}
=============< Reporting >============================== */
// =====< Reporting, public >=====
void FullJacobian::GetRowLengths(VariableType rowLengths[]) const
      {
      if (matrixStatus == MS_UNFACTORED)
            {
            const JacobianType *matP = squareMatrix;
            for (int row=0; row<numVars; row++)
                  {
                  VariableType temp = *matP++;
                  long double rowSumSquares = temp*temp;
                  for (int col=1; col<numVars; col++)
                        {
                        temp = *matP++;
                        rowSumSquares += temp*temp;
                        }
                  *rowLengths++ = sqrt(rowSumSquares);
                  }
            }
      else FATAL(ERR_NEED_UNFACTORED, 1071812);
      return;
      }

void FullJacobian::StreamPrint(void) const
      {
      if (matrixStatus == MS_UNFACTORED)
            {
            //   Print squareMatrix as an unfactored square matrix.
            //   Each block gets a heading identifying the column
            // numbers, unless all matrix columns are to be printed.
            //   For notational convenience, all row and column
            // indices run from 1 to numVars inclusive.
            //
            int printColsPerLine = utlReportColsPerLine;
            if (printColsPerLine < 1
                  || printColsPerLine > numVars)
                  printColsPerLine = numVars;
            int startCol = 1;
            int endCol = printColsPerLine;
            OnOffList printBlockHeadingMode;
```

```
          if (endCol == numVars) printBlockHeadingMode = OFF;
          else printBlockHeadingMode = ON;
          //
          while (startCol <= numVars)
               {
               if (endCol > numVars) endCol = numVars;
               if (printBlockHeadingMode == ON)
                    {
                    cout << "\ncolumns " << startCol
                         << " to " << endCol;
                    cout.put(':');
                    }
               const JacobianType *currRowStartColP, *rowStepP;
               rowStepP = currRowStartColP = squareMatrix + (startCol-1);
               for (int row=1; row<=numVars; row++)
                    {
                    // Print column startCol with no delimiter.
                    cout.put('\n') << *rowStepP;
                    // Print remaining columns with delimiter.
                    for (int colStep=startCol; colStep<endCol; colStep++)
                         cout.put('\t') << *++rowStepP;
                    currRowStartColP += numVars;
                    rowStepP = currRowStartColP;
                    }
               startCol += printColsPerLine;
               endCol += printColsPerLine;
               }
          }
     else
          cout.put('\n') << ERR_NEED_UNFACTORED;
     return;
     } // End StreamPrint().

/* ==========< End reporting >============================ */


/* ====================< End File FULLJCBN.CPP >================== */
```

NewtRaph.hpp

```
/* ====================< File NEWTRAPH.HPP >=========================
======================< rev 07/24/96 >================================

     Headers for NEWTRAPH.CPP.
     Solve a system of nonlinear equations using PLU factorization
in a pure Newton's method.
     The method takes the full Newton-Raphson step at every
iteration, unless it encounters an error in the residual
equations, when it halves the step until no error is reported.

     Sources:
     G. Dahlquist and A. Bjorck, "Numerical Methods," Prentice
Hall, 1974.
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983.
     Gilbert Strang, "Introduction to Applied Mathematics,"
Wellesley-Cambridge Press, 1986.

======================< File NEWTRAPH.HPP >====================== */

#ifndef NEWTRAPH_HPP
#define NEWTRAPH_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For form of residual and Jacobian functions.


/* ==========< Function SolveNewtonRaphson >================
============< Function SolveNewtonRaphson >============= */
SolveNonLinearReturnList SolveNewtonRaphson
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns PERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance);

/* ==========< End Function SolveNewtonRaphson >========= */

#endif


/* ====================< End File NEWTRAPH.HPP >=================== */
```

NewtRaph.cpp

```
/* ====================< File NEWTRAPH.CPP >=========================
======================< rev 08/23/96 >==============================

     Solve a system of nonlinear equations using PLU factorization
in a pure Newton-Raphson method.
     Documentation is in the header file.

======================< File NEWTRAPH.CPP >====================== */

#include "arrayops.hpp"
#include "plujcbn.hpp"
#include "termcond.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "newtraph.hpp"


/* ==========< Global constants >============================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
=============< Global constants >======================== */
const int MAX_LOOPS = 100;

/* ==========< End global constants >=================== */


/* ==========< Function SolveNewtonRaphson >================
=============< Function SolveNewtonRaphson >============= */
SolveNonLinearReturnList SolveNewtonRaphson
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns PERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance)
     {
     // Assume r has been evaluated for the given x.
     //
     // Set up dimensioned variables.
     PLUJacobian *J = new PLUJacobian(problemDimension);
     VariableType *xK = x;
     VariableType *dx = new VariableType[problemDimension];
     VariableType *xKp1 = new VariableType[problemDimension];
     if (!J || !dx || !xKp1)
          FATAL(ERR_FREE_STORE, 1191614);
     //
     // Initialize.
     SolveNonLinearReturnList exitReason = SNLR_ITERATION_COUNT;
     CheckAbsoluteZeroTolerance(absoluteZeroTolerance);
     CheckRelativeStepTolerance(relativeStepTolerance);
     double rSquareK = ArraySumSquares(problemDimension, r);
     double rSquareKp1;
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\nReporting from SolveNewtonRaphson():\nZero tol:\t"
               << *absoluteZeroTolerance
               << "\tStep tol:\t" << *relativeStepTolerance
               << "\nInitial x:";
```

```
        ArrayStreamPrint(problemDimension, xK);
        }
    //
    for (int loopCount=1; loopCount<=MAX_LOOPS; loopCount++)
        {
        // Start loop with x, r, rSquareK known; J unknown.
        //
        // Report if necessary.
        if (utlReportLevel == RL_VERBOSE)
            {
            cout << "\n\nx: iteration:\t" << loopCount;
            ArrayStreamPrint(problemDimension, xK);
            cout << "\nr: SS:\t" << rSquareK;
            ArrayStreamPrint(problemDimension, r);
            }
        //
        //   Find Newton-Raphson step, the solution to linearized
        // residual equations, J(xK - xKpl) = J(dxNR) = rK. Will
        // have xKpl = xK - dxNR
        EvaluateJacobian(xK, r, J);
        if (J->Solve(dx, r) == FAILURE)
            {
            if (utlReportLevel == RL_VERBOSE)
                cout << "\n\nJacobian singular at given x";
            exitReason = SNLR_FACTORIZATION_ERROR;
            break;
            }
        // Find xKpl = xK - dx.
        ArrayDifference(problemDimension, xK, dx, xKpl);
        //   Evaluate residuals and halve step until get
        // good residuals.
        while (EvaluateResiduals(xKpl, r) == RER_ERROR)
            {
            if (utlReportLevel == RL_VERBOSE)
                cout << "\nstep halved (residual error)";
            ArrayScalarProduct(problemDimension, 0.5, dx, dx);
            ArrayDifference(problemDimension, xK, dx, xKpl);
            }
        //   Find norm and check termination criteria in
        // preferred order (zero vector, stagnated solution,
        // stagnated norm).
        //   Note calculate norm before needed so that
        // has correct value if terminate.
        rSquareKpl = ArraySumSquares(problemDimension, r);
        if (CheckZeroVector(problemDimension,
            r, *absoluteZeroTolerance) == TC_TERMINATE)
            {
            exitReason = SNLR_ZERO_RESIDUAL_VECTOR;
            break;
            }
        if (CheckStagnatedVector(problemDimension,
            dx, xKpl, *relativeStepTolerance) == TC_TERMINATE)
            {
            exitReason = SNLR_STAGNANT_ESTIMATED_SOLUTION;
            break;
            }
        //
        //   Prepare for next iteration. Do not copy vector
        // xKpl to xK, but swap arrays the pointers reference.
        rSquareK = rSquareKpl;
        VariableType *temp = xK; xK = xKpl; xKpl = temp;
        } // End loop of pure Newton steps.
    //
    //   Copy current values to arrays passed as function
    // arguments, if necessary. Note residuals are always
    // calculated in r[], so no copy required.
    if (xKpl != x)
```

```
          ArrayCopy(problemDimension, xKp1, x);
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\n\nSolveNewtonRaphson() terminating: "
               << SolveNonLinearReturnExplanation[exitReason]
               << "\nFinal x:";
          ArrayStreamPrint(problemDimension, x);
          cout << "\nr: SS:\t" << rSquareKp1;
          ArrayStreamPrint(problemDimension, r);
          }
     //
     //   Release memory allocated from free store. Since have
     // been trading arrays around between pointers, make sure
     // don't try to delete the arrays passed as arguments to
     // this function.
     delete J;
     if (xK != x)
          delete [] xK;
     delete [] dx;
     if (xKp1 != x)
          delete [] xKp1;
     return exitReason;
     } // End SolveNewtonRaphson().

/* ==========< End Function SolveNewtonRaphson >========= */


/* ====================< End File NEWTRAPH.CPP >================== */
```

PlanarHk.hpp

```
/* ====================< File PLANARHK.HPP >==========================
=======================< rev 07/22/96 >================================

     Headers for PLANARHK.CPP.
     Solve a system of nonlinear equations using PLU factorization
in a variation on the double dogleg trust region method given
by Dennis.
     Place iterates to optimize expected decrease of the cost
function. Also, modify the computational details of the algorithm,
to exploit the special structure imposed by the equilibrium
problem.

     Source:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983; Society for Industrial and Applied Mathematics, 1996.

=======================< File PLANARHK.HPP >======================== */

#ifndef PLANARHK_HPP
#define PLANARHK_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For form of residual and Jacobian functions.


/* ===========< Function SolvePlanarHook >====================
==============< Function SolvePlanarHook >================= */
SolveNonLinearReturnList SolvePlanarHook
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance);

/* ===========< End Function SolvePlanarHook >============= */

#endif


/* ====================< End File PLANARHK.HPP >=================== */
```

PlanarHk.cpp

```
/* ====================< File PLANARHK.CPP >==========================
=======================< rev 08/27/96 >================================

     Solve a system of nonlinear equations using PLU factorization
in a variation on the double dogleg trust region method given
by Dennis.
     Place iterates to optimize expected decrease of the cost
function. Also, modify the computational details of the algorithm,
to exploit the special structure imposed by the equilibrium
problem.
     Documentation is in the header file.

     Programming notes for planar hook method:
     This file adapts the pseudocode from Appendix A of the
text by Dennis and Schnabel. Major exceptions:
(1) It replaces the double dogleg curve for choosing trial
iterates, with the planar hook curve. The planar hook curve
chooses points in the plane of the Cauchy and Newton-Raphson
steps, choosing the point at the given trust length which
minimizes the expected cost function.
(2) It does not scale the residuals.
(3) It handles termination criteria by separate functions in
"termcond.cpp".
(4) It does not perturb a Jacobian identified as singular.
(5) It subtracts steps from, rather than adding steps to, xK when
finding xKpl. Thus, xKpl = xK - dx where Dennis uses xK + dx.
(6) It reacts to residual errors, by reducing the trust region.
(7) It tests for zero residuals.
(8) It initializes the trust region to the Newton-Raphson length,
rather than to the Cauchy length.
(9) Where possible, it replaces matrix-vector multiplications,
vector-vector inner products, and vector sums of squares with
equivalent scalar expressions. (The scalar operations are derived
by applying the linear algebra operations to the equilibrium
problem structure and chosen cost function.)
(11) It uses the sum of squares of residuals, not half the sum
of squares, as the cost function.

=======================< File PLANARHK.CPP >======================= */

#include <math.h>


#include "arrayops.hpp"
#include "plujcbn.hpp"
#include "termcond.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "planarhk.hpp"


/* ==========< Global constants >============================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
=============< Global constants >======================== */
const int MAX_LOOPS = 100;
const double ALPHA = 1E-4;
const double SLOPE_TOL = 1E-5;
const double RHO_TOL = 1E-5;

/* ==========< End global constants >=================== */


/* ==========< Iteration status >============================
=============< Iteration status >======================== */
```

```
enum TrustRegionStatusList
      {
      TRS_ZERO_RESIDUALS,
      TRS_STAGNANT_SOLUTION,
      TRS_ACCEPTED,
      TRS_ACCEPTED_AFTER_REJECT,
      TRS_INHERITED,
      TRS_REDUCED,
      TRS_INCREASED
      };
      //   Dennis: retcode.
      //   Indicates if attempt to find the hook step is the
      // first try with an inherited trust region, a subsequent
      // try with a reduced trust region (retcode = 2), or a
      // subsequent try with an increased trust region
      // (retcode = 3).
      //   May also indicate that step selection is complete,
      // either because current step accepted (retcode = 0),
      // because current step is so small that the termination
      // test for a stagnant estimated solution is satisfied
      // (retcode = 1), or because found a zero residual vector.
      //   May also indicate that an increased step should be
      // rejected, and the original step, i.e. the one that
      // caused the increase, should be accepted.

/* ==========< End iteration status >==================== */


/* ==========< Global variables >============================
      "Static" keyword limits variables to file scope.
      The scalar variables used to calculate the hook points
could also be declared static in the FindStep() function,
so that they retained their values from one function call
to another.
=============< Global variables >======================== */
// =====< Global versions of function arguments >=====
static int probDim;
static VariableType relStepTol;

// =====< Global scalar variables >=====
static double stepLenNR, stepLenCauchy;
      //   Distances, respectively, to: the Newton-Raphson
      // solution of the linearized equations; and the predicted
      // minimizer of r-square in the steepest descent direction.
static double stepLenTrust, stepLenTrustPrev;
      //   Dennis del: trust region length. Would like to
      // take a step of this length. If increase trust length
      // during the current iteration, store its previous
      // value (just prior to increasing).
static TrustRegionStatusList trustRegionStatus;
static double rSquareK, rSquareKpl, rSquareKplPrev;
      //   The cost function at beginning of iteration; at trial
      // point during iteration; and, if the trust length was
      // increased during the current iteration, at the last
      // trial point for the current iteration.
static double g2Tdx;
      //   Dennis calls this the initial slope. Not exactly
      // initial slope, but g2Tdx where g2 is the gradient of
      // rSqare at xK.
      //   The value should be negative, since going downhill
      // in values of rSquare. Calculate as -gTdx, using
      // negative sign since gTdx is positive for a descent
      // direction when dx subtracted. Dennis does not use
      // negative, but he adds dx.
static double deltaRSquare, deltaRSquarePredict;
static YesNoList firstHook;
      //   Indicates whether have calculated points used for
```

```
     // hook curve for the current iteration.
static SolveNonLinearReturnList exitReason;

// =====< Scalar variables for hook point >=====
static double lenSqJTr, lenFrthJTrOnLenSqJJTr;
static double scaleJTrToCauchy;
     //   Multiply vector JTr by this to find step to Cauchy point,
     // the minimizer of rSquare in the steepest descent direction
     // when residuals linearized.
static double c1, c2, c1SqMc2;

// =====< Global dimensioned variables >=====
static PLUJacobian *J;
static VariableType *xK;
static VariableType *rCurrent;
     // Sometimes holds rK, sometimes rKp1.
static VariableType *dxNR, *dx;
     //   The Newton-Raphson step, and actual step taken,
     // respectively.
     //   Newton-Raphson step goes to solution of linearized
     // residual equations, which also minimizes the sum of
     // squares of the linearized residuals.
static VariableType *xKp1;
     // xKp1 = xK - dx. Note Dennis uses x + dx.
static VariableType *xHold, *rHold;
     //   Store intermediate value of xKp1, rKp1 for convenience
     // during one iteration. Also used for intermediate tasks.
static VariableType *vectorJTr;
     // Note gradient of rSquare = 2*JT*r.

/* ==========< End global variables >=================== */


/* ==========< Declare helper functions >==================
============< Declare helper functions >================ */
static void FindStep(void);
static TrustRegionStatusList EvaluateStep(void);

/* ==========< End declare helper functions >============ */


/* ==========< Function SolvePlanarHook >==================
============< Function SolvePlanarHook >================ */
SolveNonLinearReturnList SolvePlanarHook
     (const int problemDim,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance)
     {
     // Assume r has been evaluated for the given x.
     //
     // Allocate memory.
     J = new PLUJacobian(problemDim);
     xK = x;
     rCurrent = r;
     dxNR = new VariableType[problemDim];
     dx = new VariableType[problemDim];
     xKp1 = new VariableType[problemDim];
     xHold = new VariableType[problemDim];
     rHold = new VariableType[problemDim];
```

```
      vectorJTr = new VariableType[problemDim];
      if (!J
            || !dxNR || !dx
            || !xKp1 || !xHold || !rHold
            || !vectorJTr)
            FATAL(ERR_FREE_STORE, 1191608);
      //
      // Initialize.
      probDim = problemDim;
      exitReason = SNLR_ITERATION_COUNT;
      CheckAbsoluteZeroTolerance(absoluteZeroTolerance);
      CheckRelativeStepTolerance(relativeStepTolerance);
      relStepTol = *relativeStepTolerance;
      rSquareK = ArraySumSquares(problemDim, rCurrent);
      //
      // Report if necessary.
      if (utlReportLevel > RL_MUTE)
            {
            cout << "\nReporting from SolvePlanarHook():\nZero tol:\t"
                  << *absoluteZeroTolerance
                  << "\tStep tol:\t" << *relativeStepTolerance
                  << "\nInitial x:";
            ArrayStreamPrint(problemDim, xK);
            }
      //
      for (int loopCount=1; loopCount<=MAX_LOOPS; loopCount++)
            {
            //   Known: xK, rK (in rCurrent), rSquareK. Unknown:
            // J, grad.
            //   Vectors free for initializing calculations: dxNR,
            // dx, vectorJTr, xKp1, xHold, rHold.
            //
            //   Initialize loop variables.
            //   Find vectors requiring Jacobian product, before
            // factor Jacobian.
            EvaluateJacobian(xK, rCurrent, J);
            J->TProduct(rCurrent, vectorJTr);
            J->Product(vectorJTr, rHold);
                  //   rHold free until try to increase a step. Here,
                  // holds JJTr in case needed to find Cauchy point.
            firstHook = YES;
            //
            //   Find Newton-Raphson step, the solution to linearized
            // residual equations, J(xK - xKp1) = J(dxNR) = rK. Will
            // have xKp1 = xK - dxNR.
            if (J->Solve(dxNR, rCurrent) == FAILURE)
                  {
                  if (utlReportLevel == RL_VERBOSE)
                        {
                        cout << "\nJacobian singular at x:";
                        ArrayStreamPrint(problemDim, xK);
                        }
                  exitReason = SNLR_FACTORIZATION_ERROR;
                  break;
                  }
            stepLenNR = sqrt(ArraySumSquares(problemDim, dxNR));
            //
            // Set trust length for first iteration to NR length.
            if (loopCount == 1)
                  stepLenTrust = stepLenNR;
            //
            // Report if necessary.
            if (utlReportLevel == RL_VERBOSE)
                  {
                  cout << "\n\nx: iteration:\t" << loopCount;
                  ArrayStreamPrint(problemDim, xK);
                  cout << "\nr: SS:\t" << rSquareK;
```

```
        ArrayStreamPrint(problemDim, rCurrent);
        cout << "\nstep lengths: trust:\t" << stepLenTrust
             << "\tNR:\t" << stepLenNR;
        }
    //
    //   Inner loop: Find test point on double dogleg curve,
    // at step length = trust length. Evaluate residuals and
    // cost function there. Update the current trust region,
    // and repeat until find an acceptable point.
    trustRegionStatus = TRS_INHERITED;
    while (trustRegionStatus >= TRS_INHERITED)
        {
        FindStep();
        // Take the step and evaluate residuals.
        ArrayDifference(problemDim, xK, dx, xKpl);
        if (EvaluateResiduals(xKpl, rCurrent) == RER_ERROR)
            {
            if (trustRegionStatus == TRS_INCREASED)
                {
                if (utlReportLevel == RL_VERBOSE)
                    cout << "\trejected (residual error)";
                trustRegionStatus = TRS_ACCEPTED_AFTER_REJECT;
                }
            else
                {
                if (utlReportLevel == RL_VERBOSE)
                    cout << "\thalved (residual error)";
                stepLenTrust /= 2;
                trustRegionStatus = TRS_REDUCED;
                }
            } // end residual error processing
        else
            {
            // Here, have good residuals.
            rSquareKpl = ArraySumSquares(problemDim, rCurrent);
            if (CheckZeroVector(problemDim,
                rCurrent, *absoluteZeroTolerance) == TC_TERMINATE)
                {
                // Found a solution.
                trustRegionStatus = TRS_ZERO_RESIDUALS;
                }
            else
                trustRegionStatus = EvaluateStep();
            } // end good residual processing
        } // end search for acceptable dx
    //
    //   Here, know an acceptable dx, or met a termination
    // criterion.
    if (trustRegionStatus == TRS_ACCEPTED)
        {
        // Update trust region and prepare for next iteration.
        if (deltaRSquare > 0.1*deltaRSquarePredict)
            {
            //   Recall deltas are negative, so if actual
            // reduction < 0.1*predicted reduction, i.e. if not
            // doing well, reduce trust region for next step.
            stepLenTrust /= 2;
            }
        else if (deltaRSquare <= 0.75*deltaRSquarePredict)
            {
            //   If actual reduction >= 0.75*predicted, doing
            // well. Increase trust region for next step.
            stepLenTrust *= 2;
            }
        VariableType *temp = xK; xK = xKpl; xKpl = temp;
        rSquareK = rSquareKpl;
        }
```

```
          else if (trustRegionStatus == TRS_ACCEPTED_AFTER_REJECT)
               {
               //   Reject an increased trust region. Accept the
               // step which induced the increase, and take its
               // length as the trust length.
               stepLenTrust = stepLenTrustPrev;
               VariableType *temp = xK; xK = xHold; xHold = temp;
               temp = rCurrent; rCurrent = rHold; rHold = temp;
               rSquareK = rSquareKp1Prev;
               }
          else
               {
               //   Termination. Set exit reason and break out of
               // for-loop of MAX_LOOPS iterations.
               if (trustRegionStatus == TRS_ZERO_RESIDUALS)
                    exitReason = SNLR_ZERO_RESIDUAL_VECTOR;
               else
                    exitReason = SNLR_STAGNANT_ESTIMATED_SOLUTION;
               break;
               }
          } // end sequence of double dogleg steps
     //
     //   Copy current values to arrays passed as function
     // arguments, if necessary.
     if (xKp1 != x)
          ArrayCopy(problemDim, xKp1, x);
     if (rCurrent != r)
          ArrayCopy(problemDim, rCurrent, r);
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\n\nSolvePlanarHook() terminating: "
               << SolveNonLinearReturnExplanation[exitReason]
               << "\nFinal x:";
          ArrayStreamPrint(problemDim, x);
          cout << "\nr: SS:\t" << rSquareKp1;
          ArrayStreamPrint(problemDim, r);
          }
     //
     //   Release memory allocated from free store. Since have
     // been trading arrays around between pointers, make sure
     // don't try to delete the arrays passed as arguments to
     // this function.
     delete J;
     if (xK != x)
          delete [] xK;
     if (rCurrent != r)
          delete [] rCurrent;
     delete [] dxNR;
     delete [] dx;
     if (xKp1 != x)
          delete [] xKp1;
     if (xHold != x)
          delete [] xHold;
     if (rHold != r)
          delete [] rHold;
     delete [] vectorJTr;
     return exitReason;
     } // End SolvePlanarHook().

/* ==========< End Function SolvePlanarHook >============ */


/* ==========< Helper functions >==========================
============< Helper functions >======================== */
static void FindStep(void)
```

```
{
//   Find test point on planar hook curve, at step
// length = trust length. Also find g2Tdx and predicted
// change in cost function for this step.
//
if (stepLenTrust >= stepLenNR)
     {
     // Take the full Newton-Raphson step.
     stepLenTrust = stepLenNR;
     ArrayCopy(probDim, dxNR, dx);
     g2Tdx = -2*rSquareK;
     deltaRSquarePredict = -rSquareK;
     }
else
     {
     // Take a step smaller than the full NR step.
     if (firstHook == YES)
          {
          // Calculate points needed for planar hook curve.
          firstHook = NO;
          lenSqJTr = ArraySumSquares(probDim, vectorJTr);
          //   Find Cauchy point, the minimizer of rSquare in
          // steepest descent direction if residuals were linear.
          // Recall rHold temporarily stores JJTr.
          scaleJTrToCauchy =
               lenSqJTr/ArraySumSquares(probDim, rHold);
          stepLenCauchy = scaleJTrToCauchy*sqrt(lenSqJTr);
          // Find constants needed to calculate other points.
          lenFrthJTrOnLenSqJJTr = scaleJTrToCauchy*lenSqJTr;
          c1 = scaleJTrToCauchy*rSquareK/stepLenNR/stepLenNR;
          c2 = stepLenCauchy/stepLenNR;
          c2 *= c2;
          c1SqMc2 = 2*c1 - c2;
          } // end firstHook == YES
     //
     //   Here, have planar hook information and need dx.
     // Express dx = rho1*dxCauchy + rho2*dxNR. Need to iterate
     // to choose best rho1, which makes deltaRSquarePredict
     // most negative.
     //   Locally, x stands for rho1, y for deltaRSquarePredict,
     // and z for rho2.
     //   Set up initial left and right bounds on rho1, and
     // the corresponding rho2, deltaRSquarePredict, and slope.
     double xL = 0;
     double z = stepLenTrust/stepLenNR;
     double yL = z*(z-2)*rSquareK;
     double dydxL = 2*(z-1)*(lenFrthJTrOnLenSqJJTr-c1*rSquareK);
     double xR = stepLenTrust/stepLenCauchy;
          // Note z = 0;
     double yR = xR*(xR-2)*lenFrthJTrOnLenSqJJTr;
     double dydxR = 2*xR*(1-c2/c1)*lenFrthJTrOnLenSqJJTr;
     double xLastDiscard;
     double yLastDiscard = fabs(yL) + fabs(yR);
          // Values from the left or right bound just replaced.
     double xOpt;
          // Note zOpt will be in z, yOpt in deltaRSquarePredict.
     double c3 = stepLenTrust/stepLenNR;
     c3 *= c3;
     for (int subLoop=1; subLoop<=20; subLoop++)
          {
          //   Choose as points for parabolic fit those with
          // most negative function values.
          double x1, y1, dydx1, x2, y2;
          if (yL < yR)
               {
               // Identify x1 with xL.
               x1 = xL; y1 = yL; dydx1 = dydxL;
```

```
                    if (yLastDiscard < yR)
                        {
                        x2 = xLastDiscard; y2 = yLastDiscard;
                        }
                    else
                        {
                        x2 = xR; y2 = yR;
                        }
                    }
                else
                    {
                    // Identify x1 with xR.
                    x1 = xR; y1 = yR; dydx1 = dydxR;
                    if (yLastDiscard < yL)
                        {
                        x2 = xLastDiscard; y2 = yLastDiscard;
                        }
                    else
                        {
                        x2 = xL; y2 = yL;
                        }
                    }
                //
                // Here, have points 1 and 2 defined.
                double x2Mx1 = x2 - x1;
                xOpt = x1 - dydx1*x2Mx1*x2Mx1/2/(y2-y1-dydx1*x2Mx1);
                if (xOpt < xL || xOpt > xR)
                    // Keep next guess within current bounds.
                    xOpt = (xL + xR)/2;
                z = sqrt(xOpt*xOpt*c1SqMc2 + c3) -c1*xOpt;
                deltaRSquarePredict = xOpt*(xOpt+2*z-2)*
                    lenFrthJTrOnLenSqJJTr + z*(z-2)*rSquareK;
                double dzdxOpt = -(c2*xOpt+c1*z)/(c1*xOpt+z);
                double dydxOpt = 2*
                    ((xOpt+z+xOpt*dzdxOpt-1)*lenFrthJTrOnLenSqJJTr
                    + (z-1)*dzdxOpt*rSquareK);
                //
                //   Here, have all values at new estimate of best rho1.
                // Check for small slope, then replace the left or the
                // right bound, then check for small bounded interval.
                if (deltaRSquarePredict < 0
                    && fabs(dydxOpt*xOpt/deltaRSquarePredict) < SLOPE_TOL)
                    break;
                if (dydxOpt < 0)
                    {
                    // Replace the left bound.
                    xLastDiscard = xL; yLastDiscard = yL;
                    xL = xOpt; yL = deltaRSquarePredict; dydxL = dydxOpt;
                    }
                else
                    {
                    // Replace the right bound.
                    xLastDiscard = xR; yLastDiscard = yR;
                    xR = xOpt; yR = deltaRSquarePredict; dydxR = dydxOpt;
                    }
                if ((xR - xL) < xR*RHO_TOL)
                    break;
                } // end subLoop looking for optimal rho1, rho2.
            //
            //   Here, have best rho1 in xOpt, its rho2 in z, and
            // its yOpt in deltaRSquarePredict.
            ArrayScalarProduct(probDim,
                xOpt*scaleJTrToCauchy, vectorJTr, dx);
            ArraySaxpy(probDim, z, dxNR, dx, dx);
            g2Tdx = -2*(xOpt*lenFrthJTrOnLenSqJJTr + z*rSquareK);
            } // end finding step with lenTrust < lenNR
        //
```

```
      // Here, have step dx with length stepLenTrust.
      if (utlReportLevel == RL_VERBOSE)
           cout << "\nstep:\t" << stepLenTrust;
      return;
      } // End FindStep().

static TrustRegionStatusList EvaluateStep(void)
      {
      //   Update the trust region, to seek a different dx in
      // current iteration, or to use during next iteration.
      //
      // Check if increased trust region failed.
      if (trustRegionStatus == TRS_INCREASED
           && rSquareKp1 > rSquareKp1Prev)
           {
           //   Trust region was increased into a worse cost function.
           // Restore previous values and accept.
           //   Note that previous values of actual and estimated
           // change in cost function are still valid.
           if (utlReportLevel == RL_VERBOSE)
                cout << "\trejected (worse cost)";
           return TRS_ACCEPTED_AFTER_REJECT;
           }
      //
      //   Prepare to evaluate step's progress.
      //   Both g2Tdx ("initial slope") and deltaRSquare are negative
      // for smaller rSquare. Good progress made when delta is more
      // negative than g2Tdx, or when delta < g2Tdx.
      //   Acceptable progress when delta <= ALPHA*g2Tdx.
      //   g2Tdx and deltaRSquarePredict calculated earlier.
      deltaRSquare = rSquareKp1 - rSquareK;
      //
      // Check that cost function making progress.
      if (deltaRSquare > ALPHA*g2Tdx)
           {
           // Actual rSquare too large.
           if (CheckStagnatedVector(probDim, dx, xKp1, relStepTol)
                == TC_TERMINATE)
                {
                return TRS_STAGNANT_SOLUTION;
                }
           else if (trustRegionStatus == TRS_INCREASED)
                {
                //   Last trial step was good enough that increased
                // trust length. If here, the increased length had
                // better cost than previously, so don't reject.
                return TRS_ACCEPTED;
                }
           else
                {
                // Decrease trust length.
                if (utlReportLevel == RL_VERBOSE)
                     cout << "\tcut (poor cost)";
                double lambda = g2Tdx/2/(g2Tdx - deltaRSquare);
                if (lambda < 0.1) lambda = 0.1;
                else if (lambda > 0.5) lambda = 0.5;
                stepLenTrust *= lambda;
                return TRS_REDUCED;
                }
           } // end check that making sufficient progess
      //
      //   Cost function sufficiently small. May increase or accept
      // it for current iteration, and may also change it for next
      // iteration, all depending on success of model at predicting
      // values.
      //   Make decision about current, acceptable, trust region.
      if (trustRegionStatus == TRS_REDUCED
```

```
          || stepLenTrust == stepLenNR)
          {
          //   If just reduced, or if using NR step, then don't
          // consider increasing trust region during this iteration.
          return TRS_ACCEPTED;
          }
     else if (fabs(deltaRSquareFredict - deltaRSquare)
          <= -0.1*deltaRSquare
          ||
          deltaRSquare <= g2Tdx)
          {
          //   Increase trust region if prediction was very good,
          // or if actual decrease very large. Store current values
          // so can recover later if needed.
          if (utlReportLevel == RL_VERBOSE)
               cout << "\tdoubled (good cost)";
          stepLenTrustPrev = stepLenTrust;
          stepLenTrust *= 2;
          VariableType *temp = xHold; xHold = xKpi; xKpl = temp;
          temp = rHold; rHold = rCurrent; rCurrent = temp;
          rSquareKplPrev = rSquareKpl;
          return TRS_INCREASED;
          }
     // Here, no reason to increase the acceptable trust region.
     return TRS_ACCEPTED;
     } // End EvaluateStep().

