Copyright © 1986, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

PROXIMITY ALGORITHMS: THEORY AND IMPLEMENTATION

by

John E. Hauser

Memorandum No. UCB/ERL M86/53

20 May 1986

COVER PAGE

PROXIMITY ALGORITHMS: THEORY AND IMPLEMENTATION

by

John E. Hauser

Memorandum No. UCB/ERL M86/53

20 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

PROXIMITY ALGORITHMS: THEORY AND IMPLEMENTATION

by

John E. Hauser

Memorandum No. UCB/ERL M86/53

20 May 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Proximity Algorithms: Theory and Implementation

John E. Hauser

University of California, Berkeley

ABSTRACT

The problem of estimating the minimum norm point in (the closure of) a convex set C in a real Hilbert space is investigated for the case where the characterization of the set C allows one to evaluate its support function and the corresponding contact point in C.

An extension of Wolfe's algorithm for finding the nearest point in a polytope in $\mathbf{R}^n$ is proposed as an efficient method for solving this problem. Numerical results are given to illustrate the behavior of the new algorithm. The new algorithm was implemented in C and a listing of the code is given in an appendix.

This research was supported in part by the Armed Forces Communications and Electronics Association (AFCEA), the Air Force Institute of Technology (AFIT), and the Air Force Office of Scientific Research under Grant AFOSR-83-0361.

Table of Contents

Introduction

There are many applications where it is necessary to compute the minimum distance (and corresponding minimizer) from the origin to a convex geometric object. In robotics, for example, a robot must not collide with objects in the workspace. This requires knowledge of the distance between the robot and all objects in the workspace throughout the planned maneuver [G2]. Another application is the calculation of a descent direction for a family of nondifferentiable optimization algorithms, where the minimum norm point in convex hull of active gradients (and/or generalized gradients) weighted by a component indicating their relative importance [P4] must be determined. Optimal control problems can also be formulated as minimum norm point problems involving convex sets ([G1], [B1], [P1]).

In this paper, we study methods to solve the problem of calculating the minimum norm point (or an approximation thereof) of a convex set C in a real Hilbert space $\mathbf{H}$. Problems involving the minimum distance between two sets can be converted to a single set problem by considering the difference of the two sets (e.g., $C = R - O$ where R is a robot arm and O is an obstacle). For the methods we study, the only restriction on the description of the set C is that one be able to evaluate its support function and corresponding contact point, i.e., given a direction $x \in \mathbf{H}$ compute a $y \in C$ such that $\langle x, y \rangle = \max_{y' \in C} \langle x, y' \rangle$.

The problem of computing the minimum norm point in a convex set in $\mathbf{R}^n$ was studied by Gilbert [G1] and Barr[B1]. Wolfe [W2] has presented a method for finding the (exact) nearest (i.e., minimum norm) point in a polytope in $\mathbf{R}^n$. In this paper we prove convergence for the obvious generalization of the methods of Gilbert and Barr applied to convex sets in an arbitrary Hilbert space. Extensions to Wolfe's method are proposed for general convex sets in a Hilbert space (rather than polytopes in $\mathbf{R}^n$) and shown to be convergent. We give an implementation of one extension of Wolfe's method and compare the numerical results for an example set in $\mathbf{R}^n$ with those obtained using the methods of Gilbert and Barr.

We also show how the ideas used in these algorithms can be used in a related problem where the notion of distance is not given by a norm but by how much a parameter is to be increased to grow a convex parameterized set until it touches the origin. Optimal control problems and associated subproblems can often be formulated and solved in such a manner (see [M2], [P1], [W1], [M1]).

This paper is organized as follows: Section 1 presents some mathematical preliminaries. Section 2 presents the problem and gives algorithms and proves their convergence. Section 3 describes some implementation details and gives examples to demonstrate performance. The C program implementing the extension of Wolfe's method is given in the appendix.

1. Preliminaries

Let $\mathbf{H}$ be a real Hilbert space with inner product $\langle \cdot, \cdot \rangle$ and corresponding norm $\| \cdot \| = (\langle \cdot, \cdot \rangle)^{1/2}$. The following definitions and results for convex sets can be found in books such as [B1], [B2], [R1], [V1], and [V2].

Let $x, y \in \mathbf{H}$. The *line segment* $[x, y]$ (with endpoints x and y) is the set $\{\lambda x + (1-\lambda)y \mid 0 \leq \lambda \leq 1\}$. A set $C \subset \mathbf{H}$ is called *convex* if $[x, y] \subset C$ whenever $x, y \in C$. A *convex combination* of the points $x_1, x_2, \ldots, x_m \in \mathbf{H}$ is a point $x \in \mathbf{H}$ that can be expressed in the form

$$x = \sum_{i=1}^{m} w^i x_i, \qquad \sum_{i=1}^{m} w^i = 1, \;\; w^i \geq 0, \, i = 1, \ldots, m. \tag{1.1}$$

Let $C \subset \mathbf{H}$. The *convex hull* $\mathrm{co}(C)$ is the smallest convex set in $\mathbf{H}$ containing C. The set $\mathrm{co}(C)$ can be shown to be the set of all (finite) convex combinations of elements of C.

A subset of $\mathbf{H}$ of the form $x + L$, where $x \in \mathbf{H}$ and L is a linear subspace of $\mathbf{H}$, is called an *affine subset* (or linear manifold) of $\mathbf{H}$. The *dimension* $\dim(x + L)$ of $x + L$ is defined to be $\dim(x + L) \triangleq \dim(L)$. (Note that $\{0\}$ and $\emptyset$ are defined to be linear subspaces of dimension 0 and –1, respectively). An *affine combination* of the points $x_1, x_2, \ldots, x_m \in \mathbf{H}$ is a point $x \in \mathbf{H}$ that can be expressed in the form

$$x = \sum_{i=1}^{m} w^i x_i, \qquad \sum_{i=1}^{m} w^i = 1. \tag{1.2}$$

Let $C \subset \mathbf{H}$. The *affine hull* $\mathrm{aff}(C)$ is the smallest affine set in $\mathbf{H}$ containing C. The *dimension* $\dim(C)$ of C is the dimension of $\mathrm{aff}(C)$. The set $\mathrm{aff}(C)$ can be shown to be the set of all affine combinations of elements of C. The *relative interior* $\mathrm{ri}(C)$ of C is defined to be the interior of the set C with respect to $\mathrm{aff}(C)$.

Given a finite set $X = \{x_1, x_2, \ldots, x_m\} \subset \mathbf{H}$, the set $\mathrm{co}(X)$ is called the *polytope* generated by X. By analogy to the case where $\mathbf{H} = \mathbf{R}^n$ and X is represented as the $n \times m$ matrix whose columns are the points x_i, we write a linear combination as

$$Xw \triangleq \sum_{i=0}^{m} w^i x_i \tag{1.3}$$

where $w \in \mathbf{R}^m$. In this manner, we can consider X to be a continuous linear map from $\mathbf{R}^m$ to $\mathbf{H}$. Letting X^{T} denote the adjoint map from $\mathbf{H}$ to $\mathbf{R}^m$, we write $X^{\mathrm{T}}X$ for the $m \times m$ symmetric matrix whose elements are given by

$$[X^{\mathrm{T}}X]_{i,j} = \langle x_i, x_j \rangle. \tag{1.4}$$

With this notation we may thus write

$$\mathrm{aff}(X) = \{x \in \mathbf{H} \mid x = Xw, e^{\mathrm{T}}w = 1\}, \tag{1.5}$$

$$\mathrm{co}(X) = \{x \in \mathbf{H} \mid x = Xw, e^{\mathrm{T}}w = 1, w \geq 0\}, \tag{1.6}$$

where e is the column vector $[1, 1, \ldots, 1]^{\mathrm{T}}$ and $w \geq 0$ denotes $w^i \geq 0$ for $i = 1, \ldots, m$. In each case the vector $w = [w^1, w^2, \ldots, w^m]^{\mathrm{T}}$ constitutes the *weights*, or barycentric coordinates, of the point x with respect to X.

A finite set $X = \{x_1, x_2, \ldots, x_m\} \subset \mathbf{H}$, is said to be *affinely independent* if no point of X belongs to the affine hull of the remaining points (hence $\dim(X) = m - 1$). In this case, the weights expressing the point $x = Xw$, $e^{\mathrm{T}}w = 1$ are uniquely determined for each point $x \in \mathrm{aff}(X)$. Affine independence of X is equivalent to the property that the vectors

$$\begin{bmatrix} 1 \\ x_i \end{bmatrix} \in \mathbf{R} \times \mathbf{H}, \quad x_i \in X \tag{1.7}$$

are linearly independent which implies that the symmetric matrix $ee^{\mathrm{T}} + X^{\mathrm{T}}X$ is positive definite and that the symmetric matrix

$$\begin{bmatrix} 0 & e^{\mathrm{T}} \\ e & X^{\mathrm{T}}X \end{bmatrix} \tag{1.8}$$

is nonsingular.

Given a polytope P, a *face* F is a set $F = P \cap H$ where H is a supporting hyperplane for the set P. It is also convenient to consider P and $\emptyset$ as faces of P. A *simplex* S is a polytope that contains an affinely independent set $X = \{x_1, \ldots, x_m\}$ such that $S = \mathrm{co}(X)$. The points $\{x_1, \ldots, x_m\}$ are the *vertices* of S. Note that every point $x \in S$ is contained in the relative interior of a unique face of S.

We shall be interested in sets C for which the *support function*

$$\sigma(x) \triangleq \max_{y \in C} \langle x, y \rangle \tag{1.9}$$

is well defined for all $x \in \mathbf{H}$ (many authors use sup instead of max, but we are interested in sets where the max is achieved). Given $0 \neq x \in \mathbf{H}$, a point $y^* \in C$ such that $\langle x, y^* \rangle = \sigma(x)$ is called a *contact point* since the hyperplane

$$H(x, y^*) \triangleq \{y \in \mathbf{H} \mid \langle x, y \rangle = \langle x, y^* \rangle\ (=\sigma(x))\} \tag{1.10}$$

is a supporting hyperplane for the set C at the point y^* with outward normal x. In the algorithms that follow, we shall use only information gathered by computing contact points in the given set.

The following lemma establishes an important property of support functions.

Lemma 1.11. Let K be a bounded set in $\mathbf{H}$ with a well defined support function. Then

$$\max_{y \in K} \langle x,y \rangle = \max_{y \in \mathrm{co}(K)} \langle x,y \rangle \quad \forall\, x \in \mathbf{H}. \tag{1.12}$$

Proof. Let $x \in \mathbf{H}$ be arbitrary. Every $y' \in \mathrm{co}(K)$ can be expressed as a finite convex combination of points in K. Since a finite convex combination of real numbers is bounded above by the largest, it follows that

$$\langle x,y' \rangle \leq \max_{y \in K} \langle x,y \rangle \quad \forall\, y' \in \mathrm{co}(K). \tag{1.13}$$

Let $y^* \in K$ be such that

$$\langle x, y^* \rangle = \max_{y \in K} \langle x, y \rangle. \tag{1.14}$$

Since $y^* \in \mathrm{co}(K)$, equality holds and the lemma is true. $\boxtimes$

Proximity algorithms are concerned with finding the *nearest* point in a convex set. This nearest point is, of course, well defined for a closed convex set. We shall denote the nearest point in the set C by

$$\mathrm{Nr}(C) \triangleq \arg\min\{\, \| x \| \mid x \in C \}. \tag{1.15}$$

Since we will not always deal with closed sets, we would like to know that the approximate answer we compute is valid for both the closed and not-closed cases. The following lemma shows that the *final* answer is the same in either case.

