NAG Library Chapter Introduction

e04 – Minimizing or Maximizing a Function

Contents

1 Scope of the Chapter

This chapter provides functions for solving various mathematical optimization problems by solvers based on local stopping criteria. The main classes of problems covered in this chapter are:

Linear Programming (LP) – dense and sparse;

Quadratic Programming (QP) – convex and nonconvex, dense and sparse;

Nonlinear Programming (NLP) – dense and sparse, based on active-set SQP methods or interior point methods (IPM);

Semidefinite Programming (SDP) – both linear matrix inequalities (LMI) and bilinear matrix inequalities (BMI);

Derivative-free Optimization (DFO);

Least Squares (LSQ), data fitting – linear and nonlinear, constrained and unconstrained.

For a full overview of the functionality offered in this chapter, see Section 6 or the Chapter Contents (Chapter e04).

See also other chapters in the Library relevant to optimization:

Chapter e05 contains functions to solve **global optimization** problems;

Chapter h addresses problems arising in **operational research** and focuses on **Mixed Integer Programming (MIP)**;

Chapters f07 and f08 include functions for linear algebra and in particular unconstrained linear least squares;

Chapter e02 focuses on curve and surface fitting, in which linear data fitting in l_1 or l_∞ norm might be of interest.

This introduction is only a brief guide to the subject of optimization. It discusses a classification of the optimization problems and presents an overview of the algorithms and their stopping criteria to help with the choice of a correct solver for a particular problem. Anyone with a difficult or protracted problem to solve will find it beneficial to consult a more detailed text, see Gill *et al.* (1981), Fletcher (1987) or Nocedal and Wright (2006). If you are unfamiliar with the mathematics of the subject you may find Sections 2.1, 2.2, 2.3, 2.6 and 4 a useful starting point.

2 Background to the Problems

2.1 Introduction to Mathematical Optimization

Mathematical Optimization, also known as *Mathematical Programming*, refers to the problem of finding values of the inputs from a given set so that a function (called the **objective function**) is minimized or maximized. The inputs are called **decision variables**, *primal variables* or just *variables*. The given set from which the decision variables are selected is referred to as a **feasible set** and might be defined as a domain where **constraints** expressed as functions of the decision variables hold certain values. Each point of the feasible set is called a **feasible point**.

A general mathematical formulation of such a problem might be written as

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in \mathcal{F} \end{array}$$

where x denotes the decision variables, $f(x)$ the objective function and $\mathcal{F}$ the feasibility set. In this chapter we assume that $\mathcal{F} \subset R^n$. Since **maximization** of the objective function $f(x)$ is equivalent to minimizing $-f(x)$, only minimization is considered further in the text. Some functions allow you to specify whether you are solving a minimization or maximization problem, carrying out the required transformation of the objective function in the latter case.

A point x^* is said to be a **local minimum** of a function f if it is feasible ($x^* \in \mathcal{F}$) and if $f(x) \geq f(x^*)$ for all $x \in \mathcal{F}$ near x^*. A point x^* is a **global minimum** if it is a local minimum and $f(x) \geq f(x^*)$ for all feasible x. The solvers in this chapter are based on algorithms which seek only a local minimum,

however, many problems (such as *convex optimization* problems) have only one local minimum. This is also the global minimum. In such cases the Chapter e04 solvers find the global minimum. See Chapter e05 for solvers which try to find a global solution even for nonconvex functions.

2.2 Classification of Optimization Problems

There is no single efficient solver for all optimization problems. Therefore it is important to choose a solver which matches the problem and any specific needs as closely as possible. A more generic solver might be applied, however the performance suffers in some cases, depending on the underlying algorithm.

There are various criteria to help to classify optimization problems into particular categories. The main criteria are as follows:

Type of objective function;

Type of constraints;

Size of the problem;

Smoothness of the data and available derivative information.

Each of the criteria is discussed below to give the necessary information to identify the class of the optimization problem. Section 2.5 presents the basic properties of the algorithms and Section 4 advises on the choice of particular functions in the chapter.

2.2.1 Types of objective functions

In general, if there is a structure in the problem the solver should benefit from it. For example, a solver for problems with the *sum of squares* objective should work better than when this objective is treated as a general *nonlinear objective*. Therefore it is important to recognize typical types of the objective functions.

An optimization problem which has **no objective** is equivalent to having a constant objective, i.e., $f(x) = 0$. It is usually called a **feasible point problem**. The task is to then find any point which satisfies the constraints.

A **linear objective function** is a function which is linear in all variables and therefore can be represented as

$$f(x) = c^{\mathrm{T}}x + c_0$$

where $c \in R^n$. Scalar c_0 has no influence on the choice of decision variables x and is usually omitted. It will not be used further in this text.

A **quadratic objective function** is an extension of a linear function with quadratic terms as follows:

$$f(x) = \frac{1}{2}x^{\mathrm{T}}Hx + c^{\mathrm{T}}x.$$

Here H is a real symmetric $n \times n$ matrix. In addition, if H is positive semidefinite (all its eigenvalues are non-negative), the objective is **convex**.

A general **nonlinear objective function** is any $f : R^n \rightarrow R$ without a special structure.

Special consideration is given to the objective function in the form of a **sum of squares** of functions, such as

$$f(x) = \sum_{i=1}^{m} r_i^2(x)$$

where $r_i : R^n \rightarrow R$; often called *residual functions*. This form of the objective plays a key role in **data fitting** solved as a **least squares problem** as shown in Section 2.2.3.

2.2.2 Types of constraints

Not all optimization problems have to have constraints. If there are no restrictions on the choice of x except that $x \in \mathcal{F} = R^n$, the problem is called **unconstrained** and thus every point is a feasible point.

Simple bounds on decision variables $x \in R^n$ (also known as *box constraints* or *bound constraints*) restrict the value of the variables, e.g., $x_5 \leq 10$. They might be written in a general form as

$$l_{x_i} \leq x_i \leq u_{x_i}, \quad \text{for } i = 1, \ldots, n$$

or in the vector notation as

$$l_x \leq x \leq u_x$$

where l_x and u_x are n-dimensional vectors. Note that lower and upper bounds are specified for all the variables. By conceptually allowing $l_{x_i} = -\infty$ and $u_{x_i} = +\infty$ or $l_{x_i} = u_{x_i}$ full generality in various types of constraints is allowed, such as unconstrained variables, one-sided inequalities, ranges or equalities (fixing the variable).

The same format of bounds is adopted to linear and nonlinear constraints in the whole chapter. Note that for the purpose of passing infinite bounds to the functions, all values above a certain threshold (typically 10^{20}) are treated as $+\infty$.

Linear constraints are defined as constraint functions that are linear in all of their variables, e.g., $3x_1 + 2x_2 \geq 4$. They can be stated in a matrix form as

$$l_B \leq Bx \leq u_B$$

where B is a general $m_B \times n$ rectangular matrix and l_B and u_B are m_B-dimensional vectors. Each row of B represents linear coefficients of one linear constraint. The same rules for bounds apply as in the simple bounds case.

Although the bounds on x_i could be included in the definition of linear constraints, we recommend you distinguish between them for reasons of computational efficiency as most of the solvers treat simple bounds explicitly.

A set of m_g **nonlinear constraints** may be defined in terms of a nonlinear function $g : R^n \to R^{m_g}$ and the bounds l_g and u_g which follow the same format as simple bounds and linear constraints:

$$l_g \leq g(x) \leq u_g.$$

Although the linear constraints could be included in the definition of nonlinear constraints, again we prefer to distinguish between them for reasons of computational efficiency.

A **matrix constraint** (or *matrix inequality*) is a constraint on eigenvalues of a matrix operator. More precisely, let S^m denote the space of real symmetric matrices m by m and let $\mathcal{A}$ be a matrix operator $\mathcal{A} : R^n \to S^m$, i.e., it assigns a symmetric matrix $\mathcal{A}(x)$ for each x. The matrix constraint can be expressed as

$$\mathcal{A}(x) \succeq 0$$

where the inequality $S \succeq 0$ for $S \in S^m$ is meant in the eigenvalue sense, namely all eigenvalues of the matrix S should be non-negative (the matrix should be positive semidefinite).

There are two types of matrix constraints allowed in the current mark of the Library. The first is **linear matrix inequality (LMI)** formulated as

$$\mathcal{A}(x) = \sum_{i=1}^{n} x_i A_i - A_0 \succeq 0$$

and the second one, **bilinear matrix inequality (BMI)**, stated as

$$\mathcal{A}(x) = \sum_{i,j=1}^{n} x_i x_j Q_{ij} + \sum_{i=1}^{n} x_i A_i - A_0 \succeq 0.$$

Here all matrices A_i, Q_{ij} are given real symmetric matrices of the same dimension. Note that the latter type is in fact quadratic in x, nevertheless, it is referred to as bilinear for historical reasons.

2.2.3 Typical classes of optimization problems

Specific combinations of the types of the objective functions and constraints give rise to various classes of optimization problems. The common ones are presented below. It is always advisable to consider the closest formulation which covers your problem when choosing the solver. For more information see classical texts such as Dantzig (1963), Gill *et al.* (1981), Fletcher (1987), Nocedal and Wright (2006) or Chvátal (1983).

A **Linear Programming (LP)** problem is a problem with a linear objective function, linear constraints and simple bounds. It can be written as follows:

$$\begin{array}{ll} \underset{x \in R^n}{\text{minimize}} & c^{\mathrm{T}}x \\ \text{subject to} & l_B \leq Bx \leq u_B \\ & l_x \leq x \leq u_x \end{array}$$

Quadratic Programming (QP) problems optimize a quadratic objective function over a set given by linear constraints and simple bounds. Depending on the convexity of the objective function, we can distinguish between **convex** and **nonconvex** (or *general*) QP.

$$\begin{array}{ll} \underset{x \in R^n}{\text{minimize}} & \frac{1}{2}x^{\mathrm{T}}Hx + c^{\mathrm{T}}x \\ \text{subject to} & l_B \leq Bx \leq u_B \\ & l_x \leq x \leq u_x \end{array}$$

Nonlinear Programming (NLP) problems allow a general nonlinear objective function $f(x)$ and any of the nonlinear, linear or bound constraints. Special cases when some (or all) of the constraints are missing are termed as *unconstrained*, *bound-constrained* or *linearly-constrained nonlinear programming* and might have a specific solver as some algorithms take special provision for each of the constraint type. Problems with a linear or quadratic objective and nonlinear constraints should be still solved as general NLPs.

$$\begin{array}{ll} \underset{x \in R^n}{\text{minimize}} & f(x) \\ \text{subject to} & l_g \leq g(x) \leq u_g \\ & l_B \leq Bx \leq u_B \\ & l_x \leq x \leq u_x \end{array}$$