/* ==========< End helper functions >=================== */


/* ====================< End File PLANARHK.CPP >================== */
```

```
/* ====================< File PLUJCBN.HPP >=======================
=======================< rev 08/28/96 >============================

     Headers for PLUJCBN.CPP.
     Derives a class PLUJacobian from class FullJacobian.

     Inherited functions:
     Except as noted below, the class is used in the same way
as its base class. Documentation is in the header for class
FullJacobian.
     The class also "inherits" the macros which define user
preferences.

     New public functions:
(1) Solve. PLU factors the matrix, if it is not already factored,
and solve Jmdx = r (not Jdx = -r) by back substitution. Returns
SUCCESS.
     If factorization reveals matrix is nearly singular, returns
FAILURE, but leaves matrix in a form where matrix multiplications
are still defined.

     Affected inherited functions:
     These functions, valid for an unfactored matrix, are not
defined when the matrix is PLU factored (i.e. after invoking
Solve):
(1) Transpose.
(2) TProduct.
(3) UpdateRankOne.
(4) GetRowLengths.

=======================< File PLUJCBN.HPP >======================== */

#ifndef PLUJCBN_HPP
#define PLUJCBN_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "utlreprt.hpp"
     // For standard enumerations.


/* ==========< Class PLUJacobian >============================
==============< Class PLUJacobian >======================== */
class PLUJacobian
     : public FullJacobian

     {
public:
     // =====< Constructors >=====
     PLUJacobian(void);
          //   Default constructor: PLUJacobian J;
          // Provided for consistency only. Allocates for
          // a one-dimensional problem.
     PLUJacobian(const int problemDimension);
          // Standard constructor: PLUJacobian J(20);
     PLUJacobian(const PLUJacobian &const rval);
          //   Copy-initializer. Preferred: PLUJacobian B(J);
          // Alternate form: PLUJacobian B = J;
          // This constructor implicitly used e.g. to pass
          // Jacobian as return from functions.

     // =====< Destructor >=====
     ~PLUJacobian(void);

     // =====< Memory management >=====
     void NewProblemDimension(const int newProblemDimension);
```

```
      void SwapWith(PLUJacobian *other);

      // =====< Assign from other matrix >=====
      void CopyJacobian(const PLUJacobian &const copyFrom);

      // =====< Linear algebra >=====
      void Product
            (const VariableType rval[], VariableType result[]) const;
      void TProduct
            (const VariableType rval[], VariableType result[]) const;
      SuccessFailureList Solve
            (VariableType *const mdx, const VariableType *const r);
            // Factors automatically if required.

      // =====< Reporting >=====
      void StreamPrint(void) const;

private:
      // =====< Matrix variables >=====
      int *rowJumpIndex;
      double *auxRowInfo;
      YesNoList isSingular;

      // =====< Memory management >=====
      void InitializePLU(void);

      // =====< Linear algebra >=====
      void FactorPLU(void);

      };

/* ==========< End class PLUJacobian >================== */

#endif


/* ====================< End File PLUJCBN.HPP >==================== */
```

```
/* ====================< File PLUJCBN.CPP >==========================
=======================< rev 08/29/96 >================================

      Defines a class PLUJacobian.
      Documentation is in the header file.

      PLU factorization:
      Notation used throughout has PJ = LU, or J = (P^-1)LU.

      Programming notes for squareMatrix:
      The squareMatrix gets both L and U. L has ones on the diagonal;
these are not stored, and the diagonal elements of squareMatrix
are the diagonal elements of U.
      L is lower and U is upper diagonal in logical memory.
However they are not lower and upper diagonal in physical memory.
As pivots are identified during elimination, the row data remain
in the same memory locations and the swaps are implicit.

      Programming notes for rowJumpIndex:
      The rowJumpIndex stores the relative jump from squareMatrix
required to reach the physical memory for the logical row
indexed. Logical rows are numbered from 0 to numVars-1. Thus
rowJumpIndex[0]=2 says that row 0 of PJ is row 2 of squareMatrix,
and a pointer to the beginning of the row is set there using
squareMatrix + 2*numVars.
      Before factorization, rowJumpIndex stores no useful information.
      The rowJumpIndex is used to run through the physical rows in
their logical (swapped) order. To find the pivot order of a
given physical row, find the element of rowJumpIndex in which the
physical row number appears, then add one (since matrix rows
and array elements are indexed from zero, but the pivot order
starts from one). Since the pivot is in the diagonal element,
this same rule also tells the first column of a physical row
to contain a nonzero element of U, and the column of the
physical row which would contain the "1" element of L, if that
diagonal element was stored.
      Thus:
(1) U is implicitly row-ordered in squareMatrix. RowJumpIndex
tells how to find each logical row in physical memory. For a
given physical row, the diagonal element rule gives the leftmost
nonzero entry in U.
(2) L is implicitly row-ordered in squareMatrix. RowJumpIndex
tells how to find each logical row in physical memory. For a
given physical row, the diagonal element rule gives the "1"
entry of L, and the leftmost column of physical memory which does
not store information for L.
(3) (P^-1)*L is in the squareMatrix in its proper row order, since
to turn squareMatrix into the actual L requires premultiplication
by P. For a given physical row, the diagonal element rule gives
the "1" entry.
      Note that a typical solution to the implicit row order
problem is to define an array of pointers, initialized to the
rows of squareMatrix, and to swap these pointers as pivot rows
are chosen. Speed tests indicate that for small problems, pointer
arithmetic using rowJumpIndex is as fast as indexing into an
array and dereferencing the pointer found there. As the problem
dimension increases, the rowJumpIndex becomes faster.
      In terms of storage, the difference is between storing one
integer per row and storing one address pointer per row.

      Programming notes for auxRowInfo:
      The auxRowInfo is used by various procedures as needed
and does not store useful data from one function to another.
Thus it need not be copied when a Jacobian is duplicated. It
is declared double since at some times it stores VariableTypes
while at other times it stores JacobianTypes.
```

```
     Programming notes for isSingular:
     Set to YES when factorization indicates matrix is singular
to machine precision. Note that in the rare event an all-zero
row is identified before factorization begins, isSingular is
set to YES but the matrix is left UNFACTORED.

======================< File PLUJCBN.CPP >======================== */

#include <float.h>
#include <stdlib.h>
#include <math.h>

#include "utldepnd.hpp"
     // For macros governing user preferences.
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "plujcbn.hpp"


/* ==========< User preferences >============================
     Check user preferences.
=============< User preferences >======================== */
// =====< Variable Type >=====
#ifndef VARIABLES_ARE_DOUBLES
#error VARIABLES_ARE_DOUBLES is not defined
#endif

// =====< Jacobian element type >=====
#ifndef JACOBIAN_ELEMENTS_ARE_DOUBLES
#error JACOBIAN_ELEMENTS_ARE_DOUBLES is not defined
#endif

#if (JACOBIAN_ELEMENTS_ARE_DOUBLES == 1)
const double MIN_NONZERO_PIVOT = DBL_EPSILON;
#else
const float MIN_NONZERO_PIVOT = FLT_EPSILON;
#endif

// =====< Copy arrays >=====
#ifndef ARRAY_COPIES_USE_MEMCPY
#error ARRAY_COPIES_USE_MEMCPY is not defined
#endif

#if (ARRAY_COPIES_USE_MEMCPY == 1)
const int JACOBIAN_TYPE_SIZE = sizeof(JacobianType);
#endif

/*
Control overflows is unimplemented.
// =====< Control overflows >=====
#ifndef ARRAY_OPS_CONTROL_OVERFLOWS
#error ARRAY_OPS_CONTROL_OVERFLOWS is not defined
#endif

#if (ARRAY_OPS_CONTROL_OVERFLOWS == 1)
#if (JACOBIAN_ELEMENTS_ARE_DOUBLES == 1)
const double JACOBIAN_TYPE_MAX = DBL_MAX;
#else
const float JACOBIAN_TYPE_MAX = FLT_MAX;
#endif
#endif
*/

/*
Scale inner products is unimplemented.
```

```
// =====< Scale inner products >=====
// Note LOG_BASE and SCALE_THRESHOLD are declared extern.
#ifndef INNER_PRODUCTS_USE_SCALE_FACTOR
#error INNER_PRODUCTS_USE_SCALE_FACTOR is not defined
#endif
*/

/* ==========< End user preferences >=================== */


/* ==========< Constructors >==============================
     Create PLU-factored Jacobian matrix objects.
     Inline (defined in header file):
None.
     Inherited (defined in base class):
FullJacobian(void);
FullJacobian(const int problemDimension);
FullJacobian(const FullJacobian &const rval);
     Virtual (defined for derived classes):
None.
=============< Constructors >============================ */
PLUJacobian::PLUJacobian(void)
     // Member initialization list required for base class.
     :FullJacobian(1)
     {
     // Default constructor.
     InitializePLU();
     }

PLUJacobian::PLUJacobian(const int problemDimension)
     :FullJacobian(problemDimension)
     {
     // Construct and size matrix.
     InitializePLU();
     }

PLUJacobian::PLUJacobian
     (const PLUJacobian &const rval)
     :FullJacobian(rval.numVars)
     {
     // Copy-initializer.
     InitializePLU();
     CopyJacobian(rval);
     }

/* ==========< End constructors >======================== */


/* ==========< Destructor >================================
     Called when the Jacobian object leaves scope.
     Inline (defined in header file):
None.
     Inherited (defined in base class):
~FullJacobian(void);
=============< Destructor >============================== */
PLUJacobian::~PLUJacobian(void)
     {
     //   Release dynamically-allocated memory to free store.
     // Note squareMatrix released by destructor of base
     // class.
     delete [] rowJumpIndex;
     delete [] auxRowInfo;
     }

/* ==========< End destructor >========================== */
```

```
/* ==========< Memory management >=========================
     Control matrix object memory.
     Inline (defined in header file):
None.
     Inherited (defined in base class):
void NewProblemDimension(const int newProblemDimension);
void SwapWith(FullJacobian *other);
void InitializeFull(const int problemDimension);
=============< Memory management >====================== */
// =====< Memory management, public >=====
void PLUJacobian::NewProblemDimension
     (const int newProblemDimension)
     {
     delete [] rowJumpIndex;
     delete [] auxRowInfo;
     FullJacobian::NewProblemDimension(newProblemDimension);
     InitializePLU();
     return;
     } // End NewProblemDimension().

void PLUJacobian::SwapWith(PLUJacobian *other)
     {
     //   Swap data unique to the PLUJacobian. Recall auxRowInfo
     // stores no useful data.
     int *tempIntP = rowJumpIndex;
     rowJumpIndex = other->rowJumpIndex;
     other->rowJumpIndex = tempIntP;
     YesNoList tempYNList = isSingular;
     isSingular = other->isSingular;
     other->isSingular = tempYNList;
     //
     // Swap data defined in base class.
     FullJacobian::SwapWith(other);
     return;
     }

// =====< Memory management, private >=====
void PLUJacobian::InitializePLU(void)
     {
     rowJumpIndex = new int [numVars];
     auxRowInfo = new JacobianType [numVars];
     if (!rowJumpIndex || !auxRowInfo)
          FATAL(ERR_FREE_STORE, 1091612);
     isSingular = NO;
          // Not needed.
     return;
     }

/* ==========< End memory management >=================== */


/* ==========< Assign to unfactored matrix >================
     Set matrix elements and control factorization status.
     Inline (defined in header file):
None.
     Inherited (defined in base class):
JacobianType *GetRowPointer(const int rowToGet);
void SetRow(const int rowToSet, const JacobianType newRow[]);
void SetColumn(const int colToSet, const JacobianType newCol[]);
JacobianType *GetMatrixPointer(void);
void SetAcceptMatrix(const JacobianType newMat[]);
void SetAcceptMatrixTranspose(const JacobianType newMatT[]);
void AcceptMatrix(void);
=============< Assign to unfactored matrix >============ */
/* ==========< End assign to unfactored matrix >========= */
```

```
/* ==========< Assign from other matrix >==================
     Assign to Jacobian object.
     Inline (defined in header file):
None.
     Inherited (defined in base class):
void CopyJacobian(const FullJacobian &const copyFrom);
==============< Assign from other matrix >================ */
// =====< Assign from other matrix, public >=====
void PLUJacobian::CopyJacobian
     (const PLUJacobian &const copyFrom)
     {
     if (numVars != copyFrom.numVars)
          NewProblemDimension(copyFrom.numVars);
     FullJacobian::CopyJacobian(copyFrom);
     //
     // Copy rowJumpIndex. Recall auxRowInfo stores no useful data.
#if (ARRAY_COPIES_USE_MEMCPY == 1)
     memcpy(rowJumpIndex, copyFrom.rowJumpIndex,
          numVars*sizeof(int));
#else
     int *myRowJump = rowJumpIndex;
     int *copyFromRowJump = copyFrom.rowJumpIndex;
     for (int arrayEl=0; arrayEl<numVars; arrayEl++)
          *myRowJump++ = *copyFromRowJump++;
#endif
     //
     isSingular = copyFrom.isSingular;
     return;
     } // End CopyJacobian().

/* ==========< End assign from other matrix >============ */


/* ==========< Linear algebra >============================
     Inline (defined in header file):
None.
     Inherited (defined in base class):
void Transpose(void);
void Product
     (const VariableType rval[], VariableType result[]) const;
void TProduct
     (const VariableType rval[], VariableType result[]) const;
void UpdateRankOne
     (const VariableType u[], const VariableType vT[]);
==============< Linear algebra >========================== */
// =====< Linear algebra, public >=====
//void PLUJacobian::Transpose(void)
     //   Not defined for PLU factored matrix. Inherited
     // version already tests and prints error message.

void PLUJacobian::Product
     (const VariableType rval[], VariableType result[]) const
     {
     if (matrixStatus <= MS_UNFACTORED)
          {
          FullJacobian::Product(rval, result);
          return;
          }
     //
     //   Find result = J*rval = (P^-1)*L*U*rval.
     //   Note no test for rval == result even though this
     // makes results wrong.
     //
     //   Find auxRowInfo = U*rval. U is implicitly row-ordered
     // in squareMatrix. Do not implicitly order auxRowInfo.
     const int *jumpStepP = rowJumpIndex;
     double *auxStepP = auxRowInfo;
```

```
      for (int row=0; row<numVars; row++)
            {
            //   Find logical row of U in squareMatrix, and go
            // to diagonal element, which is first element of U.
            const JacobianType *matStepP =
                  squareMatrix + numVars*(*jumpStepP++) + row;
            // Start at corresponding row of rval.
            const VariableType *rvalStepP = rval + row;
            // Calculate inner product of remaining columns.
            long double innerProduct = (*matStepP)*(*rvalStepP);
            for (int col=row+1; col<numVars; col++)
                  innerProduct += (*(++matStepP))*(*(++rvalStepP));
            *auxStepP++ = innerProduct;
            }
      //
      //   Find result = (P^-1)*L*auxRowInfo. (P^-1)*L is already
      // in its proper row order in squareMatrix, and auxRowInfo
      // is in proper row order.
      jumpStepP = rowJumpIndex;
      VariableType *resultStepP = result;
      for (row=0; row<numVars; row++)
            {
            // Go to the physical row in squareMatrix.
            const JacobianType *matStepP = squareMatrix + numVars*row;
            //   Find diagonal element in logical row,  where have
            // an implicit "1" which is rightmost nonzero element
            // in L.
            int oneInCol = 0;
            jumpStepP = rowJumpIndex;
            while (*jumpStepP++ != row)
                  oneInCol++;
            //   Find inner product of current row of (P^-1)*L
            // and auxRowInfo up to diagonal element noninclusive.
            auxStepP = auxRowInfo;
            long double innerProduct = 0;
            for (int col=0; col<oneInCol; col++)
                  innerProduct += (*matStepP++)*(*auxStepP++);
            // Add contribution from "1" on diagonal, and store result.
            innerProduct += *auxStepP;
            *resultStepP++ = innerProduct;
            }
      return;
      } // End Product();

void PLUJacobian::TProduct
      (const VariableType rval[], VariableType result[]) const
      {
      if (matrixStatus <= MS_UNFACTORED)
            {
            FullJacobian::TProduct(rval, result);
            return;
            }
      //
      //   Find result = JT*rval = ((P^-1)*L*U)T*rval
      // = (UT)*((P^-1)L)T*rval.
      //   Note no test for rval == result even though this
      // makes results wrong.
      //
      //   Find auxRowInfo = ((P^-1)*L)T*rval. (P^-1)L is
      // in squareMatrix in its proper row order. To find implicit
      // 1 in each physical row, find the element of rowJumpIndex
      // in which the physical row number appears, then add one.
      //   L lies to left of the implicit 1; to right, L has zeros.
      double *auxStepP = auxRowInfo;
      for (int row=0; row<numVars; row++)
            {
            // Zero each element of auxRowInfo.
```

```
            *auxStepP++ = 0;
            }
      //    For each row of (P^-1)L, take corresponding element
      // of rval. This rvalCurrent multiplies each column of
      // (P^-1)L and the products add in to the elements auxRowInfo.
      const VariableType *rvalStepP = rval;
      for (row=0; row<numVars; row++)
            {
            // Find where in this row the implicit one appears.
            int oneInCol = 0;
            const int *jumpStepP = rowJumpIndex;
            while (*jumpStepP++ != row)
                  oneInCol++;
            //    Add column products to each element of auxRowInfo
            // until hit the implicit one.
            const JacobianType *matStepP = squareMatrix + numVars*row;
            auxStepP = auxRowInfo;
            VariableType rvalCurrent = *rvalStepP++;
            for (int col=0; col<oneInCol; col++)
                  *auxStepP++ += rvalCurrent*(*matStepP++);
            *auxStepP += rvalCurrent;
            }
      //
      //    Find result = UT*auxRowInfo. U is stored in squareMatrix
      // with its logical rows out of physical order. To find the
      // ith row of U, 0<=i<numVars, go to squareMatrix +
      // numVars*rowJumpIndex[i].
      //    For the ith logical row, the ith element is the diagonal,
      // leftmost, element of U.
      VariableType *resultStepP = result;
      for (row=0; row<numVars; row++)
            *resultStepP++ = 0;
      //    Each row of U is a column of UT. Taking rows in logical
      // order, multiply each row by an element of auxRowInfo and
      // add in to result.
      const int *jumpStepP = rowJumpIndex;
      for (row=0; row<numVars; row++)
            {
            //    In squareMatrix, skip in to physical start of logical
            // row, then to diagonal element. Also skip in to matching
            // element in result vector.
            const JacobianType *matStepP = squareMatrix +
                  numVars*(*jumpStepP++) + row;
            resultStepP = result + row;
            double pInvLTrValCurrent = *(auxRowInfo + row);
            for (int col=row; col<numVars; col++)
                  *resultStepP++ += pInvLTrValCurrent*(*matStepP++);
            }
      return;
      } // End TProduct().

//void PLUJacobian::UpdateRankOne
      //(const VariableType u[], const VariableType vT[])
      //    Not defined for PLU factored matrix. Inherited
      // version already tests and prints error message.

SuccessFailureList PLUJacobian::Solve
      (VariableType *const mdx, const VariableType *const r)
      {
      //    Solve J*dx = -r as J*mdx = r.
      //    To avoid having to take negative of r, this routine
      // is formulated as J*mdx = r, and mdx should be subtracted
      // from x to find the next estimate of x. This routine is
      // just the solution of the classical Ax = b. For notational
      // convenience in the code, comments refer to x and b, not
      // to mdx and r.
      //    Routine does not check to insure mdx and r are distinct
```

```
      // arrays.
      //
      if (matrixStatus <= MS_UNFACTORED)
           FactorPLU();
      //
      if (isSingular == YES)
           return FAILURE;
      //
      //   Have PAx = LUx = Pb. First solve Lx' = Pb (forward
      // substitution) then solve Ux = x' (back substitution).
      //
      //   Solve Lx' = Pb. Store x' in mdx.
      //   No physical row exchanges were performed on squareMatrix,
      // so to turn squareMatrix into the actual LU requires
      // premultiplication by P. This is done implicitly at the
      // same time it is done implicitly on b.
      //   First row of forward substitution is straight assignment
      // of (mdx)[0] = (Pb)[0].
      const int *jumpStepP = rowJumpIndex;
      *mdx = *(r + *jumpStepP);
      //
      for (int row=1; row<numVars; row++)
           {
           // Find logical row of L in physical memory.
           jumpStepP++;
           const JacobianType *matStepP =
                squareMatrix + numVars*(*jumpStepP);
           //   Find Lx' up to row noninclusive. x' is in mdx
           // and is not row-swapped.
           VariableType *xStepP = mdx;
           long double innerProduct = (*matStepP)*(*xStepP);
           for (int col=1; col<row; col++)
                innerProduct += (*(++matStepP))*(*(++xStepP));
           //   Find new element of x', which is multiplied by
           // L's diagonal "1" element.
           *(++xStepP) = *(r + *jumpStepP) - innerProduct;
           }
      //
      //   Back substitution: Solve Ux = x'.
      //   U is stored in squareMatrix and is implicitly row-
      // ordered. x' is in mdx and is in the proper row order.
      //   Work from bottom row upwards.
      for (row=(numVars-1); row>=0; row--)
           {
           //   Find logical row of U in squareMatrix, and move
           // to the rightmost column.
           const JacobianType *matStepP = squareMatrix +
                numVars*(*(rowJumpIndex + row)) + (numVars-1);
                //   Note parens in (numVars-1) used to keep
                // pointer from leaving squareMatrix bounds,
                // where behavior undefined.
           //   Find Ux, working backward from rightmost column
           // to diagonal element noninclusive. x is in mdx and is
           // not row-swapped.
           VariableType *xStepP = mdx + (numVars-1);
           long double innerProduct = 0;
           for (int col=(numVars-1); col>row; col--)
                innerProduct += (*matStepP--)*(*xStepP--);
           //   Find new element of x, which is multiplied by
           // U's diagonal element.
           *xStepP -= innerProduct;
           *xStepP /= *matStepP;
           }
      return SUCCESS;
      } // End Solve().

// =====< Linear algebra, private >=====
```

```
void PLUJacobian::FactorPLU(void)
      {
      //   Find PJ = LU by classical Gaussian elimination with
      // row pivoting.
      //
      if (matrixStatus != MS_UNFACTORED)
            FATAL(ERR_NEED_UNFACTORED, 1061612);
      //
      //   Set the implicit row scales, as largest magnitude in
      // each row. Also initialize the row jump indices.
      JacobianType *matStepP = squareMatrix;
      int *jumpStepP = rowJumpIndex;
      double *scaleStepP = auxRowInfo;
      for (int row=0; row<numVars; row++)
            {
            *jumpStepP++ = row;
            JacobianType big = fabs(*matStepP++);
            for (int col=1; col<numVars; col++)
                  {
                  JacobianType testBig = fabs(*matStepP++);
                  if (testBig > big) big = testBig;
                  }
            if (big == 0)
                  {
                  //   An all-zero row. Do not factor matrix,
                  // but mark it as singular.
                  isSingular = YES;
                  return;
                  }
            *scaleStepP++ = big;
            }
      //
      // Have row scales; factorization can go forward.
      matrixStatus = MS_MODIFIED;
      isSingular = NO;
      //
      //   Classical elimination. Choose numVars-1 pivots and
      // subtract multiples of the pivot row from the rows
      // beneath. Implicit row swapping handled by rowJumpIndex,
      // implicit row scaling handled by auxRowInfo.
      for (int pivotRow=0; pivotRow<(numVars-1); pivotRow++)
            {
            //   Use (pivotRow=0..) instead of (pivotRowCount=1..)
            // to make array indexing easy.
            //   Find current logical row in unfactored matrix.
            jumpStepP = rowJumpIndex + pivotRow;
            //   Select pivot column. Since rows only are swapped,
            // physical column and logical column are same.
            JacobianType *matColP = squareMatrix + pivotRow;
            //   Diagonal element for this column is current pivot.
            // Find relative pivot size.
            double big = fabs(*(matColP + numVars*(*jumpStepP))) /
                  (*(auxRowInfo + *jumpStepP));
            int *markP = 0;
            for (row=pivotRow; row<(numVars-1); row++)
                  {
                  // Find relative pivot in next logical row.
                  jumpStepP++;
                  JacobianType testBig =
                        fabs(*(matColP + numVars*(*jumpStepP))) /
                        (*(auxRowInfo + *jumpStepP));
                  if (testBig > big)
                        {
                        big = testBig;
                        markP = jumpStepP;
                        }
                  }
```

```
        //
        // Return to current pivot row.
        jumpStepP = rowJumpIndex + pivotRow;
        if (markP != 0)
             {
             //   Switch row indices so marked row is the true
             // pivot row, and physical row formerly pointed
             // to is still available at next pivot selection.
             int tempInt = *jumpStepP;
             *jumpStepP = *markP;
             *markP = tempInt;
             }
        //
        //   Here, jumpStepP indexes to physical row selected as
        // logical pivot row. Note the pivot element.
        JacobianType *pivotElementP = matColP + numVars*(*jumpStepP);
        //
        // Check for a good pivot.
        if (big < MIN_NONZERO_PIVOT
             && fabs(*pivotElementP) < MIN_NONZERO_PIVOT)
             {
             //   A zero scaled pivot element in every row, and
             // best one is in some absolute sense very small for
             // multiplicative operations. Leave the pivot element
             // alone, but don't eliminate-- subtract zero times
             // pivot row from remaining rows.
             isSingular = YES;
             for (row=pivotRow; row<(numVars-1); row++)
                  {
                  //   Leave pivot alone, even if nonzero, but set
                  // multipliers to zero. Multipliers are in the
                  // same column, in logical rows below pivot row.
                  jumpStepP++;
                  *(matColP + numVars*(*jumpStepP)) = 0;
                  }
             } // Done setting multipliers to zero.
        else
             {
             //   LU factorization. Steps: (1) In current column,
             // divide row elements under the pivot by the pivot
             // element. These are the L's. (2) In the remaining
             // columns to the right, for rows under the pivot
             // row, subtract that row's L times the pivot row.
             for (row=pivotRow; row<(numVars-1); row++)
                  {
                  // Move to same column in next logical row.
                  jumpStepP++;
                  matStepP = matColP + numVars*(*jumpStepP);
                  //   Find current row's L. Keep full precision
                  // even if matrix elements are floats.
                  long double currRowMult =
                       *matStepP / *pivotElementP;
                  *matStepP = currRowMult;
                  // Subtract L*pivot row from current row.
                  JacobianType *pivotRowStepP = pivotElementP;
                  for (int col=pivotRow; col<(numVars-1); col++)
                       *++matStepP -= (currRowMult * (*++pivotRowStepP));
                  }
             } // Done setting multipliers and subtracting rows.
        } // Done processing current pivot row.
   //
   //   Process last column in last row. Check for nearly zero
   // pivot, both in absolute and in scaled sense.
   jumpStepP = rowJumpIndex + (numVars-1);
   matStepP = squareMatrix + numVars*(*jumpStepP) + (numVars-1);
   scaleStepP = auxRowInfo + (*jumpStepP);
   if (fabs(*matStepP) < MIN_NONZERO_PIVOT
```

```
          && fabs(*matStepP)/(*scaleStepP) < MIN_NONZERO_PIVOT)
          isSingular = YES;
     return;
     } // End FactorPLU().