Lemma 1.16. Let $C \subset \mathbf{H}$ be convex. Then

$$\inf_{x \in C} \| x \| = \min_{x \in \overline{C}} \| x \| . \tag{1.17}$$

Proof. Let $\overline{x}$ be the unique minimum norm point in $\overline{C}$. Since $C \subset \overline{C}$ it is clear that

$$\| \overline{x} \| = \min_{x \in \overline{C}} \| x \| \leq \inf_{x \in C} \| x \| . \tag{1.18}$$

Suppose, for the sake of contradiction, that strict inequality holds in (1.18) and let

$$\epsilon = \inf_{x \in C} \| x \| - \| \overline{x} \| > 0. \tag{1.19}$$

Then, by the triangle inequality, we have that

$$\| x - \overline{x} \| \geq \| x \| - \| \overline{x} \| \geq \epsilon \quad \forall\, x \in C. \tag{1.20}$$

This contradicts the fact that $\overline{x}$ is either a point in C or a limit point of C. Hence equality holds in (1.18) and the lemma is true. $\boxtimes$

2. Theory and Algorithms

2.1. Problem and Basic Algorithm

Given a set C for which we can compute contact points, we shall consider the problem of finding an $x^* \in C$ such that $\| x^* \|$ estimates the infimum of $\| x \|$ over $x \in C$ to a given precision. Since we normally do not even know the range in which this value lies, we should express such a precision relatively. This leads to the following formal statement of the problem:

Problem 2.1. Let C be a bounded convex set in $\mathbf{H}$ such that the support function

$$\sigma(x) = \max_{y \in C} \langle x, y \rangle \tag{2.2}$$

is well defined for all x in $\mathbf{H}$. Given $\epsilon > 0$, $0 < \rho < 1$ find a point $x^* \in C$ such that either

$$\| x^* \| < \epsilon, \tag{2.3}$$

or

$$\| x^* \| - \inf_{x \in C} \| x \| < \rho \| x^* \| . \tag{2.4}$$

⊠

This problem can also be applied to the convex hull of sets that are not convex since, for any bounded set K with $\sigma(\cdot)$ well defined,

$$\sigma(x) = \max_{y \in K} \langle x, y \rangle = \max_{y \in \mathrm{co}(K)} \langle x, y \rangle. \tag{2.5}$$

We can thus state the related problem involving $\inf_{x \in \mathrm{co}(K)} \| x \|$ as Problem 2.1 using $C = \mathrm{co}(K)$ and the $\sigma(\cdot)$ for K.

As noted above, it is rare that we will actually know the value of $\inf_{x \in C} \| x \|$. The following lemma provides an estimate of $\inf_{x \in C} \| x \|$.

Lemma 2.6. Let C and $\sigma(\cdot)$ be as in Problem 2.1. Then for any $0 \neq x^* \in C$

$$\frac{-\sigma(-x^*)}{\| x^* \|} \leq \inf_{x \in C} \| x \| \leq \| x^* \| . \tag{2.7}$$

Proof. We have

$$\frac{-\sigma(-x^*)}{\| x^* \|} \leq \frac{\langle x^*, x \rangle}{\| x^* \|} \leq \| x \| \quad \forall \ x \in C \tag{2.8}$$

which leads to (2.7). ⊠

The value $\frac{-\sigma(-x^*)}{\| x^* \|}$ is simply the distance from the origin to the hyperplane $\{x \in \mathbf{H} \mid \langle x^*, x \rangle = -\sigma(-x^*)\}$ in the x^* direction. When $-\sigma(-x^*) = \| x^* \|^2$, then the upper and lower bounds are equal and x^* is the (unique) minimum norm point in C. If $\sigma(-x) < 0$ then this hyperplane properly separates C from the origin. This is very useful since, for $x^* \neq 0$,

$$\| x^* \| + \frac{\sigma(-x^*)}{\| x^* \|} < \rho \| x^* \| \tag{2.9}$$

directly implies (2.4). We use this fact in the following algorithm.

Algorithm 2.10

Given: $x_0 \in C$, $\epsilon > 0$, $0 < \rho < 1$.

Step 0: Set $i = 0$.

Step 1: If $\| x_i \| < \epsilon$, stop.

Step 2: Compute $\sigma(-x_i)$ and $y_i \in C$ such that $\langle -x_i, y_i \rangle = \sigma(-x_i)$.

Step 3: If $\| x_i \|^2 + \sigma(-x_i) < \rho \| x_i \|^2$, stop.

Step 4: Let C_i be a closed convex subset of C containing x_i and y_i and compute

$$x_{i+1} = \text{argmin } \{ \| x \|^2 \mid x \in C_i \}. \tag{2.11}$$

Step 5: Replace i by $i+1$ and go to step 1.

$\boxtimes$

Theorem 2.12. Algorithm 2.10 will terminate after a finite number of iterations yielding an x^* satisfying either (2.3) or (2.4).

Proof. Define the cost function $c(x) \triangleq \| x \|^2$. Given a point $x \in C$, the algorithm computes a point $y \in C$ satisfying $\langle -x, y \rangle = \sigma(-x)$ ($\geq -\| x \|^2$) so that $c(x) + \sigma(-x) = \langle x, x-y \rangle \geq 0$. Consider the smallest closed convex set containing x and y given by $C_i \triangleq [x, y]$. Write

$$c(x + \lambda(y-x)) - c(x) = 2\lambda\langle x, y-x \rangle + \lambda^2 \| y-x \|^2. \tag{2.13}$$

To minimize this expression (as in step 4), we use

$$\lambda_m = \min\{1, \frac{\langle x, x-y \rangle}{\| y-x \|^2}\}. \tag{2.14}$$

Let $\Delta c(x,y) \triangleq c(x + \lambda_m(y-x)) - c(x)$ denote the change in cost corresponding to a particular y (a contact point from the maximizing set). We have two cases:

Case 1: $\langle x, x-y \rangle < \| y-x \|^2 \quad (\lambda_m < 1)$. Then

$$\begin{aligned} \Delta c(x,y) &= -\frac{\langle x, x-y \rangle^2}{\| y-x \|^2} \\ &\leq -\frac{(c(x) + \sigma(-x))^2}{d} \\ &\leq 0, \end{aligned} \tag{2.15}$$

where $d = \sup_{x,y \in C} \| y-x \|^2$ (the squared diameter of C).

Case 2: $\langle x, x-y \rangle \geq \| y-x \|^2 \quad (\lambda_m = 1)$. Then

$$\begin{aligned}\Delta c(x,y) &= 2\langle x, y-x\rangle + \| y-x \|^2 \\ &\leq \langle x, y-x\rangle \\ &= -(c(x)+\sigma(-x)) \\ &\leq 0.\end{aligned} \tag{2.16}$$

Define

$$\psi(x) \triangleq \max\{-\frac{(c(x)+\sigma(-x))^2}{d}, -(c(x)+\sigma(-x))\} \tag{2.17}$$

and let $A(x)$ denote the set of all possible successors to $x \in C$ computed during an iteration of the algorithm. From above, we have that

$$c(x') - c(x) \leq \psi(x) \leq 0 \quad \forall\, x \in C, \forall\, x' \in A(x). \tag{2.18}$$

Now suppose that the algorithm does not terminate, but generates an infinite sequence $\{x_i\}$. This implies that

$$\| x_i \| \geq \epsilon \quad \forall\, i \tag{2.19}$$

and

$$\| x_i \|^2 + \sigma(-x_i) \geq \rho \| x_i \|^2 \quad \forall\, i \tag{2.20}$$

so that

$$c(x_i) + \sigma(-x_i) \geq \rho\epsilon^2 \quad \forall\, i. \tag{2.21}$$

Hence

$$\psi(x_i) \leq \max\{-\frac{\rho^2\epsilon^4}{d}, -\rho\epsilon^2\} < 0 \quad \forall\, i \tag{2.22}$$

so that $c(x_i) \rightarrow -\infty$ as $i \rightarrow \infty$ which contradicts the fact that $c(x_i)$ is bounded from below by $\inf\limits_{x \in C} \| x \|^2 \geq 0$. Therefore the algorithm terminates in a finite number of iterations. Lemma 2.6 shows the sufficiency of the stop rules in satisfying (2.3) or (2.4). $\boxtimes$

Corollary 2.23. Let Step 4 of Algorithm 2.10 be replaced by

Step 4′: Compute $x_{i+1} \in C$ such that

$$\| x_{i+1} \|^2 \leq \min_{x \in [x_i, y_i]} \| x \|^2. \tag{2.24}$$

Then the modified Algorithm 2.10 terminates after a finite number of iterations yielding an x^* satisfying either (2.3) or (2.4).

Proof. The decrease in the cost function is at least as good as that achieved by using $C_i = [x_i, y_i]$, hence the same result holds. $\boxtimes$.

2.2. Selection of the Set C_i

Our knowledge about the set C is obtained solely by computing a contact point during each iteration. Hence, from a practical point of view, our choice of C_i in Step 4 of Algorithm 2.10 is restricted to convex hulls of subsets of contact and minimizing points computed thus far.

The simplest choice for C_i is, of course, the line segment $[x_i, y_i]$. Gilbert [G1] used this method for sets in $\mathbf{R}^n$ when he extended the Frank-Wolfe procedure [F1] to the problem of minimizing a quadratic form on general convex sets for use in solving optimal control problems. This method is illustrated in figure 2.1 for $C \subset \mathbf{H} = \mathbf{R}^2$. During each iteration, we find the nearest point in the line segment between x_i (which is usually a relatively interior point) and y_i (a boundary point). Thus our sequence of x_i's tend to remain interior and not approach the boundary of the set as rapidly as desired. This behavior manifests itself as very slow convergence when the ratio of the radius of curvature to the minimum distance for a set is large. This is particularly undesirable for applications such as optimization where the nearest point must be computed for a sequence of sets that approach the origin to satisfy optimality conditions.

Another option is to use the convex hull of all points computed so far. That is, $C_i = \text{co}\{x_0, y_j, j = 0,..., i\}$. This, however, leads to a subproblem of

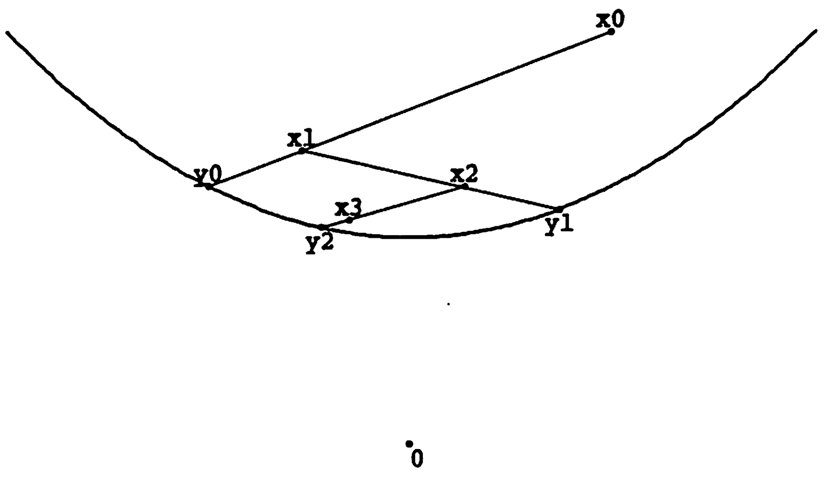

Figure 2.1 Algorithm 2.10 using minimal C_i

ever increasing complexity. Furthermore, as the example of figure 2.2 illustrates, many of the points collected thus far do not play a role in determining the nearest point in such an approximating polytope. Thus, one can imagine that, between these two extremes, there is a reasonably good method of choosing the points to determine C_i that balances the cost of the subproblem with the rate of convergence.

