MO

Mathematical Optimization

Keywords and Notation Key

Keywords

(1) list of prerequisites: vector calculus, basic numerical linear algebra

(2) list of computational methods: line search, steepest descent, conjugate gradient, discrete Newton, quasi Newton, truncated Newton, simulated annealing, genetic algorithms

(3) list of architectures/computers: general machines

Notation Key

$\mathbf{x}, \mathbf{y}, \mathbf{p}, \mathbf{r}, \mathbf{b}, \mathbf{d}, \mathbf{s}, \mathbf{u}, \mathbf{v}$	a column vector, dimension n
$\mathbf{A}, \mathbf{H}, \mathbf{B}, \mathbf{M}, \mathbf{U}$	a matrix
$c, \alpha, \beta, \lambda, \eta, \gamma$	a scalar
^{T}	a transpose (vector or matrix) operator
$\Vert \bullet \Vert$	vector norm
$f(\mathbf{x})$	objective function
$\mathbf{g}(\mathbf{x})$	gradient vector
$\mathbf{H}(\mathbf{x})$	Hessian matrix, dimension $n \times n$
$q_A(\mathbf{x})$	convex quadratic function

1 Introduction

'What's new?' is an interesting and broadening eternal question, but one which, if pursued exclusively, results only in an endless parade of trivia and fashion, the silt of tomorrow. I would like, instead, to be concerned with the question 'What is best?', a question which cuts deeply rather than broadly, a question whose answers tend to move the silt downstream.

Robert M. Pirsig
"Zen and the Art of Motorcycle Maintenance" (1974)

Mathematical optimization is the formal title given to the branch of computational science that seeks to answer the question 'What is best?' for problems in which the quality of any answer can be expressed as a numerical value. Such problems arise in all areas of mathematics, the physical, chemical and biological sciences, engineering, architecture, economics, and management, and the range of techniques available to solve them is nearly as wide.

The purpose of this chapter is not to make the reader an expert on all aspects of mathematical optimization but to provide a broad overview of the field. The beginning sections introduce the terminology of optimization and the ways in which problems and their solutions are formulated and classified. Subsequent sections consider the most appropriate methods for several classes of optimization problems, with emphasis placed on powerful, versatile algorithms well suited to optimizing functions of many variables on high performance computational platforms. High-performance computational issues, such as vectorization and parallelization of optimization codes, are beyond the scope of this chapter. This field is still in its infancy at this time, with general strategies adopted from numerical linear algebra codes (see chapter on Linear Algebra). However, the last section contains a brief overview of possible approaches.

1.1 Definitions

The goal of an optimization problem can be formulated as follows: find the combination of parameters (independent variables) which optimize a given quantity, possibly subject to some restrictions on the allowed parameter ranges. The quantity to be optimized (maximized or minimized) is termed the *objective function*; the parameters which may be changed in the quest for the optimum are called control or *decision variables*; the restrictions on allowed parameter values are known as *constraints*.

A maximum of a function f is a minimum of $-f$. Thus, the general optimization problem may be stated mathematically as:

$$\begin{array}{lll} \text{minimize} & f(\mathbf{x}), & \mathbf{x} = (x_1, x_2, \ldots, x_n)^T \\ \text{subject to} & c_i(\mathbf{x}) = 0, & i = 1, 2, \ldots, m' \\ & c_i(\mathbf{x}) \geq 0, & i = m'+1, \ldots, m. \end{array} \tag{1}$$

where $f(\mathbf{x})$ is the objective function, $\mathbf{x}$ is the column vector of the n independent variables, and $c_i(\mathbf{x})$ is the set of constraint functions. Constraint equations of the form $c_i(\mathbf{x}) = 0$ are termed *equality constraints*, and those of the form $c_i(\mathbf{x}) \geq 0$ are *inequality constraints.* Taken together, $f(\mathbf{x})$ and $c_i(\mathbf{x})$ are known as the *problem functions.*

1.2 Classifications

There are many optimization algorithms available to the computational scientist. Many methods are appropriate only for certain types of problems. Thus, it is important to be able to recognize the characteristics of a problem in order to identify an appropriate solution technique. Within each class of problems there are different minimization methods, varying in computational requirements, convergence properties, and so on. A discussion on the relative strengths and weaknesses of available techniques within a given class of problems will be the focus of the following sections. Optimization problems are classified according to the mathematical characteristics of the objective function, the constraints, and the control variables.

Probably the most important characteristic is the *nature* of the objective function. A function is *linear* if the relationship between $f(\mathbf{x})$ and the control variables is of the form

$$f(\mathbf{x}) = \mathbf{b}^T\mathbf{x} + c, \tag{2}$$

where $\mathbf{b}$ is a constant-valued vector and c is a constant; a function is *quadratic* if

$$f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c, \tag{3}$$

where $\mathbf{A}$ is a constant-valued matrix. Special methods can locate the optimal solution very efficiently for linear and quadratic functions, for example.

There is a special class of problems, examples of which are particularly common in the fields of operations research and engineering design, in which the task is to find the optimum *permutation* of some control variables. These are known as *combinatorial* optimization problems. The most famous example is the *traveling salesman problem* (TSP) in which the shortest cyclical itinerary visiting N cities is sought. The solutions to such problems are usually represented as *ordered* lists of integers (indicating, for example in the TSP, the cities in the order they are to be visited), and they are, of course, constrained, since not all integer lists represent valid solutions. These and other classifications are summarized in Table 1. Table 2 lists application examples from the wide range of fields where optimization is employed and gives their classifications under this taxonomy.

Section 2 details methods appropriate to *unconstrained continuous univariate/multivariate problems*, and Section 3 mentions methods appropriate to *constrained continuous multivariate problems.* Section 4 considers methods appropriate to *(mixed) integer multivariate problems*, and Section 5 discusses what to do if none of these methods succeed. In general, the optimization of the complex functions that occur in many practical applications is difficult. However, with persistence and resourcefulness solutions can often be obtained.

Table 1: Optimization Problem Classifications

Characteristic	Property	Classification
Number of control variables	One	Univariate
	More than one	Multivariate
Type of control variables	Continuous real numbers	Continuous
	Integers	Integer or Discrete
	Both continuous real numbers and integers	Mixed Integer
	Integers in permutations	Combinatorial
Problem functions	Linear functions of the control variables	Linear
	Quadratic functions of the control variables	Quadratic
	Other nonlinear functions of the control variables	Nonlinear
Problem formulation	Subject to constraints	Constrained
	Not subject to constraints	Unconstrained

Table 2: Examples of Optimization Applications

Field	Problem	Classification
Nuclear Engineering	In-Core Nuclear Fuel Management	Nonlinear Constrained Multivariate Combinatorial
Computational Chemistry	Energy Minimization for 3D Structure Prediction	Nonlinear Unconstrained Multivariate Continuous
Computational Chemistry and Biology	Distance Geometry	Nonlinear Constrained Multivariate Continuous

1.3 Optimality Conditions

Before continuing to consider individual optimization algorithms, we describe the conditions which hold at the optimum sought.

The strict definition of the *global optimum* $\mathbf{x}^*$ of $f(\mathbf{x})$ is that

$$f(\mathbf{x}^*) < f(\mathbf{y}) \quad \forall\, \mathbf{y} \in V(\mathbf{x}), \mathbf{y} \neq \mathbf{x}^*, \tag{4}$$

where $V(\mathbf{x})$ is the set of feasible values of the control variables $\mathbf{x}$. Obviously, for an unconstrained problem $V(\mathbf{x})$ is infinitely large.

A point $\mathbf{y}^*$ is a *strong local minimum* of $f(\mathbf{x})$ if

$$f(\mathbf{y}^*) < f(\mathbf{y}) \quad \forall\, \mathbf{y} \in N(\mathbf{y}^*, \eta), \mathbf{y} \neq \mathbf{y}^*, \tag{5}$$

where $N(\mathbf{y}^*, \eta)$ is defined as the set of feasible points contained in the neighborhood of $\mathbf{y}^*$, i.e., within some arbitrarily small distance η of $\mathbf{y}^*$. For $\mathbf{y}^*$ to be a *weak local minimum* only an inequality need be satisfied

$$f(\mathbf{y}^*) \leq f(\mathbf{y}) \quad \forall\, \mathbf{y} \in N(\mathbf{y}^*, \eta), \mathbf{y} \neq \mathbf{y}^*. \tag{6}$$

More useful definitions, i.e., more easily identified optimality conditions, can be provided if $f(\mathbf{x})$ is a smooth function with continuous first and second derivatives for all feasible $\mathbf{x}$. Then a point $\mathbf{x}^*$ is a *stationary point* of $f(\mathbf{x})$ if

$$\mathbf{g}(\mathbf{x}^*) = 0, \tag{7}$$

where $\mathbf{g}(\mathbf{x})$ is the *gradient* of $f(\mathbf{x})$. This first derivative vector $\Delta f(\mathbf{x})$ has components given by

$$g_i(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial x_i}. \tag{8}$$

The point $\mathbf{x}^*$ is also a *strong local minimum* of $f(\mathbf{x})$ if the *Hessian* matrix $\mathbf{H}(\mathbf{x})$, the symmetric matrix of second derivatives with components

$$H_{ij}(\mathbf{x}) = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}, \tag{9}$$

is *positive-definite* at $\mathbf{x}^*$, i.e., if

$$\mathbf{u}^T \mathbf{H}(\mathbf{x}^*) \mathbf{u} > 0 \quad \forall\, \mathbf{u} \neq 0. \tag{10}$$

This condition is a generalization of convexity, or positive curvature, to higher dimensions.

Figure 1 illustrates the different types of stationary points for unconstrained univariate functions.

As shown in Figure 2, the situation is slightly more complex for constrained optimization problems. The presence of a constraint boundary, in Figure 2 in the form of a simple bound on the permitted values of the control variable, can cause the global minimum to be an extreme value, an *extremum* (i.e., an endpoint), rather than a true stationary point. Some methods of treating constraints transform the optimization problem into an equivalent unconstrained one, with a different objective function. Such techniques are discussed in Section 4.

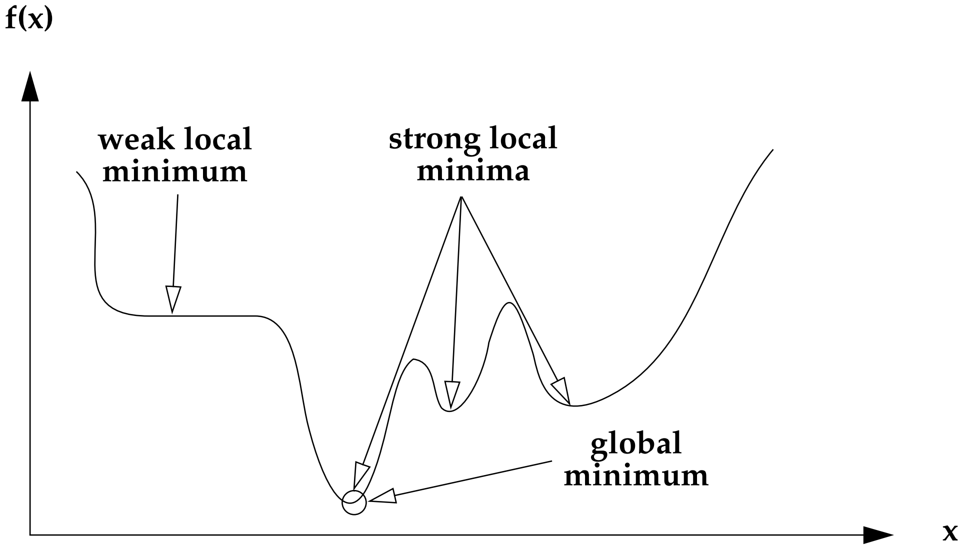

Figure 1: Types of Minima for Unconstrained Optimization Problems.

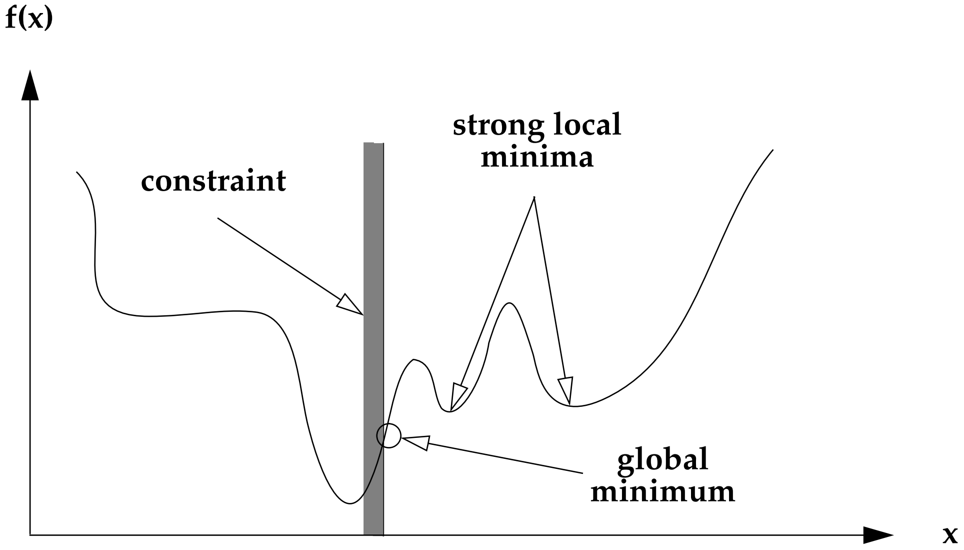

Figure 2: Types of Minima for Constrained Optimization Problems.

1.4 Numerical Example and Programming Notes

Rosenbrock's function is often used as a minimization test problem for nonlinear continuous problems, since its minimum lies at the base of a "banana-shaped valley" and can be difficult to locate. This function is defined for even integers n as the following sum:

$$f(\mathbf{x}) = \sum_{j=1,3,5,\ldots,n-1} [(1-x_j)^2 + 100(x_{j+1} - x_j^2)^2] \ . \tag{11}$$

The contour plot of Rosenbrock's function for $n = 2$ is shown in Figure 3. In general, contour maps show surfaces in the $(n+1)$-dimensional space defined by $f(\mathbf{x}) = \gamma$ where γ is a constant. For $n = 2$, *plane* curves correspond to various values of γ. We see from the figure that the minimum point (dark circle) is at $(1,1)^T$, where $f(\mathbf{x}) = 0$. The gradient components of this function are given by

$$\left.\begin{aligned} g_{j+1} &= 200(x_{j+1} - x_j^2) \\ g_j &= -2[x_j g_{j+1} + (1 - x_j)] \end{aligned}\right\} , \quad j = 1,3,5,\ldots,n-1, \tag{12}$$

and the Hessian is the 2×2 block diagonal matrix with entries

$$\left.\begin{aligned} H_{j+1,j+1} &= 200 \\ H_{j+1,j} &= -400x_j \\ H_{j,j} &= -2(x_j H_{j+1,j} + g_{j+1} - 1) \end{aligned}\right\} , \quad j = 1,3,5,\ldots,n-1. \tag{13}$$

(These formulas are given in a form most efficient for programming.) For $n = 2$, the two eigenvalues of the Hessian at the minimum are $\lambda_1 = 1001.6$ and $\lambda_2 = 0.4$, and thus the condition number $\kappa = \lambda_{max}/\lambda_{min} = 2.5 \times 10^3$. The function contours, whose axes lengths are proportional to the inverse of the eigenvalues, are thus quite elongated near the minimum (see Figure 3).

In minimization applications, the user is often required to write subroutines that compute the target function and its first and second derivatives (the latter optional) at each given point. For Rosenbrock's function, for example, the code for computing these quantities may consist of the following:

```
C*******************************************************************
      SUBROUTINE ROSFUN(N,X,F,G,NOUT)
C ------------------------------------------------------
C ROSENBROCK'S FUNCTION OF DIMENSION N - ASSEMBLE F,G,H
C ------------------------------------------------------
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      PARAMETER (MAXN = 1000)
      INTEGER N,NOUT
      DOUBLE PRECISION F,T1,T2,X(N),G(N)
      COMMON/HESIAN/HESD(MAXN),HESOD(MAXN)
```

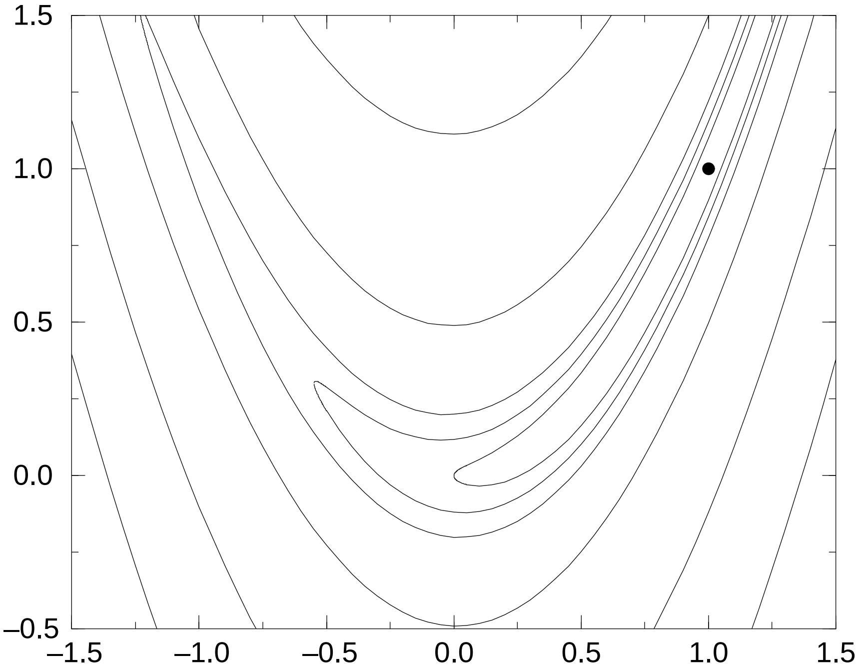

Figure 3: Contours of the Two-Dimensional Rosenbrock Function.

```
      F = 0.D0
      DO 10 J = 1, N-1, 2
         T1 =  1.D0 - X(J)
         T2 = 10.D0 * ( X(J+1) - X(J)**2 )
         F = F + (T1**2) + (T2**2)
   10 CONTINUE
      IF (NOUT .EQ. 0) RETURN

      DO 20 J = 1, N-1, 2
         T1 =  1.D0 - X(J)
         T2 = 10.D0 * ( X(J+1) - X(J)**2 )
         G(J+1) =  20.D0 * T2
         G( J ) =  -2.D0 * ( X(J) * G(J+1) + T1 )
   20 CONTINUE
      IF (NOUT .EQ. 1) RETURN
C -----------------------------------------------------
C H is stored in 2 arrays: HESD for diagonals, HESOD for
C off-diagonals, stored by rows for the upper triangle.
C HESOD is zero every even entry since H has the 2x2
C block-diagonal pattern:
C                    * *
```

```
C                     * *
C                         * *
C                         * *
C                                 Etc.
C -------------------------------------------------------
      DO 30 J = 1, N-1, 2
         HESD(J+1) =  200.D0
         HESD( J ) = -2.D0 * (200.D0*X(J+1) - 1.D0 - 600.D0*(X(J)**2) )
         HESOD(J)  = -400.D0 * X(J)
   30 CONTINUE

      RETURN
      END
C**********************************************************************
```

Note that the Hessian is stored in two one-dimensional arrays that reside in a `COMMON` block. A storage format is often not imposed on the Hessian for large-scale problems so that the user can exploit problem structure (e.g., sparsity) to save storage space.

2 Methods for Unconstrained Continuous Multivariate Problems

2.1 Overview

In this section we outline some basic techniques involving *deterministic* algorithms[1] for finding *local* minima of multivariate functions whose arguments are continuous and on which no restrictions are imposed. For constrained problems, techniques are based on those for unconstrained problems, and we mention only general approaches to them at the end of this section. It should be emphasized that finding the *global* minimum is an entirely different, and more challenging, problem which will not be addressed here.[2] Basically, stochastic methods[3] are better suited at this time for large-scale global optimization (see Figure 4) and some appropriate algorithms will be outlined in Section 5.

For comprehensive presentations on deterministic optimization techniques for multivariate functions, we refer the reader to excellent textbooks [1, 4, 9, 16, 22, 29, 45], some recent volumes and reviews [23, 48, 59]. The outline in this section is not intended to provide full algorithmic details of all available methods; rather it exposes key algorithmic concepts and modules, so that the reader could consult specialized literature for further details. Further background details can also be obtained from the linear algebra chapter.

[1] methods that do not contain any random elements

[2] Thus, the notion of *global* convergence of the algorithms in this section refers to obtaining a strict local minimum $\mathbf{x}^*$ from any given starting point $\mathbf{x}_0$ and not the global minimum of a function.