Semidefinite Programming (SDP) typically refers to *linear* semidefinite programming thus a problem with a linear objective function, linear constraints and linear matrix inequalities:

$$\begin{array}{ll} \underset{x \in R^n}{\text{minimize}} & c^{\mathrm{T}}x \\ \text{subject to} & \displaystyle\sum_{i=1}^{n} x_i A_i^k - A_0^k \succeq 0, \quad k = 1, \ldots, m_A \\ & l_B \leq Bx \leq u_B \\ & l_x \leq x \leq u_x \end{array}$$

This problem can be extended with a quadratic objective and bilinear (in fact quadratic) matrix inequalities. We refer to it as a semidefinite programming problem with **bilinear matrix inequalities (BMI-SDP)**:

$$\begin{array}{ll} \underset{x \in R^n}{\text{minimize}} & \frac{1}{2}x^{\mathrm{T}}Hx + c^{\mathrm{T}}x \\ \text{subject to} & \displaystyle\sum_{i,j=1}^{n} x_i x_j Q_{ij}^k + \sum_{i=1}^{n} x_i A_i^k - A_0^k \succeq 0, \quad k = 1, \ldots, m_A \\ & l_B \leq Bx \leq u_B \\ & l_x \leq x \leq u_x \end{array}$$

A **least squares (LSQ)** problem is a problem where the objective function in the form of sum of squares is minimized subject to usual constraints. If the residual functions $r_i(x)$ are linear or nonlinear, the problem is known as **linear** or **nonlinear least squares**, respectively. Not all types of the constraints need to be present which brings up special cases of *unconstrained*, *bound-constrained* or *linearly-constrained least squares problems* as in NLP .

$$\begin{array}{ll} \underset{x\in R^n}{\text{minimize}} & \displaystyle\sum_{i=1}^{m} r_i^2(x) \\ \text{subject to} & l_g \le g(x) \le u_g \\ & l_B \le Bx \le u_B \\ & l_x \le x \le u_x \end{array}$$

This form of the problem is very common in **data fitting** as demonstrated on the following example. Let us consider a process that is observed at times t_i and measured with results y_i, for $i = 1, 2, \ldots, m$. Furthermore, the process is assumed to behave according to a model $\phi(t; x)$ where x are parameters of the model. Given the fact that the measurements might be inaccurate and the process might not exactly follow the model, it is beneficial to find model parameters x so that the error of the fit of the model to the measurements is minimized. This can be formulated as an optimization problem in which x are decision variables and the objective function is the sum of squared errors of the fit at each individual measurement, thus:

$$\underset{x\in R^n}{\text{minimize}} \quad \sum_{i=1}^{m} r_i^2(x) \quad \text{where} \quad r_i(x) = \phi(t_i; x) - y_i$$

2.2.4 Problem size, dense and sparse problems

The size of the optimization problem plays an important role in the choice of the solver. The size is usually understood to be the number of variables n and the number (and the type) of the constraints. Depending on the size of the problem we talk about *small-scale*, *medium-scale* or *large-scale* problems.

It is often more practical to look at the data and its structure rather than just the size of the problem. Typically in a large-scale problem not all variables interact with everything else. It is natural that only a small portion of the constraints (if any) involves all variables and the majority of the constraints depends only on small different subsets of the variables. This creates many explicit zeros in the data representation which it is beneficial to capture and pass to the solver. In such a case the problem is referred to as **sparse**. The data representation usually has the form of a sparse matrix which defines the linear constraint matrix B, Jacobian matrix of the nonlinear constraints g_i or the Hessian of the objective H. Common sparse matrix formats are used, such as *coordinate storage (CS)* and *compressed column storage (CCS)* (see Section 2.1 in the f11 Chapter Introduction).

The counterpart to a sparse problem is a **dense** problem in which the matrices are stored in general full format and no structure is assumed or exploited. Whereas passing a dense problem to a sparse solver presents typically only a small overhead, *calling a dense solver on a large-scale sparse problem is ill-advised; it leads to a significant performance degradation and memory overuse.*

2.2.5 Derivative information, smoothness, noise and derivative-free optimization (DFO)

Most of the classical optimization algorithms rely heavily on derivative information. It plays a key role in necessary and sufficient conditions (see Section 2.4) and in the computation of the search direction at each iteration (see Section 2.5). Therefore it is important that accurate derivatives of the nonlinear objective and nonlinear constraints are provided whenever possible.

Unless stated otherwise, it is assumed that the nonlinear functions are sufficiently *smooth*. The solvers will usually solve optimization problems even if there are isolated discontinuities away from the solution, however you should always consider whether an alternative smooth representation of the problem exists. A typical example is an absolute value $|x_i|$ which does not have a first derivative for $x_i = 0$. Nevertheless, if the model allows it can be transformed as

$$x_i = x_i^+ - x_i^-, \quad |x_i| = x_i^+ + x_i^-, \quad \text{where} \quad x_i^+, x_i^- \ge 0$$

which avoids the discontinuity of the first derivative. If many discontinuities are present, alternative methods need to be applied such as **nag_opt_simplex_easy (e04cbc)** or stochastic algorithms in Chapter e05, **nag_glopt_bnd_pso (e05sac)** or **nag_glopt_nlp_pso (e05sbc)**.

The vector of first partial derivatives of a function is called the **gradient vector**, i.e.,

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right]^{\mathrm{T}},$$

the matrix of second partial derivatives is termed the **Hessian matrix**, i.e.,

$$\nabla^2 f(x) = \left[\frac{\partial^2 f(x)}{\partial x_i \partial x_j}\right]_{i,j=1,\ldots,n}$$

and the matrix of first partial derivatives of the vector-valued function $f : R^n \rightarrow R^m$ is known as the **Jacobian matrix**:

$$J(x) = \left[\frac{\partial f_i(x)}{\partial x_j}\right]_{i=1,\ldots,m,j=1,\ldots,n}.$$

If the function is smooth and the derivative is unavailable, it is possible to approximate it by **finite differences**, a change in function values in response to small perturbations of the variables. Many functions in the Library estimate missing elements of the gradients automatically this way. The choice of the size of the perturbations strongly affects the quality of the approximation. Too small perturbations might spoil the approximation due to the cancellation errors in floating-point arithmetic and too big reduce the match of the finite differences and the derivative (see **nag_opt_estimate_deriv (e04xac)** for optimal balance of the factors). In addition, finite differences are very sensitive to the accuracy of $f(x)$. They might be unreliable or fail completely if the function evaluation is inaccurate or *noisy* such as when $f(x)$ is a result of a stochastic simulation or an approximate solution of a PDE.

Derivative-free optimization (DFO) represents an alternative to derivative-based optimization algorithms. DFO solvers neither rely on derivative information nor approximate it by finite differences. They sample function evaluations across the domain to determine a new iteration point (for example, by a quadratic model through the sampled points). They are therefore less exposed to the relative error of the noise of the function because the sample points are never too close to each other to take the error into account. DFO might be useful even if the finite differences can be computed as the number of function evaluations is lower. This is particularly beneficial for problems where the evaluations of f are expensive. DFO solvers tend to exhibit a faster initial progress to the solution, however, they typically cannot achieve high-accurate solutions.

2.2.6 Minimization subject to bounds on the objective function

In all of the above problem categories it is assumed that

$$a \leq f(x) \leq b$$

where $a = -\infty$ and $b = +\infty$. Problems in which a and/or b are finite can be solved by adding an extra constraint of the appropriate type (i.e., linear or nonlinear) depending on the form of $f(x)$. Further advice is given in Section 4.5.

2.2.7 Multi-objective optimization

Sometimes a problem may have two or more objective functions which are to be optimized at the same time. Such problems are called **multi-objective**, *multi-criteria* or *multi-attribute* optimization. If the constraints are linear and the objectives are all linear then the terminology **goal programming** is also used.

Although there is no function dealing with this type of problems explicitly in this mark of the Library, techniques used in this chapter and in Chapter e05 may be employed to address such problems, see Section 2.5.5.

2.3 Geometric Representation

To illustrate the nature of optimization problems it is useful to consider the following example:

$$f(x) = e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right).$$

(This function is used as the example function in the documentation for the unconstrained functions.)

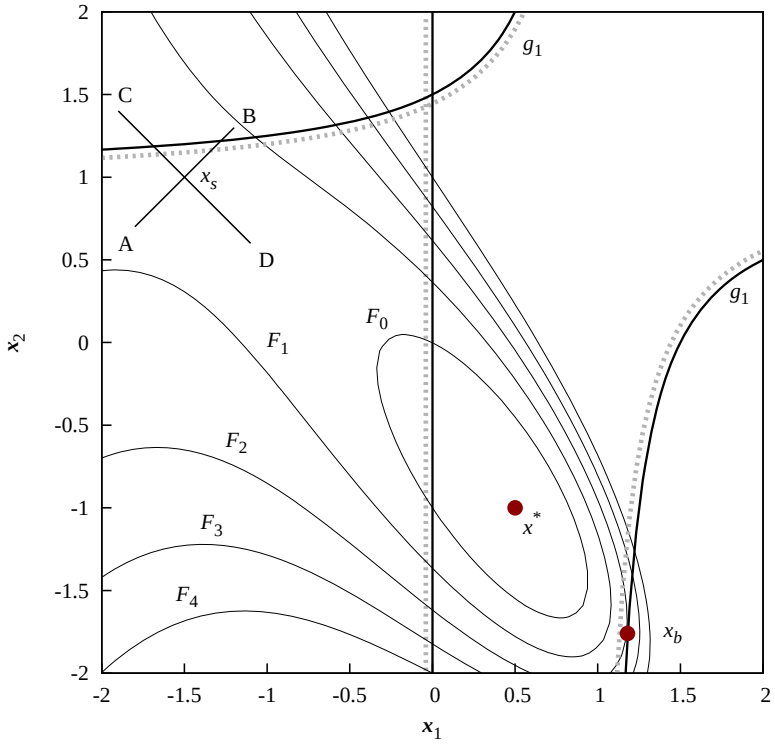


Figure 1

Figure 1 is a contour diagram of $f(x)$. The contours labelled $F_0, F_1, \ldots, F_4$ are *isovalue contours*, or lines along which the function $f(x)$ takes specific constant values. The point $x^* = \left(\frac{1}{2}, -1\right)^{\mathrm{T}}$ is a **local unconstrained minimum**, that is, the value of $f(x^*)$ $(= 0)$ is less than at all the neighbouring points. A function may have several such minima. The point x_s is said to be a **saddle point** because it is a minimum along the line AB, but a maximum along CD.

If we add the constraint $x_1 \geq 0$ (a simple bound) to the problem of minimizing $f(x)$, the solution remains unaltered. In Figure 1 this constraint is represented by the straight line passing through $x_1 = 0$, and the shading on the line indicates the unacceptable region (i.e., $x_1 < 0$).

If we add the nonlinear constraint $g_1(x) : x_1 + x_2 - x_1x_2 - \frac{3}{2} \geq 0$, represented by the curved shaded line in Figure 1, then x^* is not a feasible point because $g_1(x^*) < 0$. The solution of the new constrained problem is $x_b \simeq (1.1825, -1.7397)^{\mathrm{T}}$, the feasible point with the smallest function value (where $f(x_b) \simeq 3.0607$).