Our subproblem always consists of finding the nearest point in a polytope. It is clear that this point will be found in one of the faces. When working in $\mathbf{R}^n$ this closest face will have dimension less than n. The minimum number of points needed to determine a face of dimension $n-1$ is n. Even in infinite dimensional spaces, a set may have special structure so that the nearest face will have dimension lower than that of the polytope accumulated thus far. This leads to the task of deciding how many points to keep and which points to discard.

Barr [B1] used this idea to extend Gilbert's procedure for sets in $\mathbf{R}^n$ by choosing C_i to be the convex hull of a number of previous contact points, the current minimum point, and the corresponding contact point. Each contact

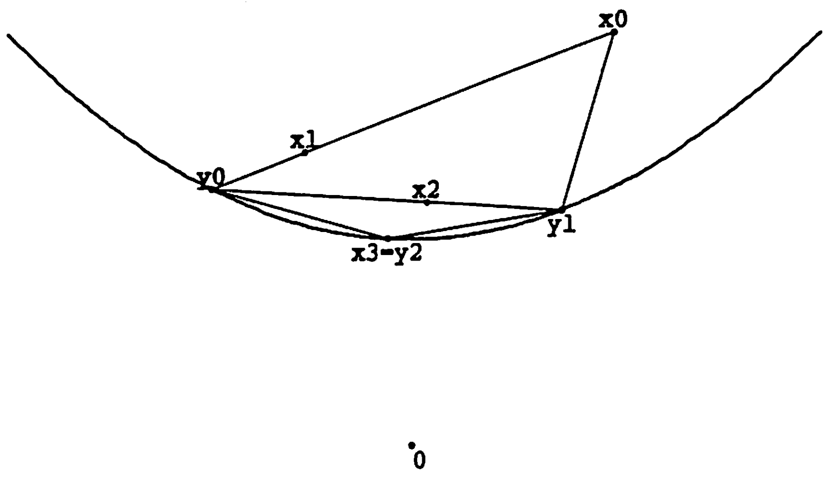

Figure 2.2 Algorithm 2.10 using maximal C_i

point is rated by the distance of its supporting hyperplane from the origin: a higher rating being given to points with a larger distance. A fixed number (p) of old contact points is kept at each stage. The addition of a new contact point forces out the old contact point having the lowest rating. Computational experience indicates much faster convergence than Gilbert's two-point method and tends to reinforce the hunch that $p = n$ (to define the nearest face with $n + 1$ points) is a good choice for the number of old contact points to keep.

There are a few questions and (minor) problems with this method. During each iteration, Barr requires that one find the nearest point in a polytope defined by $n + 2$ points (n old contact points, the current nearest point, and the new contact point). When solved by a quadratic program (as he does), we find that the quadratic form is only positive semi-definite. This can sometimes cause jamming in the quadratic program and may not be the most efficient method. Also, we never take advantage of the possibility that the nearest face in the approximating polytope may have dimension lower than $n - 1$ (i.e., be an edge, etc.). The set C may itself have dimension lower than n. In infinite dimensional spaces ($n = \infty$) we would be required to keep all previous points regardless of the structure or dimension of the set! Furthermore, since the current minimum point is a convex combination of previous points, we are carrying extra baggage by including it as a point in the nearest point subproblem (assuming that we have not discarded any of the vertices for the face containing the nearest point).

What we really need is a method that automatically determines the nearest face of the approximating polytope and keeps only those points necessary to determine that face. There may be a possibility that the nearest face we achieve by keeping a subset of points is not as good as that achieved by using every point, but computational experience tends to point to higher efficiency. Figure 2.2 indicates that, at least for $\mathbf{H} = \mathbf{R}^2$, we are probably not losing anything using such a method. Keeping fewer points not only means having a simpler subproblem to solve, the special structure of those points may also lead to a much more efficient subproblem.

Wolfe used such an approach to compute the nearest point in a polytope in $\mathbf{R}^n$ [W2]. Each iteration starts with a set of affinely independent points such that the nearest point in the convex hull of these points is contained in the relative interior of this convex hull (such a set is called a *corral*). The new contact point is added to the set and selected points (other than the new point) are deleted as necessary until the resulting set is itself a corral. If the new contact point is not in the hyperplane defined by the current nearest point (if it was we would have stopped in Step 3 of Algorithm 2.10), the union of the contact point and the current corral will be affinely independent since the affine hull of the current corral is contained in the hyperplane. It is this special structure that makes Wolfe's method particularly efficient. Furthermore, though designed for polytopes in $\mathbf{R}^n$, this method is easily extended to general convex sets (in a Hilbert space) for which we can compute contact points.

2.3. Wolfe's Method

We will now formally state the subprocedure used by Wolfe to reduce the value of $\| x \|$.

Subprocedure 2.25 (Wolfe)

Given: a corral $Q = \{x_1, \ldots, x_m\} \subset \mathbf{H}$ together with $x = \mathrm{Nr}(\mathrm{co}(Q))$ and a point $y \in \mathbf{H}$ on the near side of the hyperplane $H(x, x)$ (i.e., $\langle x, y \rangle < \langle x, x \rangle$).

Step 0: Set $j = 0$, $Q_0 = Q \cup \{y\}$, $q_0 = x$.

Step 1: Compute $\hat{q}_j = \mathrm{Nr}(\mathrm{aff}(Q_j))$; if $\hat{q}_j \in \mathrm{ri}(\mathrm{co}(Q_j))$ set $Q' = Q_j$, $x' = \hat{q}_j$ and return.

Step 2: Compute $q_{j+1} = \mathrm{Nr}([q_j, \hat{q}_j] \cap \mathrm{co}(Q_j))$ and set $Q_{j+1} = Q_j \backslash \{\tilde{q}_j\}$ with $\tilde{q}_j \in Q_j$ not in the face of $\mathrm{co}(Q_j)$ containing q_{j+1}.

Step 3: Replace j by $j + 1$ and go to step 1.

$\boxtimes$

This subprocedure can be used in Step 4 of Algorithm 2.10 with values $x = x_i$, $y = y_i$, and Q equal to the Q' returned during the previous iteration of Algorithm 2.10 ($Q = \{x_0\}$ initially). Upon return, x_{i+1} is set to the value of x'. This is the method used by Wolfe to find the nearest point in a polytope [W2]. Let $P = \{p_1, p_2, \ldots, p_l\}$ be the given set of points that generate such a polytope. Then, by Lemma 1.11, we can compute a contact point for the polytope, $\mathrm{co}(P)$, as required in Step 2 of Algorithm 2.10 by choosing a vertex, $p_k \in P$, such that $\langle x_i, p_k \rangle \leq \langle x_i, p_k \rangle$ for $j = 1, \ldots, l$. Wolfe showed that this method will terminate in a finite number of iterations with the unique nearest point in the polytope [W2] (i.e., $\| x^* \| = \min_{x \in \mathrm{co}(P)} \| x \|$). We later will discuss how the computations in Steps 1 and 2 of Subprocedure 2.25 are carried out.

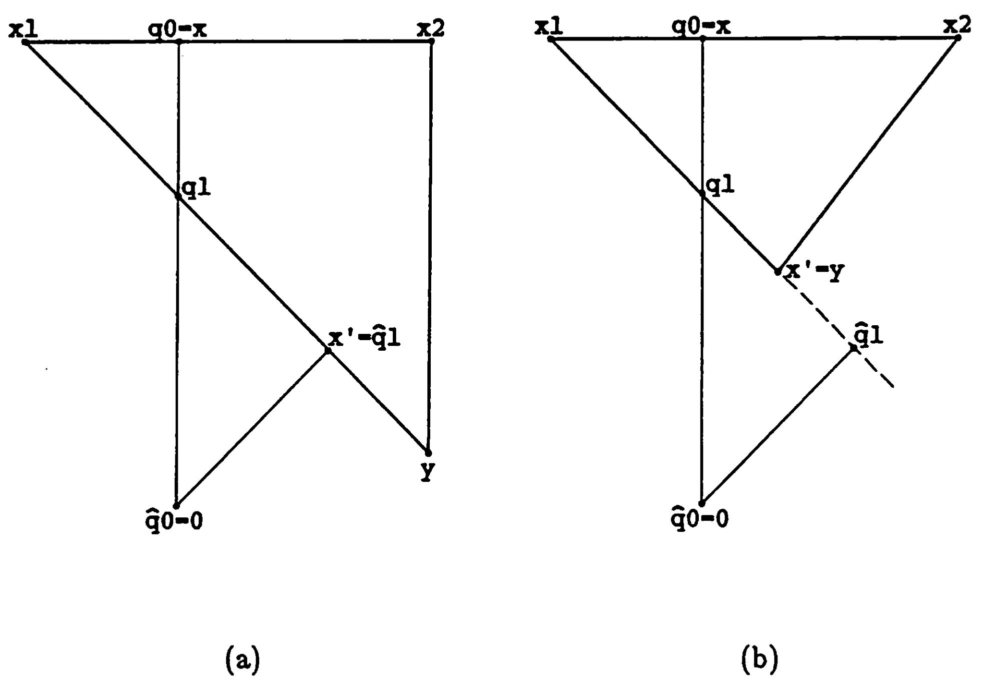

Figure 2.3 Subprocedure 2.25

To see how this method works, consider the two examples in figure 2.3. We begin with the corral $Q = \{x_1, x_2\}$ with nearest point x and the new point y. Thus $Q_0 = \{x_1, x_2, y\}$. The minimum norm point in $\mathrm{aff}(Q_0) = \mathbf{R}^2$ is, of course, $\hat{q}_0 = 0$. Since 0 is not in the relative interior of $\mathrm{co}(Q_0)$, we calculate the nearest point in $[q_0, \hat{q}_0] \cap \mathrm{co}(Q_0)$ giving q_1. Now, x_2 is not in the face of $\mathrm{co}(Q_0)$ that q_1 lies in, so we discard it (i.e., $\tilde{q}_1 = x_2$) and set $Q_1 = \{x_1, y\}$. At this point, the set $\mathrm{aff}(Q_1)$ is the line containing x_1 and y. For example (a), the minimum norm point $\hat{q}_1$ is contained in the relative interior of $[x_1, y]$ so we would stop with $x' = \hat{q}_1$ and $Q' = \{x_1, y\}$. Example (b) requires one more projection onto a nearer face ($\{y\}$) and a corresponding point deletion (x_1). Since a single point is its own convex hull, affine hull, and nearest point (hence a corral), we would stop with $x' = y$ and $Q' = \{y\}$.