[3] methods that contain random elements

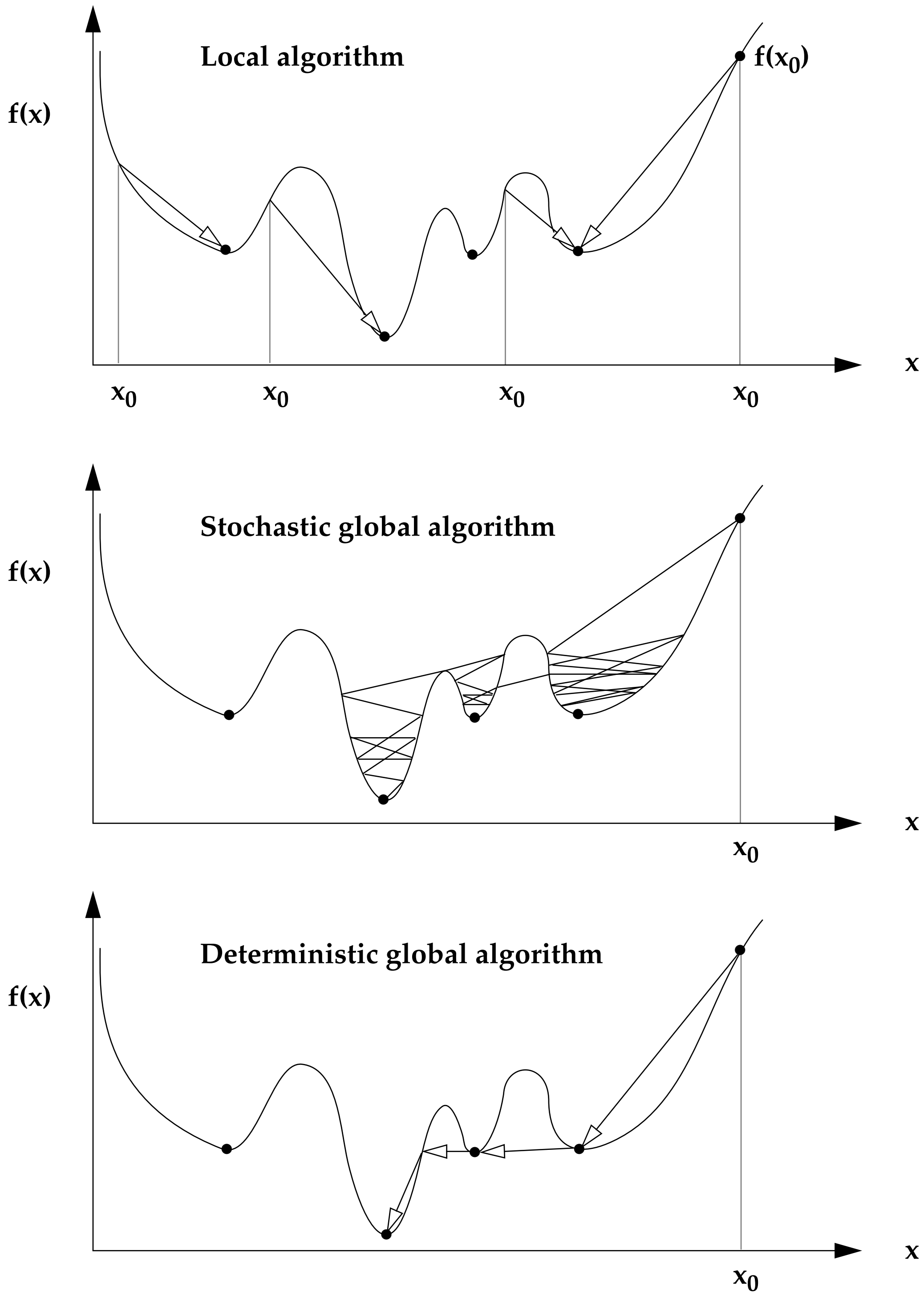

Figure 4: The Structure of Local and Global Minimization Algorithms.

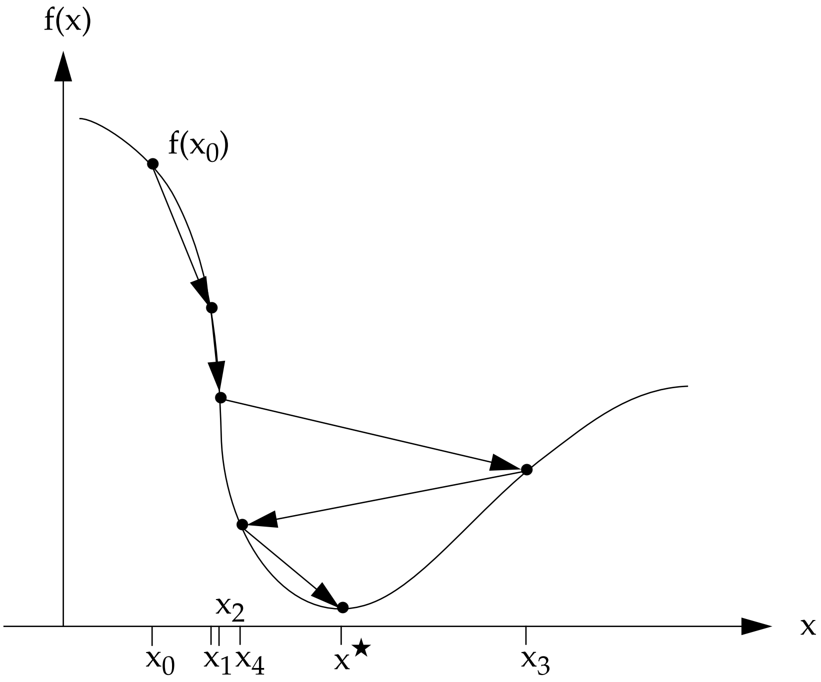

Figure 5: The Structure of a Line-Search Based Local Minimization Algorithm.

2.2 Basic Structure Of Local Methods

The fundamental structure of local iterative techniques for solving unconstrained minimization problems is simple. A starting point is chosen; a direction of movement is prescribed according to some algorithm, and a *line search* or *trust region* approach is performed to determine an appropriate next step. The process is repeated at the new point and the algorithm continues until a local minimum is found (see Figure 5). Schematically, a model local minimizer method can be sketched as follows:

Algorithm 2.1 *Basic Local Minimizer*

- *Supply an initial guess* $\mathbf{x}_0$
- *For* $k = 0, 1, 2, \ldots$ *until convergence*
  1. *Test* $\mathbf{x}_k$ *for convergence*
  2. *Calculate a search direction* $\mathbf{p}_k$
  3. *Determine an appropriate step length* λ_k *(or modified step* $\mathbf{s}_k$*)*
  4. *Set* $\mathbf{x}_{k+1}$ *to* $\mathbf{x}_k + \lambda_k \mathbf{p}_k$ *(or* $\mathbf{x}_k + \mathbf{s}_k$*)*

See exercise 1.

2.2.1 Descent Directions

It is reasonable to choose a search vector $\mathbf{p}_k$ that will be a *descent* direction; that is, a direction leading to function reduction. A descent direction $\mathbf{p}$ is defined as one along which the directional derivative is negative:

$$\mathbf{g}(\mathbf{x})^T \mathbf{p} < 0 \; . \tag{14}$$

When we write the approximation

$$f(\mathbf{x} + \lambda \mathbf{p}) - f(\mathbf{x}) \approx \lambda \mathbf{g}(\mathbf{x})^T \mathbf{p} \; , \tag{15}$$

we see that the negativity of the right-hand side guarantees that a lower function value can be found along $\mathbf{p}$ for a sufficiently small λ.

Different methods are distinguished by their choice of search directions. Algorithms can be classified into nonderivative, gradient, and second-derivative (Newton) methods depending on the technique used to determine $\mathbf{p}_k$ in Algorithm 2.1. These classes will be discussed in turn beginning in Section 2.3.

2.2.2 Line Search and Trust Region Steps

Both line search and trust region methods are essential components of basic descent schemes for guaranteeing *global* convergence [16, 45]. Of course, only one of the two methods is needed for a given minimization algorithm. To date, there has been no clear evidence for superiority of one class over another. Thus we sketch below the line search procedure, more intuitive and simpler to program.

The line search is essentially an approximate *one-dimensional minimization problem.* It is usually performed by safeguarded polynomial interpolation. That is, in a typical line step iteration, cubic interpolation is performed in a region of λ that ensures that the minimum of f along $\mathbf{p}$ has been *bracketed.* Typically, if the search directions are properly scaled, the initial trial point $\lambda_t = 1$ produces a first reasonable trial move from $\mathbf{x}_k$ (see Figure 6). The minimum is bracketed by examining the new function value and slope and decreasing or increasing the interval as needed (see Figure 7). The minimum of that polynomial interpolant in the bracketed interval then provides a new candidate for λ. The minimized one-dimensional function at the current point $\mathbf{x}_k$ is defined by $\tilde{f}(\lambda) \equiv f(\mathbf{x}_k + \lambda \mathbf{p}_k)$, and the vectors corresponding to different values of λ are set by $\tilde{\mathbf{x}}(\lambda) = \mathbf{x}_k + \lambda \mathbf{p}_k$.

For example, at the first step, a cubic polynomial can be constructed from the two function values $\tilde{f}(0)$, $\tilde{f}(\lambda_t)$ and the two slopes $\tilde{g}(0)$, $\tilde{g}(\lambda_t)$. The slopes are the directional derivatives defined as: $\tilde{g}(\lambda) = \mathbf{g}(\mathbf{x}_k + \lambda \mathbf{p}_k)^T \mathbf{p}_k$. Note in Fig. 6 a negative slope at $\lambda = 0$ since $\mathbf{p}_k$ is a descent direction. More generally, for the bracketed interval $[\lambda_1, \lambda_2]$, and corresponding function and slopes $\tilde{f}_1$, $\tilde{f}_2$, $\tilde{g}_1$, $\tilde{g}_2$, the cubic polynomial passing through $(\lambda_1, \tilde{f}_1)$ and $(\lambda_2, \tilde{f}_2)$ and having the specified slopes is given by:

$$p(\lambda) = a(\lambda - \lambda_1)^3 + b(\lambda - \lambda_1)^2 + c(\lambda - \lambda_1) + d \; , \tag{16}$$

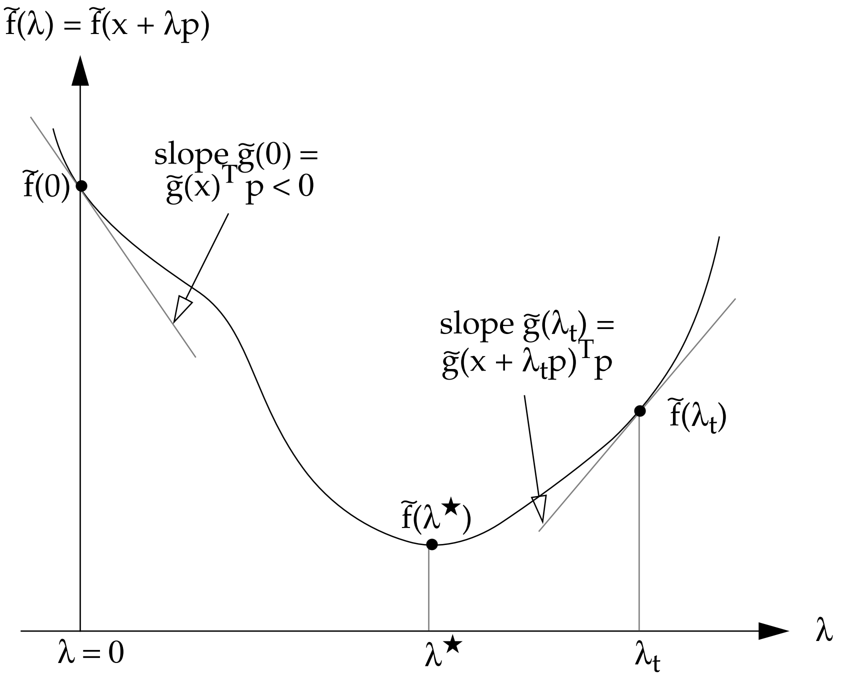

Figure 6: A Line Search Step.

where

$$a = [-2(\tilde{f}_2 - \tilde{f}_1) + (\tilde{g}_1 + \tilde{g}_2)(\lambda_2 - \lambda_1)] \, / \, (\lambda_2 - \lambda_1)^3 \tag{17}$$

$$b = [3(\tilde{f}_2 - \tilde{f}_1) - (2\tilde{g}_1 + \tilde{g}_2)(\lambda_2 - \lambda_1)] \, / \, (\lambda_2 - \lambda_1)^2 \tag{18}$$

$$c = \tilde{g}_1 \tag{19}$$

$$d = \tilde{f}_1 \; . \tag{20}$$

A minimum of $p(\lambda)$ can be obtained by setting

$$\lambda = \lambda_1 + [-b + \sqrt{b^2 - 3ac}\,] \, / \, 3a \tag{21}$$

as long as $a \neq 0$ and $b^2 - 3ac \geq 0$. Otherwise, a quadratic interpolant fitted to $\tilde{f}_1$, $\tilde{g}_1$, and $\tilde{f}_2$ can be constructed with the same coefficients:

$$p(\lambda) = b(\lambda - \lambda_1)^2 + c(\lambda - \lambda_1) + d \; , \tag{22}$$

and minimized to produce

$$\lambda = \lambda_1 - c/2b \; . \tag{23}$$

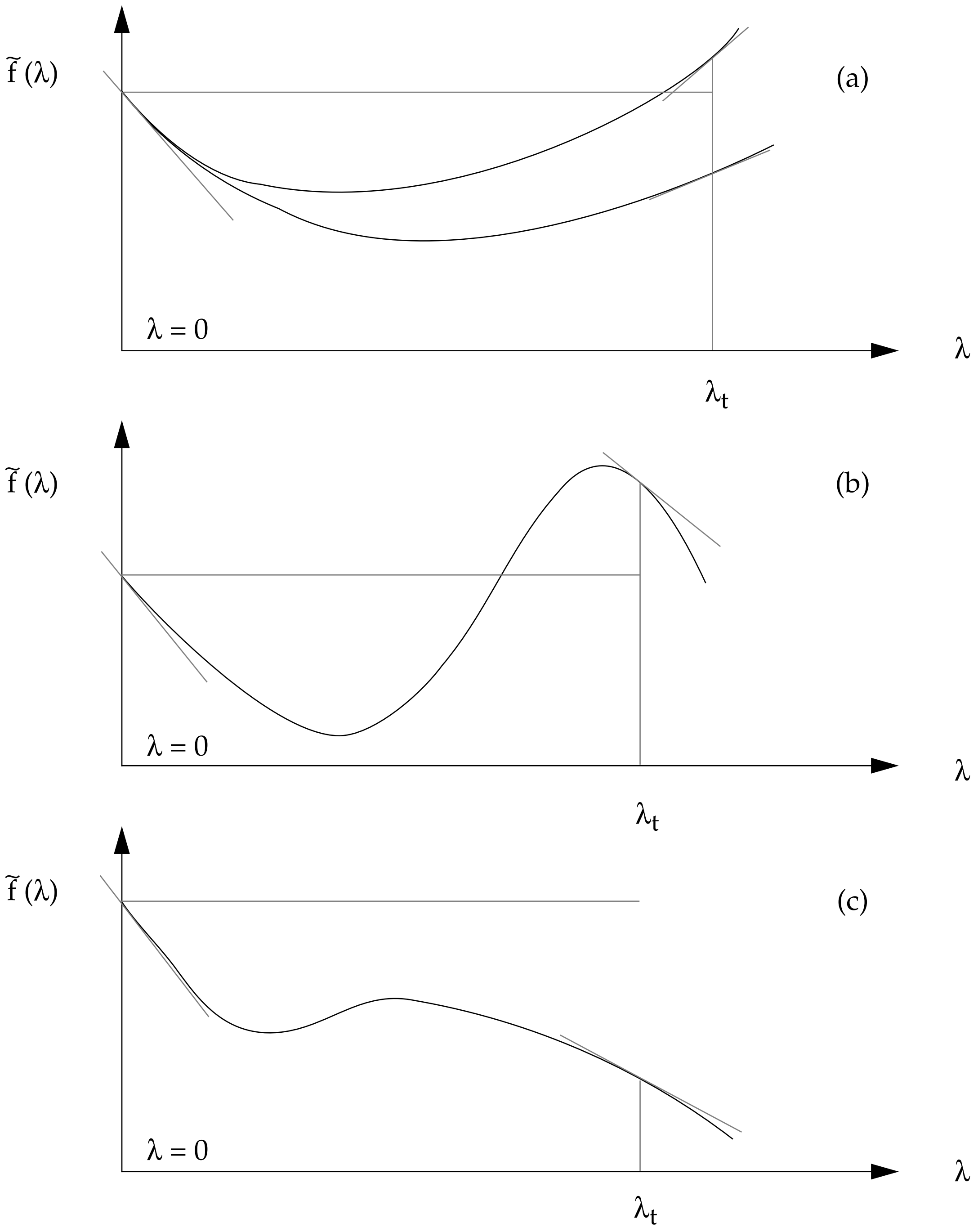

Figure 7: Possible Situations in Line Search Algorithms, with respect to the Current and Trial Points: (a) The new slope is positive; (b) The new slope is negative but function value is greater; (c) The new slope is negative and function value is lower.

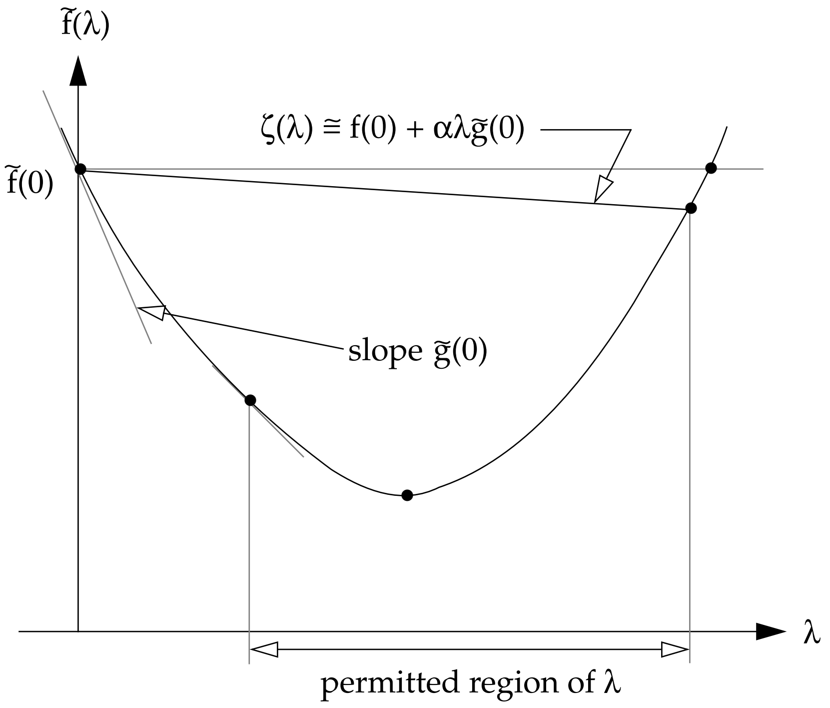

Figure 8: Line Search Conditions.

The degenerate case of $b = 0$ corresponds to a linear function rather than a quadratic and redundancy among the three values $\{\tilde{f}_1, \tilde{f}_2, \tilde{g}_1\}$; it is excluded by construction.

Once a new λ and corresponding trial point $\tilde{\mathbf{x}}(\lambda)$ have been determined in a line search iteration, conditions of sufficient progress with respect to the objective function are tested. The conditions often used in optimization algorithms are derived from the Armijo and Goldstein criteria [16]. They require that

$$f(\mathbf{x}_k + \lambda \mathbf{p}_k) \leq f(\mathbf{x}_k) + \alpha\lambda \mathbf{g}(\mathbf{x}_k)^T \mathbf{p}_k \tag{24}$$

and

$$|\mathbf{g}(\mathbf{x}_k + \lambda \mathbf{p}_k)^T \mathbf{p}_k| \leq \beta |\mathbf{g}(\mathbf{x}_k)^T \mathbf{p}_k| \tag{25}$$

hold for two constants α, β, where $0 < \alpha < \beta < 1$. Essentially, the first condition prescribes an upper limit on acceptable new function values, and the second condition imposes a lower bound on λ (see Figure 8). Typical values of α and β in line search algorithms are $\alpha = 10^{-4}$ and $\beta = 0.9$. Larger values of α make the first test more severe, and smaller β make the second more severe. The work in the line search (number of polynomial interpolations) should be balanced with the overall progress in the minimization.

See exercises 2 and 3.

2.2.3 Convergence Criteria

The simplest test for optimality of each $\mathbf{x}_k$ in the basic local minimizer algorithm 2.1 involves the following gradient condition:

$$\|\mathbf{g}_k\| \leq \epsilon_g(1 + |f(\mathbf{x}_k)|) \ . \tag{26}$$

The parameter ϵ_g is a small positive number such as square root of *machine precision*, ϵ_m (ϵ_m is the smallest number ϵ such that the floating point value of $(1+\epsilon)$ is greater than the floating representation of 1). For large-scale problems, the Euclidean norm divided by $\sqrt{n}$, or the *max* norm, $\|\mathbf{g}\|_\infty = \max_i |g_i|$, may be used to replace $\|\mathbf{g}\|_2$ or $\|\mathbf{g}\|_2/\sqrt{n}$ in the left side of equation (26). This measures instead an average gradient element.