2.4 Sufficient Conditions for a Solution

All nonlinear functions will be assumed to have continuous second derivatives in the neighbourhood of the solution.

2.4.1 Unconstrained minimization

The following conditions are sufficient for the point x^* to be an unconstrained local minimum of $f(x)$:

(i) $\|\nabla f(x^*)\| = 0$ and

(ii) $\nabla^2 f(x^*)$ is positive definite,

where $\|\cdot\|$ denotes the Euclidean norm.

2.4.2 Minimization subject to bounds on the variables

At the solution of a bounds-constrained problem, variables which are not on their bounds are termed **free variables**. If it is known in advance which variables are on their bounds at the solution, the problem can be solved as an unconstrained problem in just the free variables; thus, the sufficient conditions for a solution are similar to those for the unconstrained case, applied only to the free variables.

Sufficient conditions for a feasible point x^* to be the solution of a bounds-constrained problem are as follows:

(i) $\|\bar{g}(x^*)\| = 0$; and

(ii) $\bar{G}(x^*)$ is positive definite; and

(iii) $\dfrac{\partial}{\partial x_j} f(x^*) < 0, x_j = u_j$; $\dfrac{\partial}{\partial x_j} f(x^*) > 0, x_j = l_j$,

where $\bar{g}(x)$ is the gradient of $f(x)$ with respect to the free variables, and $\bar{G}(x)$ is the Hessian matrix of $f(x)$ with respect to the free variables. The extra condition (iii) ensures that $f(x)$ cannot be reduced by moving off one or more of the bounds.

2.4.3 Linearly-constrained minimization

For the sake of simplicity, the following description does not include a specific treatment of bounds or range constraints, since the results for general linear inequality constraints can be applied directly to these cases.

At a solution x^*, of a linearly-constrained problem, the constraints which hold as equalities are called the **active** or **binding** constraints. Assume that there are t active constraints at the solution x^*, and let $\hat{A}$ denote the matrix whose columns are the columns of A corresponding to the active constraints, with $\hat{b}$ the vector similarly obtained from b; then

$$\hat{A}^{\mathrm{T}} x^* = \hat{b}.$$

The matrix Z is defined as an $n \times (n-t)$ matrix satisfying:

$$\begin{aligned} \hat{A}^{\mathrm{T}} Z &= 0; \\ Z^{\mathrm{T}} Z &= I. \end{aligned}$$

The columns of Z form an orthogonal basis for the set of vectors orthogonal to the columns of $\hat{A}$.

Define

$g_Z(x) = Z^{\mathrm{T}} \nabla f(x)$, the **projected gradient vector** of $f(x)$;

$G_Z(x) = Z^{\mathrm{T}} \nabla^2 f(x) Z$, the **projected Hessian matrix** of $f(x)$.

At the solution of a linearly-constrained problem, the projected gradient vector must be zero, which implies that the gradient vector $\nabla f(x^*)$ can be written as a linear combination of the columns of $\hat{A}$, i.e., $\nabla f(x^*) = \sum_{i=1}^{t} \lambda_i^* \hat{a}_i = \hat{A}\lambda^*$. The scalar λ_i^* is defined as the **Lagrange multiplier** corresponding to the ith active constraint. A simple interpretation of the ith Lagrange multiplier is that it gives the gradient of $f(x)$ along the ith active constraint normal; a convenient definition of the Lagrange multiplier vector (although not a recommended method for computation) is:

$$\lambda^* = \left(\hat{A}^{\mathrm{T}} \hat{A}\right)^{-1} \hat{A}^{\mathrm{T}} \nabla f(x^*).$$

Sufficient conditions for x^* to be the solution of a linearly-constrained problem are:

(i) x^* is feasible, and $\hat{A}^{\mathrm{T}}x^* = \hat{b}$; and

(ii) $\|g_Z(x^*)\| = 0$, or equivalently, $\nabla f(x^*) = \hat{A}\lambda^*$; and

(iii) $G_Z(x^*)$ is positive definite; and

(iv) $\lambda_i^* > 0$ if λ_i^* corresponds to a constraint $\hat{a}_i^{\mathrm{T}}x^* \geq \hat{b}_i$;

$\lambda_i^* < 0$ if λ_i^* corresponds to a constraint $\hat{a}_i^{\mathrm{T}}x^* \leq \hat{b}_i$.

The sign of λ_i^* is immaterial for equality constraints, which by definition are always active.

2.4.4 Nonlinearly-constrained minimization

For nonlinearly-constrained problems, much of the terminology is defined exactly as in the linearly-constrained case. To simplify the notation, let us assume that all nonlinear constraints are in the form $c(x) \geq 0$. The set of active constraints at x again means the set of constraints that hold as equalities at x, with corresponding definitions of $\hat{c}$ and $\hat{A}$: the vector $\hat{c}(x)$ contains the active constraint functions, and the columns of $\hat{A}(x)$ are the gradient vectors of the active constraints. As before, Z is defined in terms of $\hat{A}(x)$ as a matrix such that:

$$\hat{A}^{\mathrm{T}}Z = 0;$$
$$Z^{\mathrm{T}}Z = I$$

where the dependence on x has been suppressed for compactness.

The projected gradient vector $g_Z(x)$ is the vector $Z^{\mathrm{T}}\nabla f(x)$. At the solution x^* of a nonlinearly-constrained problem, the projected gradient must be zero, which implies the existence of Lagrange multipliers corresponding to the active constraints, i.e., $\nabla f(x^*) = \hat{A}(x^*)\lambda^*$.

The **Lagrangian function** is given by:

$$L(x, \lambda) = f(x) - \lambda^{\mathrm{T}}\hat{c}(x).$$

We define $g_L(x)$ as the gradient of the Lagrangian function; $G_L(x)$ as its Hessian matrix, and $\hat{G}_L(x)$ as its projected Hessian matrix, i.e., $\hat{G}_L = Z^{\mathrm{T}}G_L Z$.

Sufficient conditions for x^* to be the solution of a nonlinearly-constrained problem are:

(i) x^* is feasible, and $\hat{c}(x^*) = 0$; and

(ii) $\|g_Z(x^*)\| = 0$, or, equivalently, $\nabla f(x^*) = \hat{A}(x^*)\lambda^*$; and

(iii) $\hat{G}_L(x^*)$ is positive definite; and

(iv) $\lambda_i^* > 0$ if λ_i^* corresponds to a constraint of the form $\hat{c}_i \geq 0$.

The sign of λ_i^* is immaterial for equality constraints, which by definition are always active.

Note that condition (ii) implies that the projected gradient of the Lagrangian function must also be zero at x^*, since the application of Z^{T} annihilates the matrix $\hat{A}(x^*)$.

2.5 Background to Optimization Methods

All the algorithms contained in this chapter generate an iterative sequence $\{x^{(k)}\}$ that converges to the solution x^* in the limit, except for some special problem categories (i.e., linear and quadratic programming). To terminate computation of the sequence, a convergence test is performed to determine whether the current estimate of the solution is an adequate approximation. The convergence tests are discussed in Section 2.7.

Most of the methods construct a sequence $\{x^{(k)}\}$ satisfying:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)}p^{(k)},$$

where the vector $p^{(k)}$ is termed the **direction of search**, and $\alpha^{(k)}$ is the **steplength**. The steplength $\alpha^{(k)}$

is chosen so that $f\left(x^{(k+1)}\right) < f\left(x^{(k)}\right)$ and is computed using one of the techniques for one-dimensional optimization referred to in Section 2.5.1.

2.5.1 One-dimensional optimization

The Library contains two special functions for minimizing a function of a single variable. Both functions are based on safeguarded polynomial approximation. One function requires function evaluations only and fits a quadratic polynomial whilst the other requires function and gradient evaluations and fits a cubic polynomial. See Section 4.1 of Gill *et al.* (1981).

2.5.2 Methods for unconstrained optimization

The distinctions between methods arise primarily from the need to use varying levels of information about derivatives of $f(x)$ in defining the search direction. We describe three basic approaches to unconstrained problems, which may be extended to other problem categories. Since a full description of the methods would fill several volumes, the discussion here can do little more than allude to the processes involved, and direct you to other sources for a full explanation.

(a) **Newton-type** Methods (Modified Newton Methods)

Newton-type methods use the Hessian matrix $\nabla^2 f\left(x^{(k)}\right)$, or its finite difference approximation , to define the search direction. The functions in the Library either require a function that computes the elements of the Hessian directly, or they approximate them by finite differences.

Newton-type methods are the most powerful methods available for general problems and will find the minimum of a quadratic function in one iteration. See Sections 4.4 and 4.5.1 of Gill *et al.* (1981).

(b) **Quasi-Newton** Methods

Quasi-Newton methods approximate the Hessian $\nabla^2 f\left(x^{(k)}\right)$ by a matrix $B^{(k)}$ which is modified at each iteration to include information obtained about the curvature of f along the current search direction $p^{(k)}$. Although not as robust as Newton-type methods, quasi-Newton methods can be more efficient because the Hessian is not computed directly, or approximated by finite differences. Quasi-Newton methods minimize a quadratic function in n iterations, where n is the number of variables. See Section 4.5.2 of Gill *et al.* (1981).

(c) **Conjugate-gradient** Methods

Unlike Newton-type and quasi-Newton methods, conjugate-gradient methods do not require the storage of an n by n matrix and so are ideally suited to solve large problems. Conjugate-gradient type methods are not usually as reliable or efficient as Newton-type, or quasi-Newton methods. See Section 4.8.3 of Gill *et al.* (1981).

2.5.3 Methods for nonlinear least squares problems

These methods are similar to those for general nonlinear optimization, but exploit the special structure of the Hessian matrix to give improved computational efficiency.

Since

$$f(x) = \sum_{i=1}^{m} r_i^2(x)$$

the Hessian matrix is of the form

$$\nabla^2 f(x) = 2\left(J(x)^{\mathrm{T}} J(x) + \sum_{i=1}^{m} r_i(x)\nabla^2 r_i(x)\right),$$

where $J(x)$ is the Jacobian matrix of $r(x)$.

In the neighbourhood of the solution, $\|r(x)\|$ is often small compared to $\left\|J(x)^{\mathrm{T}} J(x)\right\|$ (for example, when $r(x)$ represents the goodness-of-fit of a nonlinear model to observed data). In such cases,

$2J(x)^{\mathrm{T}}J(x)$ may be an adequate approximation to $\nabla^2 f(x)$, thereby avoiding the need to compute or approximate second derivatives of $\{r_i(x)\}$. See Section 4.7 of Gill *et al.* (1981).