From the above example, one might be tempted to believe that Wolfe's subprocedure will always find the nearest point and corresponding corral of the set $\text{co}(Q \cup \{y\})$. This certainly seems to be the case for any polytope you can easily imagine (i.e., in $\mathbf{R}^2$ or $\mathbf{R}^3$). This is, in fact, the usual case. In numerical tests for sets in $\mathbf{R}^n$ with n ranging from 2 to 20, we found that Wolfe's subprocedure discarded a point that was not separated by the final point x' (i.e., $x' \neq \text{Nr}(\text{co}(Q \cup \{y\}))$) only a few times per thousand calls and this only occurred when the dimension was approximately fifteen or greater. This small amount is, of course, more than enough to make the expected result all but possible. We can, however, use Corollary 2.23 to examine the convergence of the new algorithm using Wolfe's subprocedure.

We will need the following simple lemma to show that Subprocedure 2.25 produces a decrease in the cost sufficient to guarantee convergence when used for step 4 of Algorithm 2.10.

Lemma 2.26. Let $f:\mathbf{H} \to \mathbf{R}$ be convex and continuous and let $x, y, z \in \mathbf{H}$ be such that

$$f(z) \leq f(w) \quad \forall\, w \in [x, y]. \tag{2.27}$$

Then

$$\min_{w \in [x', y]} f(w) \leq \min_{w \in [x, y]} f(w) \quad \forall\, x' \in [x, z]. \tag{2.28}$$

Proof. Let $v \in [x, y]$ be a minimizer of $f(\cdot)$ on $[x, y]$. Since $f(\cdot)$ is convex and $f(z) \leq f(v)$ by (2.27), we have that $f(v') \leq f(v)$ for all $v' \in [v, z]$. Therefore, for any $x' \in [x, z]$, the segment $[x', y]$ contains a point $v' \in [v, z]$ such that $f(v') \leq f(v)$ which proves the lemma. ⊠

The following result shows that Wolfe's subprocedure produces a decrease in cost at least as large as that obtained using the two point method.

Theorem 2.29. Let Q, x, and y be given as in Subprocedure 2.25. Subprocedure 2.25 will terminate in fewer than $m+1$ iterations with a corral Q' whose nearest point x' is such that

$$\| x' \| \le \min_{v \in [x,y]} \| v \| . \tag{2.30}$$

Proof. Since a single point is a corral and the subprocedure deletes one point in each iteration it must terminate in fewer than $m+1$ iterations, and can only do so in Step 1 with a corral Q'. If the subprocedure terminates without deleting any points the result is trivially true. The result is also obvious when the original corral contains a single point (i.e., $Q = \{x\}$). Therefore, suppose that Q contains at least 2 points and that the subprocedure has terminated with $j = k \ge 1$. Now since y is on the near side of $H(x,x)$, $\mathrm{co}(Q_0) = \mathrm{co}(Q \cup \{y\})$ contains points with norm strictly less than $\| x \|$ (e.g., the minimum norm point in $[x,y]$ is one such point). Furthermore, since $q_0 = x$ is the minimum norm point in $\mathrm{aff}(Q)$, any point in $\mathrm{aff}(Q_0)$ (and hence in $\mathrm{co}(Q_0)$) with norm strictly less than $\| q_0 \|$ (e.g., $\hat{q}_0$) must be an affine combination of points in Q_0 with a strictly positive weight for y. Write

$$q_0 = Q_0 w, \qquad e^{\mathrm{T}} w = 1, \tag{2.31}$$

and

$$\hat{q}_0 = Q_0 \hat{w}, \qquad e^{\mathrm{T}} \hat{w} = 1. \tag{2.32}$$

Then, $w^l > 0$ for $l = 1,\dots,m$ and $w^{m+1} = 0$, while $\hat{w}^{m+1} > 0$. Therefore, there exists $\bar{\lambda} \in (0,1]$ such that

$$\lambda \hat{q}_0 + (1-\lambda) q_0 \in \mathrm{ri}(\mathrm{co}(Q_0)) \qquad \forall \lambda \in (0, \bar{\lambda}) \tag{2.33}$$

which implies that

$$[q_0, \hat{q}_0] \cap \mathrm{ri}(\mathrm{co}(Q_0)) \ne \emptyset. \tag{2.34}$$

Since $\| \cdot \|$ decreases strictly along the segment $[q_0, \hat{q}_0]$, we find that

$$\| q_1 \| < \| q_0 \| . \tag{2.35}$$

Now, since

$$\| q_j \| \leq \| q_{j-1} \| \quad \text{for } j = 1,\dots,k, \tag{2.36}$$

we see that $\| x' \| \leq \| q_1 \|$ which implies that $y \in Q'$. Hence, we have

$$\| x' \| = \| \hat{q}_k \| = \min_{v \in \mathrm{co}(Q_k)} \| v \| \leq \min_{v \in [q_k, y]} \| v \| . \tag{2.37}$$

Since $q_j \in [q_{j-1}, \hat{q}_{j-1}]$ and $\| \cdot \|$ is convex, Lemma 2.26 implies that

$$\min_{v \in [q_j, y]} \| v \| \leq \min_{v \in [q_{j-1}, y]} \| v \| \quad \text{for } j = 1,\dots,k. \tag{2.38}$$

Therefore

$$\| x' \| \leq \min_{v \in [q_0, y]} \| v \| = \min_{v \in [x, y]} \| v \| \tag{2.39}$$

which is the desired result. ☒

We must now show that we can compute the quantities required by Subprocedure 2.25. Given an affinely independent set of points Q_j, we must calculate the minimum norm point in $\mathrm{aff}(Q_j)$. Dropping the subscripts for convenience, the problem is

$$\min\{ \| q \|^2 \mid q = Qw, \ e^{\mathrm{T}} w = 1 \} \tag{2.40}$$

or

$$\min\{ w^{\mathrm{T}} Q^{\mathrm{T}} Q w \mid e^{\mathrm{T}} w = 1 \}. \tag{2.41}$$

Forming the Lagrangian $w^{\mathrm{T}} Q^{\mathrm{T}} Q w + 2\lambda(e^{\mathrm{T}} w - 1)$ and differentiating with respect to w, we obtain the necessary conditions

$$\left. \begin{aligned} e^{\mathrm{T}} w &= 1 \\ e\lambda + Q^{\mathrm{T}} Q w &= 0 \end{aligned} \right\} \tag{2.42}$$

which have a unique solution since the system matrix is nonsingular. Furthermore, since $w^{\mathrm{T}} Q^{\mathrm{T}} Q w$ is convex in w, these conditions are also sufficient. Hence $\hat{q} = Q\hat{w}$, with $\hat{w}$ solving (2.42), minimizes $\| \cdot \|$ over $\mathrm{aff}(Q)$. The point $\hat{q}$ is in the relative interior of $\mathrm{co}(Q)$ if and only if $\hat{w} > 0$. Thus the calculations of Step

1 are easily accomplished.

Now, for Step 2 we must determine the nearest point in $[q,\hat{q}]\cap \mathrm{co}(Q)$ and a point $\tilde{q}\in Q$ not in the face of $\mathrm{co}(Q)$ containing this nearest point. Given $q\in \mathrm{co}(Q)$ we have $q = Qw$ with $e^{\mathrm{T}}w = 1$ and $w \geq 0$ whereas $\hat{q}\notin \mathrm{ri}(\mathrm{co}(Q))$ implies that $\hat{q} = Q\hat{w}$ with $e^{\mathrm{T}}\hat{w} = 1$ and $\hat{w}^l \leq 0$ for some $l\in\{1,\ldots,m\}$ (where m is the number of points in Q). Since the value of the norm decreases strictly along the line segment $[q,\hat{q}]$, the desired point (call it $\overline{q} = Q\overline{w}$) is the point closest to $\hat{q}$ such that

$$\overline{w} = \lambda w + (1-\lambda)\hat{w} \geq 0. \tag{2.43}$$

The required λ is easily found to be

$$\lambda = \max\{\frac{-\hat{w}^l}{w^l - \hat{w}^l} \mid \hat{w}^l \leq 0\}. \tag{2.44}$$

Choosing $\tilde{q}$ to be a point in Q whose weight $\overline{w}^l$ is equal to zero (an l that achieves the max in (2.44)), we complete the calculations required by Subprocedure 2.25.

Wolfe's decision to work with affinely independent points has some important implications. Since the convex hull of an affinely independent set of m points is a simplex of dimension $m-1$, every point in the affine hull (and hence the convex hull) of this set has a unique representation with respect to the set. As we saw above, this leads to easy analytic minimization of the norm over the affine hull. Furthermore, every point in the convex hull is contained in the relative interior of a unique face allowing us to delete vertices that do not determine the active face. Also, since we can never keep more than $\dim(C)+1$ affinely independent points, this method tends to implicitly determine and work with the dimension of the set. This is particularly important when $\dim(C) << \dim(\mathrm{H})$ as might be the case when H is infinite dimensional.

One minor shortcoming of using Subprocedure 2.25 directly is that it does not always return the nearest point in $\mathrm{co}(Q\cup\{y\})$. To alleviate this problem, we suggest using Wolfe's method to find the nearest point in the simplex

$\mathrm{co}(Q \cup \{y\})$ (recall that Wolfe's method was originally designed to find the nearest point in a polytope). We will then return the (unique) corral $Q' \subset Q \cup \{y\}$ containing this nearest point and use it in the next call. The following subprocedure implements this idea by calling Subprocedure 2.25 internally.

Subprocedure 2.45

Given: a corral $Q = \{x_1, \ldots, x_m\} \subset \mathbf{H}$ together with $x = \mathrm{Nr}(\mathrm{co}(Q))$ and a point $y \in \mathbf{H}$ on the near side of the hyperplane $H(x, x)$ (i.e., $\langle x, y \rangle < \langle x, x \rangle$).

Step 0: Set $k = 0$, $Q_0 = Q$, $y_0 = y$, $\bar{x}_0 = x$.

Step 1: Call Subprocedure 2.25 with parameters $Q = Q_k$, $x = \bar{x}_k$, and $y = y_k$. Upon return, set $Q_{k+1} = Q'$ and $\bar{x}_{k+1} = x'$.

Step 2: Compute $y_{k+1} = \arg \min_{y \in Q \setminus Q_{k+1}} \langle \bar{x}_{k+1}, y \rangle$. If $\langle \bar{x}_{k+1}, y_{k+1} \rangle \geq \| \bar{x}_{k+1} \|^2$, set $Q' = Q_{k+1}$, $x' = \bar{x}_{k+1}$ and return.

Step 3: Replace k by $k + 1$ and go to step 1.

$\boxtimes$

This subprocedure calculates the nearest point and corresponding corral in the polytope $\mathrm{co}(Q \cup \{y\})$. We can thus use this subprocedure directly in Algorithm 2.10 by defining C_i to be the convex hull of the union of the corral Q' returned by the previous call and the new point y_i. The reason we only need to check points in $Q \setminus Q_{k+1}$ for separation by the hyperplane $H(\bar{x}_{k+1}, \bar{x}_{k+1})$ is that $y \in Q_{k+1}$ implies $y \in H(\bar{x}_{k+1}, \bar{x}_{k+1})$ (i.e., $\langle \bar{x}_{k+1}, y \rangle = \| \bar{x}_{k+1} \|^2$) since $\bar{x}_{k+1}$ minimizes $\| \cdot \|$ over $\mathrm{aff}(Q_{k+1})$. The following finite termination result follows that given by Wolfe [W2] for general polytopes in $\mathbf{R}^n$.