To obtain a measure of progress at each iteration (function reduction, change in $\mathbf{x}$, etc.) and possibly halt computations if necessary, the following combination can be used [29]:

$$f(\mathbf{x}_{k-1}) - f(\mathbf{x}_k) < \epsilon_f(1 + |f(\mathbf{x}_k)|) \tag{27}$$

$$\|\mathbf{x}_{k-1} - \mathbf{x}_k\| < (\epsilon_f)^{1/2}(1 + \|\mathbf{x}_k\|) \ , \tag{28}$$

$$\|\mathbf{g}_k\| < (\epsilon_f)^{1/3}(1 + |f(\mathbf{x}_k)|) \ . \tag{29}$$

Here $\epsilon_f > 0$ is a small number that specifies the desired accuracy in the function value. Each step of the algorithm 2.1 can then check conditions (26) and (27), (28), (29). For $\mathbf{x}_0$, only the first is checked. If either the triplet (27), (28), (29) or (26) hold, the iteration process can be halted. While the conditions above are quite useful in practice, many minimization algorithms only incorporate a gradient-norm test in some form.

2.2.4 Convergence Characterization

The convergence properties of an algorithm are described by two analytic quantities: convergence *order* and convergence *ratio*. A sequence $\{\mathbf{x}_k\}$ is said to converge to $\mathbf{x}^*$ if the following holds: $\lim_{k\to\infty} \|\mathbf{x}_k - \mathbf{x}^*\| = 0$. The sequence is said to converge to $\mathbf{x}^*$ with *order* p if p is the largest nonnegative number for which a finite limit β exists, where

$$0 \leq \beta \leq \lim_{k\to\infty} \frac{\|\mathbf{x}_{k+1} - \mathbf{x}^*\|}{\|\mathbf{x}_k - \mathbf{x}^*\|^p} \ . \tag{30}$$

When $p = 1$ and $\beta < 1$, the sequence is said to converge *linearly* (e.g., $\mathbf{x}_k = 2^{-k}$ for $n = 1$); when $p = 1$ and $\beta = 0$, the sequence converges *superlinearly* (e.g., $\mathbf{x}_k = k^{-k}$); and when $p = 2$, the convergence is *quadratic* (e.g., $\mathbf{x}_k = 2^{-2^k}$). Thus, quadratic convergence is more rapid than superlinear, which in turn is faster than linear. The constant β is the associated convergence ratio.

See exercise 4.

Convergence properties of most minimization algorithms are analyzed through their application to convex *quadratic functions*. Such functions can be written in the form of equation (3), where A is a positive-definite $n \times n$ matrix. We refer to this convex quadratic function throughout this chapter by $q_A(\mathbf{x})$. For such a function, the unique *global* minimum $\mathbf{x}^*$ satisfies the linear system $A\mathbf{x}^* = -\mathbf{b}$. Since general functions can be approximated by a quadratic convex function in the neighborhood of their local minima, the convergence properties obtained for convex quadratic functions are usually applied locally to general functions. However, such generalizations do not guarantee good behavior in practice on complex, large-scale functions.

2.3 Nonderivative Methods

Minimization methods that incorporate only function values generally involve some systematic method to search the conformational space. Although they are generally easy to implement, their realized convergence properties are rather poor. They may work well in special cases when the function is quite random in character or the variables are essentially uncorrelated. In general, the computational cost, dominated by the number of function evaluations, can be excessively high for functions of many variables and can far outweigh the benefit of avoiding derivative calculations. The techniques briefly sketched below are thus more interesting from a historical perspective.

Coordinate Descent methods form the basis to nonderivative methods [22, 45]. In the simplest variant, the search directions at each step are taken as the standard basis vectors. A *sweep* through these n search vectors produces a sequential modification of one function variable at a time. Through repeatedly sweeping the n-dimensional space, a local minimum might ultimately be found. Since this strategy is inefficient and not reliable, Powell's variant has been developed [51]. Rather than specifying the search vectors *a priori*, the standard basis directions are modified as the algorithm progresses. The modification ensures that, when the procedure is applied to a convex quadratic function, n mutually conjugate directions are generated after n sweeps. A set of mutually conjugate directions $\{\mathbf{p}_k\}$ with respect to the (positive-definite) Hessian $\mathbf{A}$ of such a convex quadratic is defined by $\mathbf{p}_i^T\mathbf{A}\mathbf{p}_j = 0$ for all $i \neq j$. This set possesses the important property that a successive search along each of these directions suffices to find the minimum solution [22, 45]. Powell's method thus guarantees that in exact arithmetic (i.e., in absence of round-off error), the minimum of a convex quadratic function will be found after n sweeps.

If obtaining the analytic derivatives is out of the question, viable alternatives remain. The gradient can be approximated by *finite-differences* of function values, such as

$$g_i(\mathbf{x}) \approx \frac{1}{h_i}[f(\mathbf{x} + h_i\mathbf{e_i}) - f(\mathbf{x})] \ , \tag{31}$$

for suitably chosen intervals$\{h_i\}$ [30]. Alternatively, automatic differentiation, essentially a new algebraic construct [19, 35, 53], may be used. In any case, these calculated derivatives may then be used in a gradient or quasi-Newton method. Such alternatives will generally provide significant improvement in computational cost and reliability, as will be discussed in the following sections.

2.4 Gradient Methods

Two common gradient methods are Steepest Descent (SD) [7, 45] and Conjugate Gradient (CG) [32, 36]. Both are fundamental techniques that are often incorporated into various iterative algorithms in many areas of scientific computing. For a theoretical treatment and notes on parallel implementations, see the linear algebra chapter.

2.4.1 Derivative Programming

Before we continue to describe algorithmic details of gradient methods, some programming tips are appropriate. For using gradient and second derivative minimization methods, derivative subroutines must be supplied by the user. Errors in the code for the function and the derivatives are common. Such errors, especially in the derivatives, are often difficult to find since function decrease may still be realized upon minimization. Therefore, it is essential to test the derivative routines of the objective function before applying minimization routines.

We have developed one such testing procedure using a Taylor series approach. Our general subroutine `TESTGH` tests the compatibility of the gradient and Hessian components of a function against the function routine. Its calling sequence is:

```
TESTGH(N,XC,FC,GC,Y,YHY,VEC)
```

where `N` is the dimension; `XC(N)` the current vector of control variables; `FC` the function value at `XC`; `GC(N)` the gradient vector at `XC`; `Y(N)` a random perturbation vector; `YHY` the inner product of `Y` with `HY` (the product of the Hessian evaluated at `XC` and the vector `Y`); and `VEC(N)` a work vector. On input, all quantities except for `VEC` must be given. The vector `Y` should be chosen so that the function value at `XC+Y` is in a reasonable range for the problem (see below).

Derivatives are tested using a Taylor expansion of f around a given point $\mathbf{x}_c$. The following Taylor series is formulated at $\mathbf{x}_c + \epsilon\mathbf{y}$ where ϵ is a scalar:

$$f(\mathbf{x}_c + \epsilon\mathbf{y}) = f(\mathbf{x}_c) + \epsilon\, \mathbf{g}_c^T \mathbf{y} + (\epsilon^2/2)\mathbf{y}^T H_c\, \mathbf{y} + \mathrm{O}(\epsilon^3), \tag{32}$$

where $\mathbf{g}_c$ and H_c are the gradient and Hessian, respectively, evaluated at $\mathbf{x}_c$. If only the gradient routines are tested, the second-order Taylor term `YHY` is set to zero, and the truncation error is $\mathcal{O}(\epsilon^2)$. Our test is performed by computing this Taylor approximation at smaller and smaller values of ϵ and checking whether the truncation errors are as expected: $\mathcal{O}(\epsilon^2)$ and $\mathcal{O}(\epsilon^3)$ if the approximation is correct up to the gradient and Hessian terms, respectively. At every step we half ϵ and test if indeed the truncation errors decrease as they should (i.e., if the error corresponding to ϵ is ϵ_1, the error for $\epsilon/2$ should be $\epsilon_1/4$ if the gradient is correct, and $\epsilon_1/8$ if the Hessian is also correct.)

The output consists of a series of values for `RATIO` (ratio of old to new errors) printed for each ϵ until the truncation error and/or ϵ is very small. If `RATIO` tends to 4 as ϵ is decreased (and the error is relatively small) the gradient is correct, and if `RATIO` tends to 8 both the gradient and Hessian are correct. If `RATIO` tends to 2, which is $\mathcal{O}(\epsilon)$, neither the gradient nor the Hessian are correct. If `RATIO` tends to unity, the errors may be too large given the perturbation vector $\mathbf{y}_c$. Thus in general, reliable values of `RATIO` should occur when: (1) ϵ is not too large and not too small, and (2) the difference between $f(\mathbf{x}_c + \epsilon\mathbf{y})$ and the Taylor-series approximation is of reasonable magnitude. Different starting point and/or perturbation vectors can be tried for verification. The code for `TESTGH` can be found in file `testgh.f` in connection with the online version of this paper.

For example, output from testing Rosenbrock's function for 12 variables consists of the following:

```
X0 VECTOR:
     -1.20              1.00              -1.20              1.00
     -1.20              1.00              -1.20              1.00
     -1.20              1.00              -1.20              1.00
Y VECTOR:
     -1.09              0.77              -0.88              0.64
      0.71              0.58               0.94             -0.90
     -0.62              0.77              -0.90             -0.98

     ENTERING TESTGH ROUTINE:

THE FUNCTION VALUE AT X                   =   1.45200000E+02
THE FIRST-ORDER TAYLOR TERM,   (G, Y)     =   3.19353760E+02
THE SECOND-ORDER TAYLOR TERM, (Y,HY)      =   5.39772665E+03
EPSMIN =   1.42108547E-14


EPS             F                TAYLOR            DIFF.            RATIO

 5.0000E-01  1.09854129E+03  9.79592712E+02  1.18948574E+02
 2.5000E-01  4.07080835E+02  3.93717398E+02  1.33634374E+01  8.90104621E+00
 1.2500E-01  2.28865318E+02  2.27288959E+02  1.57635878E+00  8.47740855E+00
 6.2500E-02  1.75893210E+02  1.75702045E+02  1.91165417E-01  8.24604580E+00
 3.1250E-02  1.57838942E+02  1.57815414E+02  2.35282126E-02  8.12494428E+00
 1.5625E-02  1.50851723E+02  1.50848805E+02  2.91806005E-03  8.06296382E+00
 7.8125E-03  1.47860040E+02  1.47859677E+02  3.63322099E-04  8.03160629E+00
 3.9063E-03  1.46488702E+02  1.46488657E+02  4.53255493E-05  8.01583443E+00
 1.9531E-03  1.45834039E+02  1.45834033E+02  5.66008660E-06  8.00792506E+00
 9.7656E-04  1.45514443E+02  1.45514443E+02  7.07160371E-07  8.00396463E+00
 4.8828E-04  1.45356578E+02  1.45356578E+02  8.83731524E-08  8.00198196E+00
```

```
DIFF IS SMALL (LESS THAN   2.97291798E-08 IN ABSOLUTE VALUE)
```

Note that the `RATIO` is larger than eight when `EPS` is larger and then decreases steadily. A small error in the code would produce much different values. We encourage the student to try this testing routine on several subroutines that compute objective functions and their derivatives; errors should be introduced into the derivative codes systematically to examine the ability of `TESTGH` to detect them and provide the right diagnosis, as outlined above.

See exercises 5 and 6.

2.4.2 Steepest Descent

SD is one of the oldest and simplest methods. It is actually more important as a theoretical, rather than practical, reference by which to test other methods. However, 'steepest descent' *steps* are often incorporated into other methods (e.g., Conjugate Gradient, Newton) when roundoff destroys some desirable theoretical properties, progress is slow, or regions of indefinite curvature are encountered.

At each iteration of SD, the search direction is taken as $-\mathbf{g}_k$, the negative gradient of the objective function at $\mathbf{x}_k$. Recall that a descent direction $\mathbf{p}_k$ satisfies $\mathbf{g}_k^T\mathbf{p}_k < 0$. The simplest way to guarantee the negativity of this inner product is to choose $\mathbf{p}_k = -\mathbf{g}_k$. This choice also minimizes the inner product $-\mathbf{g}_k^T\mathbf{p}$ for unit-length vectors and, thus gives rise to the name *Steepest Descent*.

SD is simple to implement and requires modest storage, $\mathcal{O}(n)$. However, progress toward a minimum may be very slow, especially near a solution. The convergence rate of SD when applied to a convex quadratic function $q_A(\mathbf{x})$ is only *linear*. The associated convergence ratio is no greater than $[(\kappa - 1)/(\kappa + 1)]^2$ where $\kappa = \lambda_{\max}(\mathbf{A}) \,/\, \lambda_{\min}(\mathbf{A})$. Since the convergence ratio measures the reduction of the error at every step ($\|\mathbf{x}_{k+1} - \mathbf{x}^*\| \leq \beta\|\mathbf{x}_k - \mathbf{x}^*\|$ for a linear rate), the relevant SD value can be arbitrarily close to 1 when κ is large. Thus, the SD search vectors may in some cases exhibit very inefficient paths toward a solution, especially close to the solution.

Minimization performance for Rosenbrock's and Beale's function with $n = 2$ are shown in Figures 9 and Figure 13 for SD and other methods that will be discussed in this section.

We show progress by superimposing the resulting iterates from each minimization application on the contour plot. Recall that the minimum point for Rosenbrock's function lies at $\mathbf{x} = (1, 1)^T$, where $f(\mathbf{x}) = 0$. We clearly see in Figure 9 the characteristic behavior of the SD method: relatively good progress at first, but very slow convergence in the vicinity of the solution. The method was stopped after 1200 iterations, where a gradient norm of only $\mathcal{O}(10^{-2})$ was obtained. For the remaining methods, gradient norms of $\mathcal{O}(10^{-9}$–$10^{-11})$ were realized.

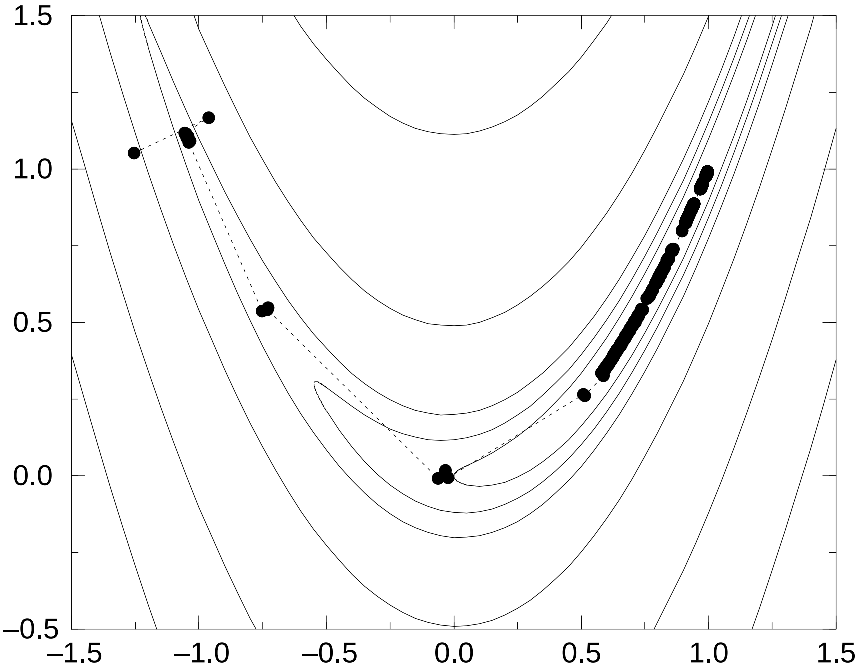

Figure 9: Steepest Descent Minimization Path for the Two-Dimensional Rosenbrock Function.

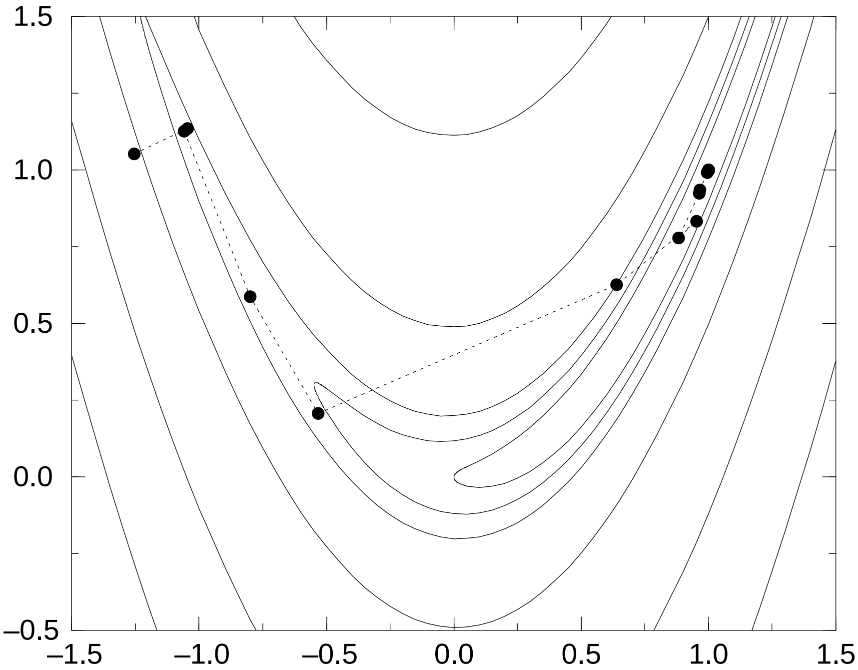

Figure 10: Conjugate Gradient Minimization Path for the Two-Dimensional Rosenbrock Function.

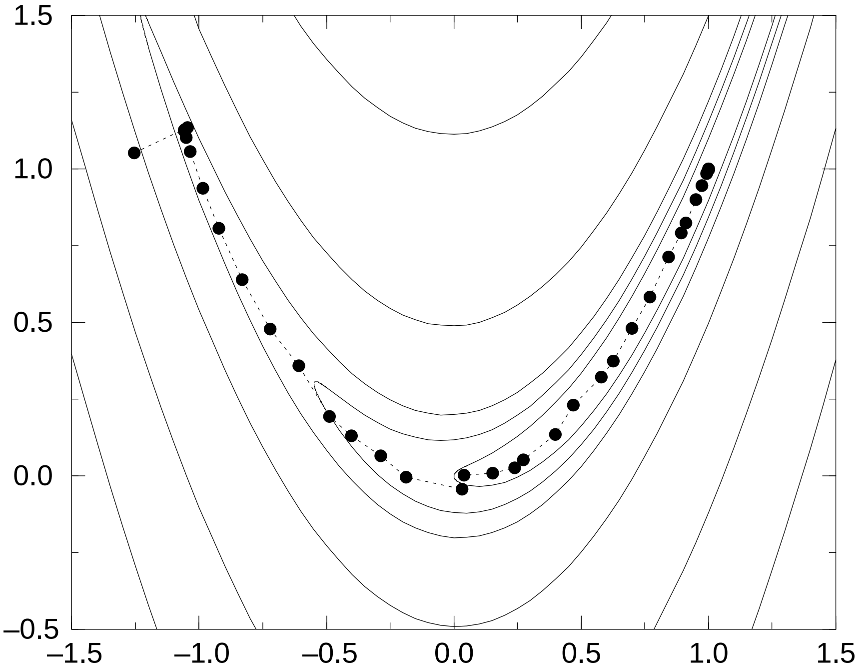

Figure 11: BFGS Quasi-Newton Minimization Path for the Two-Dimensional Rosenbrock Function.

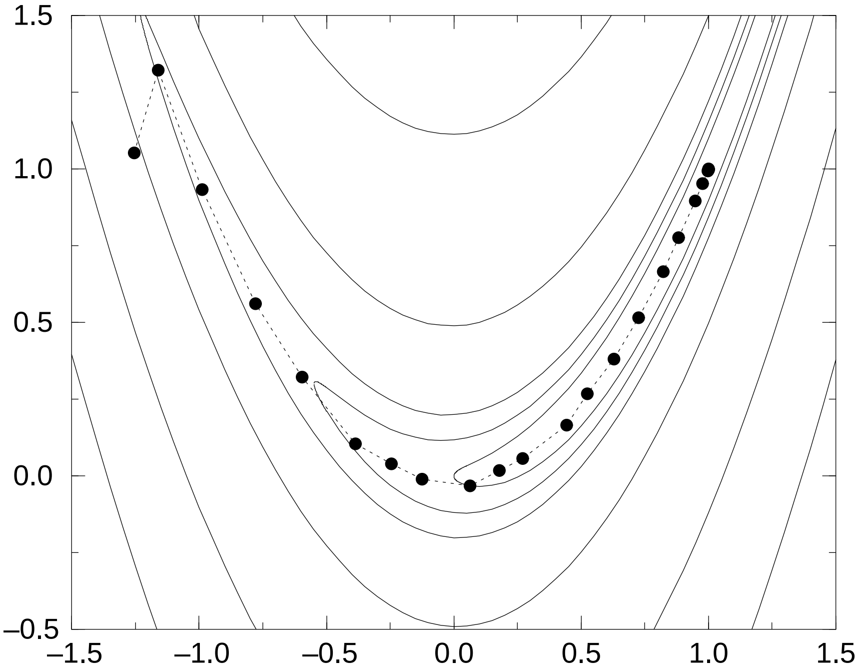

Figure 12: Truncated Newton Minimization Path for the Two-Demensional Rosenbrock Function.