/* ==========< End linear algebra >====================== */


/* ==========< Reporting >=================================
     Inline (defined in header file):
None.
     Inherited (defined in base class):
int GetProblemDimension(void) const;
void GetRowLengths(VariableType rowLengths[]) const;
void StreamPrint(void) const;
=============< Reporting >============================== */
// =====< Reporting, public >=====
//void PLUJacobian::GetRowLengths(VariableType rowLengths[]) const
     //   Not defined for PLU factored matrix. Inherited
     // version already tests and prints error message.

void PLUJacobian::StreamPrint(void) const
     {
     if (matrixStatus <= MS_UNFACTORED)
          {
          FullJacobian::StreamPrint();
          return;
          }
     //
     // Matrix is PLU factored, and perhaps singular.
     if (isSingular == YES)
          cout.put('\n') << ERR_LOOKS_SINGULAR;
     //   Print squareMatrix as the factorization PJ = LU.
     // Print out (P^-1)L, then U.
     //   Each matrix is identified by a heading. Each block
     // gets a heading identifying the column numbers, unless
     // all matrix columns are to be printed.
     //
     cout << "\nP-inverse*L:";
     //   L is in the squareMatrix with implicit row shifting.
     // To turn squareMatrix into the actual L, premultiply by
     // P. Then premultiplying by P^-1 takes the squareMatrix
     // back to its original form. Therefore (P^-1)L is in its
     // proper row order as it stands.
     int printColsPerLine = utlReportColsPerLine;
     if (printColsPerLine < 1
          || printColsPerLine > numVars)
          printColsPerLine = numVars;
     int startCol = 1;
     int endCol = printColsPerLine;
     OnOffList printBlockHeadingMode;
     if (endCol == numVars) printBlockHeadingMode = OFF;
     else printBlockHeadingMode = ON;
     //
     while (startCol <= numVars)
          {
          if (endCol > numVars) endCol = numVars;
          if (printBlockHeadingMode == ON)
               {
               cout << "\ncolumns " << startCol
                    << " to " << endCol;
               cout.put(':');
               }
          const JacobianType *currRowStartColP =
               squareMatrix + (startCol - 1);
          for (int row=0; row<numVars; row++)
               {
```

```
                // Find column requiring a one.
                int oneInCol = 1;
                const int *jumpStepP = rowJumpIndex;
                while (*jumpStepP++ != row)
                        oneInCol++;
                // Start new line.
                cout.put('\n');
                // Print columns with delimiter after.
                const JacobianType *rowStepP = currRowStartColP;
                for (int colStep=startCol;
                        (colStep<oneInCol
                        && colStep<endCol); colStep++)
                        {
                        cout << *rowStepP++;
                        cout.put('\t');
                        }
                if (colStep == oneInCol)
                        {
                        cout.put('1');
                        colStep++;
                        for (; colStep<=endCol; colStep++)
                                cout << "\t0";
                        }
                else if (colStep > oneInCol)
                        cout.put('0');
                else
                        cout << *rowStepP;
                currRowStartColP += numVars;
                }
        startCol += printColsPerLine;
        endCol += printColsPerLine;
        }

//
cout << "\nU:";
//   U is implicitly ordered. RowJumpIndex tells how to
// find each row, and first nonzero element of row is in
// diagonal element.
startCol = 1;
endCol = printColsPerLine;
//
while (startCol <= numVars)
        {
        if (endCol > numVars) endCol = numVars;
        if (printBlockHeadingMode == ON)
                {
                cout << "\ncolumns " << startCol
                        << " to " << endCol;
                cout.put(':');
                }
        const int *jumpStepP = rowJumpIndex;
        for (int rowCount=1; rowCount<=numVars; rowCount++)
                {
                // Go to the correct row.
                const JacobianType *rowStepP =
                        squareMatrix + numVars*(*jumpStepP++) + (startCol-1);
                // Start new line.
                cout.put('\n');
                // Print columns with delimiter after.
                for (int colStep=startCol;
                        (colStep<rowCount
                        && colStep<endCol); colStep++, rowStepP++)
                        cout << "0\t";
                cout << *rowStepP;
                colStep++;
                for (; colStep<=endCol; colStep++)
                        cout.put('\t') << *++rowStepP;
```

```
                }
            startCol += printColsPerLine;
            endCol += printColsPerLine;
            }
        return;
        } // End StreamPrint().

/* ==========< End reporting >============================ */


/* ====================< End File PLUJCBN.CPP >==================== */
```

PrBroyTr.cpp

```
/* ====================< File PRBROYTR.CPP >=========================
=====================< rev 07/31/96 >==============================

     Defines the variably-dimensioned Broyden tridiagonal function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is not given.

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

======================< File PRBROYTR.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utlinput.hpp"
#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


void StoredDimension(int *problemDimension, int store);


/* ==========< Problem discovery >==========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     //   Since problem dimension is not known at compile
     // time, get it from user, and store for access by
     // residual and Jacobian functions.
     int problemDim;
     cout << "Broyden tridiagonal function."
          << "\nProblem dimension: ";
     cin >> problemDim;
     if (problemDim < 1)
          problemDim = 20;
     StoredDimension(&problemDim, 1);
     return problemDim;
     }

void GetStartPoint(VariableType x[])
     {
     int problemDim;
     StoredDimension(&problemDim, 0);
     //
     cout << "Standard starting pencil?";
     if (AskOneChoice("yes,no") == 1)
          {
          float mult;
          cout << "Pencil multiplier (1): ";
          cin >> mult;
          for (int i=0; i<problemDim; i++)
               *x++ = -mult;
          }
     else
          {
          cout << "\nStarting point:\n x1: ";
          cin >> *x;
          cout << "Same for rest?";
          if (AskOneChoice("yes,no") == 1)
```

```
                {
                VariableType sameVal = *x;
                for (int i=2; i<=problemDim; i++)
                     *++x = sameVal;
                }
           else
                {
                for (int i=2; i<=problemDim; i++)
                     {
                     cout << " x" << i << ": ";
                     cin >> *++x;
                     }
                }
           }
      return;
      }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >=========================
=============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
      (const VariableType x[], VariableType r[],
      {
      residualCounter.Call();
      //
      int problemDim;
      StoredDimension(&problemDim, 0);
      //
      VariableType xLast = 0;
      VariableType xCurrent = *x++;
      for (int i=1; i<problemDim; i++)
           {
           VariableType xNext = *x++;
           *r++ = (3 - 2*xCurrent)*xCurrent - xLast - 2*xNext + 1;
           xLast = xCurrent;
           xCurrent = xNext;
           }
      *r = (3 - 2*xCurrent)*xCurrent - xLast + 1;
      return RER_SUCCESS;
      }

JacobianEvaluationReturnList GetJacobian
      (const VariableType x[], const VariableType r[],
      FullJacobian *J)
      {
      jacobianCounter.Call();
      //
      int problemDim;
      StoredDimension(&problemDim, 0);
      //
      JacobianType *matP = J->GetMatrixPointer();
      for (int row=0; row<problemDim; row++)
           {
           const VariableType *xStepP = x;
           for (int col=0; col<problemDim; col++)
                {
                if (col == row)
                     *matP++ = 3 - 4*(*xStepP);
                else if (col == (row-1))
                     *matP++ = -1;
                else if (col == (row+1))
                     *matP++ = -2;
                else
                     *matP++ = 0;
                xStepP++;
```

```
                }
            }
        J->AcceptMatrix();
        return JER_SUCCESS_ANALYTIC;
        }

/* ==========< End problem evaluation >================== */


/* ==========< Function StoredDimension >==================
============< Function StoredDimension >================= */
void StoredDimension(int *problemDimension, int store)
        {
        static int storedDimension;
        if (store == 1)
            storedDimension = *problemDimension;
        else
            *problemDimension = storedDimension;
        return;
        }

/* ==========< End Function StoredDimension >============ */


/* ====================< End File PRBROYTR.CPP >=================== */
```

PrDsctBv.cpp

```
/* ====================< File PRDSCTBV.CPP >=========================
=======================< rev 03/21/96 >================================

     Defines a variably-dimensioned discrete boundary value function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is not given.

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

=======================< File PRDSCTBV.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utlinput.hpp"
#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


void StoredDimension(int *problemDimension, int store);


/* ==========< Problem discovery >=============================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     //   Since problem dimension is not known at compile
     // time, get it from user, and store for access by
     // residual and Jacobian functions.
     int problemDim;
     cout << "Discrete Boundary Value function."
          << "\nProblem dimension: ";
     cin >> problemDim;
     if (problemDim < 1)
          problemDim = 20;
     StoredDimension(&problemDim, 1);
     return problemDim;
     }

void GetStartPoint(VariableType x[])
     {
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     cout << "Standard starting pencil?";
     if (AskOneChoice("yes,no") == 1)
          {
          float mult;
          cout << "Pencil multiplier (1): ";
          cin >> mult;
          VariableType *xStepP = x;
          for (int i=1; i<=problemDim; i++)
               {
               double tSubi = double(i)/pDp1;
               *xStepP++ = mult*tSubi*(tSubi - 1);
               }
          }
     else
```

```
          {
          cout << "\nStarting point:\n x1: ";
          cin >> *x;
          cout << "Same for rest?";
          if (AskOneChoice("yes,no") == 1)
               {
               VariableType sameVal = *x;
               for (int i=2; i<=problemDim; i++)
                    *++x = sameVal;
               }
          else
               {
               for (int i=2; i<=problemDim; i++)
                    {
                    cout << " x" << i << ": ";
                    cin >> *++x;
                    }
               }
          }
     return;
     }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >=========================
============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     double temp;
     VariableType xLast = 0;
     VariableType xCurrent = *x++;
     for (int i=1; i<problemDim; i++)
          {
          VariableType xNext = *x++;
          temp = xCurrent + double(i)/pDp1 + 1;
          *r++ = 2*xCurrent - xLast - xNext +
               temp*temp*temp/pDp1/pDp1/2;
          xLast = xCurrent;
          xCurrent = xNext;
          }
     temp = xCurrent + double(problemDim)/pDp1 + 1;
     *r = 2*xCurrent - xLast + temp*temp*temp/pDp1/pDp1/2;
     return RER_SUCCESS;
     }

JacobianEvaluationReturnList GetJacobian
     (const VariableType x[], const VariableType r[],
     FullJacobian *J)
     {
     jacobianCounter.Call();
     //
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     JacobianType *matP = J->GetMatrixPointer();
     for (int row=1; row<=problemDim; row++)
          {
          double diagEl = (*x++) + double(row)/pDp1 + 1;
```

```
            diagEl = 2 + diagEl*diagEl*3/pDpl/pDpl/2;
            for (int col=1; col<=problemDim; col++)
                  {
                  if (col == row)
                        *matP++ = diagEl;
                  else if (col == (row-1)
                        || col == (row+1))
                        *matP++ = -1;
                  else
                        *matP++ = 0;
                  }
            }
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================== */


/* ==========< Function StoredDimension >===================
============< Function StoredDimension >================ */
void StoredDimension(int *problemDimension, int store)
      {
      static int storedDimension;
      if (store == 1)
            storedDimension = *problemDimension;
      else
            *problemDimension = storedDimension;
      return;
      }

/* ==========< End Function StoredDimension >============ */


/* ====================< End File PRDSCTBV.CPP >=================== */
```

```
/* ====================< File PRDSCTIN.CPP >==========================
=======================< rev 03/21/96 >================================

     Defines a variably-dimensioned discrete integral function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is not given.

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

=======================< File PRDSCTIN.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utlinput.hpp"
#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


void StoredDimension(int *problemDimension, int store);


/* ==========< Problem discovery >==========================
=============< Problem discovery >====================== */
int GetProblemDimension(void)
     {
     //   Since problem dimension is not known at compile
     // time, get it from user, and store for access by
     // residual and Jacobian functions.
     int problemDim;
     cout << "Discrete Integral Equation function."
          << "\nProblem dimension: ";
     cin >> problemDim;
     if (problemDim < 1)
          problemDim = 20;
     StoredDimension(&problemDim, 1);
     return problemDim;
     }

void GetStartPoint(VariableType x[])
     {
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     cout << "Standard starting pencil?";
     if (AskOneChoice("yes,no") == 1)
          {
          float mult;
          cout << "Pencil multiplier (1): ";
          cin >> muit;
          VariableType *xStepP = x;
          for (int i=1; i<=problemDim; i++)
               {
               double tSubi = double(i)/pDp1;
               *xStepP++ = mult*tSubi*(tSubi - 1);
               }
          }
     else
```

```
          {
          cout << "\nStarting point:\n x1: ";
          cin >> *x;
          cout << "Same for rest?";
          if (AskOneChoice("yes,no") == 1)
               {
               VariableType sameVal = *x;
               for (int i=2; i<=problemDim; i++)
                    *++x = sameVal;
               }
          else
               {
               for (int i=2; i<=problemDim; i++)
                    {
                    cout << " x" << i << ": ";
                    cin >> *++x;
                    }
               }
          }
     return;
     }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >=========================
=============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     long double sumToi = 0;
     for (int i=1; i<=problemDim; i++)
          {
          VariableType xCurrent = *x++;
          double tSubi = double(i)/pDp1;
          double temp = xCurrent + tSubi + 1;
          temp = temp*temp*temp;
          sumToi += tSubi*temp;
          VariableType *backr = r;
          *r++ = xCurrent + (1-tSubi)*sumToi/pDp1/2;
          temp = (1-tSubi)*temp;
          for (int backi = i-1; backi>=1; backi--)
               *--backr += temp*backi/pDp1/pDp1/2;
          }
     return RER_SUCCESS;
     }

JacobianEvaluationReturnList GetJacobian
     (const VariableType x[], const VariableType r[],
     FullJacobian *J)
     {
     jacobianCounter.Call();
     //
     int problemDim;
     StoredDimension(&problemDim, 0);
     int pDp1 = problemDim + 1;
     //
     JacobianType *matP = J->GetMatrixPointer();
     for (int row=1; row<=problemDim; row++)
          {
          double tSubi = double(row)/pDp1;
```

```
            JacobianType xSubi = *(x + row - 1);
            const JacobianType *xSubjP = x;
            for (int col=1; col<=problemDim; col++)
                  {
                  double tSubj = double(col)/pDp1;
                  double temp = (*xSubjP++) + tSubj + 1;
                  temp = 3*temp*temp/pDp1/2;
                  if (row < col)
                        *matP++ = temp*tSubi*(1-tSubj);
                  else if (row == col)
                        *matP++ = temp*tSubi*(1-tSubj) + 1;
                  else
                        *matP++ = temp*tSubj*(1-tSubi);
                  }
            }
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================= */


/* ==========< Function StoredDimension >==================
============< Function StoredDimension >================ */
void StoredDimension(int *problemDimension, int store)
      {
      static int storedDimension;
      if (store == 1)
            storedDimension = *problemDimension;
      else
            *problemDimension = storedDimension;
      return;
      }

/* ==========< End Function StoredDimension >============ */


/* ====================< End File PRDSCTIN.CPP >=================== */
```

```
/* ====================< File PRDUCTFL.CPP >==========================
======================< rev 08/28/96 >================================

     Defines a 3-dimensional duct flow problem.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in the thesis. Its solution is f = 0.025,
V = 0.293127, and D = 1.2.

======================< File PRDUCTFL.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


const int PROBLEM_DIMENSION = 3;


/* ==========< Problem discovery >===========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     return PROBLEM_DIMENSION;
     }

void GetStartPoint(VariableType x[])
     {
     cout << "Duct flow problem."
          << "\nStarting point:\n f (ans 0.025): ";
     cin >> *x;
     cout << " V (ans 0.293127): ";
     cin >> *++x;
     cout << " D (ans 1.2): ";
     cin >> *++x;
     return;
     }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >==========================
=============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     // Load meaningful names.
     VariableType f = *x;
     if (f <= 0) return RER_ERROR;
     double rtf = sqrt(f);
     VariableType V = *++x;
     VariableType D = *++x;
     double temp = (1+2.7861/V/rtf)/D;
     if (temp <= 0) return RER_ERROR;
     //
     // Find residuals.
     *r = 1/rtf + 2*log10(temp) - 9.7384634;
     *++r = V*V*f/D - 0.00179008;
     *++r = D*D*V - 0.422104;
```

```
      return RER_SUCCESS;
      }

JacobianEvaluationReturnList GetJacobian
      (const VariableType x[], const VariableType r[],
      FullJacobian *J)
      {
      jacobianCounter.Call();
      // Load meaningful names.
      VariableType f = *x;
      if (f <= 0) return JER_ERROR;
      double rtf = sqrt(f);
      VariableType V = *++x;
      VariableType D = *++x;
      double c1 = -2.4199757/(2.7861 + V*rtf);
      //
      // Find Jacobian elements.
      JacobianType *matP = J->GetMatrixPointer();
      // Partials of r1 wrt f, V, D:
      *matP = (c1 - 1/rtf)/2/f;
      *++matP = c1/V;
      *++matP = -0.868589/D;
      // Partials of r2 wrt f, V, D:
      *++matP = V*V/D;
      *++matP = 2*V*f/D;
      *++matP = -V*V*f/D/D;
      // Partials of r3 wrt f, V, D:
      *++matP = 0;
      *++matP = D*D;
      *++matP = 2*D*V;
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================== */


/* ====================< End File PRDUCTFL.CPP >=================== */
```

PrGenric.hpp

```
/* ====================< File PRGENRIC.HPP >=========================
===== ==================< rev 02/08/96 >================================

     Header for problem definition files.
     Declares functions for evaluating the residuals and Jacobian
of a particular problem.

     This header provides universal function names so that the
particular problem to be solved can be changed simply by
substituting new code for the functions declared here. Use
this header when only one problem is to be solved. If multiple
problems can be solved during one run of the main problem,
i.e. the main() is a driver routine for the nonlinear solution
method, then the problem definition functions require unique
names, and this generic problem definition header should
not be used.

======================< File PRGENRIC.HPP >====================== */

#ifndef PRGENRIC_HPP
#define PRGENRIC_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For typedefs of variables, and return values enums.


/* ==========< Problem initialization >=====================
     Functions allowing a solution routine or its driver to
establish the variable arrays.
=============< Problem initialization >================== */
int GetProblemDimension(void);
void GetStartPoint(VariableType x[]);

/* ==========< End problem initialization >============== */


/* ==========< Problem evaluation >=========================
     Functions the solution routine will use to evaluate
the problem residuals and Jacobian elements.
     Note there is no need to pass the problem dimension
to the residual and Jacobian functions, since it is
known.
=============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[]);

JacobianEvaluationReturnList GetJacobian
     (const VariableType x[], const VariableType r[],
     FullJacobian *J);

/* ==========< End problem evaluation >================== */

#endif


/* ====================< End File PRGENRIC.HPP >=================== */
```

ProbDefn.hpp

```
/* ====================< File PROBDEFN.HPP >=========================
=======================< rev 02/12/96 >================================

     Supports communication between problem definition and
solution method code files.

     Declares:
(1) The types of residual and Jacobian variables;
(2) The return values for problem evaluation functions; and
(3) Arrays of strings explaining the return values.

     Standard choice for Symantec C++ on MacIntosh:
     Use doubles instead of floats. Doubles are faster, so the
only reason to use floats is in case of memory limitations. In
this case, try reducing the size of doubles by turning on the
8-byte doubles option, before going to floats.

======================< File PROBDEFN.HPP >====================== */

#ifndef PROBDEFN_HPP
#define PROBDEFN_HPP


/* ==========< Set user preferences >======================
     For each #defined macro, choose 1 or 0.
=============< Set user preferences >=================== */
#define VARIABLES_ARE_DOUBLES 1
#define JACOBIAN_ELEMENTS_ARE_DOUBLES 1

/* ==========< End set user preferences >================ */


/* ==========< User preference consequences >==============
     Header statements which depend on user preferences.
=============< User preference consequences >============ */
// =====< Variable type >=====
#if (VARIABLES_ARE_DOUBLES == 1)
typedef double VariableType;
#elif (VARIABLES_ARE_DOUBLES == 0)
typedef float VariableType;
#else
#error VARIABLES_ARE_DOUBLES must be 0 or 1
#endif

// =====< Jacobian element type >=====
#if (JACOBIAN_ELEMENTS_ARE_DOUBLES == 1)
typedef double JacobianType;
#elif (JACOBIAN_ELEMENTS_ARE_DOUBLES == 0)
typedef float JacobianType;
#else
#error JACOBIAN_ELEMENTS_ARE_DOUBLES must be 0 or 1
#endif

/* ==========< End user preference consequences >======== */


/* ==========< Problem return values >=====================
=============< Problem return values >================== */
// =====< Residuals >=====
enum ResidualEvaluationReturnList
     {
     RER_ERROR = 0,
     RER_SUCCESS_MODE_CHANGE,
     RER_SUCCESS
     };

// =====< Jacobian >=====
```

ProbDefn.hpp

```
enum JacobianEvaluationReturnList
      {
      JER_ERROR = 0,
      JER_SUCCESS_FINITE_DIFFERENCES,
      JER_SUCCESS_ANALYTIC
      };

// =====< Solution method >=====
enum SolveNonLinearReturnList
      {
      SNLR_ZERO_RESIDUAL_VECTOR = 0,
      SNLR_STAGNANT_ESTIMATED_SOLUTION,
      SNLR_STAGNANT_RESIDUAL_NORM,
      SNLR_EVALUATION_ERROR,
      SNLR_FACTORIZATION_ERROR,
      SNLR_ITERATION_COUNT
      };

extern char *SolveNonLinearReturnExplanation[];

/* ==========< End problem return values >============== */

#endif


/* ====================< End File PROBDEFN.HPP >=================== */
```

ProbDefn.cpp

```
/* ====================< File PROBDEFN.CPP >==========================
=======================< rev 02/25/96 >================================

     Supports communication between problem definition and
solution method code files.
     Documentation is in the header file.

======================< File PROBDEFN.CPP >======================== */


#include "probdefn.hpp"


/* ==========< Problem return values >=====================
============< Problem return values >================== */
// =====< Residuals >=====

// =====< Jacobian >=====

// =====< Solution method >=====
char *SolveNonLinearReturnExplanation[] =
     {
     "Zero residual vector", // 0
     "Small change in estimated solution", // 1
     "Nondecreasing residual norm", // 2
     "Residual or Jacobian evaluation error", // 3
     "Jacobian solution error", // 4
     "Exceeded iteration limit" // 5
     };

/* ==========< End problem return values >============== */


/* ====================< End File PROBDEFN.CPP >=================== */
```

PrPowScl.cpp

```
/* ====================< File PRPOWSCL.CPP >=========================
======================< rev 03/21/96 >==============================

     Defines the 2-dimensional Powell badly scaled function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is
(1.098E-5, 9.106).

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

======================< File PRPOWSCL.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


const int PROBLEM_DIMENSION = 2;


/* ==========< Problem discovery >==========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     return PROBLEM_DIMENSION;
     }

void GetStartPoint(VariableType x[])
     {
     cout << "Powell badly scaled function."
          << "\nStarting point:\n x1 (ans 1.098E-5): ";
     cin >> *x;
     cout << " x2 (ans 9.106): ";
     cin >> *++x;
     return;
     }

/* ==========< End problem discovery >=================== */


/* ==========< Problem evaluation >=========================
=============< Problem evaluation >======================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     // Load meaningful names.
     VariableType x1 = *x;
     VariableType x2 = *++x;
     //
     // Find residuals.
     *r = 1E4*x1*x2 - 1;
     *++r = exp(-x1) + exp(-x2) - 1.0001;
     return RER_SUCCESS;
     }
```

PrPowScl.cpp

```
JacobianEvaluationReturnList GetJacobian
      (const VariableType x[], const VariableType r[],
      FullJacobian *J)
      {
      jacobianCounter.Call();
      // Load meaningful names.
      VariableType x1 = *x;
      VariableType x2 = *++x;
      //
      // Find Jacobian elements.
      JacobianType *matP = J->GetMatrixPointer();
      // Partials of r1 wrt x1, x2:
      *matP = 1E4*x2;
      *++matP = 1E4*x1;
      // Partials of r2 wrt x1, x2:
      *++matP = -exp(-x1);
      *++matP = -exp(-x2);
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================== */


/* ====================< End File PRPOWSCL.CPP >=================== */
```

PrPowSng.cpp

```
/* ====================< File PRPOWSNG.CPP >=========================
======================< rev 03/21/96 >================================

     Defines the 4-dimensional Powell Singular function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is (0, 0, 0, 0).

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

======================< File PRPOWSNG.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


const int PROBLEM_DIMENSION = 4;
const double RT_FIVE = sqrt(5);
const double RT_TEN = sqrt(10);


/* ==========< Problem discovery >=========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     return PROBLEM_DIMENSION;
     }

void GetStartPoint(VariableType x[])
     {
     cout << "Powell Singular function."
          << "\nStarting point:\n x1 (ans 0): ";
     cin >> *x;
     cout << " x2 (ans 0): ";
     cin >> *++x;
     cout << " x3 (ans 0): ";
     cin >> *++x;
     cout << " x4 (ans 0): ";
     cin >> *++x;
     return;
     }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >=======================
=============< Problem evaluation >======================= */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     // Load meaningful names.
     VariableType x1 = *x;
     VariableType x2 = *++x;
     VariableType x3 = *++x;
     VariableType x4 = *++x;
```

```
      //
      // Find residuals.
      *r = x1 + 10*x2;
      *++r = RT_FIVE*(x3 - x4);
      *++r = x2*x2 - 4*x2*x3 + 4*x3*x3;
      *++r = RT_TEN*(x1*x1 - 2*x1*x4 +x4*x4);
      return RER_SUCCESS;
      }

JacobianEvaluationReturnList GetJacobian
      (const VariableType x[], const VariableType r[],
      FullJacobian *J)
      {
      jacobianCounter.Call();
      // Load meaningful names.
      VariableType x1 = *x;
      VariableType x2 = *++x;
      VariableType x3 = *++x;
      VariableType x4 = *++x;
      //
      // Find Jacobian elements.
      JacobianType *matP = J->GetMatrixPointer();
      // Partials of r1 wrt x1, x2, x3, x4:
      *matP = 1;
      *++matP = 10;
      *++matP = 0;
      *++matP = 0;
      // Partials of r2:
      *++matP = 0;
      *++matP = 0;
      *++matP = RT_FIVE;
      *++matP = -RT_FIVE;
      // Partials of r3:
      *++matP = 0;
      *++matP = 2*x2 - 4*x3;
      *++matP = 8*x3 - 4*x2;
      *++matP = 0;
      // Partials of r4:
      *++matP = 2*RT_TEN*(x1 - x4);
      *++matP = 0;
      *++matP = 0;
      *++matP = 2*RT_TEN*(x4 - x1);
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================== */


/* ====================< End File PRPOWSNG.CPP >================== */
```

PrRosnbk.cpp

```
/* ====================< File PRROSNBK.CPP >=========================
=======================< rev 07/30/96 >================================

     Defines the variably-dimensioned extended Rosenbrock function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in Dennis, extending the two-dimensional
Rosenbrock function of More. Its solution is (1, 1, .., 1).