Theorem 2.46. Let Q, x, and y be given as in Subprocedure 2.45. Subprocedure 2.45 will terminate after a finite number of iterations with a corral Q' whose nearest point x' is such that

$$\| x' \| = \min_{v \in \mathrm{co}(Q \cup \{y\})} \| v \| . \tag{2.47}$$

Proof. Since $\| \bar{x}_{k+1} \| < \| \bar{x}_k \|$ and $\bar{x}_k$ is uniquely determined by the corral Q_k (and conversely), a given corral will be considered at most once. Since $Q \cup \{y\}$ contains only a finite number of corrals, the subprocedure must terminate finitely and can only do so in Step 2 upon satisfying an optimality condition which implies (2.47). ⊠

As noted above, our experience indicates that Wolfe's subprocedure (Subprocedure 2.25) will normally return the minimum norm point in the polytope $\mathrm{co}(Q \cup \{y\})$. Thus, Subprocedure 2.45 will normally terminate in one iteration. This subprocedure, however, uses Wolfe's subprocedure in a manner that guarantees that we will always move from minimizing corral to minimizing corral. Subprocedure 2.45 therefore achieves the best we can do working within the framework of keeping the corral that determines the nearest face in the approximating polytope. Further research will be required to determine if any improvements can be obtained by selectively keeping points not in the minimizing corral.

For the sake of completeness and for easy reference, we state the complete algorithm obtained by using Wolfe's method to find the nearest point in a simplex (Subprocedure 2.45) during each iteration of the main proximity algorithm (Algorithm 2.10). Since no further analysis is required, we have dropped all subscripts. Note that e, v, and w are real vectors of varying length (equal to the number of points in the current Q). We now include the explicit formulas that determine the quantities needed by the Wolfe method. This algorithm can thus be implemented almost directly as stated.

Algorithm 2.48

```
Given:  $x_0 \in C$, $\epsilon > 0$, $0 < \rho < 1$.

Step 0: Set $x = x_0$, $Q = \{x_0\}$, $w = [1]$.

Step 1: If $\| x \| < \epsilon$, stop.

Step 2: Compute $\sigma(-x)$ and $y \in C$ such that $\langle -x, y \rangle = \sigma(-x)$.

Step 3: If $\| x \|^2 + \sigma(-x) < \rho \| x \|^2$, stop.

Step 4: (a) Set $\tilde{Q} = Q$.
        (b) Replace $Q$ by $Q \cup \{y\}$, $w$ by $[w^{\mathrm{T}}\ 0]^{\mathrm{T}}$.
        (c) Solve the equations

              $\left.\begin{aligned} e^{\mathrm{T}} v &= 1 \\ e\lambda + Q^{\mathrm{T}} Q v &= 0 \end{aligned}\right\}$        (2.49)

            If $v > 0$, set $w = v$, $x = Qw$ and go to (e).
        (d) Compute

              $\lambda = \max\{\frac{-v^l}{w^l - v^l} \mid v^l \le 0\}$        (2.50)

            and replace $w$ by $\lambda w + (1-\lambda)v$. Delete from $w$ some zero component,
            and from $Q$ the corresponding point. Go to (c).
        (e) Compute $y = \arg\min\{\langle x, y' \rangle \mid y' \in \tilde{Q} \backslash Q\}$. If $\langle x, y \rangle < \cdot \| x \|^2$, go
            to (b).

Step 5: Go to step 1.
```

$\boxtimes$

2.4. Extension to Parameterized Sets

We will now show how we can extend Algorithm 2.10 to deal with a related problem. We are given a parameterized convex set $C(\alpha)$, $\alpha \in [\alpha_{\min}, \infty)$, and we wish to find the minimum α for which $0 \in C(\alpha)$. For many applications, this set will actually be the difference of a *target* set and a *reachable* set, i.e.,

$C(\alpha) = T - R(\alpha)$. For this type of problem, we actually want to find an element of T which estimates the point where T and $R(\alpha)$ first meet as α is increased. Examples of the use of this type of formulation in optimal control algorithms can be found in [M2], [P1], [W1], and [M1]. We will state the problem as follows.

Problem 2.51. Let $C:[\alpha_{\min}, \infty) \rightarrow 2^{\mathbf{H}}$ (where $2^{\mathbf{H}}$ denotes the collection of all subsets of $\mathbf{H}$) be such that

(1) $C(\alpha)$ is convex and bounded for each $\alpha \in [\alpha_{\min}, \infty)$.

(2) $C(\alpha') \subset C(\alpha'')$ for $\alpha', \alpha'' \in [\alpha_{\min}, \infty)$, such that $\alpha' < \alpha''$.

(3) $C(\cdot)$ is upper semicontinuous [C1] on $[\alpha_{\min}, \infty)$ (given a neighborhood $N(C(\alpha))$ of $C(\alpha)$ there exists a neighborhood $N(\alpha)$ of α such that $\alpha' \in N(\alpha)$ implies $C(\alpha') \subset N(C(\alpha))$).

(4) $0 \in C(\alpha)$, for some $\alpha \in [\alpha_{\min}, \infty)$.

(5) Given $\alpha \in [\alpha_{\min}, \infty)$, the support function

$$\sigma_\alpha(x) \triangleq \max_{y \in C(\alpha)} \langle x, y \rangle \tag{2.52}$$

is well-defined for all $x \in \mathbf{H}$.

Given $\epsilon > 0$, find an $\alpha^* \in [\alpha_{\min}, \infty)$ and an $x^* \in C(\alpha^*)$ such that

$$\alpha^* \leq \inf\{\alpha \in [\alpha_{\min}, \infty) \mid 0 \in C(\alpha)\} \tag{2.53}$$

and

$$\| x^* \| < \epsilon. \tag{2.54}$$

⊠

Intuitively, we want to grow the set $C(\alpha)$ (by increasing α) until we find a point in $C(\alpha)$ *close enough* to the origin. For many real problems there may be a good way to express the precision of the result relatively as opposed to the absolute precision specified by (2.54). This will, of course, be application dependent but such a stopping condition can usually be satisfied by choosing $\epsilon > 0$

sufficiently small.

We can use the following extension of Algorithm 2.10 to solve Problem 2.51.

Algorithm 2.55

```
Given: $x_0 \in C(\alpha_{\min})$, $\epsilon > 0$.
Step 0: Set $i = 0$, $\alpha_0 = \alpha_{\min}$.
Step 1: If $\| x_i \| < \epsilon$, stop.
Step 2: Compute $\sigma_{\alpha_i}(-x_i)$.
    If $\sigma_{\alpha_i}(-x_i) \geq 0$, set $\alpha_{i+1} = \alpha_i$,
    else, compute $\alpha_{i+1} > \alpha_i$ such that $\sigma_{\alpha_{i+1}}(-x_i) = 0$.
    Compute $y_i \in C(\alpha_{i+1})$ such that $\langle -x_i, y_i \rangle = \sigma_{\alpha_{i+1}}(-x_i)$.
Step 3: Let $C_i$ be a closed convex subset of $C(\alpha_{i+1})$ containing $x_i$ and $y_i$ and
    compute
        $x_{i+1} = \operatorname{argmin} \{ \| x \|^2 \mid x \in C_i \}$.        (2.56)
Step 4: Replace $i$ by $i+1$ and go to step 1.
```

$\boxtimes$

Figure 2.4 illustrates the use of Algorithm 2.55 with maximal C_i for the case where $\mathbf{H} = \mathbf{R}^2$ and $C(\alpha)$ is a disc of radius α. The computations for this example were obtained using Subprocedure 2.45 in Step 3 of Algorithm 2.55. Note that the next contact point, y_3, will be very close to the origin (i.e., $\alpha_4 \approx \alpha^*$ for small ϵ).

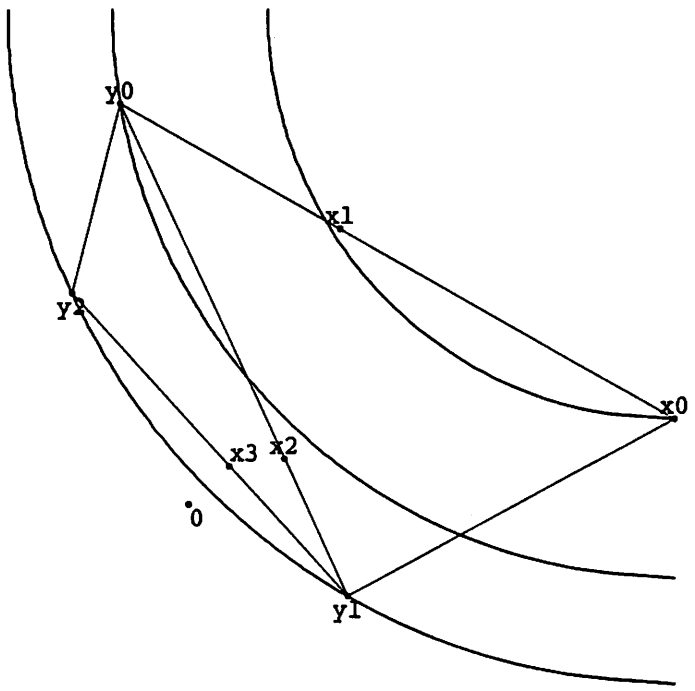

Figure 2.4 Algorithm 2.55 using maximal C_i

Theorem 2.57. Algorithm 2.55 will terminate in a finite number of iterations yielding an $\alpha^* \in [\alpha_{\min}, \infty)$ and an $x^* \in C(\alpha^*)$ satisfying (2.53) and (2.54).

Proof. First, note that since α is increased only when $C(\alpha)$ is properly separated from the origin,

$$\alpha_i \leq \inf\{\alpha \in [\alpha_{\min}, \infty) \mid 0 \in C(\alpha)\} \quad \forall\, i \geq 0 \tag{2.58}$$

so that (2.53) will be satisfied by any α^* generated by the algorithm. To see that the algorithm terminates finitely, define the cost function $c(x) \triangleq \| x \|^2$. Given a point $x \in C(\alpha)$, the algorithm computes a point $y \in C(\alpha')$, $\alpha' \geq \alpha$, such that $\langle x, y \rangle \leq 0$. Consider the smallest closed convex set containing x and y given by $C_i \triangleq [x, y]$. Write

$$\begin{aligned} c(x + \lambda(y - x)) - c(x) &= 2\lambda\langle x, y - x\rangle + \lambda^2 \| y - x \|^2 \\ &\leq -2\lambda c(x) + \lambda^2 d \end{aligned} \tag{2.59}$$

where $d = \sup_{x,y \in C(\hat{\alpha})} \| y - x \|^2$ with $\hat{\alpha}$ such that $0 \in C(\hat{\alpha})$ (d is the squared diameter of $C(\hat{\alpha})$). Since the successor point $x' \in A(x)$ to x (computed in Step 3) minimizes $c(x + \lambda(y - x))$ over $\lambda \in [0,1]$, it follows that

$$c(x') - c(x) \leq \min_{\lambda \in [0,1]} -2\lambda c(x) + \lambda^2 d. \tag{2.60}$$

Minimizing the right hand side gives

$$\lambda = \min\{\frac{c(x)}{d}, 1\} \tag{2.61}$$

so that

$$c(x') - c(x) \leq \psi(x) \leq 0 \tag{2.62}$$

where

$$\psi(x) \triangleq \begin{cases} -\frac{(c(x))^2}{d} & c(x) \leq d \\ -d & c(x) > d. \end{cases} \tag{2.63}$$

Now suppose that the algorithm does not terminate, but generates an infinite sequence $\{x_i\}$. This implies that

$$\| x_i \| \geq \epsilon \quad \forall\, i \tag{2.64}$$

so that

$$\psi(x_i) \leq \max\{-\frac{\epsilon^2}{d}, -d\} < 0 \quad \forall\, i. \tag{2.65}$$

This implies that $c(x_i) \rightarrow -\infty$ as $i \rightarrow \infty$ which contradicts the fact that $c(x_i)$ is bounded from below by zero. Hence the algorithm terminates in a finite number of iterations satisfying (2.53) and (2.54). ⊠

Corollary 2.66. Let Step 3 of Algorithm 2.55 be replaced by

Step 3′: Compute $x_{i+1} \in C(\alpha_{i+1})$ such that

$$\| x_{i+1} \|^2 \leq \min_{x \in [x_i, y_i]} \| x \|^2. \tag{2.67}$$

Then the modified Algorithm 2.55 terminates after a finite number of iterations yielding an $\alpha^* \in [\alpha_{\min}, \infty)$ and an $x^* \in C(\alpha^*)$ satisfying (2.53) and (2.54).