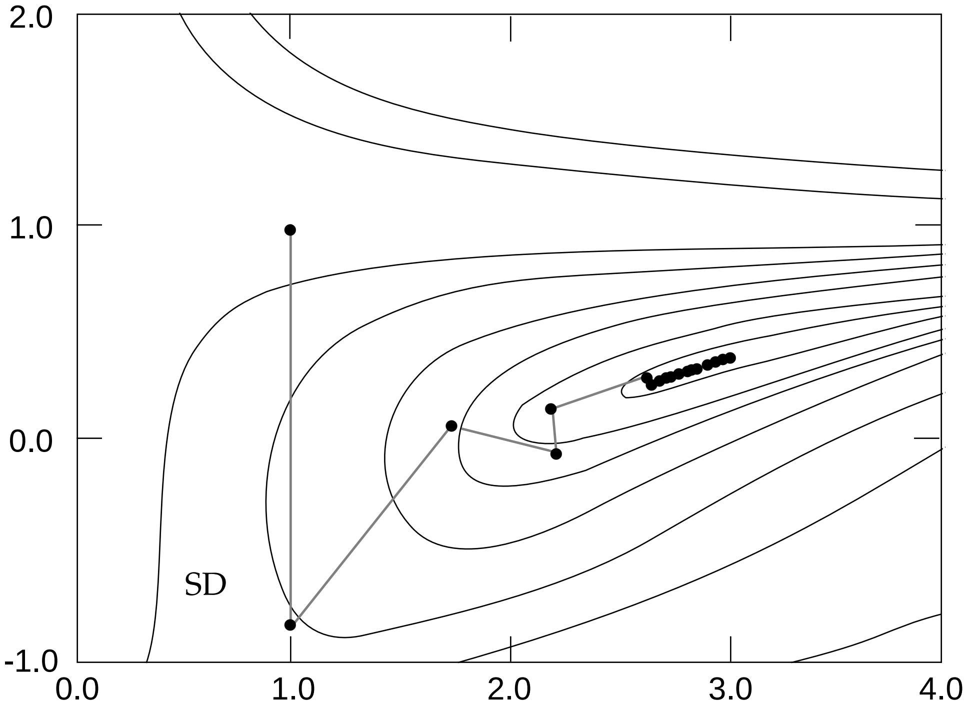

Figure 13: Steepest Descent Minimization Path for the Two-Dimensional Beale Function.

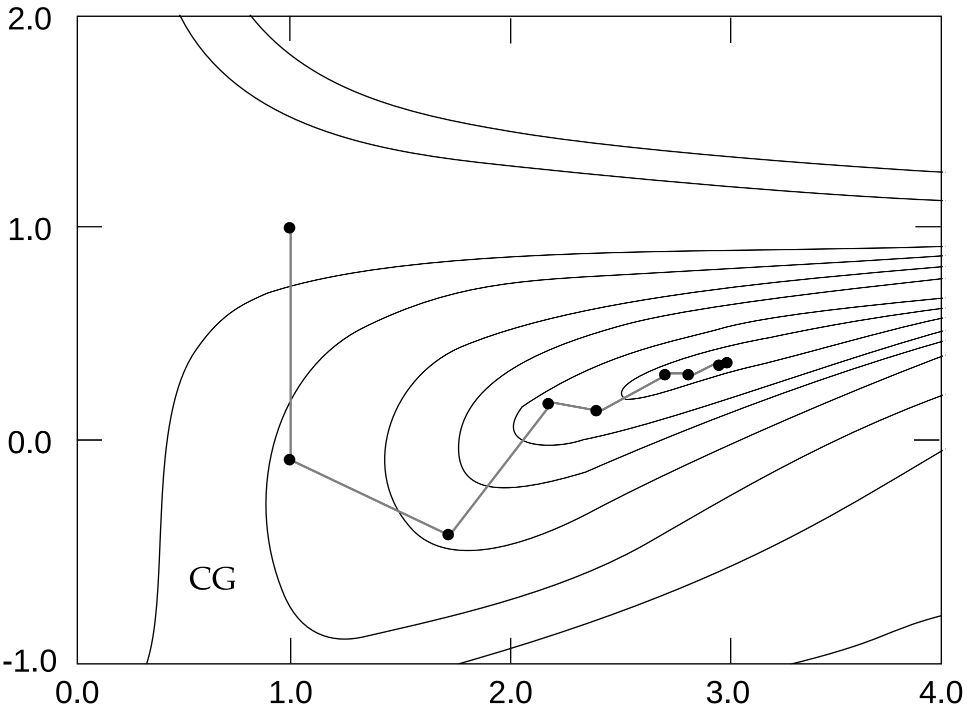

Figure 14: Conjugate Gradient Minimization Path for the Two-Dimensional Beale Function.

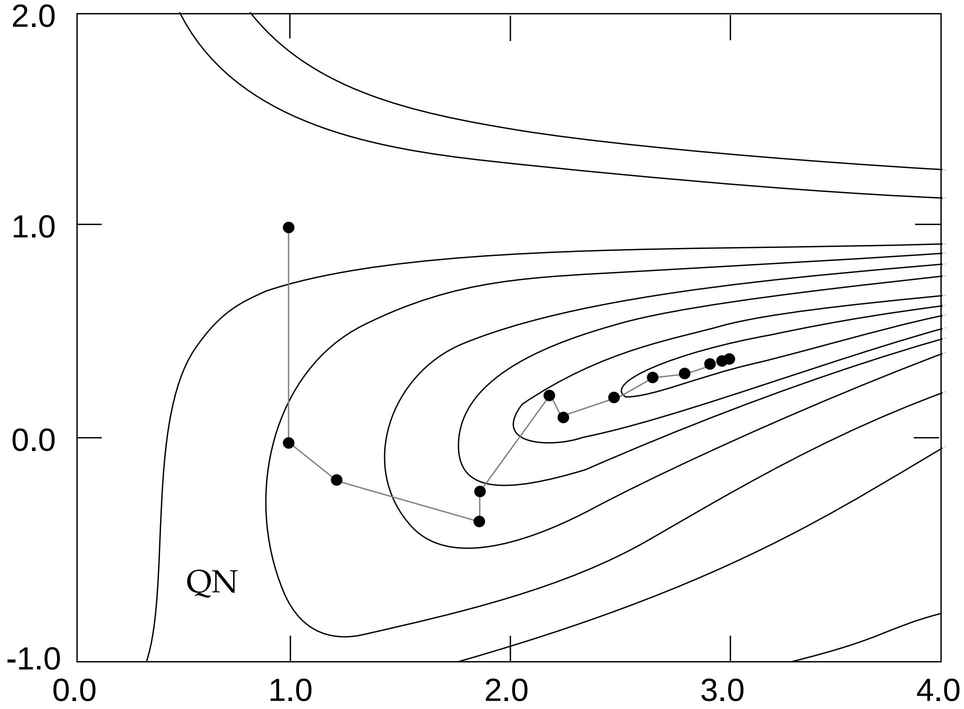

Figure 15: BFGS Quasi-Newton Minimizatin Path for the Two-Dimensional Beale Function.

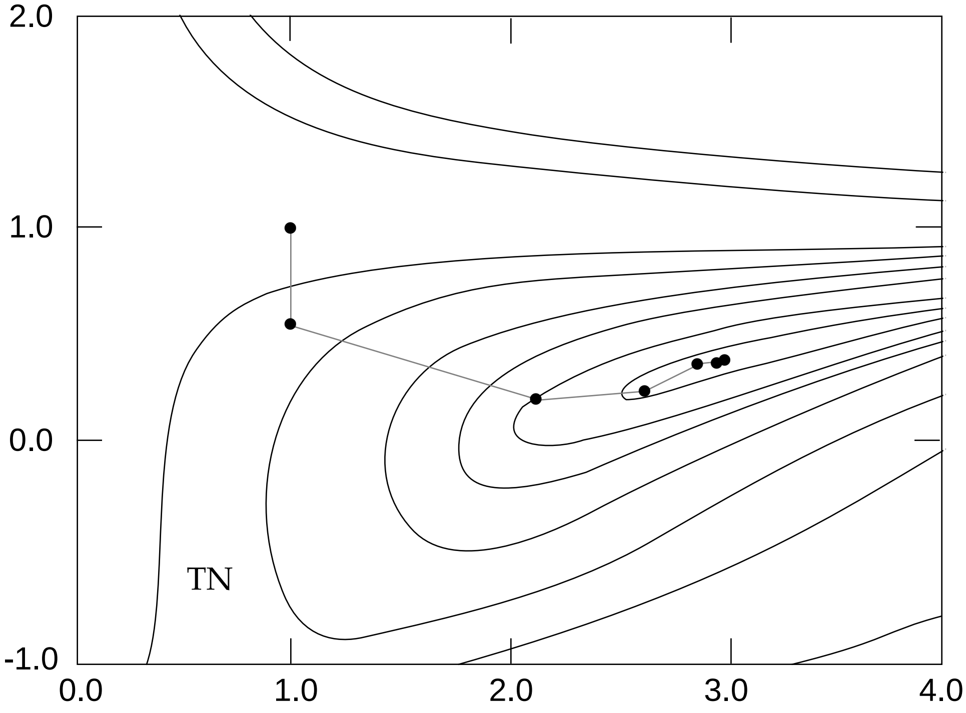

Figure 16: Truncated Newton Minimization Path for the Two-Dimensional Beale Function.

Note from the contour plots of Beale's function, Figure 13, that the function has a narrow curving valley in the vicinity of the minimum, which occurs at $\mathbf{x} = (3, 0.5)^T$. The function value at the minimum is zero and increases sharply, particularly as x_2 increases.

From Figure 13, we clearly note how the SD search vectors (the negative gradient) are perpendicular to the contour lines; progress is initially rapid but then becomes very slow.

2.4.3 Conjugate Gradient

The CG method was originally designed to minimize convex quadratic functions but, through several variations, has been extended to the general case [23, 52]. The first iteration in CG is the same as in SD, but successive directions are constructed so that they form a set of mutually conjugate vectors with respect to the (positive-definite) Hessian $\mathbf{A}$ of a general convex quadratic function $q_A(\mathbf{x})$. Whereas the rate of convergence for SD depends on the ratio of the extremal eigenvalues of $\mathbf{A}$, the convergence properties of CG depend on the entire matrix spectrum. Faster convergence is expected when the eigenvalues are clustered. In exact arithmetic, convergence is obtained in at most n steps. In particular, if $\mathbf{A}$ has m distinct eigenvalues, convergence to a solution requires m iterations.

For example, when the bound on convergence measures the size of $\mathbf{x}_k - \mathbf{x}^*$ with respect to the $\mathbf{A}$-norm,

$$\|\mathbf{x}\|_A = (\mathbf{x}^T\mathbf{A}\mathbf{x})^{1/2}, \tag{33}$$

we have [32, 45]

$$\|\mathbf{x}_k - \mathbf{x}^*\|_A^2 \leq 4\|\mathbf{x}_k - \mathbf{x}_0\|_A^2 \left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^{2k}. \tag{34}$$

Clearly rapid convergence is expected when $\kappa \approx 1$, as for SD. Further estimates of convergence bounds can only be derived when certain properties about the eigenvalue distribution are known (e.g., m large eigenvalues and $n-m$ small eigenvalues clustered in a region $[a, b]$) [32].

When one refers to the CG method, one often means the *linear* Conjugate Gradient; that is, the implementation for the convex quadratic form. In this case, minimizing $\frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x}$ is equivalent to solving the *linear* system $\mathbf{A}\mathbf{x} = -\mathbf{b}$. Consequently, the conjugate directions $\mathbf{p}_k$, as well as the step lengths λ_k, can be computed in closed form. Below we sketch such an algorithm from a given $\mathbf{x}_0$. We define the residual vector $\mathbf{r} = -(\mathbf{A}\mathbf{x} + \mathbf{b})$ and use the vectors $\{\mathbf{d}_k\}$ below to denote the CG search vectors.

Algorithm 2.2 *CG method to solve* $\mathbf{A}\mathbf{x} = -\mathbf{b}$

1. *Set* $\mathbf{r}_0 = -(\mathbf{A}\mathbf{x}_0 + \mathbf{b})$, $\mathbf{d}_0 = \mathbf{r}_0$
2. *For* $k = 0, 1, 2, \ldots$ *until* $\mathbf{r}$ *is sufficiently small, compute:*

$$\begin{aligned}
\lambda_k &= \mathbf{r}_k^T\mathbf{r}_k / \mathbf{d}_k^T\mathbf{A}\mathbf{d}_k \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + \lambda_k\mathbf{d}_k \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \lambda_k\mathbf{A}\mathbf{d}_k \\
\beta_k &= \mathbf{r}_{k+1}^T\mathbf{r}_{k+1} / \mathbf{r}_k^T\mathbf{r}_k \\
\mathbf{d}_{k+1} &= \mathbf{r}_{k+1} + \beta_k\mathbf{d}_k
\end{aligned}$$

Note here that only a few vectors are stored, the product $\mathbf{Ad}$ is required but not knowledge (or storage) of $\mathbf{A}$ *per se*, and the cost only involves several scalar and vector operations. The value of the step length λ_k can be derived in the closed form above by minimizing the quadratic function $q_A(\mathbf{x}_k + \lambda \mathbf{d}_k)$ as a function of λ (producing $\lambda_k = \mathbf{r}_k^T \mathbf{d}_k / \mathbf{d}_k^T \mathbf{A} \mathbf{d}_k$) and then using the conjugacy relation: $\mathbf{r}_k^T \mathbf{d}_j = 0$ for all $j < k$ [45].

See the linear algebra chapter for further details and examples.

Minimization performance of CG is shown in Figures 10 and 14 for Rosenbrock's and Beale's functions, respectively. Note that performance for both functions with CG is better than SD, as expected, but the paths are characteristically more 'random'. For Rosenbrock's function, for example, a large step is taken between the fourth and fifth iterates, where a distant minimum was detected in the line search. The step length varies considerably from one iteration to the next. Similarly, for Beale's function, the step lengths vary from $\mathcal{O}(10^{-4})$ to 800.

2.4.4 Preconditioning

Performance of the CG method is generally very sensitive to roundoff in the computations that may destroy the mutual conjugacy property. The method was actually neglected for many years until it was realized that a *preconditioning* technique can be used to accelerate convergence significantly [10, 21, 32].

Preconditioning involves modification of the target linear system $\mathbf{Ax} = -\mathbf{b}$ through application of a positive-definite *preconditioner* $\mathbf{M}$ that is closely related to $\mathbf{A}$. The modified system can be written as

$$\mathbf{M}^{-1/2} \mathbf{A} \mathbf{M}^{-1/2} (\mathbf{M}^{1/2} \mathbf{x}) = -\mathbf{M}^{-1/2} \mathbf{b} \ . \tag{35}$$

Essentially, the new coefficient matrix is $\mathbf{M}^{-1}\mathbf{A}$. Preconditioning aims to produce a more clustered eigenvalue structure for $\mathbf{M}^{-1}\mathbf{A}$ and/or lower condition number than for $\mathbf{A}$ to improve the relevant convergence ratio. However, preconditioning also adds to the computational effort by requiring that a linear system involving $\mathbf{M}$ (namely $\mathbf{Mz} = \mathbf{r}$) be solved at every step. Thus, it is essential for efficiency of the method that $\mathbf{M}$ be *factored* very rapidly in relation to the original $\mathbf{A}$. This can be achieved, for example, if $\mathbf{M}$ is a sparse component of the dense $\mathbf{A}$. Whereas the solution of an $n \times n$ dense linear system requires order of n^3 operations, the work for sparse systems can be as low as order n [11, 32].

The recurrence relations for the PCG method can be derived for Algorithm 2.2 after substituting $\mathbf{x}_k = \mathbf{M}^{-1/2} \tilde{\mathbf{x}}_k$ and $\mathbf{r}_k = \mathbf{M}^{1/2} \tilde{\mathbf{r}}_k$. New search vectors $\tilde{\mathbf{d}}_k = \mathbf{M}^{-1/2} \mathbf{d}_k$ can be used to derive the iteration process, and then the tilde modifiers dropped. The PCG method becomes the following iterative process.

Algorithm 2.3 *PCG Method to solve* $\mathbf{Ax} = -\mathbf{b}$.

1. Set $\mathbf{r}_0 = -(\mathbf{A}\mathbf{x}_0 + \mathbf{b})$, $\mathbf{d}_0 = \mathbf{M}^{-1} \mathbf{r}_0$

2. For $k = 0, 1, 2, \ldots$ *until* $\mathbf{r}$ *is sufficiently small, compute:*

$$
\begin{aligned}
\lambda_k &= \mathbf{r}_k^T(\mathbf{M}^{-1}\mathbf{r}_k)/\mathbf{d}_k^T\mathbf{A}\mathbf{d}_k \\
\mathbf{x}_{k+1} &= \mathbf{x}_k + \lambda_k\mathbf{d}_k \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \lambda_k\mathbf{A}\mathbf{d}_k \\
\beta_k &= \mathbf{r}_{k+1}^T(\mathbf{M}^{-1}\mathbf{r}_{k+1}) \,/\, \mathbf{r}_k^T(\mathbf{M}^{-1}\mathbf{r}_k) \\
\mathbf{d}_{k+1} &= (\mathbf{M}^{-1}\mathbf{r}_{k+1}) + \beta_k\mathbf{d}_k
\end{aligned}
\tag{36}
$$

Note above that the system $\mathbf{M}\mathbf{z}_k = \mathbf{r}_k$ must be solved repeatedly for $\mathbf{z}_k$ and that the matrix/vector products $\mathbf{A}\mathbf{d}_k$ are required as before.

2.4.5 Nonlinear Conjugate Gradient

Extensions of the linear CG method to nonquadratic problems have been developed and extensively researched [23, 52, 61]. In the common variants, the basic idea is to avoid matrix operations altogether and simply express the search directions recursively as

$$\mathbf{d}_k = -\mathbf{g}_k + \beta_k\mathbf{d}_{k-1} \ , \tag{37}$$

for $k = 1, 2, \ldots$, with $\mathbf{d}_0 = -\mathbf{g}_0$. The new iterates for the minimum point can then be set to

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda_k\mathbf{d}_k \ , \tag{38}$$

where λ_k is the step length. In comparing this iterative procedure to the linear CG of Algorithm 2.2, we note that $\mathbf{r}_k = -\mathbf{g}_k$ for a quadratic function. The parameter β_k above is chosen so that if f were a convex quadratic and λ_k is the exact one-dimensional minimizer of f along $\mathbf{d}_k$, the nonlinear CG reduces to the linear CG method and terminates in at most n steps in exact arithmetic.

Three of the best known formulas for β_k are titled Fletcher-Reeves (FR), Polak-Ribière (PR), and Hestenes-Stiefel (HS) after their developers. They are given by the following formulas:

$$\beta_k^{FR} = \mathbf{g}_k^T\mathbf{g}_k \,/\, \mathbf{g}_{k-1}^T\mathbf{g}_{k-1} \tag{39}$$

$$\beta_k^{PR} = \mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})/\mathbf{g}_{k-1}^T\mathbf{g}_{k-1} \tag{40}$$

$$\beta_k^{HS} = \mathbf{g}_k^T(\mathbf{g}_k - \mathbf{g}_{k-1})/\mathbf{d}_{k-1}^T(\mathbf{g}_k - \mathbf{g}_{k-1}) \tag{41}$$

Interestingly, the last two formulas are generally preferred in practice, though the first has better theoretical global convergence properties. In fact, very recent research has focused on combining these practical and theoretical properties for construction of more efficient schemes. The simple modification of

$$\beta_k = \max\{\beta_k^{PR}, 0\} \ , \tag{42}$$

for example, can be used to prove global convergence of this nonlinear CG method, even with inexact line searches [26]. A more general condition on β_k, including relaxation of its nonnegativity, has also been derived.

The quality of line search in these nonlinear CG algorithms is crucial to preserve the mutual conjugacy property of the search directions and to ensure that each generated direction is one of descent. A technique known as *restarting* is typically used to preserve a linear convergence rate by resetting $\mathbf{d}_k$ to the steepest descent direction, for example, after a given number of linear searches (e.g., n). Preconditioning may also be used as in the linear case to accelerate convergence.

In sum, the greatest virtues of CG methods are their modest storage and computational requirements (both order n), combined with much better convergence than SD. These properties have made them popular linear-solvers and minimization choices in many applications, perhaps the only candidates for very large problems. The linear CG is often applied to systems arising from discretizations of partial differential equations where the matrices are frequently positive-definite, sparse, and structured [20, 25].

2.5 Newton Methods

2.5.1 Newton Methods Overview

Newton methods include several classes, including *discrete* Newton, *quasi* Newton (QN) (also termed *variable metric*), and *truncated* Newton (TN). Historically, the $\mathcal{O}(n^2)$ memory requirements and $\mathcal{O}(n^3)$ computation associated with solving a linear system directly have restricted Newton methods only: (1) to small problems, (2) to problems with special sparsity patterns, or (3) near a solution, after a gradient method has been applied. Fortunately, advances in computing technology and software are making the Newton approach feasible for a wide range of problems. Particularly, with advances in automatic differentiation [35, 53], the appeal of these methods should increase further. Extensive treatments of Newton methods can be found in the literature ([16, 23, 30, 45], for example) and only general concepts will be outlined here.

For large-scale applications, essentially two specific classes are emerging as the most powerful techniques: *limited-memory* quasi-Newton (LMQN) and truncated Newton methods. LMQN methods attempt to combine the modest storage and computational requirements of CG methods with the superlinear convergence properties of standard (i.e., full memory) QN methods. Similarly, TN algorithms attempt to retain the rapid quadratic convergence rate of classic Newton methods while making computational requirements feasible for large-scale functions.