2.5.4 Methods for handling constraints

There are two main approaches for handling constraints in optimization algorithms – the **active-set sequential quadratic programming** method (or just SQP) and the **interior point method** (IPM). It is important to understand their very distinct features as both algorithms complement each other. The easiest method of comparison is to look at how the inequality constraints are treated and how the solver approaches the optimal solution (the progress of the KKT optimality measures: *optimality, feasibility, complementarity*).

Inequality constraints are the hard part of the optimization because of their 'twofold nature'. If the optimal solution strictly satisfies the inequality, i.e., the optimal point is in the interior of the constraint, the inequality constraint does not influence the result and could be removed from the model. On the other hand, if the inequality is satisfied as an equality (is *active* at the solution), the constraint must be present and could be treated as an equality from the very beginning. This is expressed by the complementarity in KKT conditions.

Solvers, based on the active-set method, solve at each iteration a quadratic approximation of the original problem; they try to estimate which constraints need to be kept (are active) and which can be ignored. A practical consequence is that the algorithm partly 'walks along the boundary' of the feasible region given by the constraints. The iterates are thus feasible early on with regard to all linear constraints (and a local linearization of the nonlinear constraints) which is preserved through the iterations. The complementarity is satisfied by default, and once the active set is determined correctly and optimality is within the tolerance, the solver finishes. The number of iterations might be high but each is relatively cheap. See Chapter 6 of Gill *et al.* (1981) for further details.

In contrast, an interior point method generates iterations that avoid the boundary defined by the inequality constraints. As the solver progresses the iterates are allowed to get closer and closer to the boundary and converge to the optimal solution which might lie on the boundary. From the practical point of view, IPM typically requires only tens of iterations. Each iteration consists of solving a large linear system of equations taking into account all variables and constraints, so each iteration is fairly expensive. All three optimality measures are reduced simultaneously.

In many cases it is difficult to predict which of the algorithms will behave better on a particular problem, however, some initial guidance can be given in the following table:

IPM advantages	**SQP advantages**
Can exploit second derivatives and its structure	Stay feasible with regard to linear constraints through most of the iterations
Efficient on unconstrained or loosely constrained problems	Very efficient for highly constrained problems
Efficient also for (both convex and nonconvex) quadratic problems (QP)	Better results on pathological problems in our experience
Better use of multi-core architecture (SMP library only)	Generally requires less function evaluations (efficient for problems with expensive function evaluations)
New interface, easier to use	Requires first derivatives but can work only with function values
	Can capitalize on a good initial point
	Allows warm starts (good for a sequence of similar problems)
	Infeasibility detection

Unless some of the specific features are required which are offered only by one algorithm, the initial decision should be based on the availability of the derivatives of the problem and the number of constraints (for example, expressed as a ratio between the numbers of variables and the sum of the number of linear and nonlinear constraints). Readiness of exact second derivatives is a clear advantage for IPM so unless the number of constraints is close to the number of variables, IPM will probably work better. Similarly, if a large-scale problem has relatively few constraints (e.g., less than 40%) IPM

might be more successful, especially as the problem gets bigger. On the other hand, if no derivatives are available, either the SQP or a specialized algorithm from the Library (see Derivative Free Optimization, Section 2.2.5) needs to be used. With more and more constraints SQP might be faster. For problems which do not fall in either of the categories, it is not easy to anticipate which solver will work better and some experimentation might be required.

2.5.5 Methods for handling multi-objective optimization

Suppose we have objective functions $f_i(x)$, $i > 1$, all of which we need to minimize at the same time. There are two main approaches to this problem:

(a) Combine the individual objectives into one composite objective. Typically this might be a weighted sum of the objectives, e.g.,

$$w_1 f_1(x) + w_2 f_2(x) + \cdots + w_n f_n(x)$$

Here you choose the weights to express the relative importance of the corresponding objective. Ideally each of the $f_i(x)$ should be of comparable size at a solution.

(b) Order the objectives in order of importance. Suppose f_i are ordered such that $f_i(x)$ is more important than $f_{i+1}(x)$, for $i = 1, 2, \ldots, n-1$. Then in the lexicographical approach to multi-objective optimization a sequence of subproblems are solved. Firstly solve the problem for objective function $f_1(x)$ and denote by r_1 the value of this minimum. If $(i-1)$ subproblems have been solved with results r_{i-1} then subproblem i becomes $\min(f_i(x))$ subject to $r_k \leq f_k(x) \leq r_k$, for $k = 1, 2, \ldots, i-1$ plus the other constraints.

Clearly the bounds on f_k might be relaxed at your discretion.

In general, if NAG functions from the Chapter e04 are used then only local minima are found. This means that a better solution to an individual objective might be found without worsening the optimal solutions to the other objectives. Ideally you seek a Pareto solution; one in which an improvement in one objective can only be achieved by a worsening of another objective.

To obtain a Pareto solution functions from Chapter e05 might be used or, alternatively, a pragmatic attempt to derive a global minimum might be tried (see **nag_glopt_nlp_multistart_sqp (e05ucc)**). In this approach a variety of different minima are computed for each subproblem by starting from a range of different starting points. The best solution achieved is taken to be the global minimum. The more starting points chosen the greater confidence you might have in the computed global minimum.

2.6 Scaling

Scaling (in a broadly defined sense) often has a significant influence on the performance of optimization methods.

Since convergence tolerances and other criteria are necessarily based on an implicit definition of ‘small’ and ‘large’, problems with unusual or unbalanced scaling may cause difficulties for some algorithms.

Although there are currently no user-callable scaling functions in the Library, scaling can be performed automatically in functions which solve sparse LP, QP or NLP problems and in some dense solver functions. Such functions have an optional parameter ‘Scale Option’ which can be set by the user; see individual function documents for details.

The following sections present some general comments on problem scaling.

2.6.1 Transformation of variables

One method of scaling is to transform the variables from their original representation, which may reflect the physical nature of the problem, to variables that have certain desirable properties in terms of optimization. It is generally helpful for the following conditions to be satisfied:

(i) the variables are all of similar magnitude in the region of interest;

(ii) a fixed change in any of the variables results in similar changes in $f(x)$. Ideally, a unit change in any variable produces a unit change in $f(x)$;

(iii) the variables are transformed so as to avoid cancellation error in the evaluation of $f(x)$.

Normally, you should restrict yourself to linear transformations of variables, although occasionally nonlinear transformations are possible. The most common such transformation (and often the most appropriate) is of the form

$$x_{\text{new}} = Dx_{\text{old}},$$

where D is a diagonal matrix with constant coefficients. Our experience suggests that more use should be made of the transformation

$$x_{\text{new}} = Dx_{\text{old}} + v,$$

where v is a constant vector.

Consider, for example, a problem in which the variable x_3 represents the position of the peak of a Gaussian curve to be fitted to data for which the extreme values are 150 and 170; therefore x_3 is known to lie in the range 150–170. One possible scaling would be to define a new variable $\bar{x}_3$, given by

$$\bar{x}_3 = \frac{x_3}{170}.$$

A better transformation, however, is given by defining $\bar{x}_3$ as

$$\bar{x}_3 = \frac{x_3 - 160}{10}.$$

Frequently, an improvement in the accuracy of evaluation of $f(x)$ can result if the variables are scaled before the functions to evaluate $f(x)$ are coded. For instance, in the above problem just mentioned of Gaussian curve-fitting, x_3 may always occur in terms of the form $(x_3 - x_m)$, where x_m is a constant representing the mean peak position.

2.6.2 Scaling the objective function

The objective function has already been mentioned in the discussion of scaling the variables. The solution of a given problem is unaltered if $f(x)$ is multiplied by a positive constant, or if a constant value is added to $f(x)$. It is generally preferable for the objective function to be of the order of unity in the region of interest; thus, if in the original formulation $f(x)$ is always of the order of 10^{+5} (say), then the value of $f(x)$ should be multiplied by 10^{-5} when evaluating the function within an optimization function. If a constant is added or subtracted in the computation of $f(x)$, usually it should be omitted, i.e., it is better to formulate $f(x)$ as $x_1^2 + x_2^2$ rather than as $x_1^2 + x_2^2 + 1000$ or even $x_1^2 + x_2^2 + 1$. The inclusion of such a constant in the calculation of $f(x)$ can result in a loss of significant figures.

2.6.3 Scaling the constraints

A 'well scaled' set of constraints has two main properties. Firstly, each constraint should be well-conditioned with respect to perturbations of the variables. Secondly, the constraints should be balanced with respect to each other, i.e., all the constraints should have 'equal weight' in the solution process.

The solution of a linearly- or nonlinearly-constrained problem is unaltered if the ith constraint is multiplied by a positive weight w_i. At the approximation of the solution determined by an active-set solver, any active linear constraints will (in general) be satisfied 'exactly' (i.e., to within the tolerance defined by ***machine precision***) if they have been properly scaled. This is in contrast to any active nonlinear constraints, which will not (in general) be satisfied 'exactly' but will have 'small' values (for example, $\hat{g}_1(x^*) = 10^{-8}$, $\hat{g}_2(x^*) = -10^{-6}$, and so on). In general, this discrepancy will be minimized if the constraints are weighted so that a unit change in x produces a similar change in each constraint.

A second reason for introducing weights is related to the effect of the size of the constraints on the Lagrange multiplier estimates and, consequently, on the active-set strategy. This means that different sets of weights may cause an algorithm to produce different sequences of iterates. Additional discussion is given in Gill *et al.* (1981).

2.7 Analysis of Computed Results

2.7.1 Convergence criteria

The convergence criteria inevitably vary from function to function, since in some cases more information is available to be checked (for example, is the Hessian matrix positive definite?), and different checks need to be made for different problem categories (for example, in constrained minimization it is necessary to verify whether a trial solution is feasible). Nonetheless, the underlying principles of the various criteria are the same; in non-mathematical terms, they are:

(i) is the sequence $\left\{x^{(k)}\right\}$ converging?

(ii) is the sequence $\left\{f^{(k)}\right\}$ converging?

(iii) are the necessary and sufficient conditions for the solution satisfied?

The decision as to whether a sequence is converging is necessarily speculative. The criterion used in the present functions is to assume convergence if the relative change occurring between two successive iterations is less than some prescribed quantity. Criterion (iii) is the most reliable but often the conditions cannot be checked fully because not all the required information may be available.

2.7.2 Checking results

Little *a priori* guidance can be given as to the quality of the solution found by a nonlinear optimization algorithm, since no guarantees can be given that the methods will not fail. Therefore, you should always check the computed solution even if the function reports success. Frequently a 'solution' may have been found even when the function does not report a success. The reason for this apparent contradiction is that the function needs to assess the accuracy of the solution. This assessment is not an exact process and consequently may be unduly pessimistic. Any 'solution' is in general only an approximation to the exact solution, and it is possible that the accuracy you have specified is too stringent.

Further confirmation can be sought by trying to check whether or not convergence tests are almost satisfied, or whether or not some of the sufficient conditions are nearly satisfied. When it is thought that a function has returned a value of **fail.code** other than **NE_NOERROR** only because the requirements for 'success' were too stringent it may be worth restarting with increased convergence tolerances.