Proof. The decrease in the cost function is at least as good as that achieved by using $C_i = [x_i, y_i]$, hence the same result holds. $\boxtimes$.

3. Implementation and Performance

Having presented the algorithms and proven their convergence, we will now discuss some details of implementation and present some computational results for some specific sets C.

3.1. Implementation of Wolfe's Subprocedure

Implementation of Subprocedure 2.25 involves finding an efficient and accurate method to minimize $\|\cdot\|$ over the affine hull of a given affinely independent point set Q. As shown above, this amounts to solving the system of linear equations

$$\begin{aligned} e^{\mathrm{T}} w &= 1 \\ e\lambda + Q^{\mathrm{T}} Q w &= 0. \end{aligned} \tag{3.1}$$

One straight-forward (and inefficient) method is to simply form the system matrix each iteration and solve it using a generic method. However, since Q changes only by the addition or deletion of a single point in each iteration, a method that updates the solution incrementally will certainly be more efficient.

Wolfe [W1] has worked with four such methods for solving (3.1) and recommends the method we will now describe. One can easily show that the system of equations

$$(ee^{\mathrm{T}} + Q^{\mathrm{T}} Q)v = e \tag{3.2}$$

$$w = v / e^{\mathrm{T}} v \tag{3.3}$$

is equivalent to (3.1) in producing the same (unique) w (we really don't care about λ). Working with this system has distinct advantages. The order of the system (3.2) is one less than that of (3.1) and the normalization of w (so that $e^{\mathrm{T}} w = 1$) is achieved to the best possible accuracy by (3.3). Furthermore, since $ee^{\mathrm{T}} + Q^{\mathrm{T}} Q$ is positive definite, we can employ techniques that are both efficient and numerically well-conditioned. In particular, we know that there is a real, upper triangular matrix R with positive diagonal elements such that

$$R^T R = ee^T + Q^T Q \tag{3.4}$$

(this is the *Cholesky Decomposition*, see, e.g., [G2], Theorem 5.2-3). Given R, we can then solve (3.2) by solving the two triangular systems

$$\begin{aligned} R^T\overline{v} &= e \\ Rv &= \overline{v}. \end{aligned} \tag{3.5}$$

We will now show how such an R can be constructed incrementally. Initially, $Q = \{x_0\}$ so that R is the 1-*by*-1 matrix $(1 + \|x_0\|^2)^{½}$. When a point y is added to Q we add a column to R as follows:

(1) Solve for r the system

$$R^T r = e + Q^T y. \tag{3.6}$$

(2) Adjoin to R on the right the column $[r^T \ \gamma]^T$, where

$$\gamma = (1 + \|y\|^2 - \|r\|^2)^{½}. \tag{3.7}$$

(Note that $\|r\|$ is the *Euclidean* norm of $r \in \mathbf{R}^m$). When a point is deleted from Q the corresponding column of R is deleted making R m-by-$(m-1)$. Then a series of orthogonal plane rotations are used to restore R to upper-triangular form (the last row is then identically zero and can be discarded). It is easy to verify that these operations maintain the relation (3.4) for the subsequent Q and R. This procedure is very efficient and there is virtually no accumulation of roundoff error [W1] unlike that common to routines that maintain an inverse and perform rank one corrections.

Theoretically, the fact that the new point y is on the near side of the hyperplane $H(x,x)$ (with $x = \mathrm{Nr}(\mathrm{co}(Q))$) guarantees that $Q \cup \{y\}$ is affinely independent if Q is. When implemented by computer, however, $\langle x, y \rangle < \langle x, x \rangle$ is no longer a guarantee that the set $Q \cup \{y\}$ will be affinely independent within the machine. This problem can occur when the point y is *very close* to the hyperplane $H(x,x)$. Fortunately, this type of problem will usually only happen when we are very close to solving the problem and would not have occurred if a slightly more modest precision had been requested.

The above method helps provide an additional check on the affine independence of the point set Q. If the system matrix $ee^{\mathrm{T}} + Q^{\mathrm{T}}Q$ loses its positive definiteness, then the point set Q is no longer affinely independent. This possibility can be detected during the incremental update by checking to see if γ^2 from (3.7) is strictly positive. If γ^2 is not positive, then $ee^{\mathrm{T}} + Q_+^{\mathrm{T}} Q_+$ is not positive definite and $Q_+ \triangleq Q \cup \{y\}$ is not affinely independent. If this happens there are two options.

The first option is to compute the *limiting* value ρ^* such that $\rho > \rho^*$ would result in the current x satisfying the stopping condition in Step 3 of Algorithm 2.3. This value is given by

$$\rho^* \triangleq 1 + \frac{\sigma(-x)}{\| x \|^2} = \frac{\langle x, x - y \rangle}{\| x \|^2}. \tag{3.8}$$

If the value of ρ^* is sufficiently small for the purposes at hand, then stop returning the current x. (In some cases, you may have no choice, the contact point routine may not return values accurate enough to improve on this precision.

The second option is to reset to two points by discarding all points except the current minimizing x and the new point y. As long as y is *far enough* away from x and $H(x, x)$, then some decrease in norm *may* be possible. However, since the two point method tends to exhibit slower convergence, it is likely that any improvement we get will be paid for dearly. Furthermore, we are probably getting close to the point where no further accuracy is possible due to finite length arithmetic.

3.2. Numerical Results

In order to compare the results obtained using Wolfe's method to previous methods, we consider the following general set suggested by Gilbert [G1]:

$$C \triangleq \{ x \in \mathbf{R}^n \mid x^1 \geq v + \frac{1}{2} \sum_{i=2}^{n} \frac{(x^i)^2}{\lambda^i},\ x^1 \leq kv \} \tag{3.9}$$

where $v > 0$, $\lambda^i > 0$, $i = 2,\ldots,n$, and $k > 1$. The nearest point in C is $x^* = [v, 0, \ldots, 0]^T$ and the λ^i are the principal radii of curvature of the boundary of C at the point x^*. Since many other sets C have a boundary surface that can be closely approximated by a similar representation in the neighborhood of x^*, this example is of general interest.

In many problems, the evaluation of the contact points for the set C will be the most time consuming part of the nearest point calculation. Since each iteration involves computing such a contact point, the number of iterations required to satisfy certain error criteria can be used as a measure of performance.

Table 3.1

Number of iterations to satisfy $\| x_i \| - \| x^* \| \leq \delta$.

		Wolfe's method			Barr's method			Gilbert's method		
		δ			δ			δ		
λ^2	λ^3	1	10^{-3}	10^{-6}	1	10^{-3}	10^{-6}	1	10^{-3}	10^{-6}
10	10	3	7	12	3	7	12	4	28	41
100	10	6	17	32	6	17	32	11	53	92
1000	10	7	18	28	7	20	29	28	189	341
100	100	4	9	14	4	9	13	10	36	93
1000	100	6	16	26	8	23	32	43	142	386
1000	1000	4	9	13	4	10	12	86	223	301

Table 3.1 present results for the set C defined by (3.9), with $n = 3$, $v = 1$, and $x_0 = [6\ 2\ 2]^T$, using the methods of Wolfe, Barr, and Gilbert. The data for Barr's method is that given in [B1]. The data tends to indicate that the speed of

convergence of the two-point method (Gilbert) depends strongly on the ratio of the maximum radius of curvature to the minimum distance, the convergence being slow when this value is high. The speed of convergence of the methods of Wolfe and Barr is much more rapid and exhibits much less dependence on this ratio. Wolfe's method seem to be doing about the same thing as Barr's method (the small differences in number of iteration is probably due to the different computers on which they were tested), except in a more efficient manner (recall that Barr used a quadratic program during each iteration). Figure 3.1 shows the logarithm plot of error versus iterations for Wolfe's method and Gilbert's two-point method.

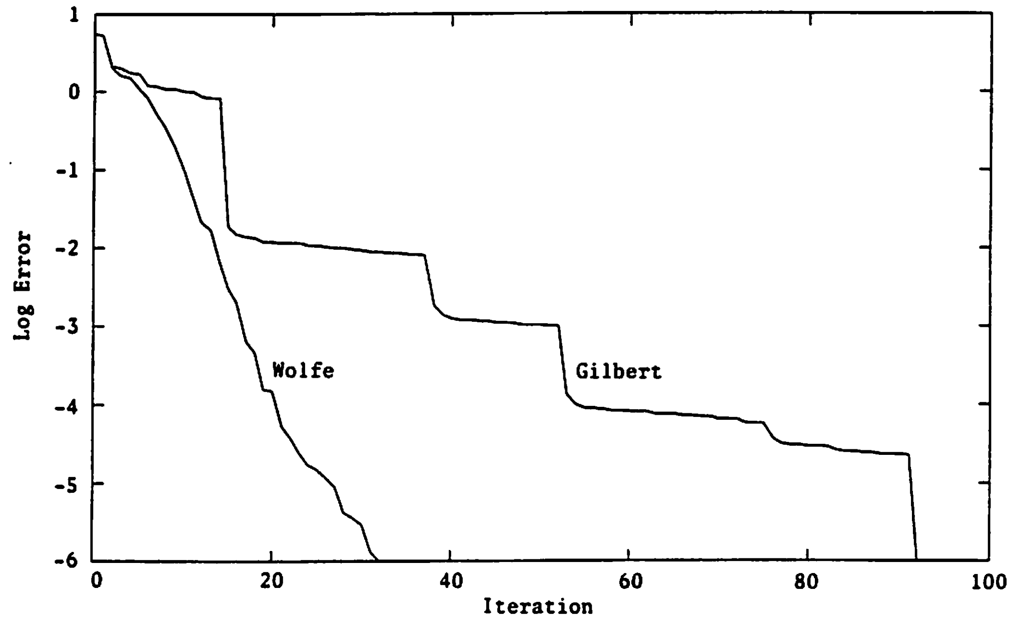

Figure 3.1 Plot of log($\| x \|$ - $\| x^* \|$) with $\lambda^2 = 100$, $\lambda^3 = 10$.

3.3. Application: Nondifferentiable Optimization

In this section we show how the proximity algorithm can be used to compute a descent direction for a nondifferentiable optimization problem. The particular problem we will investigate involves minimizing the maximum eigenvalue of a parameterized positive semi-definite matrix. Such a problem can be used to specify performance in control system design (see e.g., [P1]).