     Sources:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983; Society for Industrial and Applied Mathematics, 1996.
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

=======================< File PRROSNBK.CPP >======================== */

#include <iostream.h>
#include <math.h>

#include "utlinput.hpp"
#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


void StoredDimension(int *problemDimension, int store);


/* ==========< Problem discovery >=========================
=============< Problem discovery >====================== */
int GetProblemDimension(void)
     {
     //   Since problem dimension is not known at compile
     // time, get it from user, and store for access by
     // residual and Jacobian functions.
     int problemDim;
     cout << "Extended Rosenbrock function.\nEven problem dimension: ";
     cin >> problemDim;
     if (problemDim < 2)
          problemDim = 2;
     if ((problemDim%2) != 0)
          problemDim += 1;
     StoredDimension(&problemDim, 1);
     return problemDim;
     }

void GetStartPoint(VariableType x[])
     {
     int problemDim;
     StoredDimension(&problemDim, 0);
     cout << "Standard starting pencil?";
     if (AskOneChoice("yes,no") == 1)
          {
          VariableType mult;
          cout << "Pencil multiplier (1): ";
          cin >> mult;
          VariableType *xStepP = x;
          for (int i=0; i<problemDim/2; i++)
               {
               *xStepP++ = -1.2*mult;
               *xStepP++ = mult;
```

```
                }
            }
        else
            {
            cout << "\nStarting point:\n x1: ";
            cin >> *x;
            cout << "Same for rest?";
            if (AskOneChoice("yes,no") == 1)
                {
                VariableType sameVal = *x;
                for (int i=2; i<=problemDim; i++)
                    *++x = sameVal;
                }
            else
                {
                for (int i=2; i<=problemDim; i++)
                    {
                    cout << " x" << i << ": ";
                    cin >> *++x;
                    }
                }
            }
        return;
        }

/* ===========< End problem discovery >=================== */


/* ===========< Problem evaluation >=========================
==============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
        (const VariableType x[], VariableType r[])
        {
        residualCounter.Call();
        //
        int problemDim;
        StoredDimension(&problemDim, 0);
        //
        for (int i=0; i<problemDim/2; i++)
            {
            VariableType xFirstOfTwo = *x++;
            VariableType xSecondOfTwo = *x++;
            *r++ = 10*(xSecondOfTwo - xFirstOfTwo*xFirstOfTwo);
            *r++ = 1 - xFirstOfTwo;
            }
        return RER_SUCCESS;
        }

JacobianEvaluationReturnList GetJacobian
        (const VariableType x[], const VariableType r[],
        FullJacobian *J)
        {
        jacobianCounter.Call();
        //
        int problemDim;
        StoredDimension(&problemDim, 0);
        //
        JacobianType *matP = J->GetMatrixPointer();
        for (int row=0; row<problemDim; row++)
            {
            VariableType xSubi = *x++;
            for (int col=0; col<problemDim; col++)
                {
                if (col == row)
                    *matP++ = -20*xSubi;
                else if (col == row+1)
                    *matP++ = 10;
```

```
                else
                        *matP++ = 0;
                }
            row++;
            x++;
            for (col=0; col<problemDim; col++)
                {
                if (col == row-1)
                        *matP++ = -1;
                else
                        *matP++ = 0;
                }
            }
        J->AcceptMatrix();
        return JER_SUCCESS_ANALYTIC;
        }

/* ==========< End problem evaluation >================== */


/* ==========< Function StoredDimension >====================
=============< Function StoredDimension >================= */
void StoredDimension(int *problemDimension, int store)
        {
        static int storedDimension;
        if (store == 1)
            storedDimension = *problemDimension;
        else
            *problemDimension = storedDimension;
        return;
        }

/* ==========< End Function StoredDimension >============ */


/* ====================< End File PRROSNBK.CPP >=================== */
```

```
/* ====================< File PRTRIGNO.CPP >==========================
=======================< rev 03/21/96 >================================

     Defines a variably-dimensioned trigonometric function.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in More. Its solution is not given.

     Sources:
     Jorge More, Burton Garbow, and Kenneth Hillstrom, "Testing
Unconstrained Optimization Software," in ACM Transactions on
Mathematical Software, v. 7, n. 1, March 1981, pp. 17-41.

=======================< File PRTRIGNO.CPP >======================== */

#include <iostream.h>
#include <math.h>

#include "utlinput.hpp"
#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"

void StoredDimension(int *problemDimension, int store);


/* ==========< Problem discovery >===========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     //   Since problem dimension is not known at compile
     // time, get it from user, and store for access by
     // residual and Jacobian functions.
     int problemDim;
     cout << "Trigonometric function.\nProblem dimension: ";
     cin >> problemDim;
     if (problemDim < 2)
          problemDim = 2;
     StoredDimension(&problemDim, 1);
     return problemDim;
     }

void GetStartPoint(VariableType x[])
     {
     int problemDim;
     StoredDimension(&problemDim, 0);
     cout << "Standard starting pencil?";
     if (AskOneChoice("yes,no") == 1)
          {
          VariableType mult;
          cout << "Pencil multiplier (1): ";
          cin >> mult;
          mult /= problemDim;
          VariableType *xStepP = x;
          for (int i=0; i<problemDim; i++)
               *xStepP++ = mult;
          }
     else
          {
          cout << "\nStarting point:\n x1: ";
          cin >> *x;
          cout << "Same for rest?";
          if (AskOneChoice("yes,no") == 1)
```

```
                {
                VariableType sameVal = *x;
                for (int i=2; i<=problemDim; i++)
                     *++x = sameVal;
                }
           else
                {
                for (int i=2; i<=problemDim; i++)
                     {
                     cout << " x" << i << ": ";
                     cin >> *++x;
                     }
                }
           }
      return;
      }

/* ==========< End problem discovery >================== */


/* ==========< Problem evaluation >=========================
=============< Problem evaluation >====================== */
ResidualEvaluationReturnList GetResiduals
      (const VariableType x[], VariableType r[])
      {
      residualCounter.Call();
      //
      int problemDim;
      StoredDimension(&problemDim, 0);
      //
      VariableType *rStepP = r;
      long double sumCosines = 0;
      for (int i=1; i<=problemDim; i++)
           {
           VariableType xCurrent = *x++;
           VariableType cosxCurrent = cos(xCurrent);
           sumCosines += cosxCurrent;
           *rStepP++ = i*(1-cosxCurrent) - sin(xCurrent);
           }
      sumCosines -= problemDim;
      for (i=1; i<=problemDim; i++)
           *r++ -= sumCosines;
      return RER_SUCCESS;
      }

JacobianEvaluationReturnList GetJacobian
      (const VariableType x[], const VariableType r[],
      FullJacobian *J)
      {
      jacobianCounter.Call();
      //
      int problemDim;
      StoredDimension(&problemDim, 0);
      //
      JacobianType *matP = J->GetMatrixPointer();
      for (int col=0; col<problemDim; col++)
           {
           // Index from zero, not one, to aid pointer addition.
           VariableType xCurrent = *x++;
           VariableType sinxCurrent = sin(xCurrent);
           JacobianType *matStepP = matP + col;
           for (int row=0; row<problemDim; row++)
                {
                if (row == col)
                     // Note index starts at zero, not one.
                     *matStepP = (row+2)*sinxCurrent - cos(xCurrent);
                else
```

```
                    *matStepP = sinxCurrent;
                matStepP += problemDim;
                }
            }
        J->AcceptMatrix();
        return JER_SUCCESS_ANALYTIC;
        }

/* ==========< End problem evaluation >================== */


/* ==========< Function StoredDimension >==================
==============< Function StoredDimension >================ */
void StoredDimension(int *problemDimension, int store)
        {
        static int storedDimension;
        if (store == 1)
            storedDimension = *problemDimension;
        else
            *problemDimension = storedDimension;
        return;
        }

/* ==========< End Function StoredDimension >============ */


/* ======================< End File PRTRIGNO.CPP >=================== */
```

PrWlConv.cpp

```
/* ====================< File PRWLCONV.CPP >=========================
=======================< rev 03/21/96 >================================

     Defines a 2-dimensional wall convection problem.
     Documentation is in the generic header file "prgenric.hpp".

     The problem is defined in the thesis. Its solution is Twout =,
0.68494807 and Twin = 15.742466.

======================< File PRWLCONV.CPP >====================== */

#include <iostream.h>
#include <math.h>

#include "utltimer.hpp"
CallCounter residualCounter("Residual evaluations");
CallCounter jacobianCounter("Jacobian evaluations");


#include "prgenric.hpp"


const int PROBLEM_DIMENSION = 2;


/* ==========< Problem discovery >==========================
=============< Problem discovery >======================== */
int GetProblemDimension(void)
     {
     return PROBLEM_DIMENSION;
     }

void GetStartPoint(VariableType x[])
     {
     cout << "Wall convection problem."
          << "\nStarting point:\n Twout (ans 0.6849): ";
     cin >> *x;
     cout << " Twin (ans 15.74): ";
     cin >> *++x;
     return;
     }

/* ==========< End problem discovery >=================== */


/* ==========< Problem evaluation >=========================
=============< Problem evaluation >======================== */
ResidualEvaluationReturnList GetResiduals
     (const VariableType x[], VariableType r[])
     {
     residualCounter.Call();
     //
     // Load meaningful names.
     VariableType Twout = *x;
     VariableType Twin = *++x;
     VariableType hcin = 1.239*pow(fabs(20-Twin),1.0/3);
     //
     // Find residuals.
     *r = 13.05*Twout - 0.5678*Twin;
     *++r = 0.5678*(Twout - Twin) + (20 - Twin)*hcin;
     return RER_SUCCESS;
     }

JacobianEvaluationReturnList GetJacobian
     (const VariableType x[], const VariableType r[],
     FullJacobian *J)
     {
```

```
      jacobianCounter.Call();
      // Load meaningful names.
      //VariableType Twout = *x;
           // Note J does not depend on Twout.
      VariableType Twin = *++x;
      VariableType hcin = 1.239*pow(fabs(20-Twin),1.0/3);
      //
      // Find Jacobian elements.
      JacobianType *matP = J->GetMatrixPointer();
      // Partials of r1 wrt Twout, Twin:
      *matP = 13.05;
      *++matP = -0.5678;
      // Partials of r2:
      *++matP = 0.5678;
      *++matP = -(0.5678 + 4*hcin/3);
      J->AcceptMatrix();
      return JER_SUCCESS_ANALYTIC;
      }

/* ==========< End problem evaluation >================== */


/* ====================< End File PRWLCONV.CPP >================== */
```

SolvDriv.cpp

```
/* ====================< File SOLVDRIV.CPP >=========================
======================< rev 08/21/96 >================================

     Driver for Newton-Raphson based solution techniques.
     The problem to be solved is chosen by the problem code
file included in the project. The problem code file must
define the four functions identified in "probdefn.hpp".

======================< File SOLVDRIV.CPP >======================== */

#include <fstream.h>

#include "arrayops.hpp"
#include "utldepnd.hpp"
#include "utlinput.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;
#include "utltimer.hpp"

// Problem definition.
#include "prgenric.hpp"

// Solution methods.
#include "dbldoglg.hpp"
#include "newtraph.hpp"
#include "planarhk.hpp"
#include "stddoglg.hpp"
#include "wtddoglg.hpp"


/* ==========< Helper declarations >=========================
=============< Helper declarations >====================== */
enum SolutionMethodList
     {
     SM_NR,
     SM_DD_DENNIS,
     SM_DD_THESIS,
     SM_PLANAR,
     SM_WEIGHT_DD
     };

const char *const OPEN_HEADER = "\n\n==========< ";
const char *const CLOSE_HEADER = " >==========\n";

/* ==========< End helper declarations >================= */


/* ==========< Drive Newton-Raphson method >================
=============< Drive Newton-Raphson method >============= */
void main()
     {
     // Initialize environmental dependencies.
     DependencyInitialization();
     //
     // Initialize dimensioned variables.
     const int variableCount = GetProblemDimension();
     VariableType *x = new VariableType[variableCount];
     VariableType *r = new VariableType[variableCount];
     if (!x || !r)
          FATAL(ERR_FREE_STORE, 1130109);
     cout << OPEN_HEADER << "Initialize problem" << CLOSE_HEADER;
     GetStartPoint(x);
     //
     // Initialize control variables.
     SolutionMethodList solutionMethod;
     double zeroTolerance;
     double stepTolerance;
```

```
      cout << OPEN_HEADER << "Control solution method" << CLOSE_HEADER
           << "Search method?";
      switch(AskOneChoice
           ("Newton-Raphson,double dogleg,planar hook,weighted"))
           {
      case 1:
           solutionMethod = SM_NR;
           break;
      case 2:
           cout << "Double dogleg version?";
           switch(AskOneChoice("Dennis,thesis"))
                {
           case 1:
                solutionMethod = SM_DD_DENNIS;
                break;
           case 2:
                solutionMethod = SM_DD_THESIS;
                break;
                }
           break;
      case 3:
           solutionMethod = SM_PLANAR;
           break;
      case 4:
           solutionMethod = SM_WEIGHT_DD;
           break;
           }
      cout << "\nMaximum residual element (0 for default): ";
      cin >> zeroTolerance;
      cout << "Relative change tolerance (0 for default): ";
      cin >> stepTolerance;
      //
      // Initialize reporting variables.
      streambuf *const originalCout = cout.rdbuf();
      int reportToFile;
      ofstream outFile;
           //   Sets up outFile as an ofstream, but has no
           // associated filebuf.
      SolveNonLinearReturnList solutionResult;
      RealTimer timer1("Problem solution");
      cout << OPEN_HEADER << "Control reporting" << CLOSE_HEADER
           << "Reporting level?";
      switch(AskOneChoice("verbose,laconic,mute"))
           {
      case 1: utlReportLevel = RL_VERBOSE; break;
      case 2: utlReportLevel = RL_LACONIC; break;
      case 3: utlReportLevel = RL_MUTE; break;
           }
      if (utlReportLevel != RL_MUTE)
           {
           cout << "\nReport to file?";
           reportToFile = AskOneChoice("yes,no");
           if (reportToFile == 1)
                {
                AskOutputFile(&outFile);
                cout << "\nChange formats?";
                if (AskOneChoice("yes,no") == 1)
                     AskStreamFormats(&outFile);
                }
           }
      else reportToFile = 2;
      //
      // First evaluation.
      cout << OPEN_HEADER << "Begin solution" << CLOSE_HEADER;
      if (reportToFile == 1)
           cout = outFile;
      timer1.Reset();
```

```
      if (GetResiduals(x, r) == RER_ERROR)
           FATAL("Residual error at given starting point", 2130109);
      //
      // Solve.
      switch(solutionMethod)
           {
      case SM_NR:
           solutionResult = SolveNewtonRaphson
                (variableCount, x, r, &GetResiduals, &GetJacobian,
                &zeroTolerance, &stepTolerance);
           break;
      case SM_DD_THESIS:
           solutionResult = SolveDoubleDogleg
                (variableCount, x, r, &GetResiduals, &GetJacobian,
                &zeroTolerance, &stepTolerance);
           break;
      case SM_DD_DENNIS:
           solutionResult = SolveStandardDoubleDogleg
                (variableCount, x, r, &GetResiduals, &GetJacobian,
                &zeroTolerance, &stepTolerance);
           break;
      case SM_PLANAR:
           solutionResult = SolvePlanarHook
                (variableCount, x, r, &GetResiduals, &GetJacobian,
                &zeroTolerance, &stepTolerance);
           break;
      case SM_WEIGHT_DD:
           solutionResult = SolveWeightedDogleg
                (variableCount, x, r, &GetResiduals, &GetJacobian,
                &zeroTolerance, &stepTolerance);
           break;
           }
      //
      // Report results.
      timer1.Suspend();
      if (reportToFile == 1)
           {
           cout = originalCout;
           outFile.close();
           }
      cout << OPEN_HEADER << "Results" << CLOSE_HEADER
           << "Termination: "
           << SolveNonLinearReturnExplanation[solutionResult]
           << flush;
      //
      // Release memory allocated from free store.
      delete [] x;
      delete [] r;
      return;
      } // End main().

/* ==========< End drive Newton-Raphson method >========= */


/* ====================< End File SOLVDRIV.CPP >================== */
```

StdDoglg.hpp

```
/* ====================< File STDDOGLG.HPP >==========================
=======================< rev 07/03/96 >================================

     Headers for STDDOGLG.CPP.
     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis.

     Source:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983; Society for Industrial and Applied Mathematics, 1996.

=======================< File STDDOGLG.HPP >======================== */

#ifndef STDDOGLG_HPP
#define STDDOGLG_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For form of residual and Jacobian functions.


/* ==========< Function SolveStandardDoubleDogleg >=========
=============< Function SolveStandardDoubleDogleg >====== */
SolveNonLinearReturnList SolveStandardDoubleDogleg
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance);

/* ==========< End Function SolveStandardDoubleDogleg >== */

#endif


/* ====================< End File STDDOGLG.HPP >================== */
```

StdDoglg.cpp

```
/* ====================< File STDDOGLG.CPP >=========================
=======================< rev 08/27/96 >================================

     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis.
     Documentation is in the header file.

     Programming notes for double dogleg method:
     This file implements the pseudocode from Appendix A of the
text by Dennis and Schnabel. Major exceptions:
(1) It does not scale the residuals.
(2) It handles termination criteria by separate functions in
"termcond.cpp".
(3) It does not perturb a Jacobian identified as singular.
(4) It subtracts steps from, rather than adding steps to, xK when
finding xKp1. Thus, xKp1 = xK - dx where Dennis uses xK + dx.
(5) It reacts to residual errors, by reducing the trust region.
(6) It tests for zero residuals.
(7) It initializes the trust region to the Newton-Raphson length,
rather than to the Cauchy length.
(8) It performs the multiplication required to find J*(JT*r)
before, rather than after, factoring the Jacobian.

=======================< File STDDOGLG.CPP >======================== */

#include <math.h>


#include "arrayops.hpp"
#include "plujcbn.hpp"
#include "termcond.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "stddoglg.hpp"


/* ===========< Global constants >============================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
==============< Global constants >======================== */
const int MAX_LOOPS = 100;
const double ALPHA = 1E-4;

/* ===========< End global constants >=================== */


/* ===========< Iteration status >=========================
==============< Iteration status >======================== */
enum TrustRegionStatusList
     {
     TRS_ZERO_RESIDUALS,
     TRS_STAGNANT_SOLUTION,
     TRS_ACCEPTED,
     TRS_ACCEPTED_AFTER_REJECT,
     TRS_INHERITED,
     TRS_REDUCED,
     TRS_INCREASED
     };
     //   Dennis: retcode.
     //   Indicates if attempt to find the dogleg step is the
     // first try with an inherited trust region, a subsequent
     // try with a reduced trust region (retcode = 2), or a
     // subsequent try with an increased trust region
     // (retcode = 3).
     //   May also indicate that step selection is complete,
```

```
      // either because current step accepted (retcode = 0),
      // because current step is so small that the termination
      // test for a stagnant estimated solution is satisfied
      // (retcode = 1), or because found a zero residual vector.
      //   May also indicate that an increased step should be
      // rejected, and the original step, i.e. the one that
      // caused the increase, should be accepted.

/* ==========< End iteration status >==================== */


/* ==========< Global variables >============================
      "Static" keyword limits variables to file scope.
==============< Global variables >======================== */
// =====< Global versions of function arguments >=====
static int probDim;
static VariableType relStepTol;

// =====< Global scalar variables >=====
static double stepLenNR, stepLenCauchy, stepLenCutback;
      //   Distances, respectively, to: the Newton-Raphson
      // solution of the linearized equations; the predicted
      // minimizer of r-square in the steepest descent direction;
      // and the point at which the double dogleg curve rejoins
      // the Newton-Raphson direction.
static double stepLenTrust, stepLenTrustPrev;
      //   Dennis del: trust region length. Would like to
      // take a step of this length. If increase trust length
      // during the current iteration, store its previous
      // value (just prior to increasing).
static TrustRegionStatusList trustRegionStatus;
static double halfRSqK, halfRSqKpl, halfRSqKplPrev;
      //   The cost function at beginning of iteration; at trial
      // point during iteration; and, if the trust length was
      // increased during the current iteration, at the last
      // trial point for the current iteration.
static double eta;
      // Ratio of stepLenCutback/stepLenNR.
static double g2Tdx;
      //   Dennis calls this the initial slope. Not exactly
      // initial slope, but gTdx where g is the gradient of
      // halfRSqare at xK.
      //   The value should be negative, since going downhill
      // in values of halfRSq. Calculate as -gTdx, using
      // negative sign since gTdx is positive for a descent
      // direction when dx subtracted. Dennis does not use
      // negative, but he adds dx.
static double deltaHalfRSq, deltaHalfRSqPredict;
static YesNoList firstDog;
      //   Indicates whether have calculated points on double
      // dogleg curve for the current iteration.
static SolveNonLinearReturnList exitReason;

// =====< Global dimensioned variables >=====
static PLUJacobian *J;
static VariableType *xK;
static VariableType *rCurrent;
      // Sometimes holds rK, sometimes rKpl.
static VariableType *dxNR, *dxCauchy, *dx;
      //   The Newton-Raphson step, Cauchy step, and actual step
      // taken, respectively.
      //   Newton-Raphson step goes to solution of linearized
      // residual equations, which also minimizes the sum of
      // squares of the linearized residuals.
      //   Cauchy step goes to the minimizer of halfRSq in the
      // steepest descent direction when residuals linearized.
static VariableType *xKpl;
```

```
     // xKpl = xK - dx. Note Dennis uses x + dx.
static VariableType *xHold, *rHold;
     //   Store intermediate value of xKpl, rKpl for convenience
     // during one iteration. Also used for intermediate tasks.
static VariableType *gradHalfRSq;
static VariableType *vCauchyToCut;
     //   Dennis v. Difference vector from Cauchy point to the
     // cutback point on the double dogleg curve.

/* ==========< End global variables >==================== */


/* ==========< Declare helper functions >===================
============< Declare helper functions >================ */
static void FindStep(void);
static TrustRegionStatusList EvaluateStep(void);

/* ==========< End declare helper functions >============ */


/* ==========< Function SolveStandardDoubleDogleg >=========
============< Function SolveStandardDoubleDogleg >====== */
SolveNonLinearReturnList SolveStandardDoubleDogleg
     (const int problemDim,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance)
     {
     // Assume r has been evaluated for the given x.
     //
     // Allocate memory.
     J = new PLUJacobian(problemDim);
     xK = x;
     rCurrent = r;
     dxNR = new VariableType[problemDim];
     dxCauchy = new VariableType[problemDim];
     dx = new VariableType[problemDim];
     xKpl = new VariableType[problemDim];
     xHold = new VariableType[problemDim];
     rHold = new VariableType[problemDim];
     gradHalfRSq = new VariableType[problemDim];
     vCauchyToCut = new VariableType[problemDim];
     if (!J
          || !dxNR || !dxCauchy || !dx
          || !xKpl || !xHold || !rHold
          || !gradHalfRSq || !vCauchyToCut)
          FATAL(ERR_FREE_STORE, 119190404);
     //
     // Initialize.
     probDim = problemDim;
     exitReason = SNLR_ITERATION_COUNT;
     CheckAbsoluteZeroTolerance(absoluteZeroTolerance);
     CheckRelativeStepTolerance(relativeStepTolerance);
     relStepTol = *relativeStepTolerance;
     halfRSqK = ArraySumSquares(problemDim, rCurrent)/2;
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\nReporting from SolveStandardDoubleDogleg():"
```

```
            << "\nZero tol:\t" << *absoluteZeroTolerance
            << "\tStep tol:\t" << *relativeStepTolerance
            << "\nInitial x:";
      ArrayStreamPrint(problemDim, xK);
      }
//
for (int loopCount=1; loopCount<=MAX_LOOPS; loopCount++)
      {
      //   Known: xK, rK (in rCurrent), rSquareK. Unknown:
      // J, grad.
      //   Vectors free for initializing calculations: dxNR,
      // dx, gradHalfRSq, dxCauchy, vCauchyToCut, xKp1,
      // xHold, rHold.
      //
      // Initialize loop variables.
      //   Find vectors requiring Jacobian product, before
      // factor Jacobian.
      EvaluateJacobian(xK, rCurrent, J);
      J->TProduct(rCurrent, gradHalfRSq);
      J->Product(gradHalfRSq, rHold);
            //   rHold free until try to increase a step. Here,
            // holds JJTr in case needed to find Cauchy point.
      firstDog = YES;
      //
      //   Find Newton-Raphson step, the solution to linearized
      // residual equations, J(xK - xKp1) = J(dxNR) = rK. Will
      // have xKp1 = xK - dxNR.
      if (J->Solve(dxNR, rCurrent) == FAILURE)
            {
            if (utlReportLevel == RL_VERBOSE)
                  {
                  cout << "\n\nJacobian singular at x:";
                  ArrayStreamPrint(problemDim, xK);
                  }
            exitReason = SNLR_FACTORIZATION_ERROR;
            break;
            }
      stepLenNR = sqrt(ArraySumSquares(problemDim, dxNR));
      //
      // Set trust length for first iteration to NR length.
      if (loopCount == 1)
            stepLenTrust = stepLenNR;
      //
      // Report if necessary.
      if (utlReportLevel == RL_VERBOSE)
            {
            cout << "\n\nx: iteration:\t" << loopCount;
            ArrayStreamPrint(problemDim, xK);
            cout << "\nr: half SS:\t" << halfRSqK;
            ArrayStreamPrint(problemDim, rCurrent);
            cout << "\nstep lengths: trust:\t" << stepLenTrust
                  << "\tNR:\t" << stepLenNR;
            }
      //
      //   Inner loop: Find test point on double dogleg curve,
      // at step length = trust length. Evaluate residuals and
      // cost function there. Update the current trust region,
      // and repeat until find an acceptable point.
      trustRegionStatus = TRS_INHERITED;
      while (trustRegionStatus >= TRS_INHERITED)
            {
            FindStep();
            // Take the step and evaluate residuals.
            ArrayDifference(problemDim, xK, dx, xKp1);
            if (EvaluateResiduals(xKp1, rCurrent) == RER_ERROR)
                  {
                  if (trustRegionStatus == TRS_INCREASED)
```

```
                    {
                    if (utlReportLevel == RL_VERBOSE)
                        cout << "\trejected (residual error)";
                    trustRegionStatus = TRS_ACCEPTED_AFTER_REJECT;
                    }
               else
                    {
                    if (utlReportLevel == RL_VERBOSE)
                        cout << "\thalved (residual error)";
                    stepLenTrust /= 2;
                    trustRegionStatus = TRS_REDUCED;
                    }
               } // end residual error processing
          else
               {
               // Here, have good residuals.
               halfRSqKp1 = ArraySumSquares(problemDim, rCurrent)/2;
               if (CheckZeroVector(problemDim,
                    rCurrent, *absoluteZeroTolerance) == TC_TERMINATE)
                    {
                    // Found a solution.
                    trustRegionStatus = TRS_ZERO_RESIDUALS;
                    }
               else
                    trustRegionStatus = EvaluateStep();
               } // end good residual processing
          } // end search for acceptable dx
     //
     //   Here, know an acceptable dx, or met a termination
     // criterion.
     if (trustRegionStatus == TRS_ACCEPTED)
          {
          // Update trust region and prepare for next iteration.
          if (deltaHalfRSq > 0.1*deltaHalfRSqPredict)
               {
               //   Recall deltas are negative, so if actual
               // reduction < 0.1*predicted reduction, i.e. if not
               // doing well, reduce trust region for next step.
               stepLenTrust /= 2;
               }
          else if (deltaHalfRSq <= 0.75*deltaHalfRSqPredict)
               {
               //   If actual reduction >= 0.75*predicted, doing
               // well. Increase trust region for next step.
               stepLenTrust *= 2;
               }
          VariableType *temp = xK; xK = xKp1; xKp1 = temp;
          halfRSqK = halfRSqKp1;
          }
     else if (trustRegionStatus == TRS_ACCEPTED_AFTER_REJECT)
          {
          //   Reject an increased trust region. Accept the
          // step which induced the increase, and take its
          // length as the trust length.
          stepLenTrust = stepLenTrustPrev;
          VariableType *temp = xK; xK = xHold; xHold = temp;
          temp = rCurrent; rCurrent = rHold; rHold = temp;
          halfRSqK = halfRSqKp1Prev;
          }
     else
          {
          //   Termination. Set exit reason and break out of
          // for-loop of MAX_LOOPS iterations.
          if (trustRegionStatus == TRS_ZERO_RESIDUALS)
               exitReason = SNLR_ZERO_RESIDUAL_VECTOR;
          else
               exitReason = SNLR_STAGNANT_ESTIMATED_SOLUTION;
```

```
                break;
                }
          } // end sequence of double dogleg steps
     //
     //   Copy current values to arrays passed as function
     // arguments, if necessary.
     if (xKpl != x)
          ArrayCopy(problemDim, xKpl, x);
     if (rCurrent != r)
          ArrayCopy(problemDim, rCurrent, r);
     //
     // Report if necessary.
     if (utlReportLevel > RL_MUTE)
          {
          cout << "\n\nSolveStandardDoubleDogleg() terminating: "
               << SolveNonLinearReturnExplanation[exitReason]
               << "\nFinal x:";
          ArrayStreamPrint(problemDim, x);
          cout << "\nr: half SS:\t" << halfRSqKpl;
          ArrayStreamPrint(problemDim, r);
          }
     //
     //   Release memory allocated from free store. Since have
     // been trading arrays around between pointers, make sure
     // don't try to delete the arrays passed as arguments to
     // this function.
     delete J;
     if (xK != x)
          delete [] xK;
     if (rCurrent != r)
          delete [] rCurrent;
     delete [] dxNR;
     delete [] dxCauchy;
     delete [] dx;
     if (xKpl != x)
          delete [] xKpl;
     if (xHold != x)
          delete [] xHold;
     if (rHold != r)
          delete [] rHold;
     delete [] gradHalfRSq;
     delete [] vCauchyToCut;
     return exitReason;
     } // End SolveStandardDoubleDogleg().