For constrained problems, check whether the solution returned is feasible, or nearly feasible; if not, the solution returned is not an adequate solution.

Confidence in a solution may be increased by restarting the solver with a different initial approximation to the solution. See Section 8.3 of Gill *et al.* (1981) for further information.

2.7.3 Monitoring progress

Many of the functions in the chapter have facilities to allow you to monitor the progress of the minimization process, and you are encouraged to make use of these facilities. Monitoring information can be a great aid in assessing whether or not a satisfactory solution has been obtained, and in indicating difficulties in the minimization problem or in the ability of the function to cope with the problem.

The behaviour of the function, the estimated solution and first derivatives can help in deciding whether a solution is acceptable and what to do in the event of a return with a **fail.code** other than **NE_NOERROR**.

2.7.4 Confidence intervals for least squares solutions

When estimates of the parameters in a nonlinear least squares problem have been found, it may be necessary to estimate the variances of the parameters and the fitted function. These can be calculated from the Hessian of the objective $f(x)$ at the solution.

In many least squares problems, the Hessian is adequately approximated at the solution by $G = 2J^{\mathrm{T}}J$ (see Section 2.5.3). The Jacobian, J, or a factorization of J is returned by all the comprehensive least squares functions and, in addition, a function is available in the Library to estimate variances of the

parameters following the use of most of the nonlinear least squares functions, in the case that $G = 2J^{\mathrm{T}}J$ is an adequate approximation.

Let H be the inverse of G, and S be the sum of squares, both calculated at the solution $\bar{x}$; an unbiased estimate of the **variance** of the ith parameter x_i is

$$\operatorname{var} \bar{x}_i = \frac{2S}{m-n} H_{ii}$$

and an unbiased estimate of the covariance of $\bar{x}_i$ and $\bar{x}_j$ is

$$\operatorname{covar}\left(\bar{x}_i, \bar{x}_j\right) = \frac{2S}{m-n} H_{ij}.$$

If x^* is the true solution, then the $100(1-\beta)\%$ **confidence interval** on $\bar{x}$ is

$$\bar{x}_i - \sqrt{\operatorname{var} \bar{x}_i}.t_{(1-\beta/2,m-n)} < x_i^* < \bar{x}_i + \sqrt{\operatorname{var} \bar{x}_i}.t_{(1-\beta/2,m-n)}, \quad i = 1, 2, \ldots, n$$

where $t_{(1-\beta/2,m-n)}$ is the $100(1-\beta)/2$ percentage point of the t-distribution with $m-n$ degrees of freedom.

In the majority of problems, the residuals r_i, for $i = 1, 2, \ldots, m$, contain the difference between the values of a model function $\phi(z, x)$ calculated for m different values of the independent variable z, and the corresponding observed values at these points. The minimization process determines the parameters, or constants x, of the fitted function $\phi(z, x)$. For any value, $\bar{z}$, of the independent variable z, an unbiased estimate of the **variance** of ϕ is

$$\operatorname{var} \phi = \frac{2S}{m-n} \sum_{i=1}^{n} \sum_{j=1}^{n} \left[\frac{\partial \phi}{\partial x_i}\right]_{\bar{z}} \left[\frac{\partial \phi}{\partial x_j}\right]_{\bar{z}} H_{ij}.$$

The $100(1-\beta)\%$ **confidence interval** on f at the point $\bar{z}$ is

$$\phi(\bar{z}, \bar{x}) - \sqrt{\operatorname{var} \phi}.t_{(\beta/2,m-n)} < \phi(\bar{z}, x^*) < \phi(\bar{z}, \bar{x}) + \sqrt{\operatorname{var} \phi}.t_{(\beta/2,m-n)}.$$

For further details on the analysis of least squares solutions see Bard (1974) and Wolberg (1967).

3 Optional Facilities

The comments in this section do not apply to functions introduced at Mark 8 and later, viz. **nag_opt_sparse_convex_qp_solve (e04nqc)**, **nag_opt_nlp_revcomm (e04ufc)**, **nag_opt_sparse_nlp_solve (e04vhc)** and **nag_opt_nlp_solve (e04wdc)**. For details of their optional facilities please refer to their individual documents.

The optimization functions of this chapter provide a range of optional facilities: these offer the possibility of fine control over many of the algorithmic parameters and the means of adjusting the level and nature of the printed results.

Control of these optional facilities is exercised by a structure of type Nag_E04_Opt, the members of the structure being optional input or output arguments to the function. After declaring the structure variable, which is named **options** in this manual, you must initialize the structure by passing its address in a call to the utility function **nag_opt_init (e04xxc)**. Selected members of the structure may then be set to your required values and the address of the structure passed to the optimization function. Any member which has not been set by you will indicate to the optimization function that the default value should be used for this argument. A more detailed description of this process is given in Section 3.4.

The optimization process may sometimes terminate before a satisfactory answer has been found, for instance when the limit on the number of iterations has been reached. In such cases you may wish to re-enter the function making use of the information already obtained. Functions **nag_opt_conj_grad (e04dgc)**, **nag_opt_lsq_no_deriv (e04fcc)** and **nag_opt_lsq_deriv (e04gbc)** can simply be re-entered but the functions **nag_opt_bounds_deriv (e04kbc)**, **nag_opt_lp (e04mfc)**, **nag_opt_lin_lsq (e04ncc)**, **nag_opt_qp (e04nfc)**, **nag_opt_sparse_convex_qp (e04nkc)**, **nag_opt_nlp (e04ucc)**, **nag_opt_n lin_lsq (e04unc)** and **nag_opt_nlp_solve (e04wdc)** have a structure member which needs to be set appropriately if the function is to make use of information from the previous call. The member is named **start** in the functions listed.

3.1 Control of Printed Output

Results from the optimization process are printed by default on the `stdout` (standard output) stream. These include the results after each iteration and the final results at termination of the search process. The amount of detail printed out may be increased or decreased by setting the optional parameter **Print Level**, i.e., the structure member **Print Level**. This member is an `enum` type, Nag_PrintType, and an example value is Nag_Soln which when assigned to **Print Level** will cause the optimization function to print only the final result; all intermediate results printout is suppressed.

If the results printout is not in the desired form then it may be switched off, by setting **Print Level** = Nag_NoPrint, or alternatively you can supply your own function to print out or make use of both the intermediate and final results. Such a function would be assigned to the pointer to function member print_fun; the user-defined function would then be called in preference to the NAG print function.

In addition to the results, the values of the arguments to the optimization function are printed out when the function is entered; the Boolean member **list** may be set to Nag_FALSE if this listing is not required.

Printing may be output to a named file rather than to `stdout` by providing the name of the file in the **options** character array member **outfile**. Error messages will still appear on `stderr`, if **fail**.**print** = Nag_TRUE or the **fail** argument is not supplied (see the Section 3.7 in How to Use the NAG Library and its Documentation for details of error handling within the library).

3.2 Memory Management

The **options** structure contains a number of pointers for the input of data and the output of results. The optimization functions will manage the allocation of memory to these pointers; when all calls to these functions have been completed then a utility function **nag_opt_free (e04xzc)** can be called by your program to free the NAG allocated memory which is no longer required.

If the calling function is part of a larger program then this utility function allows you to conserve memory by freeing the NAG allocated memory before the **options** structure goes out of scope. **nag_opt_free (e04xzc)** can free all NAG allocated memory in a single call, but it may also be used selectively. In this case the memory assigned to certain pointers may be freed leaving the remaining memory still available; pointers to this memory and the results it contains may then be passed to other functions in your program without passing the structure and all its associated memory.

Although the NAG C Library optimization functions will manage all memory allocation and deallocation, it may occasionally be necessary for you to allocate memory to the **options** structure from within the calling program before entering the optimization function.

An example of this is where you store information in a file from an optimization run and at a later date wish to use that information to solve a similar optimization problem or the same one under slightly changed conditions. The pointer **state**, for example, would need to be allocated memory by you before the status of the constraints could be assigned from the values in the file. The member **Cold Start** would need to be appropriately set for functions **nag_opt_lp (e04mfc)** and **nag_opt_qp (e04nfc)**.

If you assign memory to a pointer within the **options** structure then the deallocation of this memory must also be performed by you; the utility function **nag_opt_free (e04xzc)** will only free memory allocated by NAG C Library optimization functions. When your allocated memory is freed using the standard C library function `free()` then the pointer should be set to **NULL** immediately afterwards; this will avoid possible confusion in the NAG memory management system if a NAG function is subsequently entered. In general we recommend the use of `NAG_ALLOC`, `NAG_REALLOC` and `NAG_FREE` for allocating and freeing memory used with NAG functions.

3.3 Reading Optional Parameter Values From a File

Optional parameter values may be placed in a file by you and the function **nag_opt_read (e04xyc)** used to read the file and assign the values to the **options** structure. This utility function permits optional parameter values to be supplied in any order and altered without recompilation of the program. The values read are also checked before assignment to ensure they are in the correct range for the specified option. Pointers within the **options** structure cannot be assigned to using **nag_opt_read (e04xyc)**.

3.4 Method of Setting Optional Parameters

The method of using and setting the optional parameters is:

step 1 declare a structure of type Nag_E04_Opt.

step 2 initialize the structure using **nag_opt_init (e04xxc)**.

step 3 assign values to the structure.

step 4 pass the address of the structure to the optimization function.

step 5 call **nag_opt_free (e04xzc)** to free any memory allocated by the optimization function.

If after step 4, it is wished to re-enter the optimization function, then step 3 can be returned to directly, i.e., step 5 need only be executed when all calls to the optimization function have been made.

At step 3, values can be assigned directly and/or by means of the option file reading function **nag_opt_read (e04xyc)**. If values are only assigned from the options file then step 2 need not be performed as **nag_opt_read (e04xyc)** will automatically call **nag_opt_init (e04xxc)** if the structure has not been initialized.

4 Recommendations on Choice and Use of Available Functions

The choice of function depends on several factors: the type of problem (unconstrained, etc.); the level of derivative information available (function values only, etc.); your experience (there are easy-to-use versions of some functions); whether or not a problem is sparse; and whether computational time has a high priority. Not all choices are catered for in the current version of the Library.

4.1 Reverse Communication Functions

Most of the functions in this chapter are called just once in order to compute the minimum of a given objective function subject to a set of constraints on the variables. The objective function and nonlinear constraints (if any) are specified by you and written as functions to a very rigid format described in the relevant function document.

This chapter also contains a **reverse communication** function, **nag_opt_nlp_revcomm (e04ufc)**, which solves dense NLP problems using a sequential quadratic programming method. This may be convenient to use when the minimization function is being called from a computer language which does not fully support procedure arguments in a way that is compatible with the Library. This function is also useful if a large amount of data needs to be transmitted into the function. See Section 3.3.2 in How to Use the NAG Library and its Documentation for more information about reverse communication functions.