To set the problem, let $A:\mathbf{R}^p \to \mathbf{R}^{n\times n}$ be such that $A(z)$ is positive semi-definite for all $z\in\mathbf{R}^p$. The problem can then be stated as

$$\min_{z\in\mathbf{R}^p} \lambda_{\max}(A(z)). \tag{3.10}$$

Since the maximum eigenvalue of $A(z)$ can be expressed as

$$\lambda_{\max}(A(z)) = \max_{\substack{u\in\mathbf{R}^n \\ \|u\|=1}} \langle u, A(z)u\rangle, \tag{3.11}$$

problem (3.10) can be restated as

$$\min_{z\in\mathbf{R}^p} \psi(z), \tag{3.12}$$

where $\psi(z) \triangleq \max_{\substack{u\in\mathbf{R}^n \\ \|u\|=1}} \langle u, A(z)u\rangle$. Using the techniques set forth in [P1], we can construct an *augmented convergent direction finding* map $\overline{G}\psi(z)$ (a set valued map) whose nearest point, $\overline{h}(z) = [h^0(z)\ h(z)^{\mathrm{T}}]^{\mathrm{T}}$, determines a continuous descent direction, $h(z)$. For this case, we can define

$$\overline{G}\psi(z) \triangleq \mathrm{co}(\{\begin{bmatrix}\psi(z)-\langle u, A(z)u\rangle \\ v\end{bmatrix} \mid \|u\| = 1, u\in\mathbf{R}^n\}), \tag{3.13}$$

where the vector $v\in\mathbf{R}^p$ has components $v^i = \langle u, \frac{\partial A(z)}{\partial z^i} u\rangle$. In order to use a proximity algorithm, we must be able to compute contact points. That is, given $\overline{x}\in\mathbf{R}^{p+1}$, compute $\overline{y}\in\mathbf{R}^{p+1}$ such that

$$\langle \overline{x}, \overline{y}\rangle = \max_{\overline{y}'\in\overline{G}\psi(z)} \langle \overline{x}, \overline{y}'\rangle. \tag{3.14}$$

Write (letting $\overline{x} = [x^0\ x^{\mathrm{T}}]^{\mathrm{T}}$)

$$\max_{\bar{y} \in \overline{G}\psi(z)} \langle \bar{x}, \bar{y} \rangle = \max_{\substack{u \in \mathbf{R}^n \\ \|u\|=1}} \{x^0(\psi(z) - \langle u, A(z)u \rangle) + \langle x, v \rangle \tag{3.15}$$

$$= x^0\psi(z) + \max_{\substack{u \in \mathbf{R}^n \\ \|u\|=1}} \left\langle u, \left[\sum_{i=1}^{p} x^i \frac{\partial A(z)}{\partial z^i} - A(z) \right] u \right\rangle. \tag{3.16}$$

The $u \in \mathbf{R}^n$ that achieves the maximum in (3.16) is an eigenvector corresponding to the maximum eigenvalue of the symmetric matrix

$$\left[\sum_{i=1}^{p} x^i \frac{\partial A(z)}{\partial z^i} - A(z) \right] \tag{3.17}$$

and can therefore be found using a standard routine from a package such as EISPACK [S1]. The corresponding contact point $\bar{y}$ is easily computed.

We have implemented this approach for the example given in [P1] where

$$A(z) = \begin{bmatrix} 2 + (x^1 - x^2)^2 + (x^3)^2 & (x^1 - x^2)(1 - x^3) & x^2x^3 \\ (x^1 - x^2)(1 - x^3) & (x^1)^2 + (x^2)^2 + 3(x^3)^2 & x^1x^3 \\ x^2x^3 & x^1x^3 & 2 + (x^1x^3)^2 \end{bmatrix}. \tag{3.18}$$

All computations were performed using an Armijo type step size rule [P2] and were stopped when the minimum distance to $\overline{G}\psi(z)$ was less than 10^{-5}. Initially, z was set to $[4, 4, 4]^T$ for easy comparison with the results in [P1].

The accuracy to which we compute the nearest point will certainly affect the quality of the search direction obtained. We would therefore like to find a precision that strikes a balance between the quality of the search direction and the number of iterations taken by the proximity algorithm.

Table 3.2 presents the results of using various precisions, ρ, on the number of iterations that the main optimization routine required to satisfy an optimality test ($\| \mathrm{Nr}(\overline{G}\psi(z_k)) \| < 10^{-5}$). Times are included to emphasize the fact that computing the nearest point to higher precision (smaller ρ) may increase the cost of the search direction procedure enough to negate overall savings. This evidence is, of course, inconclusive since other factors, such as step length calculation and problem scaling, may have a major influence on such computations. These

Table 3.2

Effect of ρ in computing $\min \lambda_{\max}(A(z))$.

ρ	iterations	time (sec)
.001	10	2.34
.005	11	2.26
.01	10	2.16
.05	8	1.84
.1	10	2.28
.5	15	2.52
.9	22	4.01

results tend to indicate that a value of ρ between .1 and .01 (such as .05) may be a *good* choice in determining the search direction accurately without placing more emphasis on it than is really necessary.

As Table 3.1 indicates, the proximity algorithms will typically require more iterations to compute the nearest point as the radius of curvature of the set increases relative to the minimum distance. Since the set $\overline{G}\psi(z)$ approaches the origin as z approaches a stationary point (at a stationary point z^*, $0 \in \overline{G}\psi(z^*)$), it is likely that we will encounter this situation.

Table 3.3 shows the number of iterations required in the proximity algorithms of Wolfe and Gilbert to compute a descent direction during each iteration of the main optimization algorithm. These results are tabulated for $\rho = .05$ and provide the value of the cost function ($\psi(z) = \lambda_{\max}(A(z))$), the number of iterations needed by the proximity algorithm to compute the (approximate) nearest

Table 3.3

Iterations required by proximity algorithms during $\min \lambda_{\max}(A(z))$.

	Wolfe's method			Gilbert's method		
iteration	$\lambda_{\max}(A(z))$	iterations required	$\| x^* \|$	$\lambda_{\max}(A(z))$	iterations required	$\| x^* \|$
0	260.47	3	133.41	260.47	11	132.15
1	77.889	3	33.246	79.076	3	33.909
2	28.122	3	12.433	28.256	4	12.488
3	6.8292	2	3.8199	6.8504	2	3.8348
4	2.0666	9	.21805	2.0745	316	.079028
5	2.0033	12	3.3065e-3	2.0078	**	**
6	2.0003	16	3.2130e-4			
7	2.00003	14	3.3376e-5			
8	2	20	2.8453e-6			

point ($\bar{x}^*$), and the norm of this point for each iteration. Note that the two-point method of Gilbert was unable to compute a reasonable estimate of the nearest point in iteration five (as indicated by the stars). The procedure was terminated after more than 4000 iterations (and 3 minutes). This same behavior was observed for all values of ρ tested. We therefore conclude that a two-point method is definitely *not* a good method to use in such problems.

3.4. Conclusion

We have studied methods for estimating the minimum norm point in a convex set for which contact points can be evaluated. Most of these methods have required finding the nearest point in a polytope as a subproblem. The questions of choosing the polytope and finding its nearest point were considered. We recommend using Wolfe's algorithm [W2] over general-purpose quadratic programming algorithms to find the nearest point in the polytope because of the savings in computational effort, storage, and roundoff error. Choosing the polytope to be the convex hull of the previous corral and the current contact point produces a very efficient algorithm that incrementally updates the data structures required for finding the nearest point using Wolfe's method. Furthermore, this method always works within the dimension of the set (i.e., never keeps more than $\dim(C) + 1$ points) rather than the dimension of the space (as Barr [B1] does). This property is especially useful when the set is not of full dimension (i.e., $\dim(C) \neq \dim(\mathbf{H})$). We recommend the method set forth in Algorithm 2.48 as an efficient, stable computational method for estimating the minimum norm point in a convex set.

References

[B1] Barr, R.O., "An Efficient Computational Procedure for a Generalized Quadratic Programming Problem", *SIAM Journal on Control* 7 (1969) 415-429.

[B2] Berge, C., *Topological Spaces*. New York: The Macmillan Company, 1963.

[B3] Brondsted, A., *An Introduction to Convex Polytopes*. New York: Springer-Verlag (Graduate Texts in Mathematics; v. 90), 1983.

[C1] Clarke, F.H., *Optimization and Nonsmooth Analysis*. New York: John Wiley & Sons, 1983.

[F1] Frank, M. and P. Wolfe, "An Algorithm for Quadratic Programming", *Naval Research Logistics Quarterly* 3 (1956) 95-110.

[G1] Gilbert, E.G., "An Iterative Method for Computing the Minimum of a Quadratic Form on a Convex Set", *SIAM Journal on Control* 4 (1966) 61-80.

[G2] Gilbert, E.G. and D.W. Johnson, "Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles", *IEEE Journal of Robotics and Automation* RA-1 (1985) 21-30.

[G3] Golub, G.H. and C.F. Van Loan, *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 1983.

[M1] Mayne, D.Q. and E. Polak, "An Exact Penalty Function Algorithm for Control Problems with State and Control Constraints", Memorandum No. UCB/ERL M85/52, Electronics Research Laboratory, University of California, Berkeley, 1985.

[M2] Meyer, G. and E. Polak, "A Decomposition Algorithm for Solving a Class of Optimal Control Problems", *Journal of Mathematical Analysis and Applications* 32 (1970) 118-140.

[R1] Rockafellar, R.T., *Convex Analysis.* Princeton: Princeton University Press, 1970.

[P1] Polak, E., *Computational Methods in Optimization: A Unified Approach*, New York: Academic Press, 1971.

[P2] Polak, E. and D.Q. Mayne, "A Feasible Directions Algorithm for Optimal Control Problems with Control and Terminal Inequality Contraints", *IEEE Transactions on Automatic Control* AC-22 (1977) 741-751.

[P3] Polak, E. and Y. Wardi, "Nondifferentiable Optimization Algorithm for Designing Control Systems Having Singular Value Inequalities", *Automatica* 18 (1982) 267-283.

[P4] Polak, E., "On the Mathematical Foundations of Nondifferentiable Optimization in Engineering Design", Memorandum No. UCB/ERL M85/17, Electronics Research Laboratory, University of California, Berkeley, 1985.

[S1] Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y.Ikebe, V.C. Klema, and C.B. Moler, *Matrix Eigensystem Routines - EISPACK Guide*, New York: Springer-Verlag, 1976.

[V1] Valentine, F.A., *Convex Sets.* Huntington, New York: Robert E. Krieger Publishing Company, Inc., 1976.

[V2] van Tiel, J., *Convex Analysis: An Introductory Text.* Chicester: John Wiley & Sons Ltd., 1984.

[W1] Warga, J., "Iterative Procedures for Constrained and Unilateral Optimization Problems", *SIAM Journal on Control and Optimization* 20 (1982) 360-376.

[W2] Wolfe, P., "Finding the Nearest Point in a Polytope", *Mathematical Programming* 11 (1976) 128-149.

Appendix

In this appendix we present an implementation of Algorithm 2.48. The file *wolfeg.c* contains the C functions used in this implementation. These routines have been written to allow easy use from programs written in FORTRAN as well as C.