/* ==========< End Function SolveStandardDoubleDogleg >== */


/* ==========< Helper functions >==========================
==============< Helper functions >======================== */
static void FindStep(void)
     {
     //   Find test point on double dogleg curve, at step
     // length = trust length (Dennis DOGSTEP).
     //
     if (stepLenTrust >= stepLenNR)
          {
          // Take the full Newton-Raphson step.
          stepLenTrust = stepLenNR;
          ArrayCopy(probDim, dxNR, dx);
          }
     else
          {
          // Take a step smaller than the full NR step.
          if (firstDog == YES)
               {
               // Calculate points on the double dogleg curve.
```

```
                firstDog = NO;
                double num = ArraySumSquares(probDim, gradHalfRSq);
                     //   num = gTg. This is Dennis' alpha in the
                     // DOGSTEP routine. Don't use alpha to avoid
                     // confusion with global constant ALPHA used to
                     // check step progress.
                double denom = ArraySumSquares(probDim, rHold);
                     //   Recall rHold temporarily stores JJTr.
                     //   denom = (Jg)T(Jg) = gTJTJg. Note JTJ
                     // approximates Hessian of halfRSq. This is
                     // Dennis' beta, renamed for consistency with num.
                ArrayScalarProduct(probDim,
                     num/denom, gradHalfRSq, dxCauchy);
                     //   dxCauchy = (num/denom)*g. Subtract from xK
                     // to find Cauchy point.
                stepLenCauchy = num*sqrt(num)/denom;
                     // Because sqrt(num) is length of g.
                stepLenCutback = ArrayInnerProduct(probDim,
                     gradHalfRSq, 1, dxNR, 1);
                stepLenCutback = 0.2 +
                     0.8*num*num/denom/fabs(stepLenCutback);
                     //   Here, stepLenCutback is really eta, the
                     // fraction of stepLenNR to dogleg break point.
                ArrayScalarProduct(probDim,
                     stepLenCutback, dxNR, vCauchyToCut);
                     // Here, v = eta*dxNR.
                ArrayDifference(probDim, vCauchyToCut,
                     dxCauchy, vCauchyToCut);
                     // v = eta*dxNR - dxCauchy.
                stepLenCutback *= stepLenNR;
                if (utlReportLevel == RL_VERBOSE)
                     {
                     cout << "\nstep lengths: Cauchy:\t" << stepLenCauchy
                          << "\tCutback:\t" << stepLenCutback;
                     }
                } // end firstDog == YES
           //
           // Here, have dogleg points and need dx.
           if (stepLenTrust >= stepLenCutback)
                {
                // Take partial step in NR direction.
                ArrayScalarProduct(probDim,
                     stepLenTrust/stepLenNR, dxNR, dx);
                }
           else if (stepLenTrust <= stepLenCauchy)
                {
                // Take partial step in SD direction.
                ArrayScalarProduct(probDim,
                     stepLenTrust/stepLenCauchy, dxCauchy, dx);
                }
           else
                {
                // Find convex combination of dxCauchy and dxCutback.
                double temp = ArrayInnerProduct(probDim,
                     vCauchyToCut, 1, dxCauchy, 1);
                double tempv = ArraySumSquares(probDim, vCauchyToCut);
                double lambda = temp*temp - tempv *
                     (stepLenCauchy*stepLenCauchy -
                     stepLenTrust*stepLenTrust);
                lambda = (sqrt(lambda) - temp) / tempv;
                ArraySaxpy(probDim, lambda, vCauchyToCut, dxCauchy, dx);
                     // dx = lambda*v + dxCauchy.
                }
           } // end finding step with lenTrust < lenNR
      //
      // Here, have step dx with length stepLenTrust.
      if (utlReportLevel == RL_VERBOSE)
```

```
          cout << "\nstep:\t" << stepLenTrust;
     return;
     } // End FindStep().

static TrustRegionStatusList EvaluateStep(void)
     {
     //   Update the trust region, to seek a different dx in
     // current iteration, or to use during next iteration.
     //
     // Check if increased trust region failed.
     if (trustRegionStatus == TRS_INCREASED
          && halfRSqKp1 > halfRSqKp1Prev)
          {
          //   Trust region was increased into a worse cost function.
          // Restore previous values and accept.
          //   Note that previous values of actual and estimated
          // change in cost function are still valid.
          if (utlReportLevel == RL_VERBOSE)
               cout << "\trejected (worse cost)";
          return TRS_ACCEPTED_AFTER_REJECT;
          }
     //
     //   Prepare to evaluate step's progress.
     //   Both g2Tdx ("initial slope") and deltaHalfRSq are negative
     // for smaller halfRSq. Good progress made when delta is more
     // negative than g2Tdx, or when delta < g2Tdx.
     //   Acceptable progress when delta <= ALPHA*g2Tdx.
     g2Tdx = -ArrayInnerProduct(probDim, gradHalfRSq, 1, dx, 1);
     deltaHalfRSq = halfRSqKp1 - halfRSqK;
     //
     // Check that cost function making progress.
     if (deltaHalfRSq > ALPHA*g2Tdx)
          {
          // Actual halfRSq too large.
          if (CheckStagnatedVector(probDim, dx, xKp1, relStepTol)
               == TC_TERMINATE)
               {
               return TRS_STAGNANT_SOLUTION;
               }
          else if (trustRegionStatus == TRS_INCREASED)
               {
               //   Last trial step was good enough that increased
               // trust length. If here, the increased length had
               // better cost than previously, so don't reject.
               return TRS_ACCEPTED;
               }
          else
               {
               // Decrease trust length.
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\tcut (poor cost)";
               double lambda = g2Tdx/2/(g2Tdx - deltaHalfRSq);
               if (lambda < 0.1) lambda = 0.1;
               else if (lambda > 0.5) lambda = 0.5;
               stepLenTrust *= lambda;
               return TRS_REDUCED;
               }
          } // end check that making sufficient progess
     //
     //   Cost function sufficiently small. May increase or accept
     // it for current iteration, and may also change it for next
     // iteration, all depending on success of model at predicting
     // values.
     //   Find expected change in cost function. Note if here then
     // current step has been accepted, so xHold free.
     J->Product(dx, xHold);
     deltaHalfRSqPredict = g2Tdx + ArraySumSquares(probDim, xHold)/2;
```

```
            //   Predicted reduction = gTdx + dxT*(JTJ)*dx/2
            // = gTdx + (Jdx)T*(Jdx)/2. Note JTJ approximates
            // Hessian of halfRSq.
      //
      // Make decision about current, acceptable, trust region.
      if (trustRegionStatus == TRS_REDUCED
            || stepLenTrust == stepLenNR)
            {
            //   If just reduced, or if using NR step, then don't
            // consider increasing trust region during this iteration.
            return TRS_ACCEPTED;
            }
      else if (fabs(deltaHalfRSqPredict - deltaHalfRSq)
            <= -0.1*deltaHalfRSq
            ||
            deltaHalfRSq <= g2Tdx)
            {
            //   Increase trust region if prediction was very good,
            // or if actual decrease very large. Store current values
            // so can recover later if needed.
            if (utlReportLevel == RL_VERBOSE)
                  cout << "\tdoubled (good cost)";
            stepLenTrustPrev = stepLenTrust;
            stepLenTrust *= 2;
            VariableType *temp = xHold; xHold = xKp1; xKp1 = temp;
            temp = rHold; rHold = rCurrent; rCurrent = temp;
            halfRSqKp1Prev = halfRSqKp1;
            return TRS_INCREASED;
            }
      // Here, no reason to increase the acceptable trust region.
      return TRS_ACCEPTED;
      } // End EvaluateStep().

/* ==========< End helper functions >=================== */


/* ====================< End File STDDOGLG.CPP >============================ */
```

```
/* ====================< File TERMCOND.HPP >==========================
======================< rev 06/22/96 >================================

     Headers for TERMCOND.CPP.
     Standard functions for handling a subset of termination
conditions for nonlinear solution methods.

     Functions defined:
(1) Termination criteria. Check whether termination criteria
satisfied, as described below.
(2) Default tolerances for the termination tests.

     General termination criteria:
     Ideally, a solution method should terminate when the
residuals are nearly zero. In general, solution functions do
not rely on recognizing zero residuals, since this condition
depends on the scaling of the problem. Therefore the methods
also terminate if the residual norm or the estimated solution
do not change much after an iteration. These tests may also
catch a stagnated solution, i.e. one which has converged to
a stationary point of the residual norm, or one which has
converged to an incorrect estimated solution, due to an
incorrect or inaccurate Jacobian, roundoff errors, and so on.
     Solution methods terminate on:
(1) A zero residual vector. This is reported when the magnitude
of each residual element is smaller than the input tolerance
defining absolute zero.
(2) A stagnant solution. This may indicate a nearly zero
residual or a stalled method. Stagnation is reported when the
fractional change of each component of the estimated solution
is less than the input relative step tolerance.
     Note the fractional change can be compared to either the
starting point or the ending point of the step. If the test
passes, these points will be close, so in a sense it doesn't
matter which use. However for consistency among the methods,
use the ending point.
(3) A stagnant residual norm. Stagnation is reported if the
fractional change in the residual norm is less than the
relative step tolerance. In addition, a stagnant norm is reported
if the norm increases and the method has no means of recovering.
(4) An evaluation error in the residual or the Jacobian function,
if the method has no means of recovering.
(5) An evaluation error in the Jacobian solution routine.
Effectively this means the Jacobian is singular to machine
precision.
(6) The method exceeds its iteration limit.
     Return codes for these cases are given in "probdefn.hpp".

     Termination criteria tested here:
     The functions declared in this file test criteria 1 (zero
residual vector), 2 (stagnant solution), and 3 (stagnant residual
norm).

     Default tolerances:
     These functions check a trial tolerance for reasonableness,
replacing it with a default if outside the expected range. A
function is defined for the absolute zero tolerance and for the
relative step tolerance. In both cases, a tolerance of zero is
considered unreasonable, so general solution methods called with
tolerances of zero will use default values, provided they check
the values using these functions. The general solution methods
must invoke these functions explicitly, during initialization.

     Sources:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983.
```

```
     W.H. Press et al, "Numerical Recipes in C, the art of
scientific computing," Cambridge University Press, 1988.

=======================< File TERMCOND.HPP >======================== */

#ifndef TERMCOND_HPP
#define TERMCOND_HPP


#include "probdefn.hpp"
     // For typedef of VariableType.


/* ==========< Check termination >==========================
=============< Check termination >======================== */
enum TerminateContinueList
     {
     TC_TERMINATE,
     TC_CONTINUE
     };

TerminateContinueList CheckZeroVector(int arrayCount,
     const VariableType r[],
     const VariableType zeroTolerance);

TerminateContinueList CheckStagnatedVector(int arrayCount,
     const VariableType dxK[], const VariableType xKp1[],
     const VariableType stepTolerance);

TerminateContinueList CheckStagnatedNorm
     (const double residNormK, const double residNormKp1,
     const VariableType stepTolerance);

/* ==========< End check termination >=================== */


/* ==========< Check tolerance >============================
=============< Check tolerance >======================== */
void CheckAbsoluteZeroTolerance(VariableType *zeroTolerance);

void CheckRelativeStepTolerance(VariableType *stepTolerance);

/* ==========< End check tolerance >===================== */

#endif


/* ====================< End File TERMCOND.HPP >================== */
```

```
/* ====================< File TERMCOND.CPP >==========================
=======================< rev 06/22/96 >================================

     Standard functions for handling a subset of termination
conditions for nonlinear solution methods.
     Documentation is in the header file.

     Programming notes for code efficiency:
     The criteria are coded in separate functions to enforce
uniformity among solution methods. For greater efficiency, the
termination criteria would be folded into the code, or at least
the governing tolerances and residual norms would be made global
variables, to avoid unnecessary arguments in the function call.
In addition, if a solution method uses a residual norm, the test
for a zero residual vector could be made by the norm calculation
routine.

=======================< File TERMCOND.CPP >======================== */

#include <float.h>
#include <math.h>


#include "termcond.hpp"


/* ==========< User preferences >============================
     Check user preferences.
=============< User preferences >======================== */
// =====< Variable type >=====
#ifndef VARIABLES_ARE_DOUBLES
#error VARIABLES_ARE_DOUBLES is not defined
#endif

/* ==========< End user preferences >==================== */


/* ==========< Global constants >===========================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
=============< Global constants >======================== */
#if (VARIABLES_ARE_DOUBLES == 1)
const double VARIABLE_TYPE_MIN = DBL_MIN;
const double VARIABLE_TYPE_EPSILON = DBL_EPSILON;
const double VAR_SMALL = 1000*DBL_MIN;
const double ZERO_TOLERANCE_DEFAULT =
     pow(DBL_EPSILON, 0.3333);
     // Default value suggested by Dennis.
const double STEP_TOLERANCE_DEFAULT =
     pow(DBL_EPSILON, 0.6667);
     // Default value suggested by Dennis.
#else
const float VARIABLE_TYPE_MIN = FLT_MIN;
const float VARIABLE_TYPE_EPSILON = FLT_EPSILON;
const float VAR_SMALL = 1000*FLT_MIN;
const float ZERO_TOLERANCE_DEFAULT =
     pow(FLT_EPSILON, 0.3333);
     // Default value suggested by Dennis.
const float STEP_TOLERANCE_DEFAULT =
     pow(FLT_EPSILON, 0.6667);
     // Default value suggested by Dennis.
#endif

const double RESID_NORM_SMALL = 1000*DBL_MIN;

/* ==========< End global constants >==================== */
```

```
/* ==========< Check termination >==========================
=============< Check termination >======================== */
TerminateContinueList CheckZeroVector(int arrayCount,
      const VariableType r[],
      const VariableType zeroTolerance)
      {
      //   Check whether residual vector elements are
      // nearly zero.
      for (; arrayCount>0; arrayCount--)
            {
            if (fabs(*r++) > zeroTolerance)
                  return TC_CONTINUE;
            }
      return TC_TERMINATE;
      } // End CheckZeroVector().

TerminateContinueList CheckStagnatedVector(int arrayCount,
      const VariableType dxK[], const VariableType xKp1[],
      const VariableType stepTolerance)
      {
      //   Test small change in elements of estimated solution.
      //   Stationary if change is less than stepTolerance times
      // new value. Absolute value taken since not guaranteed
      // positive. Current value perturbed by VAR_SMALL in case
      // it is nearly zero.
      //
      for (; arrayCount>0; arrayCount--)
            {
            if (fabs(*dxK++) > stepTolerance*(fabs(*xKp1++) + VAR_SMALL))
                  // Some element changed by too much.
                  return TC_CONTINUE;
            }
      return TC_TERMINATE;
      } // End CheckStagnatedVector().

TerminateContinueList CheckStagnatedNorm
      (const double residNormK, const double residNormKp1,
      const VariableType stepTolerance)
      {
      //   Test stationary residual norm.
      //   Stationary (or increasing) if change is less than
      // stepTolerance times former value. Former value is
      // perturbed by RESID_NORM_SMALL in case it is nearly
      // zero.
      //   Note assume residual norm always positive so
      // absolute value not needed.
      //   Note 1 variation is to test change against average
      // of former and present values by:
      // if (2*(residNormK - residNormKp1) > stepTolerance*
      //   (residNormK + residNormKp1 + RESID_NORM_SMALL))
      if ((residNormK - residNormKp1) >
            stepTolerance*(residNormK + RESID_NORM_SMALL))
            {
            return TC_CONTINUE;
            }
      return TC_TERMINATE;
      } // End CheckStagnatedNorm().

/* ==========< End check termination >================== */


/* ==========< Check tolerance >============================
=============< Check tolerance >======================== */
void CheckAbsoluteZeroTolerance(VariableType *zeroTolerance)
      {
      if (*zeroTolerance < VARIABLE_TYPE_MIN)
```

```
            {
            //   Do not permit the largest residual element
            // value that can be considered zero to take on
            // a zero or negative value.
            *zeroTolerance = ZERO_TOLERANCE_DEFAULT;
            }
      return;
      }

void CheckRelativeStepTolerance(VariableType *stepTolerance)
      {
      if (*stepTolerance < VARIABLE_TYPE_EPSILON)
            {
            //   Never try to resolve fractional differences in
            // the variable elements, which are smaller than
            // that supported by the variable type.
            *stepTolerance = STEP_TOLERANCE_DEFAULT;
            }
      return;
      }

/* ==========< End check tolerance >==================== */


/* ====================< End File TERMCOND.CPP >=================== */
```

```
/* ====================< File UTLDEPND.HPP >=========================
=======================< rev 02/05/96 >================================

     Headers for UTLDEPND.CPP.
     Defines implementation-dependent functions-- those which
vary with the compiler or machine.

     Environment dependencies can be set through:
(1) Macros, set in this header using #define, which control
compilation in the project code files. See "user preferences"
below.
(2) A general function to be called on program initialization.
For example, this function could:
(2a) Register other functions with atexit(), so that they are
run at the end of the program;
(2b) Set compiler-specific environment control variables;
(2c) Set the _new_handler;
(2d) Collect special user inputs; etc.
(3) Particular functions called at the user's discretion.
For example, command-line processing varies from compiler
to compiler.
(4) Extern constants, declared in this header file, and
initialized in the code file. These constants can set
environment-specific parameters, such as the maximum length
of a file name. Note many environment-specific parameters
are defined in the language, e.g. in <float.h>.

     User preferences:
     Each of the user preferences below is set by a macro
which is #defined in this header file. The macro value,
either 0 or 1, is checked in the code file.
(1) Arrays are copied using either the string command memcpy,
or an element-by-element version.
(2) Arrays are compared using either the string command memcmp,
or an element-by-element version.
(3) Array functions either check for overflow (nominally safer)
or not (faster and smaller code).
(4) Vector-vector inner products are calculated either
directly or using a scaled version.

     Standard choices for Symantec C++ v. 8.0.1 for Macintosh:
(1) Element-by-element versus string copy. The element-by-
element version is faster.
(2) Element-by-element versus string compare. The element-
by-element version is faster.
(3) Control overflow versus no overflow control. The Symantec
compiler treats overflows as INF and is able to compare
INF to numbers. Moreover, the default Symantec numerical
environment gives long doubles and doubles the same number
of decimal digits, so in the event a result exceeds the limits
for a double, the overflow check does not really compare
meaningful numbers anyway. Since the overflow checks add to
code size and reduce execution speed, there is no reason
to include them.
     For more information on the size of doubles versus
floats, see <float.h> and the compiler options documentation.
(4) Direct versus scaled inner products. The choice does not
necessarily vary with the compiler, but with the problem
dimension. Longer arrays may require scaled calculations; so
far no problems have been observed.

     Environment initialization:
main()
     {
     DependencyInitialization();
     ...
     return;
```

```
     }

     Command line arguments:
// Not defined.

     Environment-dependent constants:
     The maximum number of characters allowed in a file name.
This is given using #define rather than as an extern const int,
since it may have to be known at compile time.
     Flags for opening a file. Since different operating systems
terminate lines differently (i.e. Macintosh uses \r, and DOS
uses \n, to terminate lines in a file), the flags used to
open a file are system-dependent. Examples:
ofstream outFile.open("fileName", FILE_OPEN_FLAGS|ios::ate);
     //   Note ios::app insures writes are to end of file,
     // but still replaces old file contents. The ios::ate
     // flag specifies move to end of file on open.
ofstream outFile1.open("fileName1", FILE_OPEN_FLAGS|ios::noreplace);

=======================< File UTLDEPND.HPP >======================= */

#ifndef UTLDEPND_HPP
#define UTLDEPND_HPP


/* ==========< Set user preferences >=======================
     For each #defined macro, choose 1 or 0.
==============< Set user preferences >==================== */
#define ARRAY_COPIES_USE_MEMCPY 0
#define ARRAY_COMPARES_USE_MEMCMP 0

#define ARRAY_OPS_CONTROL_OVERFLOWS 0
#define INNER_PRODUCTS_USE_SCALE_FACTOR 0

/* ==========< End set user preferences >================= */


/* ==========< User preference consequences >===============
     Header statements which depend on user preferences.
==============< User preference consequences >============ */
// =====< Copy and compare arrays >=====
#if (ARRAY_COPIES_USE_MEMCPY == 1 \
     || ARRAY_COMPARES_USE_MEMCMP == 1)
#include <string.h>
#endif

// =====< Scale inner products >=====
#if (INNER_PRODUCTS_USE_SCALE_FACTOR == 1)
extern const double LOG_BASE;
extern const int SCALE_THRESHOLD;
#endif

/* ==========< End user preference consequences >======== */


/* ==========< Environment dependent constants >============
==============< Environment dependent constants >========== */
#define FILE_NAME_LENGTH_MAX 21
extern const long int FILE_OPEN_FLAGS;

/* ==========< End environment dependent constants >===== */


/* ==========< Environment initialization >=================
==============< Environment initialization >============== */
void DependencyInitialization(void);
```

```
/* ==========< End environment initialization >========== */

#endif

/* ====================< End File UTLDEPND.HPP >================== */
```

```
/* ====================< File UTLDEPND.CPP >=========================
=======================< rev 02/06/96 >================================

     Defines implementation-dependent functions-- those which
vary with the compiler or machine.
     Documentation is in the header file.

     This code file is for Symantec C++ v. 8.0.1 for Macintosh.

=======================< File UTLDEPND.CPP >======================== */

#include <console.h>
#include <float.h>
#include <iostream.h>
#include <lowmem.h>
#include <math.h>
#include <stdio.h>


#include "utldepnd.hpp"


/* ==========< User preferences >============================
     Check user preference values given in header file.
=============< User preferences >======================== */
// =====< Copy arrays >=====
#if (ARRAY_COPIES_USE_MEMCPY != 0 \
     && ARRAY_COPIES_USE_MEMCPY != 1)
#error ARRAY_COPIES_USE_MEMCPY must be 0 or 1
#endif

// =====< Compare arrays >=====
#if (ARRAY_COMPARES_USE_MEMCMP != 0 \
     && ARRAY_COMPARES_USE_MEMCMP != 1)
#error ARRAY_COMPARES_USE_MEMCMP must be 0 or 1
#endif

// =====< Control overflows >=====
#if (ARRAY_OPS_CONTROL_OVERFLOWS != 0 \
     && ARRAY_OPS_CONTROL_OVERFLOWS != 1)
#error ARRAY_OPS_CONTROL_OVERFLOWS must be 0 or 1
#endif

// =====< Scale inner products >=====
#if (INNER_PRODUCTS_USE_SCALE_FACTOR != 0 \
     && INNER_PRODUCTS_USE_SCALE_FACTOR != 1)
#error INNER_PRODUCTS_USE_SCALE_FACTOR must be 0 or 1
#endif

/* ==========< End user preferences >=================== */


/* ==========< User preference consequences >================
     Header statements which depend on user preferences.
=============< User preference consequences >============ */
// =====< Scale inner products >=====
#if (INNER_PRODUCTS_USE_SCALE_FACTOR == 1)
const double LOG_BASE = log(FLT_RADIX);
const int SCALE_THRESHOLD = 10;
#endif

/* ==========< End user preference consequences >======== */


/* ==========< Environment dependent constants >============
=============< Environment dependent constants >========= */
const long int FILE_OPEN_FLAGS =
```

```
      ios::out|ios::translated;
      // 'translated' replaces \n with \r on output.

/* ==========< End environment dependent constants >===== */


/* ==========< Environment initialization >=================
==============< Environment initialization >============== */
void DependencyInitialization(void)
      {
      // Insure files are created as text.
      _ftype = 'TEXT';
      //
      //   Choose the application to use when a file
      // created by the program is double-clicked:
      // MS Word-- 'MSWD'
      // MS Excel-- 'XCEL'
      // Simple Text, Teach Text-- 'ttxt'
      // Ask for application-- 'text'
      // Do nothing on double-click-- '????' (literally!)
      _fcreator = 'MSWD';
      //
      //   Give the console the name of the application. Note
      // the console structure wants a Pascal string, which
      // ordinarily would require that \p start the string,
      // but the Toolbox global CurApName already is a
      // Pascal string, so nothing special is required.
      //console_options.title = "\pProgram Name";
      console_options.title = LMGetCurApName();
      //
      // Extend console length from default 25 lines of text.
      //console_options.nrows = 30;
      return;
      }

/* ==========< End environment initialization >========== */


/* ====================< End File UTLDEPND.CPP >================== */
```

```
/* ====================< File UTLINPUT.HPP >=========================
=======================< rev 03/02/96 >================================

     Headers for UTLINPUT.CPP.
     General support for getting user inputs.

     Get one of a list of input options:
cout << "Output to file?";
int i = AskOneChoice("yes,no");
     //   Returns 1 if user enters y, ye, or yes. Returns 0 if
     // user enters n or no. Forces user to make one of these
     // choices.
cout << "\nExisting file disposition";
i = AskOneChoice("replace,rename,append");
     //   Returns 1 if user enters rep, repl, etc. Returns 2 if
     // user enters ren, rena, etc. Returns 3 if user enters a,
     // ap, etc. Forces user to make one of these choices,
     // and forces unambiguous choice (e.g. "r" input not accepted).
cout << "\nSolution method";
i = AskOneChoice("newton,,conjugate gradients,,random walk");
     //   User input required as above. Double comma notation
     // causes new line between listed options.

     Get a file name and open for output:
ofstream outFile;
cout << "\nSend output to";
AskOutputFile(&outFile);
     //   Gets file name from user. If the file exists, checks
     // whether to overwrite or append. Opens file and reports
     // error if unsuccessful.
outFile << "Output data\n";
outFile.close();

=======================< File UTLINPUT.HPP >======================== */


#ifndef UTLINPUT_HPP
#define UTLINPUT_HPP

#include <fstream.h>


int AskOneChoice(const char *const choiceString);
void AskOutputFile(ofstream *outputFile);
void AskStreamFormats(ostream *os);


#endif


/* ====================< End File UTLINPUT.HPP >=================== */
```

```
/* ====================< File UTLINPUT.CPP >==========================
=======================< rev 03/02/96 >================================

     General support for getting user inputs.
     Documentation is in the header file.

=======================< File UTLINPUT.CPP >======================== */

#include <iostream.h>
#include <string.h>

#include "utldepnd.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "utlinput.hpp"


/* ==========< User preferences >============================
     Check and set user preferences.
=============< User preferences >======================== */
// =====< Choice string >=====
static const int CHARS_TO_MATCH_P2 = 16;
     //   Maximum number of input characters to be read
     // and matched, plus one for termination character,
     // plus one for starting character.
static const char CHOICE_DELIMITER = ',';

// =====< File name length >=====
#ifndef FILE_NAME_LENGTH_MAX
#error FILE_NAME_LENGTH_MAX is not defined
#endif
     // Macro defined in "utldepnd.hpp".

/* ==========< End user preferences >=================== */


/* ==========< Function AskOneChoice >=====================
=============< Function AskOneChoice >=================== */
int AskOneChoice(const char *const choiceString)
     {
     //   Print out the choices, inserting a space after
     // each choice delimiter, and inserting "\n " after
     // two consecutive choice delimiters.
     //   After printing the choice string, get a user
     // input and match it to one of the string choices.
     //   Repeat until find a unique, unambiguous match.
     //   Assume choiceString has at least one character.
     //
     while (1)
          {
          // Write out the choices.
          cout.put(' ');
          cout.put('<');
          const char *choiceStringP = choiceString;
          char choiceChar;
          while ((choiceChar = *choiceStringP++) != '\0')
               {
               cout.put(choiceChar);
               if (choiceChar == CHOICE_DELIMITER)
                    {
                    // Start new choice, possibly on a new line.
                    if (*choiceStringP == CHOICE_DELIMITER)
                         {
                         choiceStringP++;
                         cout.put('\n');
```

```
                    }
                cout.put(' ');
                }
        }
    cout.put('>');
    cout.put(' ');
    //
    // Fetch a line from cin.
    char matchString[CHARS_TO_MATCH_P2];
    *matchString = CHOICE_DELIMITER;
    do
        {
        cin.getline(matchString+1, CHARS_TO_MATCH_P2, '\n');
            //   Gets up to CHARS_TO_MATCH characters, or
            // until hits '\n'. It extracts the '\n' from
            // cin, but stores terminating '\0' instead.
            //   Use getline instead of extraction operator
            // so can have spaces in choice string.
        } while (*(matchString+1) == '\0');
    int matchLen = strlen(matchString+1);
    //
    // Find matchString in choiceString.
    int matchCount = 1;
    choiceStringP = choiceString;
    while (1)
        {
        // Assume at a testable location.
        if (strncmp(choiceStringP, matchString+1, matchLen) == 0)
            {
            //   Found a match. Check that no other occurence
            // of delimiter plus matchString in choiceString.
            choiceStringP = strstr(choiceStringP+1, matchString);
            if (choiceStringP == 0)
                return matchCount;
            else
                //   The matchString is not unique. Break out
                // of current while-loop to present choices
                // again.
                break;
            }
        // Move on to next delimiter.
        choiceStringP = strchr(choiceStringP+1, CHOICE_DELIMITER);
            //   Finds next occurence of CHOICE_DELIMITER in
            // choiceString, or returns null if no more.
        if (choiceStringP == 0)
            // No more choices to match.
            break;
        matchCount++;
        if (*++choiceStringP == CHOICE_DELIMITER)
            // Skip a double delimiter.
            choiceStringP++;
        }
    }
return 0;
; // End AskOneChoice().