4.2 Choosing Between Variant Functions for Some Problems

As evidenced by the wide variety of functions available in Chapter e04, it is clear that no single algorithm can solve all optimization problems. It is important to try to match the problem to the most suitable function, and that is what the decision trees in Section 5 help to do.

Sometimes in Chapter e04 more than one function is available to solve precisely the same optimization problem. If their differences lay in the underlying method, refer to the sections above. Section 2.5.4 discusses key features of interior point methods (represented by **nag_opt_handle_solve_ipopt (e04stc)**) and active-set SQP methods (for example, **nag_opt_nlp_sparse (e04ugc)** or **nag_opt_sparse_nlp_solve (e04vhc)**). Alternatively, there are functions implementing slightly different variants of the same method (such as **nag_opt_nlp (e04ucc)** and **nag_opt_nlp_solve (e04wdc)**). Experience shows that in this case although both functions can usually solve the same problem and get similar results, sometimes one function will be faster, sometimes one might find a different local minimum to the other, or, in difficult cases, one function may obtain a solution when the other one fails.

After using one of these functions, if the results obtained are unacceptable for some reason, it may be worthwhile trying the other function instead. In the absence of any other information, in the first instance you are recommended to try using **nag_opt_nlp (e04ucc)**, and if that proves unsatisfactory, try using **nag_opt_nlp_solve (e04wdc)**. Although the algorithms used are very similar, the two functions

each have slightly different optional parameters which may allow the course of the computation to be altered in different ways.

Other pairs of functions which solve the same kind of problem are **nag_opt_sparse_convex_qp_solve (e04nqc)** (recommended first choice) or **nag_opt_sparse_convex_qp (e04nkc)**, for sparse quadratic or linear programming problems, and **nag_opt_sparse_nlp_solve (e04vhc)** (recommended) or **nag_opt_nlp_sparse (e04ugc)**, for sparse nonlinear programming. In these cases the argument lists are not so similar as **nag_opt_nlp (e04ucc)** or **nag_opt_nlp_solve (e04wdc)**, but the same considerations apply.

4.3 NAG Optimization Modelling Suite

Mark 26 of the Library introduced the *NAG optimization modelling suite*, a suite of functions which allows you to define and solve various optimization problems in a uniform manner. The first key feature of the suite is that the definition of the optimization problem and the call to the solver have been separated so it is possible to set up a problem in the same way for different solvers. The second feature is that the problem representation is built up from basic components (for example, a QP problem is composed of a quadratic objective, simple bounds and linear constraints), therefore different types of problems reuse the same functions for their common parts.

A connecting element to all functions in the suite is a *handle*, a pointer to an internal data structure, which is passed among the functions. It holds all information about the problem, the solution and the solver. Each handle should go through four stages in its life: *initialization*, *problem formulation*, *problem solution* and *deallocation*.

The initialization is performed by **nag_opt_handle_init (e04rac)** which creates an empty problem with n decision variables. A call to **nag_opt_handle_free (e04rzc)** marks the end of the life of the handle as it deallocates all the allocated memory and data within the handle and destroys the handle itself. After the initialization, the objective may be defined as one of the following:

nag_opt_handle_set_linobj (e04rec) – a linear objective as a dense vector;

nag_opt_handle_set_quadobj (e04rfc) – a quadratic objective or a sparse linear objective;

nag_opt_handle_set_nlnobj (e04rgc) – a nonlinear objective function;

nag_opt_handle_set_nlnls (e04rmc) – a nonlinear least squares objective function.

The functions for constraint definition are

nag_opt_handle_set_simplebounds (e04rhc) – simple bounds;

nag_opt_handle_set_linconstr (e04rjc) – linear constraints;

nag_opt_handle_set_nlnconstr (e04rkc) – nonlinear constraints;

nag_opt_handle_set_nlnhess (e04rlc) – second derivatives for the objective and/or constraints;

nag_opt_handle_set_linmatineq (e04rnc) – linear matrix inequalities;

nag_opt_handle_set_quadmatineq (e04rpc) – quadratic terms for bilinear matrix inequalities.

These functions may be called in an arbitrary order, however, a call to **nag_opt_handle_set_linmatineq (e04rnc)** must precede a call to **nag_opt_handle_set_quadmatineq (e04rpc)** for the matrix inequalities with bilinear terms and the nonlinear objective or constraints (**nag_opt_handle_set_nlnobj (e04rgc)** or **nag_opt_handle_set_nlnconstr (e04rkc)**) must precede the definition of the second derivatives by **nag_opt_handle_set_nlnhess (e04rlc)**. For further details please refer to the documentation of the individual functions.

The suite also includes the following service functions:

nag_opt_handle_print (e04ryc) – query/printing function;

nag_opt_handle_opt_set (e04zmc) – supply an optional parameter from a character string;

nag_opt_handle_opt_set_file (e04zpc) – supply one or more optional parameters from a file;

nag_opt_handle_opt_get (e04znc) – get the settings of an optional parameter;

nag_opt_handle_set_get_real (e04rxc) – read or write information into the handle.

When the problem is fully formulated, the handle can be passed to a solver which is compatible with the defined problem. At the current mark of the Library the NAG optimization modelling suite comprises of **nag_opt_handle_solve_dfls (e04ffc)**, **nag_opt_handle_solve_lp_ipm (e04mtc)**, **nag_opt_handle_solve_ipopt (e04stc)** and **nag_opt_handle_solve_pennon (e04svc)**. The solver indicates by an error flag if it cannot deal with the given formulation. A diagram of the life cycle of the handle is depicted in Figure 2.

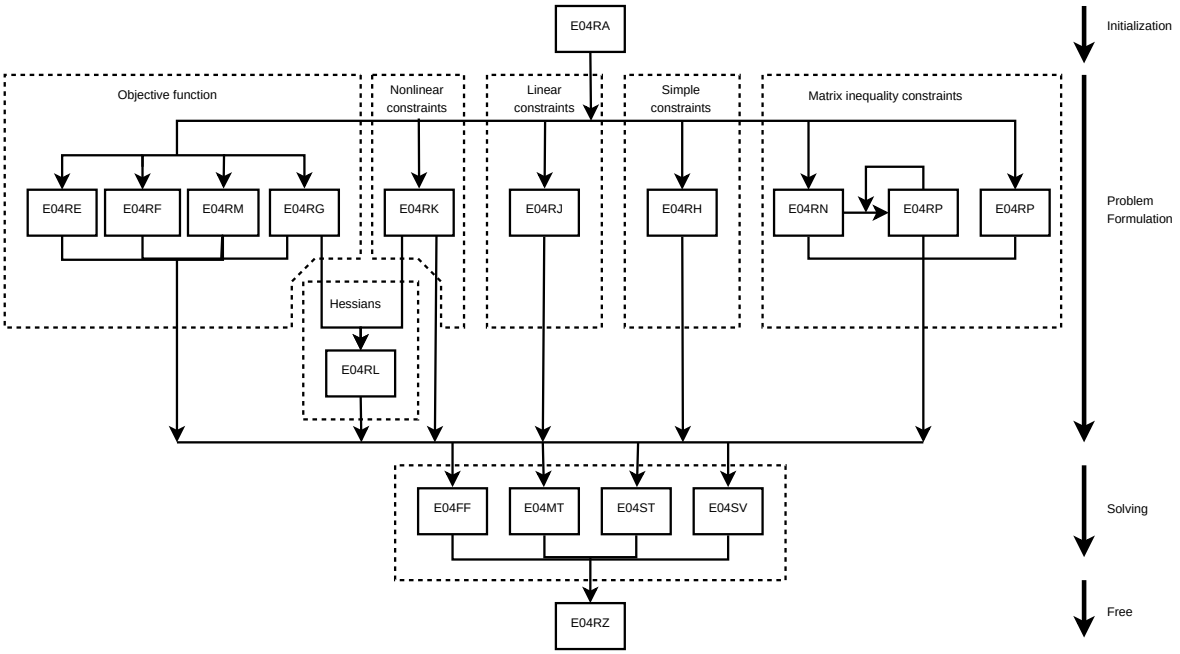


Figure 2

4.4 Service Functions

One of the most common errors in the use of optimization functions is that user-supplied functions do not evaluate the relevant partial derivatives correctly. Because exact gradient information normally enhances efficiency in all areas of optimization, you are encouraged to provide analytical derivatives whenever possible. However, mistakes in the computation of derivatives can result in serious and obscure run-time errors. Consequently, **service functions** are provided to perform an elementary check on the gradients you supplied. These functions are inexpensive to use in terms of the number of calls they require to user-supplied functions.

The appropriate checking function is as follows:

Minimization function	**Checking function(s)**
nag_opt_bounds_2nd_deriv (e04lbc)	nag_opt_check_deriv (e04hcc) and nag_opt_check_2nd_deriv (e04hdc)
nag_opt_lsq_deriv (e04gbc)	nag_opt_lsq_check_deriv (e04yac)

It should be noted that functions **nag_opt_handle_solve_ipopt (e04stc)**, **nag_opt_nlp (e04ucc)**, **nag_opt_nlp_revcomm (e04ufc)**, **nag_opt_nlp_sparse (e04ugc)**, **nag_opt_nlin_lsq (e04unc)**, **nag_opt_sparse_nlp_solve (e04vhc)** and **nag_opt_nlp_solve (e04wdc)** each incorporate a check on the derivatives being supplied. This involves verifying the gradients at the first point that satisfies the linear constraints and bounds. There is also an option to perform a more reliable (but more expensive) check on the individual gradient elements being supplied. Note that the checks are not infallible.

A second type of service function computes a set of finite differences to be used when approximating first derivatives. Such differences are required as input arguments by some functions that use only function evaluations.

nag_opt_lsq_covariance (e04ycc) estimates selected elements of the variance-covariance matrix for the computed regression parameters following the use of a nonlinear least squares function.

nag_opt_estimate_deriv (e04xac) estimates the gradient and Hessian of a function at a point, given a function to calculate function values only, or estimates the Hessian of a function at a point, given a function to calculate function and gradient values.

4.5 Function Evaluations at Infeasible Points

All the solvers for constrained problems based on an active-set method will ensure that any evaluations of the objective function occur at points which **approximately** (up to the given tolerance) satisfy any **simple bounds** or **linear constraints**.

There is no attempt to ensure that the current iteration satisfies any nonlinear constraints. If you wish to prevent your objective function being evaluated outside some known region (where it may be undefined or not practically computable), you may try to confine the iteration within this region by imposing suitable simple bounds or linear constraints (but beware as this may create new local minima where these constraints are active).

Note also that some functions allow you to return the argument (**comm** → **flag**) with a negative value to indicate when the objective function (or nonlinear constraints where appropriate) cannot be evaluated. In case the function cannot recover (e.g., cannot find a different trial point), it forces an immediate clean exit from the function. Please note that **nag_opt_sparse_convex_qp_solve (e04nqc)**, **nag_opt_sparse_nlp_solve (e04vhc)** and **nag_opt_nlp_solve (e04wdc)** use the user-supplied function **imode** instead of **comm** → **flag**.