All Newton methods are based on approximating the objective function locally by a quadratic model and then minimizing that function approximately. The quadratic model of the objective function f at $\mathbf{x}_k$ along $\mathbf{p}$ is given by the expansion

$$f(\mathbf{x}_k + \mathbf{p}) \approx f(\mathbf{x}_k) + \mathbf{g}_k^T \mathbf{p} + \frac{1}{2} \mathbf{p}^T \mathbf{H}_k \mathbf{p} \ . \tag{43}$$

The minimum of the right-hand side is achieved when $\mathbf{p}_k$ is the minimum of the quadratic function:

$$q_{H_k}(\mathbf{p}) = \mathbf{g}_k^T \mathbf{p} + \frac{1}{2} \mathbf{p}^T \mathbf{H}_k \mathbf{p} \ . \tag{44}$$

Table 3: Newton's Iteration for Solving $f(x) = x^2 - a = 0$

Single Precision		Double Precision	
x	$\lvert x - s\rvert/s$	x	$\lvert x - s\rvert/s$
6.000000	$1.0000000E+00$	6.000000000000000	$1.0000000000000000E+00$
3.750000	$2.5000000E-01$	3.750000000000000	$2.5000000000000000E-01$
3.075000	$2.5000015E-02$	3.075000000000000	$2.5000000000000004E-02$
3.000915	$3.0485788E-04$	3.000914634146342	$3.0487804878050658E-04$
3.000000	$7.9472862E-08$	3.000000139383442	$4.6461147336825566E-08$
3.000000	$0.0000000E+00$	3.000000000000003	$1.0732155904709847E-15$
3.000000	$0.0000000E+00$	3.000000000000000	$1.8503717077085942E-17$
3.000000	$0.0000000E+00$	3.000000000000000	$0.0000000000000000E+00$
(x = iterate, s = solution)		(x = iterate, s = solution)	

Alternatively, such a *Newton direction* $\mathbf{p}_k$ satisfies the linear system of n simultaneous equations, known as the *Newton equation*:

$$\mathbf{H}_k \mathbf{p} = -\mathbf{g}_k \; . \tag{45}$$

In the "classic" Newton method, the Newton direction is used to update each previous iterate by the formula $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$, until convergence. The reader may recognize the one-dimensional version of Newton's method for solving a nonlinear equation $f(x) = 0$: $x_{k+1} = x_k - f(x_k)/f'(x_k)$. The analogous iteration process for minimizing $f(x)$ is: $x_{k+1} = x_k - f'(x_k)/f''(x_k)$. Note that the one-dimensional search vector $(-f'(x_k)/f''(x_k))$ is replaced by the Newton direction $-\mathbf{H}_k^{-1}\mathbf{g}_k$ in the multivariate case. This direction is defined for nonsingular $\mathbf{H}_k$ but its solution may be unstable. When $\mathbf{x}_0$ is sufficiently close to a solution $\mathbf{x}^*$, quadratic convergence can be proven for Newton's method [16, 23, 45]. In practice, this means that the number of digits of accuracy in the solution is approximately doubled at every step! This rapid convergence can be seen from the program output for a simple one-dimensional application of Newton's method to finding the root of a (equivalently, solving $f(x) = x^2 - a = 0$ or minimizing $f(x) = x^3/3 - ax$) (see Table 3). See the linear algebra chapter for related details. Note in the double precision version the round-off in the last steps.

See exercise 7 and 8.

In practice, modifications of the classic Newton iteration are essential for guaranteeing global convergence, with quadratic convergence rate near the solution. First, when $\mathbf{H}_k$ is not positive-definite, the search direction may not exist or may not be a descent direction. Strategies to produce a related positive-definite matrix $\overline{\mathbf{H}}_k$, or alternative search directions, become necessary. Second, far away from $\mathbf{x}^*$, the quadratic approximation of (43) may be poor, and the Newton direction must be adjusted. A line search, for example, can *dampen* (scale) the Newton direction when it exists, ensuring sufficient decrease and guaranteeing uniform progress towards a solution. These modifications lead to the following *modified* Newton framework using a line search:

Algorithm 2.4 *Modified Newton*

$*$ *For* $k = 0, 1, 2, \ldots,$ *until convergence, given* $\mathbf{x}_0$*:*

1. Test $\mathbf{x}_k$ *for convergence*

2. Compute a descent direction $\mathbf{p}_k$ *so that*

$$\|\mathbf{H}_k \mathbf{p}_k + \mathbf{g}_k\| \leq \eta_k \|\mathbf{g}_k\| \ , \tag{46}$$

where η_k *controls the accuracy of the solution and some symmetric matrix* $\overline{\mathbf{H}}_k$ *may approximate* $\mathbf{H}_k$*.*

3. Compute a step length λ *so that for* $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda \mathbf{p}_k$*,*

$$f(\mathbf{x}_{k+1}) \ \leq \ f(\mathbf{x}_k) + \alpha \lambda \mathbf{g}_k^T \mathbf{p} \ , \tag{47}$$

$$|\mathbf{g}_{k+1}^T \mathbf{p}_k| \ \leq \ \beta |\mathbf{g}_k^T \mathbf{p}_k| \ , \tag{48}$$

with $0 < \alpha < \beta < 1$*.*

4. Set $\mathbf{x}_{k+1} = \mathbf{x}_k + \lambda\, \mathbf{p}_k$*.*

Newton variants are constructed by combining various strategies for the individual components above. These involve procedures for formulating $\mathbf{H}_k$ or $\overline{\mathbf{H}}_k$, dealing with structures of indefinite Hessians, and solving for the *modified Newton* search direction. For example, when $\mathbf{H}_k$ is approximated by *finite differences*, the discrete Newton subclass emerges [16]. When $\mathbf{H}_k$, or its inverse, is approximated by some modification of the previously constructed matrix (see below), QN methods are formed [16, 17]. When η_k is nonzero, TN methods result [18, 47, 55, 56, 57] since the solution of the Newton system is *truncated* before completion.

2.5.2 Discrete Newton

Standard discrete Newton methods require n gradient evaluations and $\mathcal{O}(n^2)$ operations to compute and symmetrize every Hessian $\mathbf{H}_k$. Each column i of $\mathbf{H}_k$ can be approximated by the vector

$$\tilde{\mathbf{h}}_i = \frac{1}{h_i} [\mathbf{g}(\mathbf{x}_k + h_i \mathbf{e_i}) - \mathbf{g}_k] \ , \tag{49}$$

where h_i is a suitably chosen interval [30]. This interval must balance the *roundoff* error, proportional to $(1/h_i)$, by formulation, with the *truncation* error, proportional to h_i. A simple estimate for a well-scaled problem to balance the two errors is $\mathcal{O}(\sqrt{\epsilon_m})$.

See exercise 9.

To symmetrize the resulting matrix $\widetilde{\mathbf{H}}_k$, whose columns are the vectors $\{\tilde{\mathbf{h}}_i\}$, $i = 1, \ldots, n$, from (45) the matrix

$$\overline{\mathbf{H}}_k = \frac{1}{2}(\widetilde{\mathbf{H}}_k + \widetilde{\mathbf{H}}_k^T) \tag{50}$$

is constructed.

With exact arithmetic, discrete Newton methods converge quadratically if each h_i goes to zero as $\|\mathbf{g}\|$ does [16]. However, the roundoff error limits the smallest feasible size of difference interval in practice and, hence, the accuracy (a combination of roundoff and truncation errors) that can be obtained. As the gradient becomes very small, considerable loss of accuracy may also result from *cancellation* errors in the numerator (a large relative error, $|s - s^*|/|s^*|$, from the subtraction of two quantities, s and s^*, of similar magnitude). Consequently, discrete Newton methods are inappropriate for large-scale problems unless the Hessian has a known sparsity structure and this structure is exploited in the differencing scheme.

2.5.3 Quasi Newton

QN methods form an interesting class of algorithms that are theoretically closely-related to nonlinear CG methods and to perform well in practice. Surprising recent discoveries yet increased their appeal.

As extensions of nonlinear CG methods, QN methods are additional curvature information to accelerate convergence. However, memory and computational requirements are kept as low as possible. Essentially, curvature information is built up progressively. At each step of the algorithm, the current approximation to the Hessian (or inverse Hessian) is updated by using new gradient information. (The updated matrix itself is not necessarily stored explicitly, since the updating procedure may be defined compactly in terms of a small set of stored vectors.) For the expansion of the gradient:

$$\mathbf{g}_{k+1} = \mathbf{g}_k + \mathbf{H}_k(\mathbf{x}_{k+1} - \mathbf{x}_k) + \cdots \ , \tag{51}$$

we obtain the following relation if $\mathbf{H}_k$ were a constant (equivalently, $f(\mathbf{x})$ a quadratic), equal to $\mathbf{H}$:

$$\mathbf{H}\mathbf{s}_k = \mathbf{y}_k \ , \tag{52}$$

where $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. Since each gradient difference $\mathbf{y}_k$ provides information about one *column* of $\mathbf{H}$, we attempt to construct a family of successive approximation matrices $\{B_k\}$ so that, if $\mathbf{H}$ were a constant, the procedure would be consistent with equation (52). This forms the basis for the *QN condition* on the new update, $\mathbf{B}_{k+1}$:

$$\mathbf{B}_{k+1}\mathbf{s}_k = \mathbf{y}_k \ . \tag{53}$$

To specify $\mathbf{B}_{k+1}$ uniquely additional conditions are required. For instance, it is reasonable to assume that $\mathbf{B}_{k+1}$ differs from $\mathbf{B}_k$ by a *low rank* 'updating' matrix (i.e., a matrix of rank much less than n) that depends on $\mathbf{s}_k$, $\mathbf{y}_k$, and possibly $\mathbf{B}_k$:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{U}_k(\mathbf{s}_k, \mathbf{y}_k, \mathbf{B}_k) \ . \tag{54}$$

For an update of rank one, written as $\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{u}\mathbf{v}^T$, we obtain from (53) the condition that $\mathbf{u}$ is a vector in the direction of $(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)$. This produces the *general rank one* update formula as:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{1}{\mathbf{v}^T\mathbf{s}_k}(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)\mathbf{v}^T \ , \tag{55}$$

for $\mathbf{v}^T\mathbf{s}_k \neq 0$. Broyden's QN method, for example, uses $\mathbf{v} = \mathbf{s}_k$. While Broyden's update does not guarantee symmetry, it is useful for solving nonlinear equations and for deriving a more effective, rank two update. To restrict the general rank one update form of (53) further, we can impose the condition of symmetry. Symmetry will be "inherited" from $\mathbf{B}_k$ to $\mathbf{B}_{k+1}$ if $\mathbf{u}\mathbf{v}^T = \alpha\mathbf{u}\mathbf{u}^T$ for some scalar α. Letting that α be $1/\mathbf{v}^T\mathbf{s}_k$, we obtain the *general symmetric rank 1* update (SR1) as follows:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{1}{(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)^T\mathbf{s}_k}(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)^T \ . \tag{56}$$

Now, SR1 will only be positive-definite if $(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)^T\mathbf{s}_k > 0$. Thus, rank two updates (e.g., $\mathbf{u}_1\mathbf{v}_1^T + \mathbf{u}_2\mathbf{v}_2^T$) were thought until very recently to be more suitable for optimization.

One of the most successful and widely used updating formulas is known as BFGS for its four developers, Broyden, Fletcher, Goldfarb, and Shanno. It is a rank two update with inherent positive-definiteness (i.e., $\mathbf{B}_k$ positive-definite $\Rightarrow$ $\mathbf{B}_{k+1}$ positive-definite), and was derived by "symmetrizing" the Broyden rank one update. A sequence of matrices $\{\widetilde{\mathbf{B}}_k\}$ is generated from a positive-definite $\widetilde{\mathbf{B}}_0$ (which may be taken as the identity), by the following BFGS formula:

$$\widetilde{\mathbf{B}}_{k+1} = \widetilde{\mathbf{B}}_k + U_k \ , \tag{57}$$

where

$$\mathbf{U}(\mathbf{s}, \mathbf{y}, \widetilde{\mathbf{B}}) = \frac{\mathbf{s}\mathbf{s}^T}{\mathbf{y}^T\mathbf{s}}\left[\frac{\mathbf{y}^T\widetilde{\mathbf{B}}\mathbf{y}}{\mathbf{y}^T\mathbf{s}} + 1\right] - \frac{1}{\mathbf{y}^T\mathbf{s}}[\mathbf{s}\mathbf{y}^T\widetilde{\mathbf{B}} + \widetilde{\mathbf{B}}\mathbf{y}\mathbf{s}^T] \ . \tag{58}$$

In Figures 11 and 15 QN progress for Rosenbrock's and Beale's functions is shown. The QN code is taken from the package CONMIN of Shanno & Phua available in the NAG library. The QN paths are clearly more systematic than those associated with the gradient methods, following down the valley. Behavior of the limited- memory QN methods is essentially identical for these two-dimensional problems.

We refer the reader to additional papers [27, 44] discussing state-of-the-art LMQN methods, along with practical schemes for initial scaling ($\tilde{\mathbf{B}}_0$) and for calculating each $\mathbf{p}_k$ efficiently by exploiting the low-rank update procedure.

2.5.4 Truncated Newton

Truncated Newton methods were introduced in the early 1980s [18] and have since been gaining popularity. They are based on the idea that an exact solution of the Newton equation at every step is unnecessary and can be computationally wasteful in the framework of a basic descent method. Any descent direction will suffice when the objective function is not well approximated by a convex quadratic and, as a solution to the minimization problem is approached, more effort in solution of the Newton equation may be warranted. Their appeal to scientific applications is their ability to exploit function structure to accelerate convergence.

Thus, the *approximate* a nonzero residual norm $r_k = \|\mathbf{r}_k\| = \|\mathbf{H}_k\mathbf{p}_k + \mathbf{g}_k\|$ is allowed at each step, its size monitored systematically according to the progress made. This formulation leads to a doubly-nested iteration structure: for every outer Newton iteration k (associated with $\mathbf{x}_k$) there corresponds an inner loop for $\mathbf{p}_k$ $\{\mathbf{p}_k^0, \mathbf{p}_k^1, \ldots\}$. As a computationally-economical method for solving large positive-definite linear systems, PCG is the most suitable method for the inner loop in this context. Function structure is introduced by using a preconditioner $\mathbf{M}$ that is a sparse approximation to the Hessian.

The PCG process can be terminated when either one of the following conditions is satisfied: (1) The residual r_k is sufficiently small, (2) the quadratic model $q_{H_k}(\mathbf{p}_k^i)$ of eq. (34) is sufficiently reduced, or (3) a direction of *negative curvature* $\mathbf{d}$ is encountered (i.e., $\mathbf{d}^T\mathbf{H}_k\mathbf{d}_k < 0$), possible since $\mathbf{H}_k$ may not be positive-definite. Two effective truncation tests monitor the relative residual norm (RT) [18] and the decrease of the quadratical model $\mathbf{q}_{H_k}(\mathbf{p})$ (QT) [47].

The inner loop of a TN algorithm at Newton step k (step 2 of algorithm 2.4) can then be sketched as follows. For clarity, we omit the subscript k from $\mathbf{p}$, $\mathbf{g}$, $\mathbf{H}$, $\mathbf{M}$, and q. The sequence of vectors $\{\mathbf{p}^i\}$ denotes the PCG iterates for $\mathbf{p}_k$, and a small positive number δ for the negative curvature test, such as $\sqrt{\epsilon_m}$, is chosen, along with appropriate values of c_r or c_q (for truncation), around 0.5.

Algorithm 2.5 *Truncated Newton (Inner Loop of Outer Step k)*

0. Initialization.

Set $\mathbf{p}^0 = 0$, $q^0 \equiv q_k(\mathbf{p}^0) = 0$, $\mathbf{r}_0 = -\mathbf{g}$, *and* $\mathbf{d}_0 = \mathbf{M}^{-1}\mathbf{r}_0$.

For $i = 0, 1, 2, \ldots$ *proceed as follows:*

1. Negative Curvature Test.

If $\mathbf{d}_i^T\mathbf{H}\mathbf{d}_i < \delta\mathbf{d}_i^T\mathbf{d}_i$,

exit inner loop with search direction

$$\mathbf{p} = \begin{cases} \mathbf{d}_0 & \textit{if } i = 0 \\ \mathbf{p}^i & \textit{otherwise} \end{cases} \tag{59}$$

2. Truncation Test.

$$\begin{aligned} \alpha_i &= \mathbf{r}_i^T(\mathbf{M}^{-1}\mathbf{r}_i)/\mathbf{d}_i^T\mathbf{H}\mathbf{d}_i \\ \mathbf{p}^{i+1} &= \mathbf{p}^i + \alpha_i\mathbf{d}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - \alpha_i\mathbf{H}\mathbf{d}_i \\ q^{i+1} &= (1/2)(\mathbf{r}_{i+1} + \mathbf{g})^T\,\mathbf{p}^{i+1} \qquad \textit{(for QT)} \end{aligned}$$

If $\|\mathbf{r}_{i+1}\| \leq \min\{c_r/k,\ \|\mathbf{g}\|\} \cdot \|\mathbf{g}\|$ *(for RT)*
or *If* $(1 - q^i/q^{i+1}) \leq c_q/i$ *(for QT)*
exit inner loop with search direction $\mathbf{p} = \mathbf{p}^{i+1}$
3. Continuation of PCG.

$$\begin{aligned} \beta_i &= \mathbf{r}_{i+1}^T(\mathbf{M}^{-1}\mathbf{r}_{i+1})/\mathbf{r}_i^T(\mathbf{M}^{-1}\mathbf{r}_i) \\ \mathbf{d}_{i+1} &= (\mathbf{M}^{-1}\mathbf{r}_{i+1}) + \beta_i\mathbf{d}_i \end{aligned} \tag{60}$$

Note that in case of negative curvature, $\mathbf{p}$ is set in step 1 to $-\mathbf{M}^{-1}\mathbf{g}$ if this occurs at the first PCG iteration, or to the current iterate for $\mathbf{p}_k$ thereafter. These choices are guaranteed direction of descent [18]. Alternate descent directions can also be used, such as $-\mathbf{g}$ or $\mathbf{d}_i$, but the 'default' choices above have been found to be satisfactory in practice.

For computational efficiency, the products $\mathbf{Hd}$ in step (2) can generally be computed satisfactorily by the following finite-difference design of gradients at the expense of only one additional gradient evaluation per inner iteration:

$$\mathbf{H}_k\mathbf{d} \approx \frac{\mathbf{g}(\mathbf{x}_k + h\mathbf{d}) - \mathbf{g}(\mathbf{x}_k)}{h}\ , \tag{61}$$

where h is a suitably chosen small number, such as

$$h = \frac{2\sqrt{\epsilon_m}(1 + \|\mathbf{x}_k\|)}{\|\mathbf{d}\|}\ . \tag{62}$$

The paths of TN minimization by the package TNPACK of Schlick & Fogelson [56, 57] for Rosenbrock's and Beale's functions are shown in Figure 12 and Figure 16. Note again how efficiently the paths trace the valley toward the minimum and appear even more direct than QN.

For discussions on practical ways of choosing $\mathbf{M}$, factoring $\mathbf{M}$ when the problem does not guarantee that $\mathbf{M}$ is positive definite by the modified Cholesky factorization [29, 58, 60], and performing efficient Hessian/vector products, see [55, 56, 57].

See exercise 10.

---

3 Methods for Constrained Continuous Multivariate Problems

Techniques for constrained nonlinear programming problems are clearly more challenging than their unconstrained analogs, and the best approaches to use are still unknown. When the constraints are linear, the problems are simpler, and a better framework for analysis is available. In general, algorithms for constrained problems are based on the optimization methods for the unconstrained case, as introduced in the previous section. The basic approach for solution of a constrained nonlinear multivariate problem is to simplify the problem so it can be reformulated by a sequence of related problems, each of which can be solved by more familiar methods for the unconstrained case. For example, Newton methods for unconstrained problems based on a local quadratic approximation to the objective function may be developed for the constrained problem by restricting the region in which the quadratic model is valid. This restriction is similar in flavor to trust- region methods, mentioned in section 2.2.2. A search vector $\mathbf{p}_k$ for the objective function might be chosen by a similar procedure to that described for the unconstrained case (e.g., a Quasi Newton method), but the step length along $\mathbf{p}_k$ might be restricted so that the next iterate is a feasible point, i.e., satisfies the constraints of the problem.

An excellent introduction to the optimization of constrained nonlinear functions can be found in the chapter by Gill et al. (pp. 171-210) in the optimization volume edited by Nemhauser et al. [48].

4 Methods for Integer and Mixed Integer Multivariate Problems

This section will appear in the next release of this chapter.

5 Methods of Last Resort

Unfortunately, there are many optimization problems which cannot be satisfactorily solved using any of the foregoing algorithms — inevitably many problems of practical interest fall into this category. When these systematic search methods fail, one must resort to non-systematic, i.e., random search, techniques. The algorithms discussed in this section all employ some form of random search.

5.1 Simulated Annealing

As its name implies, the Simulated Annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system.

The algorithm is based upon that of Metropolis et al. [46], which was originally proposed as a means of finding the equilibrium configuration of a collection of atoms at a given temperature. The connection between this algorithm and mathematical minimization was first noted by Pincus [50], but it was Kirkpatrick et al. [40] who proposed that it form the basis of an optimization technique for combinatorial (and other) problems.