/* ==========< End Function AskOneChoice >=============== */


/* ==========< Function AskOutputFile >====================
==============< Function AskOutputFile >================== */
void AskOutputFile(ofstream *outputFile)
    {
    char fileName[FILE_NAME_LENGTH_MAX + 1];
    cout << " File name: ";
    do
        {
```

```
            cin.width(FILE_NAME_LENGTH_MAX + 1);
                  //   Applies to next I/O operation; then, unless
                  // the "stickywidth" flag is set, reverts to
                  // width(0).
            cin >> fileName;
                  //   Use extraction operator instead of getline,
                  // to prevent white space characters in file name.
            } while (*fileName == '\0');
      outputFile->open(fileName, FILE_OPEN_FLAGS|ios::noreplace);
      if (outputFile->fail())
            {
            cout << " Existing file disposition";
            int tempInt = AskOneChoice("append,replace,new name,quit");
            if (tempInt == 1)
                  outputFile->open(fileName, FILE_OPEN_FLAGS|ios::app);
            else if (tempInt == 2)
                  outputFile->open(fileName, FILE_OPEN_FLAGS);
            else if (tempInt == 3)
                  AskOutputFile(outputFile);
            if (outputFile->fail())
                  FATAL(ERR_FILE_OPERATION, 1011506);
            }
      return;
      }

/* ==========< End Function AskOutputFile >============== */


/* ==========< Function AskStreamFormats >==================
==========< Function AskStreamFormats >================ */
void AskStreamFormats(ostream *os)
      {
      cout << " Columns per line (0): ";
      cin >> utlReportColsPerLine;
      cout << " Notation";
      if (AskOneChoice("mixed,scientific") == 1)
            StreamSetMixed(*os);
      else
            StreamSetScientific(*os);
      cout << " Print precision (6): ";
      int tempInt;
      cin >> tempInt;
      StreamSetPrecision(*os, tempInt);
      return;
      }

/* ==========< End Function AskStreamFormats >========== */


/* ====================< End File UTLINPUT.CPP >=================== */
```

```
/* ====================< File UTLREPRT.HPP >==========================
=======================< rev 06/28/96 >================================

     Headers for UTLREPRT.CPP.
     General support for output.

     Based on a utility file implemented by Guido Buzzi-Ferraris
in "Scientific C++: Building Numerical Libraries the Object-
Oriented Way," Addison-Wesley, 1993. The C stdio functions are
replaced by C++ IOstreams functions, and stream output utilities
are included.

     Global enumerations:
OnOffList commaDelimitedMode = ON;
SuccessFailureList itWorked = SUCCESS;
YesNoList initialized = NO;

     Standard output stream:
cout << "Beginning outer loop.\n";
streambuf *const originalCout = cout.rdbuf();
     // Store original cout for later use.
ofstream outFile("Program.out", ios::out);
     // Opens a file for output.
cout = outFile;
     // Directs output to the file.
cout << ...;
outFile.close();
cout = originalCout;
     // Sets cout back to standard cout.
cout << "Done with internal processing." << endl;

     Change stream format:
int oldPrecision = StreamSetPrecision(cout, 6);
StreamSetScientific(cout);
cout << "\nScientific notation " << 60.2;
StreamSetMixed(cout);
cout << "\nMixed notation " << 60.2;

     Control reporting:
utlReportLevel = RL_VERBOSE;
     // Default value is RL_LACONIC. Alternate RL_MUTE.
utlReportColsPerLine = 5;
     //   Default is 0. Behavior for value of zero should be
     // to print all related numbers on one line.
//   Values affect reporting operations in any file which includes
// this header and uses the variables. Values can be changed from
// any file which includes the header.

     Report a fatal error:
PREPARE_FATAL;
     //   Should be in global part of file if FATAL is called
     // from multiple functions in file.
FATAL("Value too high", 011496);
     // Preferred; unique integer based on function name.
FATAL(ERR_INDEX_RANGE, __LINE__);
     // Print line number in code.
FATAL_LINE(ERR_INDEX_RANGE);
     // Same as above.
FATAL(ERR_NEED_UNFACTORED, 47);

=======================< File UTLREPRT.HPP >======================== */

#ifndef UTLREPRT_HPP
#define UTLREPRT_HPP


#include <iostream.h>
```

```
/* ==========< Global enumerations >=======================
     Declares standard enumerations.
     Note enumeration elements cannot be repeated, so
these are declared globally to allow multiple uses.
=============< Global enumerations >===================== */
enum OnOffList
     {
     OFF = 0,
     ON
     };

enum SuccessFailureList
     {
     FAILURE = 0,
     SUCCESS
     };

enum YesNoList
     {
     NO = 0,
     YES
     };

/* ==========< End global enumerations >================= */


/* ==========< Global strings >============================
     "Extern" since defined once, in the code file, for use
in multiple files. Constant pointers to constant strings,
so they cannot be modified.
=============< Global strings >========================== */
// =====< File >=====
extern const char *const ERR_FILE_OPERATION;

// =====< Programmer >=====
extern const char *const ERR_HOHO;
extern const char *const ERR_NOT_IMPLEMENTED;

// =====< Run-time >=====
extern const char *const ERR_FREE_STORE;
extern const char *const ERR_FUNCTION_EVALUATION;
extern const char *const ERR_INPUT_RANGE;
extern const char *const ERR_ITERATION_COUNT;

// =====< Vector-matrix >=====
extern const char *const ERR_DIMENSION_MATCH;
extern const char *const ERR_LOOKS_SINGULAR;
extern const char *const ERR_NEED_UNFACTORED;

/* ==========< End global strings >====================== */
```

```
/* ==========< Stream manipulation >=======================
     Support stream output operations.
=============< Stream manipulation >===================== */
// =====< Format control bits >=====
extern const long int STREAM_MIXED_FLAGS;
extern const long int STREAM_SCIENTIFIC_FLAGS;

// =====< Change stream format >=====
inline int StreamSetPrecision
     (ostream &const os, const int newPrecision)
     {return os.precision(newPrecision);}
     // Returns old precision.
```

```
inline void StreamSetMixed(ostream &const os)
      {os.unsetf(STREAM_MIXED_FLAGS);}

inline void StreamSetScientific(ostream &const os)
      {os.setf(STREAM_SCIENTIFIC_FLAGS, ios::fixed);}

/* ==========< End stream manipulation >================= */


/* ==========< Control reporting >=========================
      Three program-wide variables, declared extern, and
initialized in the code file, control common aspects of
reporting.
=============< Control reporting >======================= */
// =====< Level of detail >=====
enum ReportLevelList
      {
      RL_MUTE,
      RL_LACONIC,
      RL_VERBOSE
      };

extern ReportLevelList utlReportLevel;

// =====< Format >=====
extern int utlReportColsPerLine;

/* ==========< End control reporting >=================== */


/* ==========< Report fatal error >=======================
=============< Report fatal error >=================== */
#define PREPARE_FATAL \
      static const char *const PREPARE_FATAL_NAME = __FILE__
      // Static to enforce file scope only.

#define FATAL(ErrString, ErrInt) \
      FatalErrorFileName(PREPARE_FATAL_NAME, ErrString, ErrInt);

void FatalErrorFileName
      (const char *const fileName,
      const char *const errorDescription,
      const int errorLocator);

/* ==========< End report fatal error >================== */

#endif


/* ====================< End File UTLREPRT.HPP >=================== */
```

UtlReprt.cpp

```
/* ====================< File UTLREPRT.CPP >=========================
=======================< rev 06/28/96 >==============================

      General support for output.
      Documentation is in the header file.

=======================< File UTLREPRT.CPP >======================= */

#include <stdlib.h>


#include "utlreprt.hpp"


/* ==========< Global strings >=============================
=============< Global strings >========================== */
// =====< File >=====
const char *const ERR_FILE_OPERATION =
      "Unsuccessful file operation";

// =====< Programmer >=====
const char *const ERR_HOHO =
      "Programmer note (HOHO)";
const char *const ERR_NOT_IMPLEMENTED =
      "Not implemented";

// =====< Run-time >=====
const char *const ERR_FREE_STORE =
      "Free store exhausted";
const char *const ERR_FUNCTION_EVALUATION =
      "Evaluation error";
const char *const ERR_INPUT_RANGE =
      "Input out of range";
const char *const ERR_ITERATION_COUNT =
      "Hit iteration limit";

// =====< Vector-matrix >=====
const char *const ERR_DIMENSION_MATCH =
      "Dimensions do not match";
const char *const ERR_LOOKS_SINGULAR =
      "Singular to machine precision";
const char *const ERR_NEED_UNFACTORED =
      "Unfactored matrix required";

/* ==========< End global strings >====================== */


/* ==========< Stream manipulation >=======================
      The stream format is controlled by flag bits stored
in a long integer. The relevant bits are:
(1) ios::showpoint, which when set forces the decimal
point and full precision to print;
(2) ios::scientific, which when set uses scientific
notation; and
(3) ios::fixed, which when set uses fixed-point notation.
      If neither the scientific nor fixed bits are set, the
stream uses mixed notation as appropriate. The two bits
are combined in ios::floatfield.
      Inline functions, defined in header file:
inline int StreamSetPrecision
      (ostream &const os, const int newPrecision)
      {return os.precision(newPrecision);}
      // Returns old precision.
inline void StreamSetMixed(ostream &const os)
      {os.setf(STREAM_MIXED_FLAGS, STREAM_CLEAR_PRINT_FLAGS);}
inline void StreamSetScientific(ostream &const os)
      {os.setf(STREAM_SCIENTIFIC_FLAGS, STREAM_CLEAR_PRINT_FLAGS);}
```

```
/* =============< Stream manipulation >====================== */
// =====< Format control bits >=====
const long int STREAM_MIXED_FLAGS =
      (ios::showpoint | ios::floatfield);
      //   To mix formats as appropriate, neither floatfield
      // bit should be set, and the showpoint bit should not
      // be set. This flag identifies bits to clear.
const long int STREAM_SCIENTIFIC_FLAGS =
      (ios::showpoint | ios::scientific);
      //   Flags for scientific notation. Set scientific, and
      // set showpoint bit to force full precision to print.
      // In addition, the ios::fixed flag should be cleared.

/* ===========< End stream manipulation >================== */


/* ===========< Control reporting >=========================
      Declared extern in header file, so any file which
includes the header has access to and can change values.
==============< Control reporting >======================= */
// =====< Level of detail >=====
ReportLevelList utlReportLevel = RL_LACONIC;

// =====< Format >=====
int utlReportColsPerLine = 0;

/* ===========< End control reporting >=================== */


/* ===========< Report fatal error >=======================
==============< Report fatal error >======================= */
void FatalErrorFileName
      (const char *const fileName,
      const char *const errorDescription,
      const int errorLocator)
      {
      cerr << "\nFATAL error from location "
            << errorLocator << " in file " << fileName << ":\n"
            << errorDescription;
      exit(0);
      return;
      }

/* ===========< End report fatal error >================== */


/* ====================< End File UTLREPRT.CPP >=================== */
```

UtlTimer.hpp

```
/* ====================< File UTLTIMER.HPP >=========================
=======================< rev 01/27/96 >================================

     Headers for UTLTIMER.CPP.
     Defines a class RealTimer, for timing routines, and a class
CallCounter, for counting iterations of a routine.

     An object of class RealTimer stores the time it was instantiated
or reset. At the end of its duration, it prints to screen the time,
in seconds, since it was instantiated or last reset. A message text,
if defined, also is printed to screen.
     The message may be changed. The timer may be suspended and
restarted, e.g. for user input-output. The elapsed time, in seconds,
since instantiation or reset may be accessed.

     Instantiation (construction):
// Default constructor.
RealTimer timer1;
// Construction with a message text.
RealTimer timer2("Message text"); // Preferred form.
RealTimer timer3 = RealTimer("Message text"); // Explicit form.

     Public member functions:
float timeSince = timer1.Read();
timer1.Reset();
timer2.Reset("New message text");
timer3.SetMessage("New message text");
timer2.Suspend();
timer2.Resume();

     Suggested use: timing a whole program.
RealTimer timer1("Main program");
main() {...}

     Suggested use: timing a loop. Enclosing brackets force
the timer to leave scope after the loop is over.
{
RealTimer timer2("Loop 10000 times");
for (int i=0; i<10000; i++) {...}
}

     Not supported:
timer2 = timer1; // Cannot assign to an existing timer
     // from another existing timer.
RealTimer timer3(timer1); // Cannot copy-initialize. Note this
     // means should not pass timers as function returns.


     An object of class CallCounter records how many times it was
called. At the end of its duration, it prints to screen the call
count since it was instantiated or reset. A message text is
printed out if one is defined.
     The counter may be incremented or reset.

     Instantiation (construction):
// Default constructor.
CallCounter counter1;
// Construction with a message text.
CallCounter counter1("Message text"); // Preferred form.
CallCounter counter1 = CallCounter("Message text"); // Explicit form.

     Public member functions:
counter1.Call();
int counts = counter1.Read();
counter1.Reset();
counter1.Reset("New message text");
counter1.SetMessage("New message text");
```

```
/* ======================< File UTLTIMER.HPP >====================== */


#ifndef UTLTIMER_HPP
#define UTLTIMER_HPP

#include <time.h>


/* ==========< Class RealTimer >============================
============< Class RealTimer >========================== */
class RealTimer
      {
public:
      // =====< Constructors >=====
      RealTimer(const char *text);
            // Call by: RealTimer timer1("Outer loop"); (preferred form)
            // or by: RealTimer timer1 = RealTimer("Outer loop")
            // (explicit form)
            // or by: RealTimer timer1 = "Outer loop" (shorthand form)
      RealTimer(void);
            // Default constructor. Call by: RealTimer timer1;

      // =====< Destructor >=====
      ~RealTimer(void);

      // =====< Interface functions >=====
      float Read(void);
            // Time in seconds since instantiated or reset.
      void Reset(void);
            // Reset start time to current time.
      void Reset(const char *text);
      void SetMessage(const char *text);
            //   Create or change message to print on destruction.
            // Clear message by SetMessage("");
      void Suspend(void);
            // Do not count elapsed time until Resume.
      void Resume(void);

private:
      // =====< Class variables >=====
      clock_t holdTime;
            //   Holds starting time if running, and running
            // time so far if suspended.
      enum {NO = 0, YES = 1} suspnd;
      char *ptmessage; // Set to null if no message.

      };

/* ==========< End Class RealTimer >==================== */


/* ==========< Class CallCounter >=========================
============< Class CallCounter >======================== */
class CallCounter
      {
public:
      // =====< Constructors >=====
      CallCounter(const char *text);
      CallCounter(void);

      // =====< Destructor >=====
      ~CallCounter(void); // Destructor.

      // =====< Interface functions >=====
      void Call(void)
```

```
            {callCount++;}
      int Read(void) const
            {return callCount;}
      void Reset(void)
            {callCount = 0;}
      void Reset(const char *text);
      void SetMessage(const char *text);

private:
      // =====< Class variables >=====
      int callCount;
      char *ptmessage;

      };

/* ==========< End Class CallCounter >================== */

#endif

/* ====================< End File UTLTIMER.HPP >=================== */
```

```
/* ====================< File UTLTIMER.CPP >=========================
=======================< rev 01/24/96 >================================

     Defines classes RealTimer and CallCounter.
     Documentation is in header file.

     Use of printf instead of cout:
     These classes, on destruction, are supposed to print to stdout.
When a class object is declared globally, its destructor may be
called after the destructor for cout, which is itself an object
(of class ostream_withassign). Therefore these functions use the
C function printf, rather than C++ stream-based I/O.

======================< File UTLTIMER.CPP >====================== */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "utlreprt.hpp"
PREPARE_FATAL;


#include "utltimer.hpp"


/* ==========< Class RealTimer >============================
     A class for timing in seconds.
=============< Class RealTimer >========================== */
// =====< Constructors >=====
RealTimer::RealTimer(const char *text)
     {
     // Generic constructor.
     ptmessage = 0; // Set to null required by SetMessage.
     SetMessage(text);
     suspnd = NO;
     holdTime = clock();
     // Note constructors have no return.
     }

RealTimer::RealTimer(void)
     {
     // Default constructor. Instantiate with null message.
     ptmessage = 0;
     suspnd = NO;
     holdTime = clock();
     }

// =====< Destructor >=====
RealTimer::~RealTimer(void)
     {
     //   Destructor. Print elapsed time and message text.
     // Read time before start screen operations.
     float endTime = Read();
     printf("\nTimer seconds: %f", endTime);
     if (ptmessage != 0)
          printf(": %s", ptmessage);
     // Release message text memory to free store.
     delete [] ptmessage;
     }

// =====< Interface functions >=====
float RealTimer::Read(void)
     {
     if (suspnd == YES)
          return float(holdTime)/CLOCKS_PER_SEC;
     return float(clock() - holdTime)/CLOCKS_PER_SEC;
```

```
      }

void RealTimer::Reset(void)
      {
      if (suspnd == YES) suspnd = NO;
      holdTime = clock();
      return;
      }

void RealTimer::Reset(const char *text)
      {
      SetMessage(text);
      Reset();
      return;
      }

void RealTimer::SetMessage(const char *text)
      {
      delete [] ptmessage; // Does not hurt to apply to null.
      //   Input text may go out of scope if it's of local extent,
      // or if it has been allocated on free store. Therefore copy
      // text to a new string pointer allocated just for timer.
      ptmessage = new char[strlen(text) + 1]; // Get pointer to char
            // from free store. Use strlen to get enough characters.
            // Note destructor must release memory taken from free store.
      if (ptmessage == 0) FATAL(ERR_FREE_STORE, 1181913);
      strcpy(ptmessage, text); // Copy text from input pointer.
      return;
      }

void RealTimer::Suspend(void)
      {
      if (suspnd == YES) return;
      //   To suspend, change holdTime-- from clock at start,
      // to running time so far.
      holdTime = (clock() - holdTime);
      suspnd = YES;
      return;
      }

void RealTimer::Resume(void)
      {
      if (suspnd == NO) return;
      //   To resume, change holdTime-- from running time so far,
      // to a presumed starting time.
      suspnd = NO;
      holdTime = clock() - holdTime;
      }

/* ==========< End Class RealTimer >====================== */


/* ==========< Class CallCounter >==========================
     Inline member functions, defined in the header file:
void Call(void) {callCount++;}
int Read(void) const {return callCount++;}
void Reset(void) {callCount = 0;}
=============< Class CallCounter >======================= */
// =====< Constructors >=====
CallCounter::CallCounter(const char *text)
      {
      // Generic constructor.
      ptmessage = 0; // Set to null required by SetMessage.
      SetMessage(text);
      callCount = 0;
      // Note constructors have no return.
      }
```

```
CallCounter::CallCounter(void)
      {
      // Default constructor. Instantiate with null message.
      ptmessage = 0;
      callCount = 0;
      }

// =====< Destructor >=====
CallCounter::~CallCounter(void)
      {
      //   Destructor.
      printf("\nCounter calls: %d", callCount);
      if (ptmessage != 0)
            printf(": %s", ptmessage);
      // Release message text memory to free store.
      delete [] ptmessage;
      }

// =====< Interface functions >=====
void CallCounter::Reset(const char *text)
      {
      SetMessage(text);
      callCount = 0;
      return;
      }

void CallCounter::SetMessage(const char *text)
      {
      delete [] ptmessage; // Does not hurt to apply to null.
      //   Input text may go out of scope if it's of local extent,
      // or if it has been allocated on free store. Therefore copy
      // text to a new string pointer allocated just for timer.
      ptmessage = new char[strlen(text) + 1]; // Get pointer to char
            // from free store. Use strlen to get enough characters.
            // Note destructor must release memory taken from free store.
      if (ptmessage == 0) FATAL(ERR_FREE_STORE, 1031913);
      strcpy(ptmessage, text); // Copy text from input pointer.
      return;
      }

/* ==========< End Class CallCounter >================== */


/* ====================< End File UTLTIMER.CPP >=================== */
```

WtdDoglg.hpp

```
/* ====================< File WTDDOGLG.HPP >=========================
=======================< rev 08/21/96 >==============================

     Headers for WTDDOGLG.CPP.
     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis. Use
weighted r-square as cost function.

     Source:
     J.E. Dennis and R.B. Schnabel, "Numerical Methods for
Unconstrained Optimization and Nonlinear Equations," Prentice
Hall, 1983.

=======================< File WTDDOGLG.HPP >======================== */

#ifndef WTDDOGLG_HPP
#define WTDDOGLG_HPP


#include "fulljcbn.hpp"
     // For class FullJacobian.
#include "probdefn.hpp"
     // For form of residual and Jacobian functions.


/* ==========< Function SolveWeightedDogleg >===============
=============< Function SolveWeightedDogleg >============ */
SolveNonLinearReturnList SolveWeightedDogleg
     (const int problemDimension,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance);

/* ==========< End Function SolveWeightedDogleg >======== */

#endif


/* ====================< End File WTDDOGLG.HPP >=================== */
```

```
/* ====================< File WTDDOGLG.CPP >==========================
=======================< rev 09/23/96 >================================

     Solve a system of nonlinear equations using PLU factorization
in the double dogleg trust region method given by Dennis. Use
weighted r-square as cost function.
     Documentation is in the header file.

     Programming notes for double dogleg method:
     This file implements the pseudocode from Appendix A of the
text by Dennis and Schnabel. Major exceptions:
(1) It does not scale the residuals.
(2) It handles termination criteria by separate functions in
"termcond.cpp".
(3) It does not perturb a Jacobian identified as singular.
(4) It subtracts steps from, rather than adding steps to, xK when
finding xKp1. Thus, xKp1 = xK - dx where Dennis uses xK + dx.
(5) It reacts to residual errors, by reducing the trust region.
(6) It tests for zero residuals.
(7) It initializes the trust region to the Newton-Raphson length,
rather than to the Cauchy length.
(8) It performs the multiplication required to find J*(JT*r)
before, rather than after, factoring the Jacobian.
(9) It uses a weighted sum of squares of residuals, not half the sum
of squares, as the cost function.
(10) Where possible, it replaces matrix-vector multiplications,
vector-vector inner products, and vector sums of squares with
equivalent scalar expressions. (The scalar operations are derived
by applying the linear algebra operations to the equilibrium
problem structure and chosen cost function.)
(11) Where required to support the new computational details, it
changes the computational sequence, but not the logic, of the
algorithm.

=======================< File WTDDOGLG.CPP >======================== */

#include <math.h>


#include "arrayops.hpp"
#include "plujcbn.hpp"
#include "termcond.hpp"
#include "utlreprt.hpp"
PREPARE_FATAL;


#include "wtddoglg.hpp"

/* ==========< Select weighting rule >=======================
     Preprocessor definition of weighting rule selects
the actual code at compile time.
==============< Select weighting rule >==================== */
#define WT_RULE 24

#if (WT_RULE < 1 || WT_RULE > 24)
#error WT_RULE out of range
#endif

#if (WT_RULE == 1)
#define RULE_INITIALIZES_WEIGHTS_TO_ONE 1
#else
#define RULE_INITIALIZES_WEIGHTS_TO_ONE 0
#endif

#if (WT_RULE == 2 \
     || WT_RULE == 3 \
     || WT_RULE == 4 \
```

```
      || WT_RULE == 5 \
      || WT_RULE == 6 \
      || WT_RULE == 7 \
      || WT_RULE == 8 \
      || WT_RULE == 10 \
      || WT_RULE == 11 \
      || WT_RULE == 17 \
      || WT_RULE == 18)
#define RULE_USES_NEAR_ZERO 1
#else
#define RULE_USES_NEAR_ZERO 0
#endif

#if (WT_RULE == 5 \
     || WT_RULE == 6)
#define RULE_FINDS_WEIGHTS_AFTER_NR 1
#else
#define RULE_FINDS_WEIGHTS_AFTER_NR 0
#endif

#if (WT_RULE == 7 \
     || WT_RULE == 11)
#define RULE_USES_RT_NEAR_ZERO 1
#else
#define RULE_USES_RT_NEAR_ZERO 0
#endif

#if (WT_RULE == 9 \
     || WT_RULE == 10 \
     || WT_RULE == 11 \
     || WT_RULE == 12 \
     || WT_RULE == 13 \
     || WT_RULE == 14 \
     || WT_RULE == 15 \
     || WT_RULE == 16 \
     || WT_RULE == 19 \
     || WT_RULE == 20 \
     || WT_RULE == 21 \
     || WT_RULE == 22 \
     || WT_RULE == 23 \
     || WT_RULE == 24)
#define RULE_FINDS_JAC_ROW_LENGTHS 1
#else
#define RULE_FINDS_JAC_ROW_LENGTHS 0
#endif

#if (WT_RULE == 12 \
     || WT_RULE == 13 \
     || WT_RULE == 14 \
     || WT_RULE == 15 \
     || WT_RULE == 16 \
     || WT_RULE == 17 \
     || WT_RULE == 18 \
     || WT_RULE == 19 \
     || WT_RULE == 20 \
     || WT_RULE == 21 \
     || WT_RULE == 22 \
     || WT_RULE == 23 \
     || WT_RULE == 24)
#define RULE_INITIALIZES_TRUST_LEN_NEGATIVE 1
#else
#define RULE_INITIALIZES_TRUST_LEN_NEGATIVE 0
#endif

/* ==========< End select weighting rule >=============== */
```

```
/* ==========< Global constants >============================
     Note "const" variables are static by default-- that
is, they have global scope, but for file scope only.
==============< Global constants >======================== */
const int MAX_LOOPS = 100;
const double ALPHA = 1E-4;

/* ==========< End global constants >=================== */


/* ==========< Iteration status >============================
==============< Iteration status >======================== */
enum TrustRegionStatusList
     {
     TRS_ZERO_RESIDUALS,
     TRS_STAGNANT_SOLUTION,
     TRS_ACCEPTED,
     TRS_ACCEPTED_AFTER_REJECT,
     TRS_INHERITED,
     TRS_REDUCED,
     TRS_INCREASED
     };
     //   Dennis: retcode.
     //   Indicates if attempt to find the dogleg step is the
     // first try with an inherited trust region, a subsequent
     // try with a reduced trust region (retcode = 2), or a
     // subsequent try with an increased trust region
     // (retcode = 3).
     //   May also indicate that step selection is complete,
     // either because current step accepted (retcode = 0),
     // because current step is so small that the termination
     // test for a stagnant estimated solution is satisfied
     // (retcode = 1), or because found a zero residual vector.
     //   May also indicate that an increased step should be
     // rejected, and the original step, i.e. the one that
     // caused the increase, should be accepted.

/* ==========< End iteration status >=================== */


/* ==========< Global variables >============================
     "Static" keyword limits variables to file scope.
==============< Global variables >======================== */
// =====< Global versions of function arguments >=====
static int probDim;
static VariableType relStepTol;

// =====< Global scalar variables >=====
static double stepLenNR, stepLenCauchy, stepLenCutback;
     //   Distances, respectively, to: the Newton-Raphson
     // solution of the linearized equations; the predicted
     // minimizer of r-square in the steepest descent direction;
     // and the point at which the double dogleg curve rejoins
     // the Newton-Raphson direction.
static double stepLenTrust, stepLenTrustPrev;
     //   Dennis del: trust region length. Would like to
     // take a step of this length. If increase trust length
     // during the current iteration, store its previous
     // value (just prior to increasing).
static TrustRegionStatusList trustRegionStatus;
static double wrSquareK, wrSquareKp1, wrSquareKp1Prev;
     //   The cost function at beginning of iteration; at trial
     // point during iteration; and, if the trust length was
     // increased during the current iteration, at the last
     // trial point for the current iteration.
static double lenSqJTWr, lenFrthJTWrOnLenSqRtWJJTr;
static double scaleJTWrToCauchy;
```

```
      //   Multiply vector JTr by this to find step to Cauchy point,
      // the minimizer of rSquare in the steepest descent direction
      // when residuals linearized.
static double eta;
      // Ratio of stepLenCutback/stepLenNR.
static double gwTdx;
      //   Dennis calls this the initial slope. Not exactly
      // initial slope, but gTdx where g is the gradient of
      // weighted rSquare at xK.
      //   The value should be negative, since going downhill
      // in values of wrSquare. Calculate as -gTdx, using
      // negative sign since gTdx is positive for a descent
      // direction when dx subtracted. Dennis does not use
      // negative, but he adds dx.
static double deltaWRSquare, deltaWRSquarePredict;
static YesNoList firstDog;
      //   Indicates whether have calculated points on double
      // dogleg curve for the current iteration.
static SolveNonLinearReturnList exitReason;

// =====< Global scalars for weighting rules >=====
#if (RULE_USES_NEAR_ZERO == 1)
static VariableType nearZero;
      //   For one-weighted norms, a small number considered
      // nearly zero. Smaller residual magnitudes are treated as
      // zero by weighting rules.
      //   In the code, value is same as the user input of
      // absolute zero tolerance used for termination purposes
      // (i.e. nearZero just serves as a global instance of
      // absZeroTol).
#endif
#if (RULE_USES_RT_NEAR_ZERO == 1)
static VariableType rtNearZero;
#endif

// =====< Global dimensioned variables >=====
static PLUJacobian *J;
static VariableType *xK;
static VariableType *rCurrent;
      // Sometimes holds rK, sometimes rKp1.
static VariableType *dxNR, *dx;
      //   The Newton-Raphson step, and actual step taken,
      // respectively.
      //   Newton-Raphson step goes to solution of linearized
      // residual equations, which also minimizes the sum of
      // squares of the linearized residuals.
static VariableType *xKp1;
      // xKp1 = xK - dx. Note Dennis uses x + dx.
static VariableType *xHold, *rHold;
      //   Store intermediate value of xKp1, rKp1 for convenience
      // during one iteration. Also used for intermediate tasks.
static VariableType *vectorJTWr;
      // Note gradient of wrSquare = 2*JT*r.
static VariableType *weightsK;
      // Store weights used at current iteration.