4.6 Related Problems

Apart from the standard types of optimization problem, there are other related problems which can be solved by functions in this or other chapters of the Library.

nag_ip_bb (h02bbc) solves **dense integer LP** problems.

Several functions in Chapters f04 and f08 solve **linear least squares problems**, i.e., minimize $\sum_{i=1}^{m} r_i(x)^2$ where $r_i(x) = b_i - \sum_{j=1}^{n} a_{ij}x_j$.

nag_lone_fit (e02gac) solves an overdetermined system of linear equations in the l_1 norm, i.e., minimizes $\sum_{i=1}^{m} |r_i(x)|$, with r_i as above.

nag_linf_fit (e02gcc) solves an overdetermined system of linear equations in the l_∞ norm, i.e., minimizes $\max_i |r_i(x)|$, with r_i as above.

Chapter e05 contains functions for global minimization.

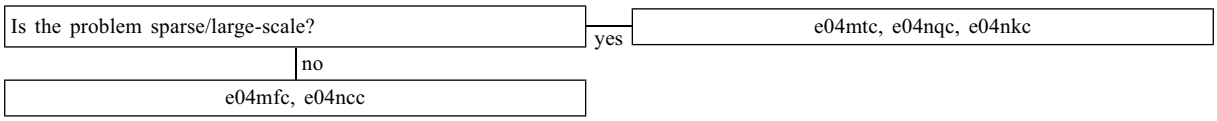
Section 2.5.5 describes how a multi-objective optimization problem might be addressed using functions from this chapter and from Chapter e05.

5 Decision Trees

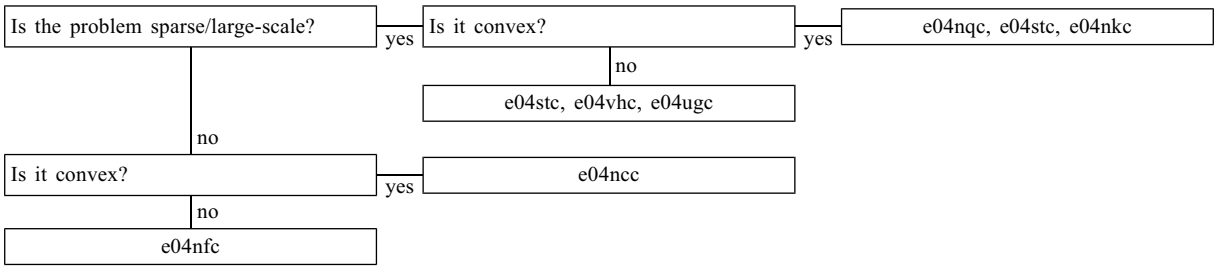
	no objective	linear	quadratic	nonlinear	sum of squares
unconstrained			QP See Tree 2	NLP See Tree 3	LSQ See Tree 4
simple bounds	LP See Tree 1	LP See Tree 1	QP See Tree 2	NLP See Tree 3	LSQ See Tree 4
linear	LP See Tree 1	LP See Tree 1	QP See Tree 2	NLP See Tree 3	LSQ See Tree 4
nonlinear	NLP See Tree 3	NLP See Tree 3	NLP See Tree 3	NLP See Tree 3	LSQ See Tree 4
matrix inequalities	e04svc	e04svc	e04svc		

Table 1
Decision Matrix

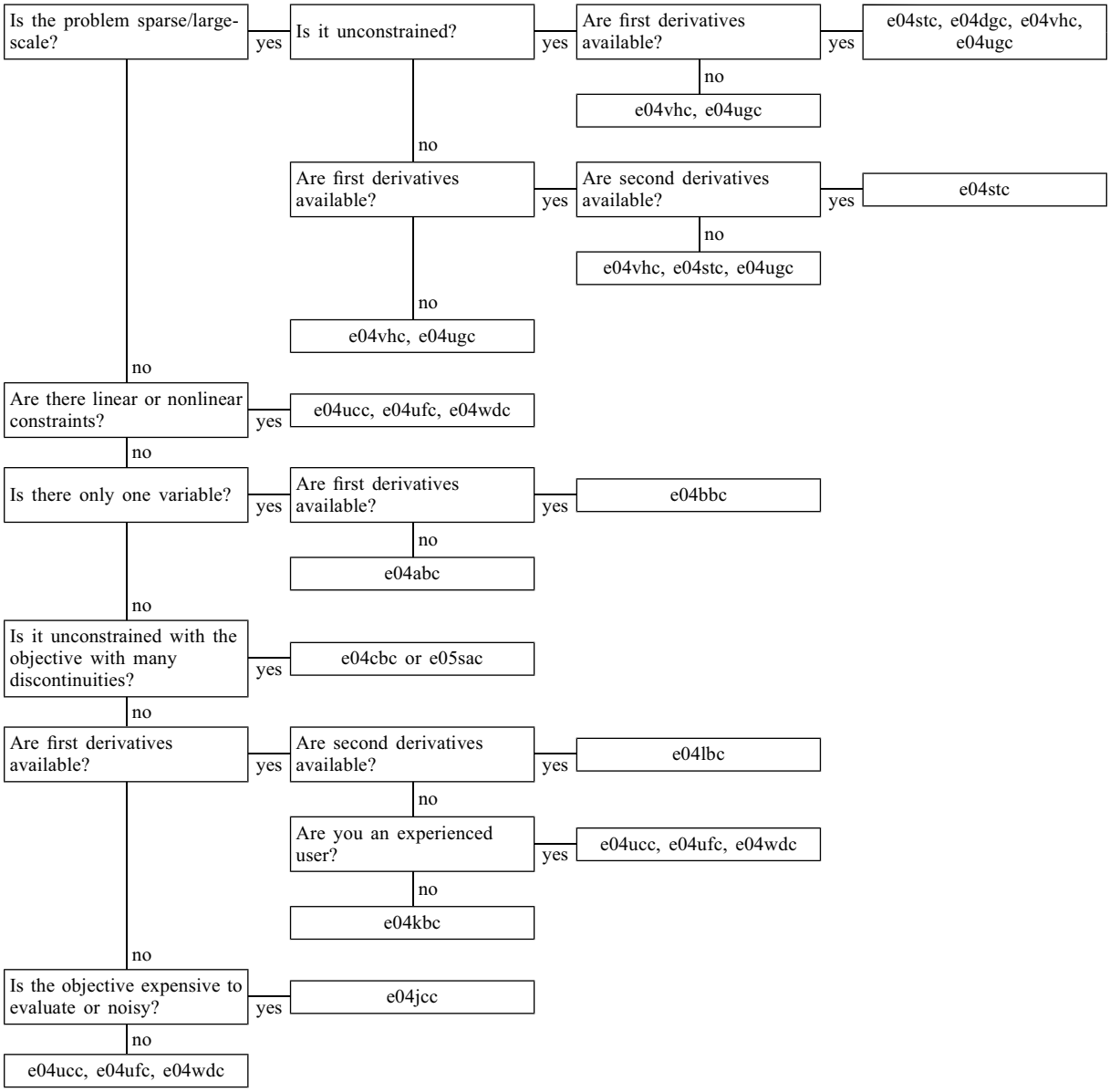
Tree 1: Linear Programming (LP)



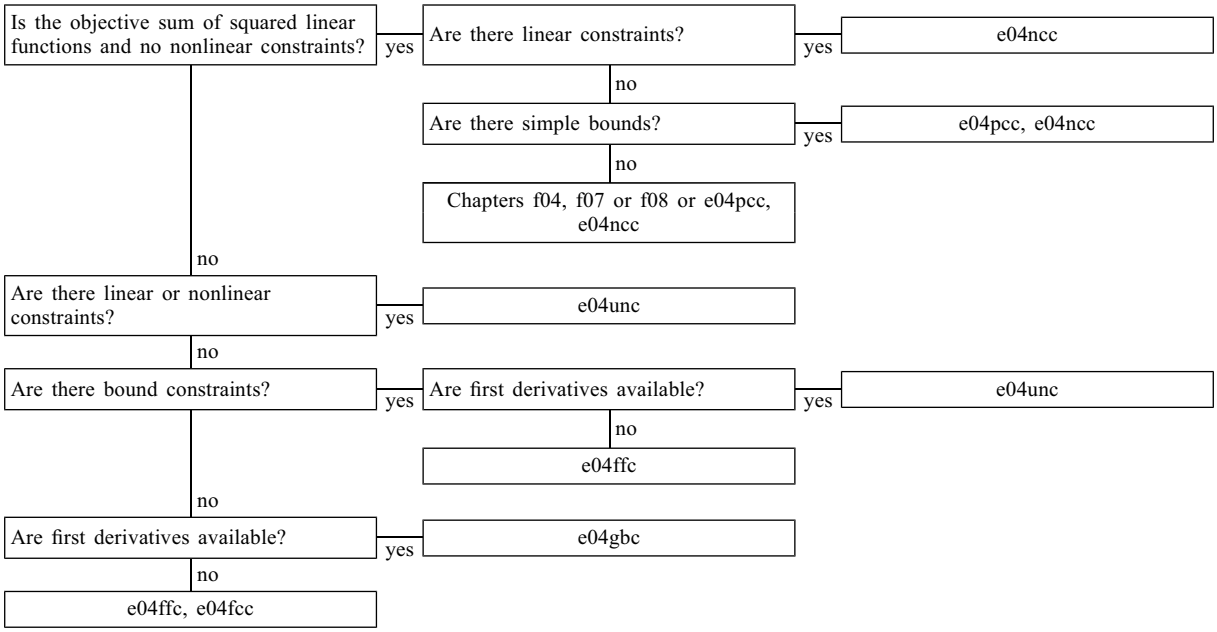
Tree 2: Quadratic Programming (QP)



Tree 3: Nonlinear Programming (NLP)



Tree 4: Least squares problems (LSQ)



6 Functionality Index

Linear programming (LP),
- dense,
  - active-set method/primal simplex,
    - alternative 1 ..... nag_opt_lp (e04mfc)
    - alternative 2 ..... nag_opt_lin_lsq (e04ncc)
- sparse,
  - interior point method (IPM) ..... nag_opt_handle_solve_lp_ipm (e04mtc)
  - active-set method/primal simplex,
    - recommended (see Section 4.2) ..... nag_opt_sparse_convex_qp_solve (e04nqc)
    - alternative ..... nag_opt_sparse_convex_qp (e04nkc)

Quadratic programming (QP),
- dense,
  - active-set method for (possibly nonconvex) QP problem ..... nag_opt_qp (e04nfc)
  - active-set method for convex QP problem ..... nag_opt_lin_lsq (e04ncc)
- sparse,
  - active-set method sparse convex QP problem,
    - recommended (see Section 4.2) ..... nag_opt_sparse_convex_qp_solve (e04nqc)
    - alternative ..... nag_opt_sparse_convex_qp (e04nkc)
    - interior point method (IPM) for (possibly nonconvex) QP problems ..... nag_opt_handle_solve_ipopt (e04stc)