SA's major advantage over other methods is an ability to avoid becoming trapped at local minima. The algorithm employs a random search which not only accepts changes that decrease objective function f, but also some changes that increase it. The latter are accepted with a probability

$$p = exp\left(-\frac{\delta f}{T}\right), \tag{63}$$

where δf is the increase in f and T is a control parameter, which by analogy with the original application is known as the system 'temperature' irrespective of the objective function involved.

The implementation of the SA algorithm is remarkably easy. Figure 17 shows its basic structure. The following elements must be provided:

- a representation of possible solutions,
- a generator of random changes in solutions,
- a means of evaluating the problem functions, and
- an *annealing schedule* - an initial temperature and rules for lowering it as the search progresses.

5.1.1 Solution Representation and Generation

When attempting to solve an optimization problem using the SA algorithm, the most obvious representation of the control variables is usually appropriate. However, the way in which new solutions are generated may need some thought. The solution generator should

- introduce small random changes, and
- allow all possible solutions to be reached.

For problems with continuous control values, Vanderbilt and Louie [62] recommend that new trial solutions be generated according to the formula:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{Q}\mathbf{u}, \tag{64}$$

where $\mathbf{u}$ is a vector of random numbers in the range $(-\sqrt{3}, \sqrt{3})$, so that each has zero mean and unit variance, and $\mathbf{Q}$ is a matrix that controls the step size distribution. In order to generate random steps with a covariance matrix $\mathbf{S}$, $\mathbf{Q}$ is found by solving

$$\mathbf{S} = \mathbf{Q}\mathbf{Q}^T. \tag{65}$$

by Cholesky decomposition, for example (see the chapter on Linear Algebra for more details of this method). $\mathbf{S}$ should be updated as the search progresses to include information about the local topography:

$$\mathbf{S}_{i+1} = (1-\alpha)\mathbf{S}_i + \alpha\omega\mathbf{X}, \tag{66}$$

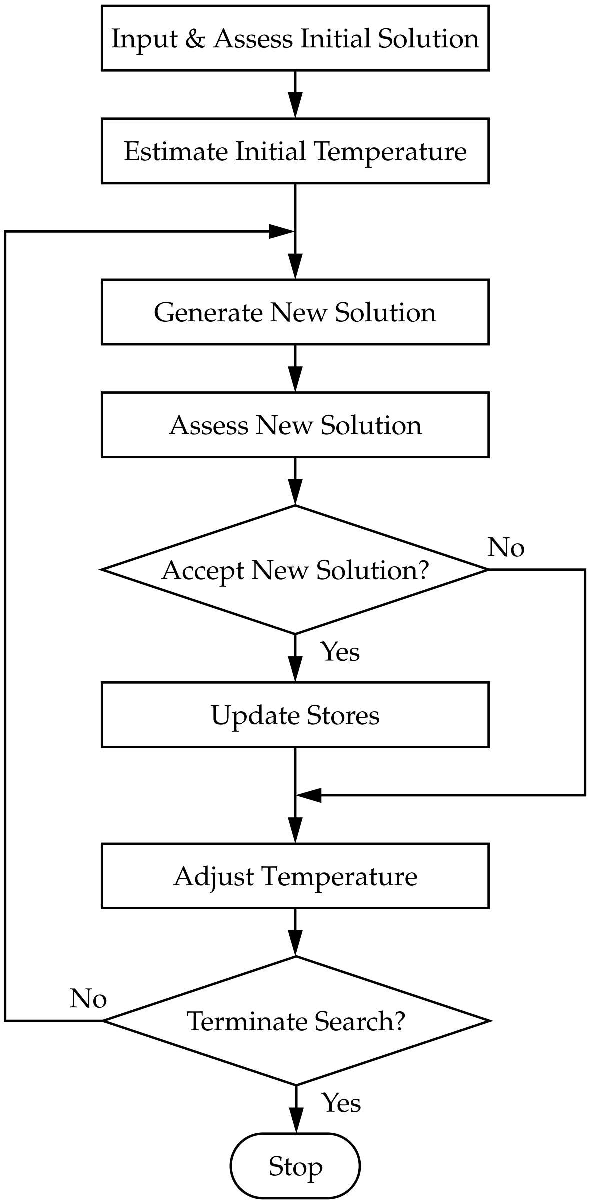

Figure 17: The Structure of the Simulated Annealing Algorithm

where matrix $\mathbf{X}$ measures the covariance of the path actually followed and the damping constant α controls the rate at which information from $\mathbf{X}$ is folded into $\mathbf{S}$ with weighting ω. One drawback of this scheme is that the solution of equation (65), which must be done every time $\mathbf{S}$ is updated, can represent a substantial computational overhead for problems with high dimensionality. In addition, because the probability of an objective function increase being accepted, given by equation (63), does not reflect the size of the step taken, $\mathbf{S}$ must be estimated afresh every time the system temperature is changed.

An alternative strategy suggested by Parks [49] is to generate solutions according to the formula:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{D}\mathbf{u}, \tag{67}$$

where $\mathbf{u}$ is now a vector of random numbers in the range $(-1, 1)$ and $\mathbf{D}$ is a diagonal matrix which defines the maximum change allowed in each variable. After a successful trial, i.e. after an accepted change in the solution, $\mathbf{D}$ is updated:

$$\mathbf{D}_{i+1} = (1 - \alpha)\mathbf{D}_i + \alpha\omega\mathbf{R}, \tag{68}$$

where α and ω perform similar roles and $\mathbf{R}$ is a diagonal matrix the elements of which consist of the magnitudes of the successful changes made to each control variable. This tunes the maximum step size associated with each control variable towards a value giving acceptable changes.

When using this strategy it is recommended that the probability of accepting an increase in f be changed from that given by equation (63) to:

$$p = exp\left(-\frac{\delta f}{T\bar{d}}\right), \tag{69}$$

where $\bar{d}$ is the average step size, so that $\delta f/\bar{d}$ is a measure of the effectiveness of the change made. As the size of the step taken is considered in calculating p, $\mathbf{D}$ does not need to be adjusted when T is changed.

For problems with integer control variables, the simple strategy whereby new trial solutions are generated according to the formula:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{u}, \tag{70}$$

where $\mathbf{u}$ is a vector of random integers in the range $(-1, 1)$ often suffices.

For combinatorial, or permutation, optimization problems, the solution representation and generation mechanism(s) will necessarily be problem-specific. For example, in the famous *traveling salesman problem* (TSP) of finding the shortest cyclical itinerary visiting N cities, the most obvious representation is a list of the cities in the order they are to be visited. The solution generation mechanism(s), or *move set*, must, obviously, be compatible with the chosen representation. For combinatorial problems, it is common for the move set to permute a small, randomly chosen, part of the solution. For example, the move set suggested by Lin [43] for the TSP makes two types of change:

- a section of the route is removed and replaced by the same cities running in the opposite order, e.g. 87|164|253 → 87|461|253; or
- a section of the route is moved (cut and pasted) from one part of the tour to another, e.g. 87|164|253 → 8725|164|3.

5.1.2 Solution Evaluation

The SA algorithm does not require or deduce derivative information, it merely needs to be supplied with an objective function for each trial solution it generates. Thus, the evaluation of the problem functions is essentially a 'black box' operation as far as the optimization algorithm is concerned. Obviously, in the interests of overall computational efficiency, it is important that the problem function evaluations should be performed efficiently, especially as in many applications these function evaluations are overwhelming the most computationally intensive activity. Depending on the nature of the system equations advice on accelerating these calculations may be found in other chapters within this project.

Some thought needs to be given to the handling of constraints when using the SA algorithm. In many cases the routine can simply be programmed to reject any proposed changes which result in constraint violation, so that a search of feasible space only is executed. However, there are two important circumstances in which this approach cannot be followed:

- if there are any equality constraints defined on the system, or
- if the feasible space defined by the constraints is (suspected to be) disjoint, so that it is not possible to move between all feasible solutions without passing through infeasible space.

In either case the problem should be transformed into an unconstrained one by constructing an augmented objective function incorporating any violated constraints as penalty functions:

$$f_A(\mathbf{x}) = f(\mathbf{x}) + \frac{1}{T}\mathbf{w}^T\mathbf{c}_V(\mathbf{x}), \tag{71}$$

where $\mathbf{w}$ is a vector of nonnegative weighting coefficients and the vector $\mathbf{c}_V$ quantifies the magnitudes of any constraint violations. The inverse dependence of the penalty on temperature biases the search increasingly heavily towards feasible space as it progresses. Consult section 4 for more details on the penalty function method.

5.1.3 Annealing Schedule

Through equation (63) or (69), the annealing schedule determines the degree of uphill movement permitted during the search and is, thus, critical to the algorithm's performance. The principle underlying the choice of a suitable annealing schedule is easily stated — the initial temperature should be high enough to 'melt' the system completely and should be reduced towards its 'freezing point' as the search progresses — but "choosing an annealing schedule for practical purposes is still something of a black art" (Bounds, [6]).

The standard implementation of the SA algorithm is one in which homogeneous Markov chains of finite length [4] are generated at decreasing temperatures. The following parameters should therefore be specified:

- an initial temperature T_0;
- a final temperature T_f or a stopping criterion;
- a length for the Markov chains; and
- a rule for decrementing the temperature.

Initial Temperature

Kirkpatrick [41] suggested that a suitable initial temperature T_0 is one that results in an average increase acceptance probability χ_0 of about 0.8.[5] The value of T_0 will clearly depend on the scaling of f and, hence, be problem-specific. It can be estimated by conducting an initial search in which all increases are accepted and calculating the average objective increase observed $\bar{\delta f}^+ T_0$ is then given by:

$$T_0 = -\frac{\bar{\delta f}^+}{ln(\chi_0)}. \tag{72}$$

Final Temperature

In some simple implementations of the SA algorithm the final temperature is determined by fixing

- the number of temperature values to be used, or
- the total number of solutions to be generated.

Alternatively, the search can be halted when it ceases to make progress. Lack of progress can be defined in a number of ways, but a useful basic definition is

- no improvement (i.e. no new best solution) being found in an entire Markov chain at one temperature,

combined with

- the acceptance ratio falling below a given (small) value χ_f.

The sample code supplied with this section uses this definition of convergence (lack of progress).

Length of Markov Chains

[4]In this context an homogeneous Markov chain is a series of random changes in the control variables.

[5]In other words, there is an 80% chance that a change which increases the objective function will be accepted.

An obvious choice for L_k, the length of the k-th Markov chain, is a value that depends on the size of the problem, so L_k is independent of k. Alternatively it can be argued that a minimum number of transitions η_{min} should be accepted at each temperature. However, as T_k approaches 0, transitions are accepted with decreasing probability so the number of trials required to achieve η_{min} acceptances approaches ∞. Thus, in practice, an algorithm in which each Markov chain is terminated after

- L_k transitions or
- η_{min} acceptances,

whichever comes first, is a suitable compromise.

Decrementing the Temperature

The simplest and most common temperature decrement rule is:

$$T_{k+1} = \alpha T_k, \tag{73}$$

where α is constant close to, but smaller than, 1. This *exponential cooling scheme* (ECS) was first proposed Kirkpatrick et al. [39] with $\alpha = 0.95$. Randelman and Grest [54] compared this strategy with a *linear cooling scheme* (LCS) in which T is reduced every L trials:

$$T_{k+1} = T_k - \Delta T. \tag{74}$$

They found the reductions achieved using the two schemes to be comparable, and also noted that the final value of f was, in general, improved with slower cooling rates, at the expense, of course, of greater computational effort. Finally, they observed that the algorithm performance depended more on the cooling rate $\Delta T/L$ than on the individual values of ΔT and L. Obviously, care must be taken to avoid negative temperatures when using the LCS.

Many researchers have proposed more elaborate annealing schedules, most of which are in some respect adaptive, using statistical measures of the algorithm's current performance to modify its control parameters. These are well reviewed by van Laarhoven and Aarts [42].

5.1.4 SA Computational Considerations

As the procedures controlling the generation and acceptance of new solutions are so simple, the computational cost of implementing the SA algorithm is almost invariably dominated by that associated with the evaluation of the problem functions. It is essential that these evaluations should be performed as efficiently as possible. Sometimes, as, for example, in the Nuclear Fuel Management Case Study presented elsewhere in this project, it is possible to use a Generalized Perturbation Theory method to expedite these calculations. However, the viability of such an approach will obviously be problem-dependent. In general, any efforts to improve performance (reduce run times) should be directed towards exploiting the vectorization or parallelization capabilities of the intended computational platform to accelerate the problem function evaluations. Advice on how this can be achieved can be found in the chapter appropriate to the form of the system equations to be solved.

SA is an intrinsically sequential algorithm, due to its recursive structure. Some possible parallel designs (reviewed by Arts and Korst [2]) have been developed, making use of the idea of *multiple trial parallelism*, in which several different trial solutions are simultaneously generated, evaluated and tested on individual processors. Whenever one of these processors accepts its solution, it becomes the new one from which others are generated. Although such an approach results in N times as many solutions being investigated per unit time (where N is the number of processors), it is found that the total (elapsed) time required for convergence is not proportionally reduced. This is due to the fact that the instantaneous concurrent efficiency h, which in this context can be defined as

$$\eta = \frac{\delta t_S}{N \delta t_P}, \tag{75}$$

where δt is the time taken for a new solution to be accepted by the serial or parallel algorithm, varies as the search progresses. It is initially only about $1/N$, because the vast majority of the solutions generated are accepted, and, therefore, $N-1$ of the processors are redundant. As the annealing temperature is reduced (and the solution acceptance probability falls), η increases, approaching 100% as T nears zero. However, the overall incentive for parallelizing the optimization scheme is not great, especially as in many instances the problem function evaluation procedure can be multitasked with much greater ease and effect.

A significant component of an SA code is the random number generator, which is used both for generating random changes in the control variables and for the temperature dependent increase acceptance test. (See, for example, the sample code supplied with this section.) Random number generators are often provided in standard function libraries or as machine-specific functions. It is important, particularly when tackling large scale problems requiring thousands of iterations, that the random number generator used have good spectral properties — see the chapter on Random Number Generators for more details.

5.1.5 SA Algorithm Performance

Figure 18 shows the progress of a SA search on the two-dimensional Rosenbrock function, $f = (1 - x)^2 + 100 * x_2 - x_1^2)^2$. Although one would not ordinarily choose to use SA on a problem which is amenable to solution by more efficient methods, it is interesting to do so for purposes of comparison. Each of the solutions accepted in a 1000 trial search is shown (marked by symbols). The algorithm employed the adaptive step size selection scheme of equations (67) and (68). It is apparent that the search is wide-ranging but ultimately concentrates in the neighborhood of the optimum.

Figure 19 shows the progress in reducing the objective function for the same search. Initially, when the annealing temperature is high, some large increases in f are accepted and some areas far from the optimum are explored. As execution continues and T falls, fewer uphill excursions are tolerated (and those that are tolerated are of smaller magnitude). The last 40% of the run is spent searching around the optimum. This performance is typical of the SA algorithm.

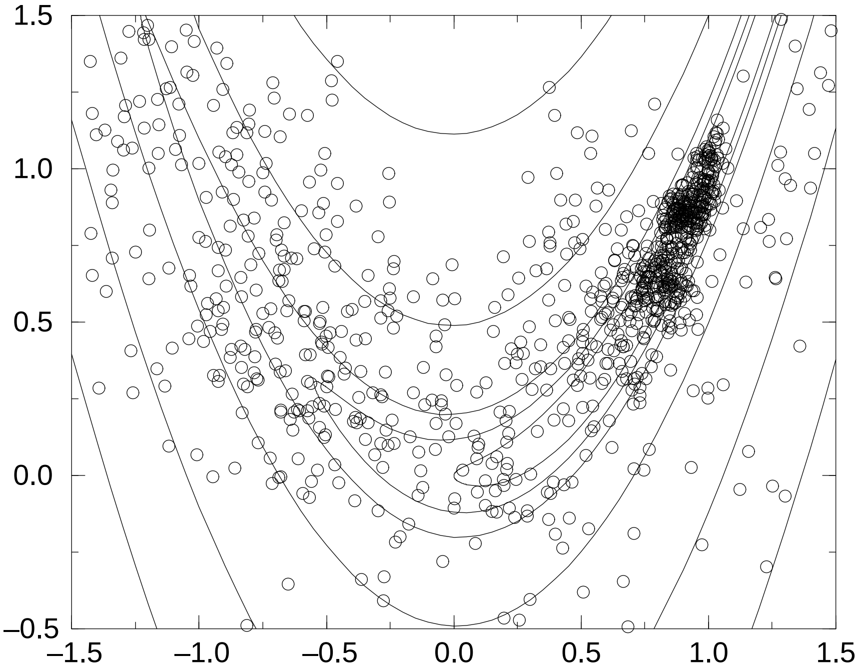

Figure 18: Minimization of the Two-dimensional Rosenbrock Function by Simulated Annealing — Search Pattern.

The code used in this example is `mo_sa.f`. It uses the input data available in file `mo_sa_in.dat`. Both of these files may be viewed with an html browser.

5.2 Genetic Algorithms

Kirkpatrick [40] has described SA as "an example of an evolutionary process modeled accurately by purely stochastic means," but this is more literally true of another class of new optimization routines known collectively as Genetic Algorithms (GAs). These attempt to simulate the phenomenon of natural evolution first observed by Darwin [12] and recently elaborated by Dawkins [14].

In natural evolution each species searches for beneficial adaptations in an ever-changing environment. As species evolve these new attributes are encoded in the chromosomes of individual members. This information does change by random mutation, but the real driving force behind evolutionary development is the combination and exchange of chromosomal material during breeding.

Although sporadic attempts to incorporate these principles in optimization routines have been made since the early 1960s (see a review in Chapter 4 of Goldberg [31]), GAs were first established on a sound theoretical basis by Holland [37]. The two key axioms underlying this innovative work were that complicated nonbiological structures could be described by simple bit strings and that these structures could be improved by the application of simple transformations to these strings.

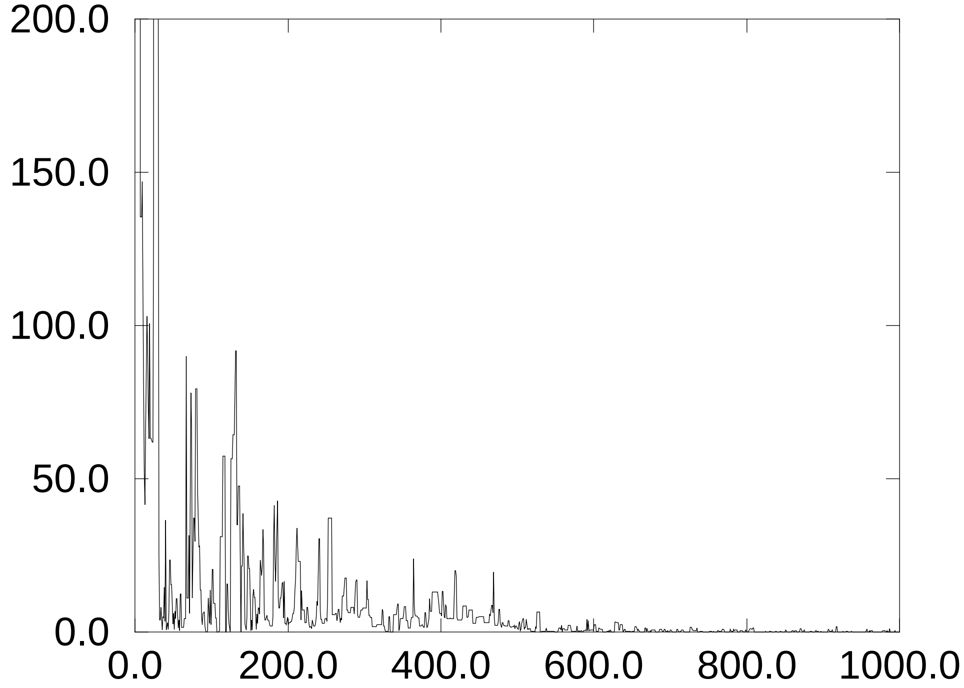

Figure 19: Minimization of the Two-Dimensional Rosenbrock Function by Simulated Annealing — Objective Reduction.

GAs differ from traditional optimization algorithms in four important respects:

- They work using an encoding of the control variables, rather than the variables themselves.
- They search from one population of solutions to another, rather than from individual to individual.
- They use only objective function information, not derivatives.
- They use probabilistic, not deterministic, transition rules.

Of course, GAs share the last two attributes with SA and, not surprisingly, have found applications in many of the same areas.

The basic structure of a GA is shown in Figure 20. One minor change from the standard optimization routine flow diagram is the use of the word 'population' rather than 'solution.' A more major difference is that the usual operation of generating a new solution has been replaced by three separate activities — population selection, recombination and mutation.

5.2.1 Solution Representation

In order to tackle a problem using a GA, candidate solutions must be encoded in a suitable form. In the traditional GA solutions are represented by binary bit strings ('chromosomes').