// =====< Global dimensioned variables for weighting rules >=====
#if (RULE_FINDS_JAC_ROW_LENGTHS == 1)
static VariableType *jacRowLengths;
#endif

/* ==========< End global variables >=================== */


/* ==========< Declare weighting functions >================
     Finds weights according to the current weighting rule,
as set by #define WT_RULE.
```

```
     Also finds wrSquareK and stores Wr in xHold.
=============< Declare weighting functions >============= */
static YesNoList FindWeightsBeforeNR(void);

#if (RULE_FINDS_WEIGHTS_AFTER_NR == 1)
static void FindWeightsAfterNR(void);
#endif

/* ==========< End declare weighting functions >========= */


/* ==========< Declare helper functions >==================
=============< Declare helper functions >================ */
static double WeightedSumSquares(const VariableType x[]);
     //   Find xTWx where diagonal weighting matrix W
     // stored in weightsK.
static void FindStep(void);
static TrustRegionStatusList EvaluateStep(void);
static void FindDoglegBreakPoints(void);

#if (RULE_FINDS_WEIGHTS_AFTER_NR == 1)
static void FindNRDirectionPoint
     (ResidualEvaluationReturnList (*EvalResids)
          (const VariableType x[], VariableType r[]));
     //   FindNRDirectionPoint is a function whose single
     // argument is a pointer to a function of (x, r)
     // which returns RERList.
static TrustRegionStatusList EvaluateNRDirectionStep(void);
#endif

/* ==========< End declare helper functions >============ */


/* ==========< Function SolveWeightedDogleg >===============
=============< Function SolveWeightedDogleg >============ */
SolveNonLinearReturnList SolveWeightedDogleg
     (const int problemDim,
     VariableType x[], VariableType r[],
     ResidualEvaluationReturnList (*EvaluateResiduals)
          (const VariableType x[], VariableType r[]),
          //   EvaluateResiduals is a pointer to a function
          // of (x, r) which returns RERList.
     JacobianEvaluationReturnList (*EvaluateJacobian)
          (const VariableType x[], const VariableType r[],
          FullJacobian *J),
     VariableType *absoluteZeroTolerance,
     VariableType *relativeStepTolerance)
     {
     // Assume r has been evaluated for the given x.
     //
     // Allocate memory.
     J = new PLUJacobian(problemDim);
     xK = x;
     rCurrent = r;
     dxNR = new VariableType[problemDim];
     dx = new VariableType[problemDim];
     xKp1 = new VariableType[problemDim];
     xHold = new VariableType[problemDim];
     rHold = new VariableType[problemDim];
     vectorJTWr = new VariableType[problemDim];
     weightsK = new VariableType[problemDim];
     if (!J
          || !dxNR || !dx
          || !xKp1 || !xHold || !rHold
          || !vectorJTWr
          || !weightsK)
          FATAL(ERR_FREE_STORE, 1192304);
```

```
//
#if (RULE_FINDS_JAC_ROW_LENGTHS == 1)
jacRowLengths = new VariableType[problemDim];
if (!jacRowLengths)
     FATAL(ERR_FREE_STORE, 2192304);
#endif
//
// Initialize.
probDim = problemDim;
exitReason = SNLR_ITERATION_COUNT;
CheckAbsoluteZeroTolerance(absoluteZeroTolerance);
CheckRelativeStepTolerance(relativeStepTolerance);
relStepTol = *relativeStepTolerance;
//
#if (RULE_INITIALIZES_WEIGHTS_TO_ONE == 1)
VariableType *wStepP = weightsK;
for (int arrayCount=0; arrayCount<problemDim; arrayCount++)
     *wStepP++ = 1;
#endif
//
#if (RULE_USES_NEAR_ZERO == 1)
nearZero = *absoluteZeroTolerance;
#endif
#if (RULE_USES_RT_NEAR_ZERO == 1)
rtNearZero = sqrt(nearZero);
#endif
//
#if (RULE_INITIALIZES_TRUST_LEN_NEGATIVE == 1)
//   Initialize trust length to negative number to signify
// special processing when find weights at first iteration.
stepLenTrust = -1;
#endif
//
// Report if necessary.
if (utlReportLevel > RL_MUTE)
     {
     cout << "\nReporting from SolveWeightedDogleg():\nRule:\t"
          << WT_RULE
          << "\nZero tol:\t" << *absoluteZeroTolerance
          << "\tStep tol:\t" << *relativeStepTolerance
          << "\nInitial x:";
     ArrayStreamPrint(problemDim, xK);
     }
//
for (int loopCount=1; loopCount<=MAX_LOOPS; loopCount++)
     {
     //   Known: xK, rK (in rCurrent). Unknown:
     // J, grad, weightsK, wrSquareK.
     //   Vectors free for initializing calculations: dxNR,
     // dx, vectorJTWr, xKp1, xHold, rHold.
     //
     //   Initialize loop variables.
     EvaluateJacobian(xK, rCurrent, J);
     //
     #if (RULE_FINDS_JAC_ROW_LENGTHS == 1)
     // Weighting rule needs lengths of Jacobian rows.
     J->GetRowLengths(jacRowLengths);
     #endif
     //
     YesNoList foundWeightsBeforeNR = FindWeightsBeforeNR();
          //   Here, if returns YES, have weightsK, wrSquareK,
          // and Wr (in xHold). If returns NO, need to finish
          // finding weights at a later point.
     if (foundWeightsBeforeNR == YES)
          {
          //   Find vectors requiring Jacobian product, before
          // factor Jacobian. Recall have Wr in xHold.
```

```
            J->TProduct(xHold, vectorJTWr);
            J->Product(vectorJTWr, rHold);
                  // Here, rHold = JJTWr.
            }
      //
      //   Find Newton-Raphson step, the solution to linearized
      // residual equations, J(xK - xKp1) = J(dxNR) = rK. Will
      // have xKp1 = xK - dxNR.
      if (J->Solve(dxNR, rCurrent) == FAILURE)
            {
            if (utlReportLevel == RL_VERBOSE)
                  {
                  cout << "\n\nJacobian singular at x:";
                  ArrayStreamPrint(problemDim, xK);
                  }
            exitReason = SNLR_FACTORIZATION_ERROR;
            break;
            }
      stepLenNR = sqrt(ArraySumSquares(problemDim, dxNR));
      //
      // Set trust length for first iteration to NR length.
      if (loopCount == 1)
            stepLenTrust = stepLenNR;
      //
      firstDog = YES;
      trustRegionStatus = TRS_INHERITED;
      //
      // Report if necessary.
      if (utlReportLevel == RL_VERBOSE)
            {
            cout << "\n\nx: iteration:\t" << loopCount;
            ArrayStreamPrint(problemDim, xK);
            cout << "\nr: SS:\t"
                  << ArraySumSquares(problemDim, rCurrent);
            ArrayStreamPrint(problemDim, rCurrent);
            if (foundWeightsBeforeNR == YES)
                  {
                  cout << "\nweights: wtd SS:\t" << wrSquareK;
                  ArrayStreamPrint(problemDim, weightsK);
                  }
            cout << "\nstep lengths: trust:\t" << stepLenTrust
                  << "\tNR:\t" << stepLenNR;
            }
      //
      #if (RULE_FINDS_WEIGHTS_AFTER_NR == 1)
      //   Weights may not be established. If not, take a NR
      // step, adjusting trust region if necessary. After find
      // a successful step, finish finding the weights.
      // Evaluate the step in the NR direction, using special
      // rules, before go into general inner loop.
      if (foundWeightsBeforeNR == NO)
            {
            FindNRDirectionPoint(EvaluateResiduals);
            FindWeightsAfterNR();
            if (utlReportLevel == RL_VERBOSE)
                  {
                  cout << "\nweights: wtd SS:\t" << wrSquareK;
                  ArrayStreamPrint(problemDim, weightsK);
                  }
            wrSquareKp1 = WeightedSumSquares(rCurrent);
            if (CheckZeroVector(problemDim,
                  rCurrent, *absoluteZeroTolerance) == TC_TERMINATE)
                  trustRegionStatus = TRS_ZERO_RESIDUALS;
            else
                  {
                  //   Need dogleg curve before can evaluate NR
                  // step, since trust region update depends on
```

```
                // whether past cutback point.
                J->TProduct(xHold, vectorJTWr);
                J->Product(vectorJTWr, rHold);
                FindDoglegBreakPoints();
                trustRegionStatus = EvaluateNRDirectionStep();
                }
          }
     // End RULE_FINDS_WEIGHTS_AFTER_NR == 1.
     #endif
     //
     //   Inner loop: Find test point on double dogleg curve,
     // at step length = trust length. Evaluate residuals and
     // cost function there. Update the current trust region,
     // and repeat until find an acceptable point.
     while (trustRegionStatus >= TRS_INHERITED)
          {
          FindStep();
          // Take the step and evaluate residuals.
          ArrayDifference(problemDim, xK, dx, xKpl);
          if (EvaluateResiduals(xKpl, rCurrent) == RER_ERROR)
               {
               if (trustRegionStatus == TRS_INCREASED)
                    {
                    if (utlReportLevel == RL_VERBOSE)
                         cout << "\trejected (residual error)";
                    trustRegionStatus = TRS_ACCEPTED_AFTER_REJECT;
                    }
               else
                    {
                    if (utlReportLevel == RL_VERBOSE)
                         cout << "\thalved (residual error)";
                    stepLenTrust /= 2;
                    trustRegionStatus = TRS_REDUCED;
                    }
               } // end residual error processing
          else
               {
               // Here, have good residuals.
               wrSquareKpl = WeightedSumSquares(rCurrent);
               if (CheckZeroVector(problemDim,
                    rCurrent, *absoluteZeroTolerance) == TC_TERMINATE)
                    {
                    // Found a solution.
                    trustRegionStatus = TRS_ZERO_RESIDUALS;
                    }
               else
                    trustRegionStatus = EvaluateStep();
               } // end good residual processing
          } // end search for acceptable dx
     //
     //   Here, know an acceptable dx, or met a termination
     // criterion.
     if (trustRegionStatus == TRS_ACCEPTED)
          {
          // Update trust region and prepare for next iteration.
          if (deltaWRSquare > 0.1*deltaWRSquarePredict)
               {
               //   Recall deltas are negative, so if actual
               // reduction < 0.1*predicted reduction, i.e. if not
               // doing well, reduce trust region for next step.
               stepLenTrust /= 2;
               }
          else if (deltaWRSquare <= 0.75*deltaWRSquarePredict)
               {
               //   If actual reduction >= 0.75*predicted, doing
               // well. Increase trust region for next step.
               stepLenTrust *= 2;
```

```
                }
            VariableType *temp = xK; xK = xKpl; xKpl = temp;
            //wrSquareK = wrSquareKpl; // Not needed since wts change.
            }
      else if (trustRegionStatus == TRS_ACCEPTED_AFTER_REJECT)
            {
            //   Reject an increased trust region. Accept the
            // step which induced the increase, and take its
            // length as the trust length.
            stepLenTrust = stepLenTrustPrev;
            VariableType *temp = xK; xK = xHold; xHold = temp;
            temp = rCurrent; rCurrent = rHold; rHold = temp;
            //wrSquareK = wrSquareKplPrev; // Not needed.
            }
      else
            {
            //   Termination. Set exit reason and break out of
            // for-loop of MAX_LOOPS iterations.
            if (trustRegionStatus == TRS_ZERO_RESIDUALS)
                  exitReason = SNLR_ZERO_RESIDUAL_VECTOR;
            else
                  exitReason = SNLR_STAGNANT_ESTIMATED_SOLUTION;
            break;
            }
      } // end sequence of double dogleg steps
//
//   Copy current values to arrays passed as function
// arguments, if necessary.
if (xKpl != x)
      ArrayCopy(problemDim, xKpl, x);
if (rCurrent != r)
      ArrayCopy(problemDim, rCurrent, r);
//
// Report if necessary.
if (utlReportLevel > RL_MUTE)
      {
      cout << "\n\nSolveWeightedDogleg() terminating: "
            << SolveNonLinearReturnExplanation[exitReason]
            << "\nFinal x:";
      ArrayStreamPrint(problemDim, x);
      cout << "\nr: SS:\t" << ArraySumSquares(problemDim, r);
      ArrayStreamPrint(problemDim, r);
      }
//
//   Release memory allocated from free store. Since have
// been trading arrays around between pointers, make sure
// don't try to delete the arrays passed as arguments to
// this function.
delete J;
if (xK != x)
      delete [] xK;
if (rCurrent != r)
      delete [] rCurrent;
delete [] dxNR;
delete [] dx;
if (xKpl != x)
      delete [] xKpl;
if (xHold != x)
      delete [] xHold;
if (rHold != r)
      delete [] rHold;
delete [] vectorJTWr;
delete [] weightsK;
#if (RULE_FINDS_JAC_ROW_LENGTHS == 1)
delete jacRowLengths;
#endif
return exitReason;
```

```
     } // End SolveWeightedDogleg().

/* ==========< End Function SolveWeightedDogleg >======== */


/* ==========< Weighting functions >=======================
     Find weights according to the current weighting rule,
as set by #define WT_RULE above.
     Also store Wr in xHold and find wrSquareK.
     Return YES if weights completed. Return NO if still
need some further processing to find weights.
=============< Weighting functions >==================== */
#if (WT_RULE == 1)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 1 is rSquare.
     //   Rule 1 sets w_i = 1, already done in initialization.
     // Thus (Wr)_i = r_i. Also ith contribution to rTWr is (r_i)^2.
     VariableType *rStepP = rCurrent;
     VariableType *wrStepP = xHold;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          *wrStepP++ = rSubi;
          wrSquareK += rSubi*rSubi;
          }
     return YES;
     } // End FindWeightsBeforeNR() for WT_RULE == 1.
#endif

#if (WT_RULE == 2)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 2 is a one-weighted r-square.
     //   Rule 2 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
     // when w_i = 1/nearZero.
     //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr)_i =
     // sign(r_i). Also ith contribution to rTWr is |r_i|.
     //   If |r_i| < nearZero, w_i = 1/nearZero. Thus (Wr)_i =
     // r_i/nearZero. Also ith contribution to rTWr is
     // ((r_i)^2)/nearZero.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          VariableType absRSubi = fabs(rSubi);
          if (absRSubi >= nearZero)
               {
               *wStepP = 1/absRSubi;
               if (rSubi > 0) *wrStepP = 1;
               else *wrStepP = -1;
               wrSquareK += absRSubi;
               }
          else
               {
               *wStepP = 1/nearZero;
               VariableType wrSubi = rSubi/nearZero;
               *wrStepP = wrSubi;
               wrSquareK += wrSubi*rSubi;
               }
          wStepP++;
          wrStepP++;
          }
```

```
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 2.
#endif

#if (WT_RULE == 3)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 3 is a one-weighted r-square.
      //   Rule 3 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i = 0.
      //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr)_i =
      // sign(r_i). Also ith contribution to rTWr is |r_i|.
      //   If |r_i| < nearZero, w_i = 0. Thus (Wr)_i = 0.
      // Also ith contribution to rTWr is 0.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            if (absRSubi >= nearZero)
                  {
                  *wStepP = 1/absRSubi;
                  if (rSubi > 0) *wrStepP = 1;
                  else *wrStepP = -1;
                  wrSquareK += absRSubi;
                  }
            else
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 3.
#endif

#if (WT_RULE == 4)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 4 is a one-weighted r-square.
      //   Rule 4 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i is smallest weight assigned to elements greater
      // than nearZero.
      //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr)_i =
      // sign(r_i). Also ith contribution to rTWr is |r_i|.
      //   If |r_i| < nearZero, w_i = smallest. Thus (Wr)_i =
      // smallest*r_i. Also ith contribution to rTWr is
      // smallest*((r_i)^2).
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      VariableType smallWeight = 1/nearZero;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            if (absRSubi >= nearZero)
                  {
                  VariableType wSubi = 1/absRSubi;
                  *wStepP = wSubi;
                  if (rSubi > 0) *wrStepP = 1;
```

```
                else *wrStepP = -1;
                wrSquareK += absRSubi;
                if (wSubi < smallWeight)
                     smallWeight = wSubi;
                }
           else
                {
                *wStepP = 0;
                }
           wStepP++;
           wrStepP++;
           }
     rStepP = rCurrent;
     wStepP = weightsK;
     wrStepP = xHold;
     for (arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           if (*wStepP == 0)
                {
                *wStepP = smallWeight;
                *wrStepP = smallWeight*(*rStepP);
                wrSquareK += (*wrStepP)*(*rStepP);
                }
           rStepP++;
           wStepP++;
           wrStepP++;
           }
     return YES;
     } // End FindWeightsBeforeNR() for WT_RULE == 4.
#endif

#if (WT_RULE == 5)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 5 is a one-weighted r-square.
     //   Rule 5 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
     // when w_i depends on the result of a NR step.
     //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr_i) =
     // sign(r_i). Also ith contribution to rTWr is |r_i|.
     //   If |r_i| < nearZero, set w_i = 0, store r_i in rHold,
     // and return NO to indicate further processing required.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     wrSquareK = 0;
     YesNoList haveAllWeights = YES;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           VariableType rSubi = *rStepP++;
           VariableType absRSubi = fabs(rSubi);
           if (absRSubi >= nearZero)
                {
                *wStepP = 1/absRSubi;
                if (rSubi > 0) *wrStepP = 1;
                else *wrStepP = -1;
                wrSquareK += absRSubi;
                }
           else
                {
                *wStepP = 0;
                rHold[arrayCount] = rSubi;
                haveAllWeights = NO;
                }
           wStepP++;
           wrStepP++;
           }
     return haveAllWeights;
```

```
     } // End FindWeightsBeforeNR() for WT_RULE == 5.

static void FindWeightsAfterNR(void)
     {
     //   Rule 5, continued.
     //   Here, had some near-zero residuals at xK, marked by
     // a zero entry in weightsK. For these elements, use average
     // magnitude of residuals at xK, and at the xKpl found in
     // the NR direction. For this averaging, call rK_i = 0.
     //   Note rKpl in rCurrent and required elements of rK
     // in rHold.
     //   If |rKpl_i|/2 >= nearZero, w_i = 2/|rKpl_i|. Thus (WR)_i =
     // 2*rK_i/|rKpl_i|. Also ith contribution to rTWr is
     // 2*(rK_i^2)/|rKpl_i|.
     //   If |rKpl_i|/2 < nearZero, w_i = 0. Thus (WR)_i = 0.
     // Also ith contribution to rTWr is 0.
     VariableType *rKplStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          if (*wStepP == 0)
               {
               VariableType absRKplSubiOnTwo = fabs(*rKplStepP)/2;
               if (absRKplSubiOnTwo >= nearZero)
                    {
                    *wStepP = 1/absRKplSubiOnTwo;
                    VariableType rSubi = rHold[arrayCount];
                    *wrStepP = (*wStepP)*rSubi;
                    wrSquareK += (*wrStepP)*rSubi;
                    }
               else
                    {
                    *wStepP = 0;
                    *wrStepP = 0;
                    }
               } // end weight_i == 0
          rKplStepP++;
          wStepP++;
          wrStepP++;
          }
     return;
     } // End FindWeightsAfterNR() for WT_RULE == 5.
#endif

#if (WT_RULE == 6)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 6 is a one-weighted r-square.
     //   Rule 6 sets w_i using average residual from current
     // residuals rK and residuals rKpl in the NR direction.
     //   Because need NR direction before choose weights, just
     // store current residuals for later processing, and
     // return NO.
     ARRAY_COPY_INLINE(probDim, rCurrent, weightsK);
     return NO;
     } // End FindWeightsBeforeNR() for WT_RULE == 6.

static void FindWeightsAfterNR(void)
     {
     //   Rule 6, continued.
     //   Here, have rK stored in weightsK, and have good residuals
     // rKpl at trust length in NR direction, stored in rCurrent.
     //   Find rAveMag_i = (|rK_i| + |rKpl_i|)/2. Note rAveMag_i >= 0.
     //   If rAveMag_i >= nearZero, w_i = 1/rAveMag_i. Thus
     // (Wr)_i = rK_i/rAveMag_i. Also ith contribution to rTWr
     // is (rK_i^2)/rAveMag_i.
```

```
      //   If rAveMag_i < nearZero, w_i = 0. Thus (Wr)_i = 0.
      // Also ith contribution to rTWr is 0.
      VariableType *wStepP = weightsK;
      VariableType *rKplStepP = rCurrent;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           VariableType rSubi = *wStepP;
           VariableType rAveMagSubi =
                (fabs(rSubi) + fabs(*rKplStepP++)) / 2;
           if (rAveMagSubi >= nearZero)
                {
                *wStepP = 1/rAveMagSubi;
                *wrStepP = rSubi/rAveMagSubi;
                wrSquareK += (*wrStepP)*rSubi;
                }
           else
                {
                *wStepP = 0;
                *wrStepP = 0;
                }
           wStepP++;
           wrStepP++;
           }
      return;
      } // End FindWeightsAfterNR() for WT_RULE == 6.
#endif

#if (WT_RULE == 7)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 7 is geometric mean of rules 1 and 2.
      //   Rule 7 sets w_i = 1/sqrt(|r_i|), unless |r_i| < nearZero,
      // when w_i = 1/sqrt(nearZero).
      //   If |r_i| >= nearZero, w_i = 1/sqrt(|r_i|). Thus (Wr)_i =
      // sqrt(|r_i|)*sign(r_i). Also ith contribution to rTWr is
      // |r_i|*sqrt(|r_i|).
      //   If |r_i| < nearZero, w_i = 1/sqrt(nearZero). Thus (Wr)_i =
      // r_i/sqrt(nearZero). Also ith contribution to rTWr is
      // ((r_i)^2)/sqrt(nearZero).
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           VariableType rSubi = *rStepP++;
           VariableType absRSubi = fabs(rSubi);
           if (absRSubi >= nearZero)
                {
                VariableType rtAbsRSubi = sqrt(absRSubi);
                *wStepP = 1/rtAbsRSubi;
                if (rSubi > 0) *wrStepP = rtAbsRSubi;
                else *wrStepP = -rtAbsRSubi;
                wrSquareK += absRSubi*rtAbsRSubi;
                }
           else
                {
                *wStepP = 1/rtNearZero;
                VariableType wrSubi = rSubi/rtNearZero;
                *wrStepP = wrSubi;
                wrSquareK += wrSubi*rSubi;
                }
           wStepP++;
           wrStepP++;
           }
```

```
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 7.
#endif

#if (WT_RULE == 8)
static YesNoList FindWeightsBeforeNR(void)
      {
      //    Rule 8 is geometric mean of rules 1 and 3.
      //    Rule 8 sets w_i = 1/sqrt(|r_i|), unless |r_i| < nearZero,
      // when w_i = 0.
      //    If |r_i| >= nearZero, w_i = 1/sqrt(|r_i|). Thus (Wr)_i =
      // sqrt(|r_i|)*sign(r_i). Also ith contribution to rTWr is
      // |r_i|*sqrt(|r_i|).
      //    If |r_i| < nearZero, w_i = 0. Thus (Wr)_i = 0.
      // Also ith contribution to rTWr is 0.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            if (absRSubi >= nearZero)
                  {
                  VariableType rtAbsRSubi = sqrt(absRSubi);
                  *wStepP = 1/rtAbsRSubi;
                  if (rSubi > 0) *wrStepP = rtAbsRSubi;
                  else *wrStepP = -rtAbsRSubi;
                  wrSquareK += absRSubi*rtAbsRSubi;
                  }
            else
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 8.
#endif

#if (WT_RULE == 9)
static YesNoList FindWeightsBeforeNR(void)
      {
      //    Rule 9 is rSquare rule 1, adjusted for gradient length.
      //    Rule 9 sets w_i = 1/||del_r_i|| unless ||del_r_i|| = 0.
      // Thus (Wr)_i = r_i/||del_r_i||. Also ith contribution to
      // rTWr is (r_i)^2/||del_r_i||.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType jRLSubi = *jacRowLenP++;
            if (jRLSubi == 0)
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            else
                  {
                  *wStepP = 1/jRLSubi;
```

```
                *wrStepP = rSubi/jRLSubi;
                wrSquareK += rSubi*(*wrStepP);
                }
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 9.
#endif

#if (WT_RULE == 10)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 10 is one-weighted rule 2, adjusted for gradient length.
      //   If ||del_r_i|| = 0, rule 10 sets w_i = 0. Thus (Wr)_i = 0.
      // Also the ith contribution to rTWr is 0.
      //   Otherwise, rule 10 divides the entries of rule 2 by
      // ||del_r_i||.
      //   Rule 2 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i = 1/nearZero.
      //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr)_i =
      // sign(r_i). Also ith contribution to rTWr is |r_i|.
      //   If |r_i| < nearZero, w_i = 1/nearZero. Thus (Wr)_i =
      // r_i/nearZero. Also ith contribution to rTWr is
      // ((r_i)^2)/nearZero.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType jRLSubi = *jacRowLenP++;
            if (jRLSubi == 0)
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            else if (absRSubi >= nearZero)
                  {
                  *wStepP = 1/absRSubi/jRLSubi;
                  if (rSubi > 0) *wrStepP = 1/jRLSubi;
                  else *wrStepP = -1/jRLSubi;
                  wrSquareK += absRSubi/jRLSubi;
                  }
            else
                  {
                  *wStepP = 1/nearZero/jRLSubi;
                  VariableType wrSubi = rSubi/nearZero/jRLSubi;
                  *wrStepP = wrSubi;
                  wrSquareK += wrSubi*rSubi;
                  }
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 10.
#endif

#if (WT_RULE == 11)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 11 is one-weighted rule 3, adjusted for gradient length.
      //   If ||del_r_i|| = 0, rule 11 sets w_i = 0. Thus (Wr)_i = 0.
      // Also the ith contribution to rTWr is 0.
```

```
      //   Otherwise, rule 11 divides the entries of rule 3 by
      // ||del_r_i||.
      //   Rule 3 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i = 0.
      //   If |r_i| >= nearZero, w_i = 1/|r_i|. Thus (Wr)_i =
      // sign(r_i). Also ith contribution to rTWr is |r_i|.
      //   If |r_i| < nearZero, w_i = 0. Thus (Wr)_i = 0.
      // Also ith contribution to rTWr is 0.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType jRLSubi = *jacRowLenP++;
            if (jRLSubi == 0
                  || absRSubi < nearZero)
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            else
                  {
                  // Else absRSubi >= nearZero.
                  *wStepP = 1/absRSubi/jRLSubi;
                  if (rSubi > 0) *wrStepP = 1/jRLSubi;
                  else *wrStepP = -1/jRLSubi;
                  wrSquareK += absRSubi/jRLSubi;
                  }
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 11.
#endif

#if (WT_RULE == 12)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 12 mixes normalized gradient rSquare rule 9,
      // and one-norm weights.
      //   If ||del_r_i|| = 0, w_i = 0 and (Wr)_i = 0 and ith
      // contribution to rTWr is zero.
      //   If stepLenTrust > |r_i|/||del_r_i||, or if on first
      // iteration (denoted by negative trust length), then
      // w_i = 1/||del_r_i||. Thus (Wr)_i = r_i/||del_r_i||.
      // Also ith contribution to rTWr is (r_i)^2/||del_r_i||.
      //   If conditions above not met, then w_i = 1/|r_i|. Thus
      // (Wr)_i = sign(r_i). Also ith contribution to rTWr is |r_i|.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType jRLSubi = *jacRowLenP++;
            if (jRLSubi == 0)
                  {
                  *wStepP = 0;
                  *wrStepP = 0;
                  }
            else
```

```
                {
                VariableType rOnJRLSubi = rSubi/jRLSubi;
                if (fabs(rOnJRLSubi) < stepLenTrust
                      || stepLenTrust <= 0)
                      {
                      // Normalized gradient r-square weights.
                      *wStepP = 1/jRLSubi;
                      *wrStepP = rOnJRLSubi;
                      wrSquareK += rSubi*rOnJRLSubi;
                      }
                else
                      {
                      // One-norm weights.
                      VariableType absRSubi = fabs(rSubi);
                      *wStepP = 1/absRSubi;
                      if (rSubi > 0) *wrStepP = 1;
                      else *wrStepP = -1;
                      wrSquareK += absRSubi;
                      }
                }
          wStepP++;
          wrStepP++;
          }
     return YES;
     } // End FindWeightsBeforeNR() for WT_RULE == 12.
#endif

#if (WT_RULE == 13)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 13 mixes normalized gradient rSquare rule 9,
     // and normalized gradient one-norm weighted rule.
     //   If ||del_r_i|| = 0, w_i = 0 and (Wr)_i = 0 and ith
     // contribution to rTWr is zero.
     //   If stepLenTrust > |r_i|/||del_r_i||, or if on first
     // iteration (denoted by negative trust length), then
     // w_i = 1/||del_r_i||. Thus (Wr)_i = r_i/||del_r_i||.
     // Also ith contribution to rTWr is (r_i)^2/||del_r_i||.
     //   If conditions above not met, then w_i = 1/|r_i|/||del_r_i||.
     // Thus (Wr)_i = sign(r_i)/||del_r_i||. Also ith contribution
     // to rTWr is |r_i|/||del_r_i||.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     VariableType *jacRowLenP = jacRowLengths;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          VariableType jRLSubi = *jacRowLenP++;
          if (jRLSubi == 0)
                {
                *wStepP = 0;
                *wrStepP = 0;
                }
          else
                {
                VariableType rOnJRLSubi = rSubi/jRLSubi;
                VariableType absROnJRLSubi = fabs(rOnJRLSubi);
                if (absROnJRLSubi < stepLenTrust
                      || stepLenTrust <= 0)
                      {
                      // Normalized gradient r-square weights.
                      *wStepP = 1/jRLSubi;
                      *wrStepP = rOnJRLSubi;
                      wrSquareK += rSubi*rOnJRLSubi;
                      }
```