Nonlinear programming (NLP),
- dense,
  - active-set sequential quadratic programming (SQP),
    - recommended (see Section 4.2) ..... nag_opt_nlp (e04ucc)
    - alternative ..... nag_opt_nlp_solve (e04wdc)
    - reverse communication ..... nag_opt_nlp_revcomm (e04ufc)
- sparse,
  - interior point method (IPM) ..... nag_opt_handle_solve_ipopt (e04stc)
  - active-set sequential quadratic programming (SQP),
    - recommended (see Section 4.2) ..... nag_opt_sparse_nlp_solve (e04vhc)
    - alternative ..... nag_opt_nlp_sparse (e04ugc)

Nonlinear programming (NLP) – derivative free optimization (DFO),
- model-based method for bound-constrained optimization ..... nag_opt_bounds_qa_no_deriv (e04jcc)

Nelder–Mead simplex method for unconstrained optimization ..... nag_opt_simplex_easy (e04cbc)

Nonlinear programming (NLP) – special cases,
unidimensional optimization (one-dimensional) with bound constraints,
method based on quadratic interpolation, no derivatives ..... nag_opt_one_var_no_deriv (e04abc)
method based on cubic interpolation ..... nag_opt_one_var_deriv (e04bbc)
unconstrained,
preconditioned conjugate gradient method ..... nag_opt_conj_grad (e04dgc)
bound-constrained,
quasi-Newton algorithm, first derivatives ..... nag_opt_bounds_deriv (e04kbc)
modified Newton algorithm, first and second derivatives ..... nag_opt_bounds_2nd_deriv (e04lbc)

Semidefinite programming (SDP),
generalized augmented Lagrangian method for SDP and SDP with bilinear matrix inequalities (BMI-SDP) ..... nag_opt_handle_solve_pennon (e04svc)

Linear least squares, linear regression, data fitting,
constrained,
bound-constrained least squares problem ..... nag_opt_bnd_lin_lsq (e04pcc)
linearly-constrained active-set method ..... nag_opt_lin_lsq (e04ncc)

Nonlinear least squares, data fitting,
unconstrained,
combined Gauss–Newton and modified Newton algorithm,
no derivatives ..... nag_opt_lsq_no_deriv (e04fcc)
combined Gauss–Newton and quasi-Newton algorithm,
first derivatives ..... nag_opt_lsq_deriv (e04gbc)
covariance matrix for nonlinear least squares problem (unconstrained)
..... nag_opt_lsq_covariance (e04ycc)
model-based derivative free algorithm ..... nag_opt_handle_solve_dfls (e04ffc)
constrained,
nonlinear constraints active-set sequential quadratic programming (SQP)
..... nag_opt_nlin_lsq (e04unc)
bound constrained,
model-based derivative free algorithm ..... nag_opt_handle_solve_dfls (e04ffc)

NAG optimization modelling suite,
initialization of a handle for the NAG optimization modelling suite ..... nag_opt_handle_init (e04rac)
define a linear objective function ..... nag_opt_handle_set_linobj (e04rec)
define a linear or a quadratic objective function ..... nag_opt_handle_set_quadobj (e04rfc)
define a nonlinear least square objective function ..... nag_opt_handle_set_nlnls (e04rmc)
define a nonlinear objective function ..... nag_opt_handle_set_nlnobj (e04rgc)
define bounds of variables ..... nag_opt_handle_set_simplebounds (e04rhc)
define a block of linear constraints ..... nag_opt_handle_set_linconstr (e04rjc)
define a block of nonlinear constraints ..... nag_opt_handle_set_nlnconstr (e04rkc)
define a structure of Hessian of the objective, constraints or the Lagrangian
..... nag_opt_handle_set_nlnhess (e04rlc)
add one or more linear matrix inequality constraints ..... nag_opt_handle_set_linmatineq (e04rnc)
define bilinear matrix terms ..... nag_opt_handle_set_quadmatineq (e04rpc)
print information about a problem handle ..... nag_opt_handle_print (e04ryc)
set/get information in a problem handle ..... nag_opt_handle_set_get_real (e04rxc)
destroy the problem handle ..... nag_opt_handle_free (e04rzc)
interior point method (IPM) for linear programming (LP)
..... nag_opt_handle_solve_lp_ipm (e04mtc)
interior point method (IPM) for nonlinear programming (NLP)
..... nag_opt_handle_solve_ipopt (e04stc)
generalized augmented Lagrangian method for SDP and SDP with bilinear matrix inequalities (BMI-SDP) ..... nag_opt_handle_solve_pennon (e04svc)
supply optional parameter values from a character string ..... nag_opt_handle_opt_set (e04zmc)
get the setting of option ..... nag_opt_handle_opt_get (e04znc)
supply optional parameter values from external file ..... nag_opt_handle_opt_set_file (e04zpc)

Service functions,
- input and output (I/O),
  - read MPS data file defining LP, QP, MILP or MIQP problem ..... nag_opt_miqp_mps_read (e04mxc)
  - write MPS data file defining LP, QP, MILP or MIQP problem ..... nag_opt_miqp_mps_write (e04mwc)
  - read sparse SPDA data files for linear SDP problems ..... nag_opt_sdp_read_sdpa (e04rdc)
  - read MPS data file defining LP or QP problem (deprecated) ..... nag_opt_sparse_mps_read (e04mzc)
  - free memory allocated by reader nag_opt_sparse_mps_read (e04mzc) (deprecated) ..... nag_opt_sparse_mps_free (e04myc)
- derivative check and approximation,
  - check user's function for calculating first derivatives of function ..... nag_opt_check_deriv (e04hcc)
  - check user's function for calculating second derivatives of function ..... nag_opt_check_2nd_deriv (e04hdc)
  - check user's function for calculating Jacobian of first derivatives ..... nag_opt_lsq_check_deriv (e04yac)
  - estimate (using numerical differentiation) gradient and/or Hessian of a function ..... nag_opt_estimate_deriv (e04xac)
  - determine the pattern of nonzeros in the Jacobian matrix for nag_opt_sparse_nlp_solve (e04vhc) ..... nag_opt_sparse_nlp_jacobian (e04vjc)
- covariance matrix for nonlinear least squares problem (unconstrained) ..... nag_opt_lsq_covariance (e04ycc)
- option setting functions,
  - NAG optimization modelling suite,
    - supply optional parameter values from a character string ..... nag_opt_handle_opt_set (e04zmc)
    - get the setting of option ..... nag_opt_handle_opt_get (e04znc)
    - supply optional parameter values from external file ..... nag_opt_handle_opt_set_file (e04zpc)
  - nag_opt_sparse_convex_qp_solve (e04nqc),
    - initialization function ..... nag_opt_sparse_convex_qp_init (e04npc)
    - supply optional parameter values from external file ..... nag_opt_sparse_convex_qp_option_set_file (e04nrc)
    - set a single option from a character string ..... nag_opt_sparse_convex_qp_option_set_string (e04nsc)
    - set a single option from an integer argument ..... nag_opt_sparse_convex_qp_option_set_integer (e04ntc)
    - set a single option from a real argument ..... nag_opt_sparse_convex_qp_option_set_double (e04nuc)
    - get the setting of an integer valued option ..... nag_opt_sparse_convex_qp_option_get_integer (e04nxc)
    - get the setting of a real valued option ..... nag_opt_sparse_convex_qp_option_get_double (e04nyc)
  - nag_opt_nlp (e04ucc) and nag_opt_nlp_revcomm (e04ufc),
    - initialization function for nag_opt_nlp (e04ucc) and nag_opt_nlp_revcomm (e04ufc) ..... nag_opt_nlp_revcomm_init (e04wbc)
    - supply optional parameter values from external file ..... nag_opt_nlp_revcomm_option_set_file (e04udc)
    - supply optional parameter values from a character string ..... nag_opt_nlp_revcomm_option_set_string (e04uec)
  - nag_opt_sparse_nlp_solve (e04vhc),
    - initialization function ..... nag_opt_sparse_nlp_init (e04vgc)
    - supply optional parameter values from external file ..... nag_opt_sparse_nlp_option_set_file (e04vkc)
    - set a single option from a character string ..... nag_opt_sparse_nlp_option_set_string (e04vlc)
    - set a single option from an integer argument ..... nag_opt_sparse_nlp_option_set_integer (e04vmc)

set a single option from a real argument ..... nag_opt_sparse_nlp_option_set_double (e04vnc)
get the setting of an integer valued option
..... nag_opt_sparse_nlp_option_get_integer (e04vrc)
get the setting of a real valued option ..... nag_opt_sparse_nlp_option_get_double (e04vsc)
nag_opt_nlp_solve (e04wdc),
initialization function ..... nag_opt_nlp_init (e04wcc)
supply optional parameter values from external file ..... nag_opt_nlp_option_set_file (e04wec)
set a single option from a character string ..... nag_opt_nlp_option_set_string (e04wfc)
set a single option from an integer argument ..... nag_opt_nlp_option_set_integer (e04wgc)
set a single option from a real argument ..... nag_opt_nlp_option_set_double (e04whc)
get the setting of an integer valued option ..... nag_opt_nlp_option_get_integer (e04wkc)
get the setting of a real valued option ..... nag_opt_nlp_option_get_double (e04wlc)
general option setting,
initialization function ..... nag_opt_init (e04xxc)
read options from a text file ..... nag_opt_read (e04xyc)
memory freeing function ..... nag_opt_free (e04xzc)

7 Auxiliary Functions Associated with Library Function Arguments

None.

8 Functions Withdrawn or Scheduled for Withdrawal

The following lists all those functions that have been withdrawn since Mark 23 of the Library or are scheduled for withdrawal at one of the next two marks.

Withdrawn Function	Mark of Withdrawal	Replacement Function(s)
nag_opt_simplex (e04ccc)	24	nag_opt_simplex_easy (e04cbc)
nag_opt_bounds_no_deriv (e04jbc)	26	nag_opt_nlp (e04ucc)

9 References

Bard Y (1974) *Nonlinear Parameter Estimation* Academic Press

Chvátal V (1983) *Linear Programming* W.H. Freeman

Dantzig G B (1963) *Linear Programming and Extensions* Princeton University Press

Fletcher R (1987) *Practical Methods of Optimization* (2nd Edition) Wiley

Gill P E and Murray W (ed.) (1974) *Numerical Methods for Constrained Optimization* Academic Press

Gill P E, Murray W and Wright M H (1981) *Practical Optimization* Academic Press

Murray W (ed.) (1972) *Numerical Methods for Unconstrained Optimization* Academic Press

Nocedal J and Wright S J (2006) *Numerical Optimization* (2nd Edition) Springer Series in Operations Research, Springer, New York

Wolberg J R (1967) *Prediction Analysis* Van Nostrand