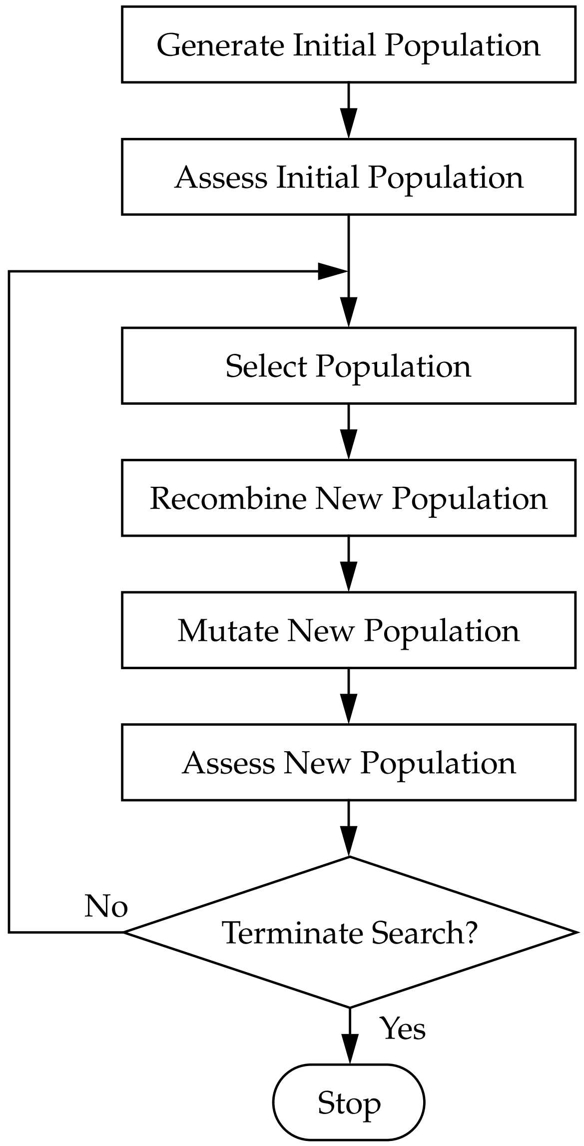

Figure 20: The Basic Structure of a Genetic Algorithm.

While integer and decision variables are easily encoded in this form, the representation of continuous control variables is not so simple. In general, the only option available is to approximate them (rescaled as necessary) by equivalent integer variables. The accuracy with which an optimum solution can be resolved then depends on the encoded bit length of these integers, leading to an inevitable compromise between precision and execution time. [6]

For combinatorial optimization problems, problem-specific solution encodings, such as ordered lists, are necessary. For example, a solution to the TSP can be represented by a string listing the cities in the order they are to be visited. Problem-specific operators are also required to manipulate such strings validly.

5.2.2 Population Selection

The initial population for a GA search is usually selected randomly, although there may be occasions when heuristic selection is appropriate (Grefenstette [34]). Within the algorithm, population selection is based on the principle of 'survival of the fittest.' The standard procedure is to define the probability of a particular solution i's survival to be:

$$P_{Si} = \frac{f_i}{f_{\sum}}, \tag{76}$$

if the objective function is to be maximized, where f_i is the *fitness* (objective value) of solution i, and

$$f_{\sum} = \sum_{i=1}^{N} f_i \tag{77}$$

is the total fitness of the population (of size N), or

$$P_{Si} = 1 - \frac{f_i}{f_{\sum}}, \tag{78}$$

if f is to be minimized. The new population is then selected by simulating the spinning of a suitably weighted roulette wheels N times.

It is clear that f must always be positive for this scheme to be used. Its range and scaling are also important. For instance, early in a search it is possible for a few superindividuals (solutions with fitness values significantly better than average) to dominate the selection process. Various schemes have been suggested to overcome this potential danger, of which the simplest is linear scaling, whereby f is rescaled through an equation of the form:

$$\tilde{f} = af + b. \tag{79}$$

Coefficients a and b are chosen each generation so that the average values of f and $\tilde{f}$ are equal and so that the maximum value of $\tilde{f}$ is a specified multiple of (usually twice) the

[6] In fact, Evolution Strategies (see section 5.3 almost always perform better than Genetic Algorithms on optimization problems with continuous control variables and do not incur quantization errors, so there is, in practice, little incentive to use a GA on such problems.

average. Linear scaling risks the introduction of negative values of $\tilde{f}$ for low performance solutions and must, therefore be used with caution.

Baker [3] suggested that P_{Si} should simply be made a (linear) function of the solution's rank within the population. For example, the best solution might be allocated a survival probability of $2/N$. If this is the case, that for the worst solution is then constrained to be zero, because of the normalization condition that the survival probabilities sum to unity. This ranking scheme has been found to overcome the problems of over- or underselection, without revealing any obvious drawbacks.

Roulette wheel selection suffers from the disadvantage of being a high-variance process with the result that there are often large differences between the actual and expected numbers of copies made — there is no guarantee that the best solution will be copied. De Jong [15] tested an *elitist* scheme, which gave just such a guarantee by enlarging the population to include a copy of the best solution if it hadn't been retained. He found that on problems with just one maximum (or minimum) the algorithm performance was much improved, but on multimodal problems it was degraded.

Numerous schemes which introduce various levels of determinism into the selection process have been investigated. Overall, it seems that a procedure entitled *stochastic remainder selection without replacement* offers the best performance. In this, the expected number of copies of each solution is calculated as

$$E_i = NP_{Si}. \tag{80}$$

Each solution is then copied I_i times, I_i being the integer part of E_i. The fractional remainder

$$R_i = E_i - I_i \tag{81}$$

is treated as the probability of further duplication. For example, a solution for which $E_i = 1.8$ would certainly be copied once and would be copied again with probability 0.8. Each solution is successively subjected to an appropriately weighted simulated coin toss until the new population is complete.

5.2.3 Population Recombination

Whatever selection procedure is used, it does not, of course, introduce any new solutions. Solutions which survive do so in order to serve as progenitors (breeding stock) for a new generation. It is in the recombination phase that the algorithm attempts to create new, improved solutions. The key procedure is that of *crossover*, in which the GA seeks to construct better solutions by combining the features of good existing ones.

In the simplest form of crossover *(one point crossover)* proceeds as follows. First, the entire population is paired off at random to give $N/2$ sets of potential parents. Second, pairs of solutions are chosen to undergo crossover with probability P_C. If the simulated weighted coin toss rejects crossover for a pair, then both solutions remain in the population unchanged. However, if it is approved, then two new solutions are created by exchanging all the bits following a randomly selected locus on the strings. For example, if crossover after position 5 is proposed between solutions

1 0 0 1 1 1 0 1

and

1 1 1 1 0 0 0 0,

the resulting offspring are

1 0 0 1 1 0 0 0

and

1 1 1 1 0 1 0 1

which replace their parents in the population.

A slightly more complex operator, first proposed by Cavicchio [8], is *two point crossover* in which two crossover points are randomly selected and the substrings between (and including) those positions are exchanged. If necessary, strings are treated as continuous rings. Thus, if crossover between points 6 (selected first) and 2 is proposed for strings

1 0 0 1 1 1 0 1

and

1 1 1 1 0 0 0 0,

the resulting offspring are

1 1 0 1 1 0 0 0

and

1 0 1 1 0 1 0 1.

Whereas, if crossover is between points 2 (selected first) and 6, the offspring are

1 1 1 1 0 0 0 1

and

1 0 0 1 1 1 0 0.

The even-handedness of two point crossover is appealing. There is no intuitive reason why right-hand bits should be exchanged more frequently than left-hand ones, unless the string represents the binary coding of a single integer or continuous variable, in which case, of course, the significance of individual bits decreases from left to right.

De Jong [15] tested multiple point crossover operators, which exchange more than one substring, and found that performance is degraded as more substrings are swapped. Although it is essential to introduce some changes in order to make progress, too many alterations make the probability of destroying the good features of a solution unacceptably high. An effective GA search requires a balance between *exploitation* (of good features in existing solutions) and *exploration* - introducing new (combinations of) features.

The performance of the recombination phase of a GA can also be improved by requiring that crossover introduce variation whenever possible. For instance, if the strings

1 0 1 0 1 0 1 1

and

1 1 0 1 1 0 1 1.

are chosen partners, then only exchanges involving bits 2, 3 or 4 will result in offspring differing from their parents. Booker [5] suggested that a general solution to this problem is to perform crossover between points in the parents' *reduced surrogates*, which contain just the nonmatching bits.

For many real applications, problem-specific solution representations and crossover operators have been developed. This flexibility is one of the attractions of GAs — it is very easy to introduce heuristic operators, which can substantially improve algorithm performance. The state of the art has been well reviewed recently by Davis [13].

5.2.4 Population Mutation

Although *mutation* merits a separate box in the flow diagram, it is very much a background operator in most GA applications (as it is in nature). The purpose of the mutation stage is to provide insurance against the irrevocable loss of genetic information and hence to maintain diversity within the population. For instance, if every solution in the population has 0 as the value of a particular bit, then no amount of crossover will produce a solution with a 1 there instead.

In traditional GAs every bit of every solution is potentially susceptible to mutation. Each bit is subjected to a simulated weighted coin toss with a probability of mutation P_M, which is usually very low (of the order of 0.01 or less). If mutation is approved, the bit changes value (in the case of a binary coding from 0 to 1 or 1 to 0).

There are schools of thought in the GA community which believe that mutation should only take place in solutions for which crossover was not approved or that only one mutation per solution should occur. Undoubtedly there are classes of problems for which each scheme is the most effective.

5.2.5 Advanced Operators

The simple operators and representations described above form the backbone of all GAs, but, because natural genetics is in reality a much more complex phenomenon than that portrayed so far, it is possible to conceive of several alternative representations and operators which have particular advantages for some GA applications, including:

- Introducing the concepts of *diploidy* and *dominance*, whereby solutions are represented by (several) pairs of chromosomes. The decoding of these (which determines between blue and brown eyes, say) then depends on whether individual bits are dominant or recessive. Such a representation allows alternative solutions to be held in abeyance, and can prove particularly useful for optimization problems where the solution space is time-varying.

- Introducing the ideas of *niche* and *speciation* in multimodal problems, whereby one deliberately tries to maintain diversity (to breed different species exploiting different niches in the environment) in order to locate several of the local optima. This can be achieved by elaborating the selection and recombination rules described above.

- Introducing some sort of intelligent control over the selection of mating partners, such as the "inbreeding with intermittent crossbreeding" scheme of Hollstien [38]. In this scheme similar individuals are mated with each other as long as the "family" fitness continues to improve. When this ceases new genetic material is added by crossbreeding with other families.

These and other advanced operators are discussed in detail in Chapter 5 of the seminal book by Goldberg [31].

5.2.6 Population Assessment

Like the SA algorithm, a GA does not use derivative information, it just needs to be supplied with a fitness value for each member of each population. Thus, the evaluation of the problem functions is essentially a "black box" operation as far as the GA is concerned. Obviously, in the interests of overall computational efficiency, the problem function evaluations should be performed efficiently. Depending on the nature of the system equations advice on accelerating these calculations may be found in other chapters within this project.

The guidelines for constraint handling in a GA are basically identical to those outlined in Section 5.1.2 for the SA algorithm. As long as there are no equality constraints and the feasible space is not disjoint, then infeasible solutions can simply be "rejected". In a GA this means ensuring that those particular solutions are not selected as parents in the next generation, e.g. by allocating them a zero survival probability.

If these conditions on the constraints are not met, then a penalty function method be used. A suitable form for a GA is:

$$f_A(x) = f(x) + M^k \mathbf{w}^T \mathbf{c}_V(\mathbf{x}), \tag{82}$$

where $\mathbf{w}$ is a vector of nonnegative weighting coefficients, the vector $\mathbf{c}_V$ quantifies the magnitudes of any constraint violations, M is the number of the current generation and k is a suitable exponent. The dependence of the penalty on generation number biases the search increasingly heavily towards feasible space as it progresses. Consult section 4 for more details on the penalty function method.

5.2.7 Control Parameters

The efficiency of a GA is highly dependent on the values of the algorithm's control parameters. Assuming that basic features like the selection procedure are predetermined, the control parameters available for adjustment are:

- the population size N,
- the crossover probability P_C, and
- the mutation probability P_M.

De Jong [15] made some recommendations based on his observations of the performance of GAs on a test bed of 5 problems, which included examples with difficult characteristics such as discontinuities, high dimensionality, noise and multimodality. His work suggested that settings of

$$(N, P_C, P_M) = (50, 0.60, 0.001) \tag{83}$$

would give satisfactory performance over a wide range of problems.

Grefenstette [33] went one stage further and used a GA to optimize these parameters for a test bed of problems. He concluded that

$$(N, P_C, P_M) = (30, 0.95, 0.010) \tag{84}$$

resulted in the best performance when the average fitness of each generation was used as the indicator, while

$$(N, P_C, P_M) = (80, 0.45, 0.010) \tag{85}$$

gave rise to the best performance when the fitness of the best individual member in each generation was monitored. The latter is, of course, the more usual performance measure for optimization routines.

In general, the population size should be no smaller than 25 or 30 whatever the problem being tackled, and for problems of high dimensionality larger populations (of the order of hundreds) are appropriate.

5.2.8 GA Computational Considerations

As with the Simulated Annealing algorithm, the procedures controlling the generation of new solutions are so simple that the computational cost of implementing a GA is usually dominated by that associated with the evaluation of the problem functions. It is therefore important that these evaluations should be performed efficiently and essential if the optimization is to be performed on a serial computer. Advice on how these calculations can be accelerated can be found in the chapter appropriate to the form of the system equations to be solved.

Unlike SA, which is intrinsically a sequential algorithm, GAs are particularly well-suited to implementation on parallel computers. Evaluation of the objective function and constraints can be done simultaneously for a whole population, as can the production of the new population by mutation and crossover. Thus, on a highly parallel machine, a GA can be expected to run nearly N times as fast for many problems, where N is the population size.

If it is possible to parallelize the evaluation of individual problem functions effectively, some thought and, perhaps, experimentation will be needed to determine the level at which multitasking should be performed. This will obviously depend on the number of processors available, the intended population size and the potential speed-ups available. If the number of processors exceeds the population size (a highly parallel machine!), multi-level parallelization may be possible.

A significant component of a GA code is the random number generator, which is essential to the processes of selection, crossover and mutation. (See, for example, the sample code

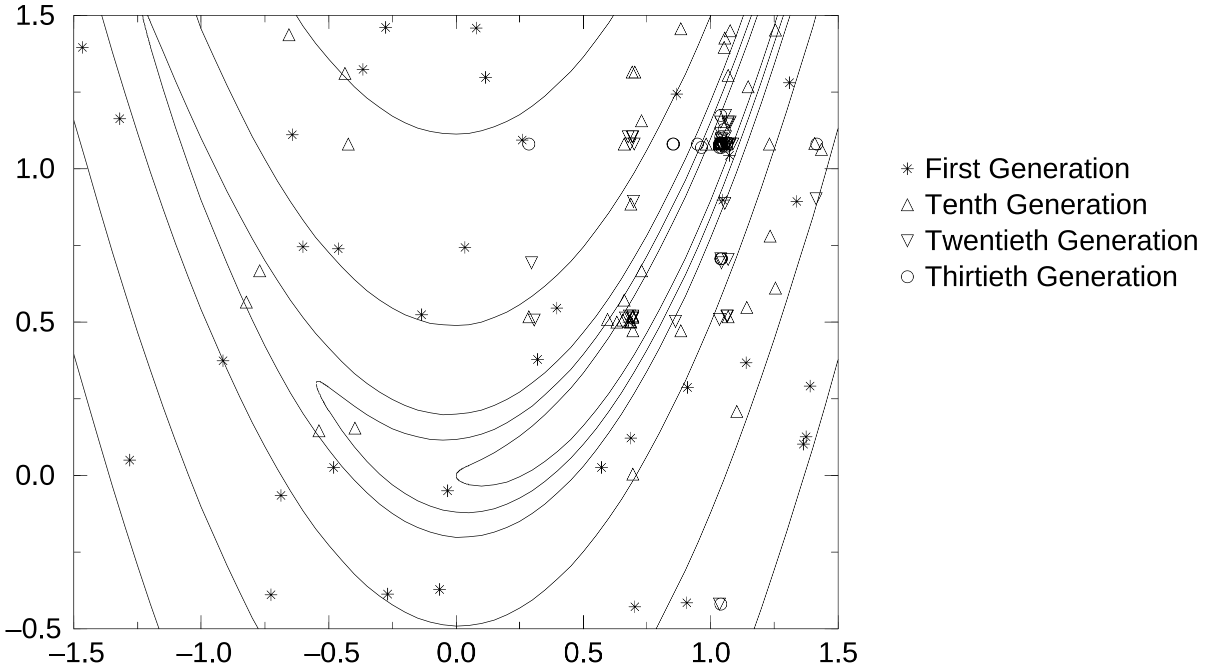

Figure 21: Minimization of the Two-Dimensional Rosenbrock Function by a Genetic Algorithm — Population Distribution of the First, Tenth, Twentieth, and Thirtieth Generations.

supplied with this section.) Random number generators are often provided in standard function libraries or as machine-specific functions. It is important, particularly when tackling large scale problems requiring thousands of iterations, that the random number generator used have good spectral properties — see the chapter on Random Number Generators for more details.

5.2.9 GA Algorithm Performance

Figure 21 shows the progress of a GA on the two-dimensional Rosenbrock function, $f = (1-x_1)^2 + 100(x_2 - x_1^2)^2$. This is presented purely for purposes of comparison. Each member of the 1st, 10th, 20th and 30th generations is shown (by a symbol). The convergence of the population to the neighborhood of the optimum at $(1,1)$ is readily apparent.

Within the GA each control variable was represented by a 30 bit binary number (chromosome), or equivalently in decimal integers between $(2^{31}-1)(2147483647)$ and 0. So, when rescaled into real numbers with a range of 3, this representation gives quantization errors of just 1.4×10^{-9}.

Figure 22 shows the progress in reducing the objective function for the same search. Both the fitness of the best individual within each population and the population average fitness are shown (note that the scales are different). These are the two standard measures of progress in a GA run. The difference between these two measures is indicative of the degree of convergence in the population.

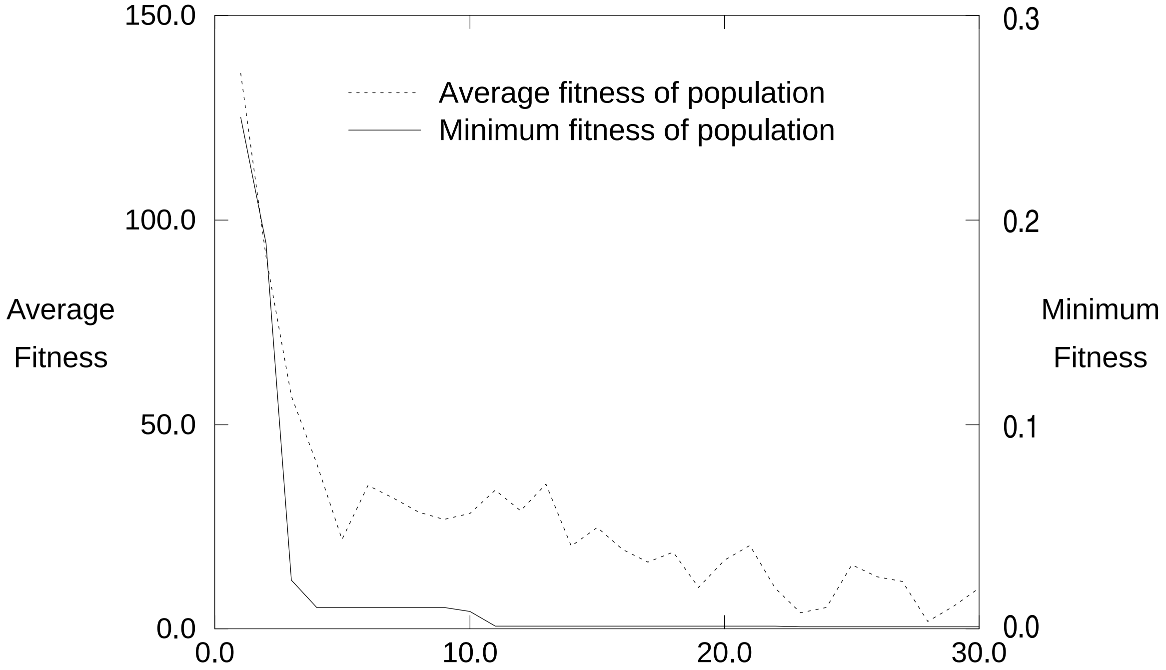

Figure 22: Minimization of the Two-Dimensional Rosenbrock Function by a Genetic Algorithm — Population Distribution of the First, Tenth, Twentieth, and Thirtieth Generations.

In this particular example, the GA has been successful in locating the global optimum, but it must be noted that GAs are often found to experience convergence difficulties. In many applications GAs locate the neighborhood of the global optimum extremely efficiently but have problems converging onto the optimum itself. In such instances a hybrid algorithm, for example using a GA initially and then switching to a low temperature SA search, can prove effective.

The code used in this example is `mo_ga.f`. It uses input data available in file `mo_ga_in.dat`. Both of these files may be viewed with an html browser.

5.3 Evolutionary Strategies

This section will appear in the next release of this chapter.

6 High-Performance Optimization Programming

Future developments in the field of optimization will undoubtedly be influenced by recent interest and rapid developments in new technologies — powerful vector and parallel machines. Indeed, their exploitation for algorithm design and solution of "grand challenge" applications is expected to bring new advances in many fields, such as computational chemistry and computational fluid dynamics.