```
                else
                     {
                     // Normalized gradient one-norm weights.
                     *wStepP = 1/fabs(rSubi)/jRLSubi;
                     if (rSubi > 0) *wrStepP = 1/jRLSubi;
                     else *wrStepP = -1/jRLSubi;
                     wrSquareK += absROnJRLSubi;
                     }
                }
           wStepP++;
           wrStepP++;
           }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 13.
#endif

#if (WT_RULE == 14)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 14 mixes rSquare rule 1, and the one-norm weights.
      //   If ||del_r_i|| = 0, w_i = 0 and (Wr)_i = 0 and ith
      // contribution to rTWr is zero.
      //   If stepLenTrust > |r_i|/||del_r_i||, or if on first
      // iteration (denoted by negative trust length), then
      // w_i = 1. Thus (Wr)_i = r_i. Also ith contribution to rTWr
      // is (r_i)^2.
      //   If conditions above not met, then w_i = 1/|r_i|. Thus
      // (Wr)_i = sign(r_i). Also ith contribution to rTWr is |r_i|.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           VariableType rSubi = *rStepP++;
           VariableType jRLSubi = *jacRowLenP++;
           if (jRLSubi == 0)
                {
                *wStepP = 0;
                *wrStepP = 0;
                }
           else
                {
                if (fabs(rSubi/jRLSubi) < stepLenTrust
                     || stepLenTrust <= 0)
                     {
                     // R-square weights.
                     *wStepP = 1;
                     *wrStepP = rSubi;
                     wrSquareK += rSubi*rSubi;
                     }
                else
                     {
                     // One-norm weights.
                     VariableType absRSubi = fabs(rSubi);
                     *wStepP = 1/absRSubi;
                     if (rSubi > 0) *wrStepP = 1;
                     else *wrStepP = -1;
                     wrSquareK += absRSubi;
                     }
                }
           wStepP++;
           wrStepP++;
           }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 14.
```

```
#endif

#if (WT_RULE == 15)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 15 mixes normalized gradient rSquare rule 9,
     // and one-norm weights.
     //   If ||del_r_i|| = 0, w_i = 0 and (Wr)_i = 0 and ith
     // contribution to rTWr is zero.
     //   If stepLenTrust > |r_i|/||del_r_i||, or if on first
     // iteration (denoted by negative trust length), then
     // w_i = 1/||del_r_i||. Thus (Wr)_i = r_i/||del_r_i||.
     // Also ith contribution to rTWr is (r_i)^2/||del_r_i||.
     //   If conditions above not met, then let
     // f_i = exp(1-|r_i|/stepLenTrust/||del_r_i||). Then
     // w_i = f_i/||del_r_i|| + (1-f_i)/|r_i|. (Wr)_i and ith
     // contribution to rTWr follow directly from definitions.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     VariableType *jacRowLenP = jacRowLengths;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          VariableType jRLSubi = *jacRowLenP++;
          if (jRLSubi == 0)
               {
               *wStepP = 0;
               *wrStepP = 0;
               }
          else
               {
               VariableType rOnJRLSubi = rSubi/jRLSubi;
               VariableType fSubi = fabs(rOnJRLSubi)/stepLenTrust;
                    // Note if stepLenTrust negative, fSubi negative.
               if (fSubi < 1)
                    {
                    // Normalized gradient r-square weights.
                    *wStepP = 1/jRLSubi;
                    *wrStepP = rOnJRLSubi;
                    wrSquareK += rSubi*rOnJRLSubi;
                    }
               else
                    {
                    // Mix normalized r-square and one-norm weights.
                    fSubi = exp(1 - fSubi);
                    VariableType wSubi =
                         fSubi/jRLSubi + (1-fSubi)/fabs(rSubi);
                    *wStepP = wSubi;
                    *wrStepP = wSubi*rSubi;
                    wrSquareK += (*wrStepP)*rSubi;
                    }
               }
          wStepP++;
          wrStepP++;
          }
     return YES;
     } // End FindWeightsBeforeNR() for WT_RULE == 15.
#endif

#if (WT_RULE == 16)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 16 mixes normalized gradient rSquare rule 9,
     // and scaled one-norm weights.
     //   If ||del_r_i|| = 0, w_i = 0 and (Wr)_i = 0 and ith
```

```
      // contribution to rTWr is zero.
      //   If stepLenTrust > |r_i|/||del_r_i||, or if on first
      // iteration (denoted by negative trust length), then
      // w_i = 1/||del_r_i||. Thus (Wr)_i = r_i/||del_r_i||.
      // Also ith contribution to rTWr is (r_i)^2/||del_r_i||.
      //   If conditions above not met, then w_i = stepLenTrust/|r_i|.
      // Thus (Wr)_i = stepLenTrust*sign(r_i). Also ith contribution
      // to rTWr is stepLenTrust*|r_i|.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
           VariableType rSubi = *rStepP++;
           VariableType jRLSubi = *jacRowLenP++;
           if (jRLSubi == 0)
                {
                *wStepP = 0;
                *wrStepP = 0;
                }
           else
                {
                VariableType rOnJRLSubi = rSubi/jRLSubi;
                if (fabs(rOnJRLSubi) < stepLenTrust
                     || stepLenTrust <= 0)
                     {
                     // Normalized gradient r-square weights.
                     *wStepP = 1/jRLSubi;
                     *wrStepP = rOnJRLSubi;
                     wrSquareK += rSubi*rOnJRLSubi;
                     }
                else
                     {
                     // Scaled one-norm weights.
                     VariableType absRSubi = fabs(rSubi);
                     *wStepP = stepLenTrust/absRSubi;
                     if (rSubi > 0) *wrStepP = stepLenTrust;
                     else *wrStepP = -stepLenTrust;
                     wrSquareK += stepLenTrust*absRSubi;
                     }
                }
           wStepP++;
           wrStepP++;
           }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 16.
#endif

#if (WT_RULE == 17)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 17 is an arithmetic averaged one-weighted r-square.
      //   On first iteration, denoted by negative trust length,
      // rule 17 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i = 0.
      //   On subsequent iterations, rule 17 averages the former
      // w_i with 1/|r_i|, unless |r_i| < nearZero, when just
      // takes half of former w_i.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
           {
```

```
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType wSubi;
            if (absRSubi >= nearZero)
                  {
                  wSubi = 1/absRSubi;
                  if (stepLenTrust > 0)
                        wSubi = (wSubi + *wStepP)/2;
                  }
            else if (stepLenTrust <= 0)
                  wSubi = 0;
            else
                  wSubi = (*wStepP)/2;
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 17.
#endif

#if (WT_RULE == 18)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 18 is a geometric averaged one-weighted r-square.
      //   On first iteration, denoted by negative trust length,
      // rule 18 sets w_i = 1/|r_i|, unless |r_i| < nearZero,
      // when w_i = 1.
      //   On subsequent iterations, rule 18 takes geometric mean
      // of the former w_i and 1/|r_i|, unless |r_i| < nearZero,
      // when just takes square root of former w_i.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType wSubi;
            if (absRSubi >= nearZero)
                  {
                  wSubi = 1/absRSubi;
                  if (stepLenTrust > 0)
                        wSubi = sqrt(wSubi*(*wStepP));
                  }
            else if (stepLenTrust <= 0)
                  wSubi = 1;
            else
                  wSubi = sqrt(*wStepP);
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 18.
#endif

#if (WT_RULE == 19)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 19 is an arithmetic averaged, normalized gradient,
```

```
      // r-square.
      //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
      //   On first iteration, denoted by negative trust length,
      // rule 20 sets w_i = 1/||del_r_i||.
      //   On subsequent iterations, rule 20 takes geometric mean
      // of the former w_i and 1/||del_r_i||.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType jRLSubi = *jacRowLenP++;
            VariableType wSubi;
            if (jRLSubi == 0)
                  wSubi = 0;
            else if (stepLenTrust <= 0)
                  wSubi = 1/jRLSubi;
            else
                  wSubi = (*wStepP + 1/jRLSubi)/2;
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 19.
#endif

#if (WT_RULE == 20)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 20 is a geometric averaged, normalized gradient,
      // r-square.
      //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
      //   On first iteration, denoted by negative trust length,
      // rule 19 sets w_i = 1/||del_r_i||.
      //   On subsequent iterations, rule 19 averages the former
      // w_i with 1/||del_r_i||.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType jRLSubi = *jacRowLenP++;
            VariableType wSubi;
            if (jRLSubi == 0)
                  wSubi = 0;
            else if (stepLenTrust <= 0)
                  wSubi = 1/jRLSubi;
            else
                  wSubi = sqrt(*wStepP/jRLSubi);
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
```

```
     } // End FindWeightsBeforeNR() for WT_RULE == 20.
#endif

#if (WT_RULE == 21)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 21 is an arithmetic averaged mix rule, combining
     // normalized gradient r-square and one-norm weighted r-square.
     //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
     //   On first iteration, denoted by negative trust length,
     // rule 21 sets w_i = 1/||del_r_i||.
     //   On subsequent iterations, rule 21 takes arithmetic mean
     // of the former w_i and mix rule w_i.
     //   If stepLenTrust > |r_i|/||del_r_i||, then mix rule
     // w_i = 1/||del_r_i||. Otherwise mix rule w_i = 1/|r_i|.
     //   (Wr)_i and ith contribution to rTWr follow by definition.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     VariableType *jacRowLenP = jacRowLengths;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          VariableType absRSubi = fabs(rSubi);
          VariableType jRLSubi = *jacRowLenP++;
          VariableType wSubi;
          if (jRLSubi == 0)
               wSubi = 0;
          else if (stepLenTrust <= 0)
               wSubi = 1/jRLSubi;
          else if (stepLenTrust > absRSubi/jRLSubi)
               wSubi = (*wStepP + 1/jRLSubi)/2;
          else
               wSubi = (*wStepP + 1/absRSubi)/2;
          *wStepP = wSubi;
          *wrStepP = wSubi*rSubi;
          wrSquareK += (*wrStepP)*rSubi;
          wStepP++;
          wrStepP++;
          }
     return YES;
     } // End FindWeightsBeforeNR() for WT_RULE == 21.
#endif

#if (WT_RULE == 22)
static YesNoList FindWeightsBeforeNR(void)
     {
     //   Rule 22 is a geometric averaged mix rule, combining
     // normalized gradient r-square and one-norm weighted r-square.
     //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
     //   On first iteration, denoted by negative trust length,
     // rule 22 sets w_i = 1/||del_r_i||.
     //   On subsequent iterations, rule 22 takes geometric mean
     // of the former w_i and mix rule w_i.
     //   If stepLenTrust > |r_i|/||del_r_i||, then mix rule
     // w_i = 1/||del_r_i||. Otherwise mix rule w_i = 1/|r_i|.
     //   (Wr)_i and ith contribution to rTWr follow by definition.
     VariableType *rStepP = rCurrent;
     VariableType *wStepP = weightsK;
     VariableType *wrStepP = xHold;
     VariableType *jacRowLenP = jacRowLengths;
     wrSquareK = 0;
     for (int arrayCount=0; arrayCount<probDim; arrayCount++)
          {
          VariableType rSubi = *rStepP++;
          VariableType absRSubi = fabs(rSubi);
```

```
            VariableType jRLSubi = *jacRowLenP++;
            VariableType wSubi;
            if (jRLSubi == 0)
                  wSubi = 0;
            else if (stepLenTrust <= 0)
                  wSubi = 1/jRLSubi;
            else if (stepLenTrust > absRSubi/jRLSubi)
                  wSubi = sqrt(*wStepP/jRLSubi);
            else
                  wSubi = sqrt(*wStepP/absRSubi);
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 22.
#endif

#if (WT_RULE == 23)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 23 is an arithmetic averaged mix rule, combining
      // normalized gradient r-square and one-norm weighted r-square.
      //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
      //   On first iteration, denoted by negative trust length,
      // rule 23 sets w_i = 1/||del_r_i||.
      //   On subsequent iterations, rule 23 takes arithmetic mean
      // of the former w_i and mix rule w_i.
      //   If stepLenTrust > 2*|r_i|/||del_r_i||, then mix rule
      // w_i = 1/||del_r_i||. Otherwise mix rule w_i = 1/|r_i|.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType jRLSubi = *jacRowLenP++;
            VariableType wSubi;
            if (jRLSubi == 0)
                  wSubi = 0;
            else if (stepLenTrust <= 0)
                  wSubi = 1/jRLSubi;
            else if (stepLenTrust > 2*absRSubi/jRLSubi)
                  wSubi = (*wStepP + 1/jRLSubi)/2;
            else
                  wSubi = (*wStepP + 1/absRSubi)/2;
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 23.
#endif

#if (WT_RULE == 24)
static YesNoList FindWeightsBeforeNR(void)
      {
      //   Rule 24 is a geometric averaged mix rule, combining
      // normalized gradient r-square and one-norm weighted r-square.
```

```
      //   At every iteration, if ||del_r_i|| = 0, then w_i = 0.
      //   On first iteration, denoted by negative trust length,
      // rule 24 sets w_i = 1/||del_r_i||.
      //   On subsequent iterations, rule 24 takes geometric mean
      // of the former w_i and mix rule w_i.
      //   If stepLenTrust > 2*|r_i|/||del_r_i||, then mix rule
      // w_i = 1/||del_r_i||. Otherwise mix rule w_i = 1/|r_i|.
      //   (Wr)_i and ith contribution to rTWr follow by definition.
      VariableType *rStepP = rCurrent;
      VariableType *wStepP = weightsK;
      VariableType *wrStepP = xHold;
      VariableType *jacRowLenP = jacRowLengths;
      wrSquareK = 0;
      for (int arrayCount=0; arrayCount<probDim; arrayCount++)
            {
            VariableType rSubi = *rStepP++;
            VariableType absRSubi = fabs(rSubi);
            VariableType jRLSubi = *jacRowLenP++;
            VariableType wSubi;
            if (jRLSubi == 0)
                  wSubi = 0;
            else if (stepLenTrust <= 0)
                  wSubi = 1/jRLSubi;
            else if (stepLenTrust > 2*absRSubi/jRLSubi)
                  wSubi = sqrt(*wStepP/jRLSubi);
            else
                  wSubi = sqrt(*wStepP/absRSubi);
            *wStepP = wSubi;
            *wrStepP = wSubi*rSubi;
            wrSquareK += (*wrStepP)*rSubi;
            wStepP++;
            wrStepP++;
            }
      return YES;
      } // End FindWeightsBeforeNR() for WT_RULE == 24.
#endif

/* ==========< End weighting functions >================ */


/* ==========< Helper functions >==========================
============< Helper functions >======================== */
static double WeightedSumSquares(const VariableType x[])
      {
      //   Find xTWx where diagonal weighting matrix W
      // stored in array w.
      VariableType temp = *x;
      VariableType *wStepP = weightsK;
      double weightedSumSquares = temp*temp*(*wStepP);
      for (int arrayCount=1; arrayCount<probDim; arrayCount++)
            {
            temp = *++x;
            weightedSumSquares += temp*temp*(*++wStepP);
            }
      return weightedSumSquares;
      } // End WeightedSumSquares().

static void FindStep(void)
      {
      //   Find test point on double dogleg curve, at step
      // length = trust length. Also find gwTdx and predicted
      // change in cost function for this step.
      //
      if (stepLenTrust >= stepLenNR)
            {
            // Take the full Newton-Raphson step.
            stepLenTrust = stepLenNR;
```

```
        ArrayCopy(probDim, dxNR, dx);
        gwTdx = -2*wrSquareK;
        deltaWRSquarePredict = -wrSquareK;
        }
    else
        {
        // Take a step smaller than the full NR step.
        if (firstDog == YES)
            FindDoglegBreakPoints();
        // Here, have dogleg points and need dx.
        double rho1, rho2;
            // Scales for dx = rho1*dxCauchy + rho2*dxNR.
        if (stepLenTrust >= stepLenCutback)
            {
            // Take partial step in NR direction.
            rho1 = 0;
            rho2 = stepLenTrust/stepLenNR;
            }
        else if (stepLenTrust <= stepLenCauchy)
            {
            // Take partial step in SD direction.
            rho1 = stepLenTrust/stepLenCauchy;
            rho2 = 0;
            }
        else
            {
            // Find convex combination of dxCauchy and dxNR.
            double b = 2*eta*wrSquareK/lenFrthJTWrOnLenSqRtWJJTr
                - 2;
            double c = stepLenCauchy*stepLenCauchy;
            double a = stepLenCutback*stepLenCutback/c
                - b - 1;
            c = 1 - stepLenTrust*stepLenTrust/c;
            double lambda = (sqrt(b*b-4*a*c) - b)/2/a;
            rho1 = 1 - lambda;
            rho2 = lambda*eta;
            }
        //
        // Here, have rho1 and rho2.
        if (rho1 == 0)
            {
            // Entirely in NR direction.
            ArrayScalarProduct(probDim, rho2, dxNR, dx);
            gwTdx = -2*rho2*wrSquareK;
            deltaWRSquarePredict = gwTdx + rho2*rho2*wrSquareK;
            }
        else
            {
            // A component in SD direction...
            ArrayScalarProduct(probDim,
                rho1*scaleJTWrToCauchy, vectorJTWr, dx);
            gwTdx = -2*rho1*lenFrthJTWrOnLenSqRtWJJTr;
            deltaWRSquarePredict = rho1*(rho1 + 2*rho2)*
                lenFrthJTWrOnLenSqRtWJJTr;
            if (rho2 != 0)
                {
                // ...and a component in NR direction.
                ArraySaxpy(probDim, rho2, dxNR, dx, dx);
                gwTdx -= 2*rho2*wrSquareK;
                deltaWRSquarePredict += (rho2*rho2*wrSquareK);
                }
            deltaWRSquarePredict += gwTdx;
            }
        } // end finding step with lenTrust < lenNR
    //
    // Here, have step dx with length stepLenTrust.
    if (utlReportLevel == RL_VERBOSE)
```

```
          cout << "\nstep:\t" << stepLenTrust;
     return;
     } // End FindStep().

static TrustRegionStatusList EvaluateStep(void)
     {
     //   Update the trust region, to seek a different dx in
     // current iteration, or to use during next iteration.
     //
     // Check if increased trust region failed.
     if (trustRegionStatus == TRS_INCREASED
          && wrSquareKp1 > wrSquareKp1Prev)
          {
          //   Trust region was increased into a worse cost function.
          // Restore previous values and accept.
          //   Note that previous values of actual and estimated
          // change in cost function are still valid.
          if (utlReportLevel == RL_VERBOSE)
               cout << "\trejected (worse cost)";
          return TRS_ACCEPTED_AFTER_REJECT;
          }
     //
     //   Prepare to evaluate step's progress.
     //   Both gwTdx ("initial slope") and deltaWRSquare are negative
     // for smaller rSquare. Good progress made when delta is more
     // negative than gwTdx, or when delta < gwTdx.
     //   Acceptable progress when delta <= ALPHA*gwTdx.
     //   gwTdx and deltaWRSquarePredict calculated earlier.
     deltaWRSquare = wrSquareKp1 - wrSquareK;
     //
     // Check that cost function making progress.
     if (deltaWRSquare > ALPHA*gwTdx)
          {
          // Actual rSquare too large.
          if (CheckStagnatedVector(probDim, dx, xKp1, relStepTol)
               == TC_TERMINATE)
               {
               return TRS_STAGNANT_SOLUTION;
               }
          else if (trustRegionStatus == TRS_INCREASED)
               {
               //   Last trial step was good enough that increased
               // trust length. If here, the increased length had
               // better cost than previously, so don't reject.
               return TRS_ACCEPTED;
               }
          else
               {
               // Decrease trust length.
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\tcut (poor cost)";
               double lambda = gwTdx/2/(gwTdx - deltaWRSquare);
               if (lambda < 0.1) lambda = 0.1;
               else if (lambda > 0.5) lambda = 0.5;
               stepLenTrust *= lambda;
               return TRS_REDUCED;
               }
          } // end check that making sufficient progess
     //
     //   Cost function sufficiently small. May increase or accept
     // it for current iteration, and may also change it for next
     // iteration, all depending on success of model at predicting
     // values.
     //   Make decision about current, acceptable, trust region.
     if (trustRegionStatus == TRS_REDUCED
          || stepLenTrust == stepLenNR)
          {
```

```
            //   If just reduced, or if using NR step, then don't
            // consider increasing trust region during this iteration.
            return TRS_ACCEPTED;
            }
      else if (fabs(deltaWRSquarePredict - deltaWRSquare)
            <= -0.1*deltaWRSquare
            ||
            deltaWRSquare <= gwTdx)
            {
            //   Increase trust region if prediction was very good,
            // or if actual decrease very large. Store current values
            // so can recover later if needed.
            if (utlReportLevel == RL_VERBOSE)
                  cout << "\tdoubled (good cost)";
            stepLenTrustPrev = stepLenTrust;
            stepLenTrust *= 2;
            VariableType *temp = xHold; xHold = xKpl; xKpl = temp;
            temp = rHold; rHold = rCurrent; rCurrent = temp;
            wrSquareKplPrev = wrSquareKpl;
            return TRS_INCREASED;
            }
      // Here, no reason to increase the acceptable trust region.
      return TRS_ACCEPTED;
      } // End EvaluateStep().

static void FindDoglegBreakPoints(void)
      {
      //   Calculate points on the double dogleg curve.
      //   Normal double dogleg code defers these calculations
      // until sure they're needed, but for some weighted
      // dogleg trials, would need two more vectors of storage
      // to carry necessary information forward. Therefore made
      // into a function, called at discretion of code.
      //
      firstDog = NO;
      lenSqJTWr = ArraySumSquares(probDim, vectorJTWr);
      //   Find Cauchy point, the minimizer of rSquare in
      // steepest descent direction if residuals were linear.
      scaleJTWrToCauchy = lenSqJTWr/WeightedSumSquares(rHold);
            // or lenSqJTWr/(rHoldT*W*rHold)
            // or lenSqJTWr/((JJTWR)T*W*JJTWr)
            // or lenSqJTWr/lenSqRtWJJTWr
      stepLenCauchy = scaleJTWrToCauchy*sqrt(lenSqJTWr);
      //   Find cutback point, where double dogleg
      // search path rejoins NR direction.
      lenFrthJTWrOnLenSqRtWJJTr = scaleJTWrToCauchy*lenSqJTWr;
      eta = 0.2 + 0.8*lenFrthJTWrOnLenSqRtWJJTr/wrSquareK;
      stepLenCutback = eta*stepLenNR;
      //
      // Report if necessary.
      if (utlReportLevel == RL_VERBOSE)
            {
            cout << "\nstep lengths: Cauchy:\t" << stepLenCauchy
                  << "\tCutback:\t" << stepLenCutback;
            }
      return;
      } // End FindDoglegBreakPoints().

#if (RULE_FINDS_WEIGHTS_AFTER_NR == 1)
static void FindNRDirectionPoint
      (ResidualEvaluationReturnList (*EvalResids)
            (const VariableType x[], VariableType r[]))
      {
      //   Find the point at the proper trust length, in the
      // NR direction, which does not induce a residual error.
      // Halve trust length as necessary.
      if (stepLenTrust > stepLenNR)
```

```
          stepLenTrust = stepLenNR;
     while (1)
          {
          ArrayScalarProduct(probDim,
               stepLenTrust/stepLenNR, dxNR, dx);
          if (utlReportLevel == RL_VERBOSE)
               cout << "\nNR direction step:\t" << stepLenTrust;
          ArrayDifference(probDim, xK, dx, xKpl);
          if (EvalResids(xKpl, rCurrent) == RER_ERROR)
               {
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\thalved (residual error)";
               stepLenTrust /= 2;
               trustRegionStatus = TRS_REDUCED;
               }
          else
               break;
          }
     return;
     } // End FindNRDirectionPoint().

static TrustRegionStatusList EvaluateNRDirectionStep(void)
     {
     //   Evaluate a good step in the Newton-Raphson direction.
     //   This function called when finding weights required
     // a good NR step. Therefore all that has been done is
     // to reduce trust length if necessary until didn't find
     // residual evaluation error.
     //   Follow same rules as in function EvaluateStep, except:
     // (1) Need to calculate gwTdx and deltaWRSquarePredict.
     // (2) Trust length is either inherited or reduced; i.e.
     // trust length was not increased to find this step. So
     // ignore possibility that trust length increased.
     // (3) If step evaluates poorly due to insufficient
     // decrease, and trust length is less than cutback length,
     // then don't reduce trust length, just allow same
     // trust length but along cutback or steepest descent path.
     //
     //   Re-report step, due to intervening Cauchy and cutback
     // points reported before evaluate NR direction step.
     if (utlReportLevel == RL_VERBOSE)
          cout << "\nNR direction step:\t" << stepLenTrust;
     //
     double rho2 = stepLenTrust/stepLenNR;
     gwTdx = -2*rho2*wrSquareK;
     deltaWRSquarePredict = gwTdx + rho2*rho2*wrSquareK;
     //
     deltaWRSquare = wrSquareKpl - wrSquareK;
     //
     // Check that cost function making progress.
     if (deltaWRSquare > ALPHA*gwTdx)
          {
          // Actual rSquare too large.
          if (CheckStagnatedVector(probDim, dx, xKpl, relStepTol)
               == TC_TERMINATE)
               {
               return TRS_STAGNANT_SOLUTION;
               }
          else if (stepLenTrust < stepLenCutback)
               {
               //   Force next trial step at same trust length
               // but on double dogleg path.
               if (utlReportLevel == RL_VERBOSE)
                    cout << "\tmoved to dogleg path (poor cost)";
               return TRS_INHERITED;
               }
          else
```

```
                {
                // Decrease trust length.
                if (utlReportLevel == RL_VERBOSE)
                     cout << "\tcut (poor cost)";
                double lambda = gwTdx/2/(gwTdx - deltaWRSquare);
                if (lambda < 0.1) lambda = 0.1;
                else if (lambda > 0.5) lambda = 0.5;
                stepLenTrust *= lambda;
                return TRS_REDUCED;
                }
          } // end check that making sufficient progess
     //
     //   Cost function sufficiently small. Make decision about
     // current, acceptable, trust region.
     if (trustRegionStatus == TRS_REDUCED
          || stepLenTrust == stepLenNR)
          {
          //   If just reduced, or if using NR step, then don't
          // consider increasing trust region during this iteration.
          return TRS_ACCEPTED;
          }
     else if (fabs(deltaWRSquarePredict - deltaWRSquare)
          <= -0.1*deltaWRSquare
          ||
          deltaWRSquare <= gwTdx)
          {
          //   Increase trust region if prediction was very good,
          // or if actual decrease very large. Store current values
          // so can recover later if needed.
          if (utlReportLevel == RL_VERBOSE)
               cout << "\tdoubled (good cost)";
          stepLenTrustPrev = stepLenTrust;
          stepLenTrust *= 2;
          VariableType *temp = xHold; xHold = xKpl; xKpl = temp;
          temp = rHold; rHold = rCurrent; rCurrent = temp;
          wrSquareKplPrev = wrSquareKpl;
          return TRS_INCREASED;
          }
     // Here, no reason to increase the acceptable trust region.
     return TRS_ACCEPTED;
     } // End EvaluateNRDirectionStep().

#endif

/* ==========< End helper functions >=================== */


/* ====================< End File WTDDOGLG.CPP >================== */
```

ACKNOWLEDGMENTS

This thesis work was funded in part by a grant from the Toda Corporation, administered through the MIT Department of Architecture and the Building Technology group.

Professor Les Norford, my thesis advisor, contributed substantially with his critical approach and willingness to help me talk out new ideas. Over the years, Les has stuck his neck out for me, perhaps more times than he cares to remember.

Professors Leon Glicksman and Qingyan Chen, of my thesis committee, pushed me through the sometimes-painful process of establishing and clarifying the main arguments of the thesis.

Axel Bring and Per Sahlin, at KTH, the Swedish Royal Institute of Technology, fired my interest in equation solving and component-based modeling. In addition they are excellent hosts.

My family stood behind me during those critical times when I seemed never likely to finish. Mom and Dad in particular never failed in their encouragement, despite probably wondering what exactly I was doing all these years.

Grandpa Lorenzetti has always been a great source of comfort, wisdom, and enjoyment.

My thanks to Cherí Long for her forgiveness and friendship.

Cassandra Morabito made a lot tolerable, or at least understandable.

Jim Axley started me in the right direction; I hope he approves of where I ended up.

Dorrit Schuchter helped me clear the practical hurdles.

Arlene and Charlie Marge, and Len and Suzanne Morse-Fortier, opened their homes and shared with me the realities of daily life.

Rhonda Ferrino, Anne Marie Michel, and Virginia Brambilla, kindred dog-people, and Shelley McGraw, Tia Tilson, and Erica Kates, gave MIT a human face.

The doctors and staff of Fresh Pond Animal Hospital kept my good friend and companion, Bahati (G), healthy and active through it all.

The numeric work was done using MATLAB® and Symantec C++ for Macintosh. All figures were produced using MATLAB.