Supercomputers can provide speedup over traditional architectures by optimizing both scalar and vector computations. This can be accomplished by pipelining data as well as offering special hardware instructions for calculating intrinsic functions (e.g., $\exp(x), \sqrt{x}$), arithmetic, and array operations. In addition, parallel computers can execute several operations concurrently. Multiple instructions can be specified for multiple data streams in MIMD designs, whereas the same instructions can be applied to multiple data streams in SIMD prototypes. Communication among processors is crucial for efficient algorithm design so that the full parallel apparatus is exploited. These issues will only increase in significance as massively parallel networks enter into regular use.

In general, one of the first steps in optimizing codes for these architectures is implementation of standard basic linear algebra subroutines (BLAS). These routines — continuously being improved, expanded, and adapted optimally to more machines — perform operations such as dot products ($x^T y$) and vector manipulations ($ax + y$), as well as matrix/vector and matrix/matrix operations. Thus, operations, such as in equations (49), (53), or (57) and (58) can be executed very efficiently. In particular, if n is very large, segmentation among the processors may also be involved. A new library of FORTRAN 77 subroutines, LAPACK, focuses on design and implementation of standard numerical linear algebra tasks (e.g., systems of linear equations, eigenvalue and singular value problems) to achieve high efficiency and accuracy on vector processors, high-performance workstations, and shared-memory multiprocessors. At this writing, up-to-date information may be obtained by sending the message `send index from LAPACK` to the electronic mail address `netlib@ornl.gov`.

Specific strategies for optimization algorithms have been quite recent and are not yet unified. For parallel computers, natural improvements may involve the following ideas: (1) performing multiple minimization procedures concurrently from different starting points; (2) evaluating function and derivatives concurrently at different points (e.g., for a finite-difference approximation of gradient or Hessian or for an improved line search); (3) performing matrix operations or decompositions in parallel for special structured systems (e.g., Cholesky factorizations of block-band preconditioners).

With increased computer storage and speed, the feasible methods for solution of very large (e.g., $\mathcal{O}(10^5)$ or more variable) nonlinear optimization problems arising in important applications (macromolecular structure, meteorology, economics) will undoubtedly expand considerably and make possible solution of larger and far more complex problems in all fields of science and engineering.

7 Exercises

Exercise 1 *The determination of a minimum of a function.*

Consider finding the minimum of the function $f(x) = (x-1)^2 + 0.1\sin(20(x-1))$ by the structure described in algorithm 2.1. This nonlinear function is essentially a quadratic function perturbed by small sinusoidal fluctuations. One can use several simple techniques as a first approach to solving this problem. These include graphics, tabulation of function values, and bisection to bracket the interval where a minimum lies.

a) Plot this function.

b) Plot its derivative $g(x) = f'(x)$; solving for the roots of $g(x)$ is a closely related problem to minimizing $f(x)$.

c) Choose various initial points devise descent paths toward the minimum of $f(x)$.

d) Plot your paths toward the minimum and analyze the results.

Exercise 2 *Derive the interpolation formulas for the line search.*

a) Derive the interpolation formulas above for the line search by using the form of $p(\lambda)$ in equation (16) and using the four known conditions involving $\tilde{f}_1, \tilde{f}_2, \tilde{g}_1$ and $\tilde{g}_2$.

b) Now use the quadratic form of equation (22), and derive the coefficients b, c, d by using the conditions for $\tilde{f}_1, \tilde{f}_2$ and $\tilde{g}_1$.

Exercise 3 *The construction of a subroutine for the line search interpolation procedure.*

a) Write a subroutine to construct the line search interpolation procedure described above once the values of $\tilde{f}_1, \tilde{f}_2, \tilde{g}_1$ and $\tilde{g}_2$ are provided. You can switch to the quadratic interpolant when the coefficient a is small or by examining the four values supplied.

b) Construct input data to cover all situations shown in Figure 7 and test your programs. Plot your results by showing your computed interpolant. Is your subroutine robust enough to produce a minimum in each case? How can it be improved?

Exercise 4 *Derive the convergence rates for the three sequences mentioned above.*

Exercise 5 *Derivative testing.*

Use routine `TESTGH` for testing Rosenbrock's function for n=2,4,10 and 100 on a wide range of starting points.

Exercise 6 *The determination of the two-dimenstional Beale function by numerical techniques.*

Write code for the two-dimensional function known as Beale:

$$f(x_1, x_2) = [1.5 - x_1(1 - x_1)]^2 + [2.25 - x_1(1 - x_2)]^2 + [2.625 - x_1(1 - x_2^3)]^2, \tag{86}$$

its gradient vector, and Hessian matrix. Test the derivative routines with program `TESTGH` above until they are all correct.

Exercise 7 *The determination of the roots of a function.*

Set up a program to find the root of $x^2 = a$, and test it for various values of a from various starting points. Plot your minimization trajectories by displaying each iterate and connecting it to the next. When is convergence satisfactory and when are problems encountered?

Exercise 8 *The reliability of the root-finding program.*

Test your program on finding the roots of the equation $f(x) = x - e^{-x} = 0$. Plot this function and your minimization paths from different starting points.

Exercise 9 *The forward and central difference quotients for the determination of a derivative of a function.*

Consider approximating the derivative of the function $f(x) = e^x$ for $x = 1$. Use the forward and central difference quotients as defined below.

$$D_F(h) = [f(x+h) - f(x)]/h \tag{87}$$

$$D_C(h) = [f(x+h) - f(x-h)]/2h \tag{88}$$

Using both single and double-precision programs, make tables showing in each case the value for h, the difference approximations $D_F(h)$ and $D_C(h)$, and the errors associated with these values (since we know the answer here). Use a wide range of values for the difference interval h: from 1.0 to 10^{-10}. What can you say about the quality of the two different approximations and their associated optimal values for h? Can you suggest how an optimal value for h can be estimated?

Exercise 10 *The minimization of a one-dimensional function by Newton's method.*

Consider minimizing the one-dimensional function

$$f(x) = \sin(x) + x - \exp(x) \tag{89}$$

by Newton's method. You might want to transform this problem into a root-finding one: $g(x) = 0$. Newton's method was originally devised for solving nonlinear equations and then extended for minimization. Roughly speaking, the Newton techniques we learned above for minimizing $f(x)$ can be adapted to solve nonlinear equations of form $g(x) = f(x) = 0$.

While good graphics skills can be very effective for minimizing or finding the roots of one-dimensional problems, we cannot expect to solve such problems systematically when many variables are involved and for repeated applications. The following exercises will give you practical experience in solving such problems using graphics and suitable programs.

(i) Sketch $g(x)$ and $g'(x)$ by making use of your basic knowledge of trigonometric and exponential functions.

(ii) Write a program to implement Newton's method for root finding. Write it in a flexible way so that you can use it later for other problems. Choose your stopping criteria carefully. You should balance the desire to obtain as accurate a result as possible with the amount of computation involved. In other words, there is no point in continuing a loop when the answer can no longer be improved. What are reasonable evaluations of progress to use? Consider, for example, changes in the magnitudes of successive iterates x and the associated function values $g(x)$. Also consider the amount of progress realized (in terms of function value) in relation to the size of step taken. (Sketch a situation where this may be relevant).

(iii) First find the largest root, i.e., the one furthest to the right on the x-axis. Experiment with different starting points. What can you say about the performance and accuracy of Newton's method? (Consider $g'(x)$). What convergence do you see in practice, in terms of digits of accuracy obtained in each step? Be sure to illustrate progress by listing $x, g(x)$, measures of error, and measures of progress between successive iterates for each step of the method. Print the results to as many digits as you think are relevant. What accuracy do you expect from Newton's method in terms of machine precision ϵ_m ?

(iv) Now try to find a few of the next largest roots to the left. Select and experiment with different starting points by considering various regions on your function sketch. When convergence is satisfactory, is it as fast as you observed for the large root? Are there examples of points for which your method fails? Why? Can you suggest a class of strategies to improve the method in those cases? Implement the following modification: Newton's method with bisection safeguards. In this version, we will keep two points a and b for which f always has opposite signs (i.e, we will always bracket a root). Let $x_k = a$ if $|g(a)| < |g(b)|$ and b otherwise. The method then takes a Newton step, defined by the tangent to the curve through the point $(x_k, g(x_k))$, if the Newton point x_{k+1} lies in the interval $[a, b]$ and $|g(x_{k+1})| \leq \frac{1}{2}|g(x_k)|$; otherwise, a bisection step is taken. Make a sketch of this scheme so that you understand what it does. Compare performance now for finding several roots. Are there cases where the straightforward implementation of Newton's method failed but this version succeeds? Why? What can you say about performance?

(v) Now that you understand some of the difficulties involved even in some very simple problems, you might appreciate available software tools better. Use a library package to minimize $f(x)$ or find some roots of $g(x)$. You can use an appropriate routine from comprehensive libraries such as NAG or you might search for a minimizer in `Netlib`, a network of free numerical analysis software. To get started in `Netlib`, at the time of writing you can send the message ``send index'' to `netlib@ornl.gov`. (You may also refer to the introduction of `netlib` in the linear algebra chapter. You may want to search for a keyword such as `Brent`. You may also need to obtain other supporting software that may not be included in the file. Information on such requirements should be given with the code. Once you obtain a suitable code, describe briefly the algorithm, going back to the original cited literature if you must. Try some of the good and bad starting points you diagnosed earlier. What can you say about performance of the package?

References

[1] Acton, F. S., Numerical Methods that Usually Work, Chapter 17, The Mathematical Association of America, Washington, D. C., 1990 (updated from the 1970 edition).

[Introductory numerical methods book, fun and lively in style.]

[2] Arts, E. H. L., and Korst, J.H.M., Simulated Annealing and Boltzmann Machines, Wiley (Interscience), New York, 1989.

[3] Baker, J. E., Adaptive Selection Methods for Genetic Algorithms, pp. 101-111, Proceedings of an International Conference on Genetic Algorithms and their Applications (J. J. Grefenstette, editor), Lawrence Erlbaum Associates, Hillsdale, NJ.

[4] Boggs, P.T., Byrd, R.H. and Schnabel, R.B., Eds., Numerical Optimization 1984, SIAM, Philadelphia, 1985. [A collection of papers presented at the SIAM optimization conference, varied and interesting approaches.]

[5] Booker, L. B., Improving Search in Genetic Algorithms, pp. 61-73, Genetic Algorithms and Simulated Annealing (L. Davis, editor), Pitman, London, 1987.

[6] Bounds, D. G., New Optimization Methods from Physics and Biology, Nature 329, 215-218, 1987.

[7] Cauchy, A., Methode Generale pour la Resolution des Systemes d'Equations Simultanees, Comp. Rend. Acad. Sci. Paris, 536-538, 1847. [Advanced article, Cauchy's steepest descent method.]

[8] Cavicchio, D. J., Adaptive Search Using Simulated Evolution, Ph.D. Thesis, University of Michigan, Ann Arbor, MI, 1970.

[9] Ciarlet, P.G., Introduction to Numerical Linear Algebra and Optimization, Cambridge University Press, Cambridge, Great Britain, 1989. [Advanced optimization book, theoretical angle.]

[10] Concus, P., Golub, G. H and O'Leary, D.P., A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations, Sparse Matrix Computations, J. R. Bunch and D. J. Rose, Eds., Academic Press, New York, 1976, 309-332. [Advanced article describing preconditioned conjugate gradient applications.]

[11] Dahlquist, G. and Bjork, A., Numerical Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1974. [Classic numerical methods book, covers numerous areas in broad detail.]

[12] Darwin, C., On The Origin of Species, 1st edition (facsimile - 1964), Harvard University Press, Cambridge, MA, 1859.

[13] Davis, L., Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, NY, 1991.

[14] Dawkins, R., The Blind Watchmaker, Penguin, London, 1986.

[15] De Jong, K. A., An Analysis of the Behavior of a Class of Genetic Adaptive Systems, Ph.D. Thesis, University of Michigan, Ann Arbor, MI., 1975.

[16] Dennis, J.E. Jr. and Schnabel, R.B., Numerical Methods for Unconstrained Optimization and Nonlinear Equations, Prentice-Hall, Englewood Cliffs, New Jersey, 1983. [`Advanced optimization book, providing theoretical background for algorithms.`]

[17] Dennis Jr., J.E. and More, J.J., Quasi-Newton Methods, Motivation and Theory, SIAM Review 19, 46-89, 1977. [`Advanced review of Quasi-Newton methods.`]

[18] Dembo, R.S. and Steihaug, T., Truncated-Newton Algorithms for Large-Scale Unconstrained Optimization, Math. Prog. 26, 190-212 (1983). [`Advanced article presenting the truncated Newton method in theoretical detail.`]

[19] Dixon, L.C.W., On the Impact of Automatic Differentiation on the Relative Performance of Parallel Truncated Newton and Variable Metric Algorithms, SIAM J. Opt. 1, 475-486, 1991. [`Review on automatic differentiation and its usage.`]

[20] Duff, I.S., Erisman, A. M. and Reid, J.K., Direct Methods for Sparse Matrices, Clarendon Press, Oxford (1986). [`Advanced textbook on direct methods for solving sparse linear systems.`]

[21] Evans, J.D., The Use of Pre-conditioning in Iterative Methods for Solving Linear Equations With Symmetric Positive Definite Matrices, J. Inst. Math. Applic. 4, 295-314, 1967. [`Advanced article introducing preconditioning for conjugate gradient methods.`]

[22] Fletcher, R., Practical Methods of Optimization, Second Edition, (A Wiley- Interscience Publication), John Wiley and Sons, Tiptree, Essex, Great Britain, 1987. [`Advanced book on optimization techniques.`]

[23] Fletcher, R. and Reeves, C.M., Function Minimization by Conjugate Gradients, Comp, J. 7, 149-154, 1964. [`Advanced article on using conjugate gradient for nonconvex functions.`]

[24] Floudas, C.A. and Pardalos, P. M., Eds., Recent Advances in Global Optimization, Princeton Series in Computer Science, Princeton University Press, New Jersey, 1991. [`Recent volume on various approaches to global optimization.`]

[25] George, A. and Liu, J.W., Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1981. [`Advanced textbook on solving large sparse positive definite systems of equations.`]

[26] Gilbert, J.C. and Nocedal, J., Global Convergence Properties of Conjugate Gradient Methods for Optimization. SIAM J. Opt. 2, 21-42, 1992. [`Advanced article summarizing the various conjugate gradient methods and their properties.`]

[27] Gilbert, J.C., and Lemarechal, C., Some Numerical Experiments with Variable-Storage Quasi-Newton Algorithms, Math. Prog. 45, 407-435, 1989. [`Advanced article describing practical experience with limited-memory Quasi-Newton methods on large-scale crystallography and meteorology problems.`]

[28] Gill, P.E., Murray, W. and Wright, M. H., Numerical Linear Algebra and Optimization, Volume 1, Addison-Wesley, Redwood City, California, 1991. [`Introductory numerical linear algebra and optimization book.`]

[29] Gill, P.E., Murray, W. and Wright, M.H., Practical Optimization, Academic Press, New York, 1983. [`Standard reference book on practical optimization methods.`]

[30] Gill, P.E., Murray, W., Saunders, M. A. and Wright, M.H., Computing Forward-Difference Intervals for Numerical Optimization, SIAM J. Sci. Stat. Comput. 4, 310-321, 1983. [`Advanced article, excellent description on choosing finite-difference intervals.`]

[31] Goldberg, D. E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, Reading, MA, 1989.

[32] Golub, G.H. and Van Loan, C.F., Matrix Computations, Johns Hopkins University Press, Baltimore, Maryland, 1983. [`Excellent linear algebra book.`]

[33] Grefenstette, J. J., Optimization of Control Parameters for Genetic Algorithms, IEEE Trans. Syst., Man, Cyber. SMC-16, 122-128, 1986.

[34] Grefenstette, J. J., Incorporating Problem Specific Knowledge into Genetic Algorithms, pp. 42-60, Genetic Algorithms and Simulated Annealing (L. Davis, editor), Pitman, London, 1987.

[35] Griewank, A., On Automatic Differentiation, Mathematical Programming 1988, Kluwer Academic Publishers, Japan, 1988, pp. 83-107. [`Review on automatic differentiation.`]

[36] Hestenes, M.R., Conjugate Direction Methods in Optimization, Springer-Verlag, New York, 1980. [`Advanced textbook on methods related to conjugate gradient.`]

[37] Holland, J. H., Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.

[38] Hollstien, R. B., Artificial Genetic Adaptation in Computer Control Systems, Ph.D. Thesis, University of Michigan, Ann Arbor, MI., 1971.

[39] Kirkpatrick, S., Gerlatt, C. D. Jr., and Vecchi, M. P., Optimization by Simulated Annealing, IBM Research Report RC 9355, 1982.

[40] Kirkpatrick, S., Gerlatt, C. D. Jr., and Vecchi, M.P., Optimization by Simulated Annealing, Science 220, 671-680, 1983.

[41] Kirkpatrick, S., Optimization by Simulated Annealing - Quantitative Studies, J. Stat. Phys. 34, 975-986, 1984.

[42] Laarhoven, P. J. M. van, and Aarts, E. H. L., Simulated Annealing: Theory and Applications, Reidel, Dordrecht, Holland, 1987.

[43] Lin, S., Computer Solutions of the Traveling Salesman Problem, Bell Syst. Tech. J. 44, 2245-2269, 1965.

[44] Liu, D.C. and Nocedal, J., On the Limited Memory BFGS Method for Large Scale Optimization, Math. Prog. 45, 503-528, 1989. [`Advanced article presenting the limited-memory Quasi Newton method.`]

[45] Luenberger, D.G., Linear and Nonlinear Programming, Second Edition, Addison-Wesley, Reading, Massachusetts, 1984. [`Introductory optimization book, providing easy-to-grasp explanations of algorithmic strategies.`]

[46] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M. N., Teller, A.H. and Teller, E., Equations of State Calculations by Fast Computing Machines, J. Chem. Phys. 21, 1087-1092, 1958.

[47] Nash, S.G., Solving Nonlinear Programming Problems using Truncated-Newton Techniques, Numerical Optimization 1984, P. T. Boggs, R. H. Byrd, and R. B. Schnabel, Eds., SIAM, Philadelphia, 1985, pp. 119-136. [`Advanced article presenting a practical truncated Newton algorithm.`]

[48] Nemhauser, G.L., Rinnooy Kan, A.H.G. and Todd, M.J., Eds., Handbook in Operations Research Management Science, Vol. 1, Optimization, Elsevier Science Publishers (North-Holland), Amsterdam, The Netherlands, and New York, 1989. [`Excellent volume on recent technique in optimization research.`]

[49] Parks, G. T., An Intelligent Stochastic Optimization Routine for Nuclear Fuel Cycle Design, Nucl. Technol. 89, 233-246, 1990.

[50] Pincus, M., A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems, Oper. Res. 18, 1225-1228, 1970.

[51] Powell, M.J.D., An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives, Comp. J. 7, 155-162, 1964. [`Advanced articles, presents Powell minimization method.`]

[52] Powell, M.J.D., Nonconvex Minimization Calculations and the Conjugate Gradient Method, Lecture Notes in Mathematics, Vol. 1066, pp. 122-141, 1984. [`Advanced review on conjugate gradient for nonconvex functions.`]

[53] Rall, L.B., Automatic Differentiation - Techniques and Applications, Lecture Notes in Computer Science 120, Springer-Verlag, Berlin, New York, 1981. [Textbook on automatic differentiation.]

[54] Randelman, R. E., and Grest, G.S., N-City Traveling Salesman Problem - Optimization by Simulated Annealings, J. Stat. Phys. 45, 885-890, 1986.

[55] Schlick, T. and Overton, M., A Powerful Truncated Newton Method for Potential Energy Minimization, J. Comp. Chem. 8, 1025-1039, 1987. [Article presenting development and application of a truncated Newton method to computational chemistry problems.]

[56] Schlick, T. and Fogelson, A., TNPACK — A Truncated Newton Minimization Package for Large-Scale Problems: I. Algorithm and Usage and II. Implementation Examples. ACM Trans. Math. Softw. 18, 41-111, 1992. [Article describing a truncated-Newton minimization package TNPACK and illustrating its application in several scientific applications.]

[57] Derreumaux, P., Zhang, G., Schlick, T., and Brooks, B., A Truncated Newton Minimizer Adapted for CHARMM and Biomolecular Applications, J. Comp. Chem. 15, 532-552, 1994. [Article describing the adaptation of a minimization package for optimization of molecular structures.]

[58] Schlick, T., Modified Cholesky Factorizations for Sparse Preconditioners, SIAM J. Sci. Stat. Comp. 14, 424-445, 1993. [Article presenting practical strategies of choosing sparse preconditioners for optimization.]

[59] Schlick, T., Optimization Methods in Computational Chemistry, Reviews in Computational Chemistry, Vol. III, K. B. Lipkowitz and D. B. Boyd, Eds., VCH Publisher, New York, pp 1-71, 1992. [Review on optimization techniques in computational chemistry applications.]

[60] Schnabel, R.B. and Eskow, E., A New Modified Cholesky Factorization, SIAM J. Sci. Stat. Comput. 11, 1136-1158, 1990. [Advanced article presenting a new modified Cholesky factorization.]

[61] Shanno, D.F., Conjugate Gradient Methods with Inexact Searches, Math. Oper. Res. 3, 244-256, 1978. [Advanced article on conjugate gradient for nonconvex functions.]

[62] Vanderbilt, D., and Louie, S. G., A Monte Carlo Simulated Annealing Approach to Optimization over Continuous Variables, J. Comput. Phys. 56, 259- 271, 1984.