To allow for this flexibility, we must follow the convention that C routines which are called from FORTRAN programs must end in an underscore, e.g., *wolfeg_*. The trailing underscore is needed to allow for the correct linking of these routines. Under UNIX†, the C compiler, *cc*, produces labels of the form *_proc*, where *proc* is the function name, while the FORTRAN compiler, *F77*, produces labels of the form *_proc_*. The linker, *ld*, then uses these labels to set up the calling sequences. Therefore, we use *wolfeg_* in C to refer to the same function that will be called *wolfeg* in FORTRAN. This convention is, of course, arbitrary and will likely vary under different operating systems. For the purposes of the following discussion we shall simply refer to the function as *wolfeg*.

In order to use *wolfeg*, the user must provide an initial point in the set of interest and a subroutine that computes contact points for this set. This user-defined subroutine will be called from within *wolfeg* each time a contact point is needed (i.e., in Step 2 of Algorithm 2.48). The contact point subroutine should accept three parameters: x, y, and *param*. The variable x contains the direction of interest (a vector of n components). Given x, the subroutine should compute a contact point and place it in y (also a vector of n components). Thus, with C denoting the set of interest, we have that

$$\langle x, y \rangle \leq \langle x, y' \rangle \quad \forall\, y' \in C. \tag{A.1}$$

The variable *param* is a pointer (or address) that can be used to pass parameters to this user-defined subroutine through *wolfeg*. For example, in FORTRAN,

† UNIX is a trademark of Bell Laboratories.

param could be (the address of) an array of parameters needed by the contact point subroutine. In C, a pointer to a structure of parameters could be passed allowing arbitrary information exchange between the calling program and the contact point subroutine via *param*.

To allow for exceptions to the normal stopping conditions, *wolfeg* requires two additional integer parameters, *imax* and *ierr*. As the name indicates, *imax* gives the user the option of specifying a maximum number of iterations that the main loop of *wolfeg* (i.e., number of contact points computed) will execute. If *imax* is set to a value other than zero, then *wolfeg* will execute no more than *imax* iterations. Setting *imax* to zero implies no restriction. On return, imax contains the actual number of iterations that were executed. The parameter *ierr* is used to indicate the actual stopping condition used. *Ierr* is set to zero if the final x satisfies the stopping condition of Step 3 of Algorithm 2.48 so that

$$\| x \|^2 + \sigma(-x) < \rho \| x \|^2. \tag{A.2}$$

If *wolfeg* stopped at Step 1 of Algorithm 2.48 with

$$\| x \| < \epsilon \tag{A.3}$$

then *ierr* is set to one. If *ierr* has been set to either two or three then computations were stopped for numerical reasons. The value of four in *ierr* implies that *imax* iterations were performed without satisfying any other stopping rule. In any case, the parameter *rho* is set to the value of the limiting ρ^* defined by (3.8) corresponding to the final x. This value can then be used to judge the quality of the final x in light of the actual stopping conditions.

The calling sequence and nature of the parameters are described in detail in the comments of the program itself. To aid the potential user, we have also included an ex

```
#define DEBUG

double wolfeg_(x,  n,  eps,  rho,   tanpt,     imax,  ierr,  param)
double        *x,     *eps, *rho;
int              *n,               (*tanpt)(), *imax, *ierr;
char                                                        *param;
{

/*
**      wolfeg_         estimate nearest point in general convex set
**
**
**      This function estimates the nearest point in a convex set to
**      a desired precision.  The method used is an extension of
**      P. Wolfe's method for finding the nearest point in a polytope
**      (P. Wolfe, "Finding the Nearest Point in a Polytope",
**      Mathematical Programming 11 (1976) 128-149.).
**      This extension corresponds to Algorithm 2.48 in
**      J. Hauser, "Proximity Algorithms: Theory and Implementation".
**
**
**      Starting with initial point 'x', wolfeg_ will find a point
**      in the set whose contact points are determined by 'tanpt'
**      that is either 'close' to the origin (as measured by 'eps')
**      or 'close' to the nearest point in the (closure of the) set.
**
**      on input-
**
**        x     is the initial point in the set ('n' component vector)
**
**        n     is the dimension of 'x' (passed by address)
**
**        eps   is the zero threshold (passed by address)
**
**        rho   is the desired ratio precision (passed by address)
**
**        tanpt is a subroutine used to determine contact points
**              in the set of interest.  this subroutine is called with
**              three parameters 'x', 'y', and 'param' as in
**
**                      tanpt(x, y, param);
**
**              'x' is the input direction and 'y' is the output
**              contact point.  i.e.,
**
**                      < x , y >  <=  < x , y' >
**
**              for all y' in C.  'param' is used to pass parameters.
**
**        imax  is the maximum number of iterations to be performed.
**              if 'imax' is set to zero, then no limit is imposed.
**              (passed by address).
**
**        param is a pointer to a parameter area that can be used
**              to pass parameters to the function tanpt.
**              it is declared
**                     char *param;
**              to imply 'generic pointer'.  In C, there is great
**              flexibility.  In FORTRAN, one is probably limited
**              to passing a single variable or array.  At any rate,
**              as long as the calling procedure and the subroutine
**              'tanpt' agree, there should be no trouble.
**
**
**      on output-
**
**        x     contains the resulting approximation to the nearest
**              point in the set.
**
**        n     is unaltered.
**
**        eps   is unaltered.
**
**        rho   contains the limiting rho achieved by the final point.
**              i.e., rho = ( < x , x > - < x , y > ) / < x , x >,
**              unless || x || < eps in which case rho is unaltered.
**
**        imax  contains the actual number of iterations executed.
**
**        ierr  is set to
**
**              zero    if a nominal 'x' was obtained.  i.e.,
**                      < x , x >  -  < x , y >  <  rho * < x , x >.
**
**              one     if 'x' approximates zero.  i.e.,
**                      || x ||  <  eps.
**
**              two     if computation was stopped because the 'rho'
**                      requested could not be achieved without the
**                      system matrix losing its positive definiteness.
**                      (as a result of numerical errors).
**
**              three   if computation was stopped because the norm
**                      was not decreased (because of numerical errors).
**
**              four    if the imax iterations were executed without
**                      satisfying any other stopping condition.
**
**
*******************************************
**
**      example usage
**
*******************************************
**
**
**      FORTRAN
**
**              program foo
**              external tanpt
**              double precision x(20), eps, rho, dummy
**              integer n, imax, ierr
**                  .
**                  .
**                  .
**              dist = wolfeg(x,n,eps,rho,tanpt,imax,ierr,dummy)
**                  .
**                  .
**                  .
**              end
**
**              subroutine tanpt(x, y, d)
**              double precision x(1), y(1), d(1)
**                  .
**                  .
**                  .
**              end
**
*****************************************
**
**      C
**
**              main()
**              {
**                  double wolfeg_();
**                  double x[20], eps, rho, params[3];
**                  int n, imax, ierr, tanpt();
**                      .
**                      .
**                      .
**                  dist = wolfeg_(x, &n, &eps, &rho, tanpt,
**                          &imax, &ierr, (char *) params);
**                      .
**                      .
**                      .
**                  }
**
**              tanpt(x, y, params)
**              double *x, *y;
**              char *params;
**              {
**                      .
**                      .
**                      .
**                  }
**
*/


        /* zero thresholds */

#define Z1              1e-10
#define Z2              1e-10
#define Z3              1e-10

        /* constants for stopping & reseting */

#define MINEPS          1e-12           /* min zero threshold */
```

```
#define MINRHO          1e-10           /* min stopping ratio */
#define MINGAMMA2       0               /* square of smallest R diag */

        /* other useful items */

#define FALSE           0
#define TRUE            ~FALSE
#define max(a,b)        ((a)>(b)?(a):(b))
#define min(a,b)        ((a)<(b)?(a):(b))
#define llog(x)         (log10((max(1e-38,(x)))))

#include <math.h>

#ifdef DEBUG
#include <stdio.h>
#endif

    char *calloc();

    static int worksize;        /* size of working storage */

    static double *work;        /* pointer to working storage */


    int i,
        j,
        k,              /* iteration counter */
        il,
        nn,             /* dimension of points */
        qq,             /* length of column in r */
        ncpts,          /* number of points in corral */
        ndpts,          /* number of points in discard */
        nfpts,          /* number of points in freepts */
        small,          /* boolean toggle */
        zeroi;          /* point with zero weight */

    double
        norm2x,         /* squared norm of current x */
        onorm2x,        /* squared norm of previous x */
        xdoty,          /* inner product of x and contact point y */
        eps2,           /* zero threshold for stopping */
        rho_,           /* ratio for stopping */
        **corral,       /* ptr to array of ptrs to corral points */
        **discard,      /* ptr to array of ptrs to discarded points */
        **freepts,      /* ptr to array of ptrs to free points */
        *rr,            /* ptr to R array */
        *w,             /* ptr to weights for 'x' */
        *v,             /* ptr to weights for minimum on affine hull */
        *bb,            /* ptr to scratch vector */
        lambda,         /* convex multiplier */
        a, b, c, d,     /* used for plane rotation */
        dot(),
        norm2();


    /*
    **  work    is partitioned as follows:
    **
    **          r       the first (n + 1)(n + 1) elements set up
    **                  as a square matrix (see define below).
    **
    **          w       is stored in the next (n + 1) elements.
    **
    **          v       is stored in the next (n + 1) elements.
    **
    **          bb      is stored in the next (n + 1) elements.
    **
    **          corral  is stored in the next (n + 1) elements.
    **                  (an array of pointers to vectors)
    **
    **          discard is stored in the next (n + 1) elements.
    **                  (an array of pointers to vectors)
    **
    **          freepts is stored in the next (n + 1) elements.
    **                  (an array of pointers to vectors)
    **
    **          ***     the next n(n+2) elements contain the (n+2)
    **                  vectors of size n that are allocated to the
    **                  corral, discard, and freepts areas.
    **
    **  hence the required storage is 2n^2 + 10n + 7
    **  this storage is allocated dynamically as one block when
    **  needed and reallocated if a larger block is needed.
    */

#define r(i,j)  rr[(i) + (j)*qq]                /* stored columnwise */


    /* initialize pointers for working vectors, etc. */

    nn = *n;                        /* dimension of points */
    qq = nn + 1;                    /* size of maximal corral */

    /* get working storage (if needed) and allocate it */

    i = 2*nn*(nn + 5) + 7;          /* calculate storage needed */
    if (worksize < i)  {            /* present storage insufficient */
        if (work)                   /* free up previously allocated */
            free( (char *) work);       /* storage */
        worksize = i;               /* save size */

        /* get needed storage */

        work = (double *) calloc( (unsigned) i, sizeof(double));

        }

    rr = work;                      /* start at beginning */
    w = &r(0,nn + 1);               /* weights for 'x' */
    v = &r(0,nn + 2);               /* weights for 'y' */
    bb = &r(0,nn + 3);              /* scratch vector */
    corral = (double **) &r(0,nn + 4);          /* corral points */
    discard = (double **) &r(0,nn + 5);          /* corral points */
    freepts = (double **) &r(0,nn + 6);          /* corral points */
    ncpts = 1;                          /* initial number of corral points */
    nfpts = qq;                         /* number of free points */
    ndpts = 0;                          /* initially no discarded points */

    corral[0] = (double *) &r(0,nn + 7);    /* set up for first point */

    /* set up pointers to free space */

    i = nfpts;
    freepts[--i] = corral[0] + nn;
    for (; i--; )
        freepts[i] = freepts[i + 1] + nn;

    eps2 = max(MINEPS,*eps);
    eps2 *= eps2;          /* use square in stop rule */
    rho_ = max(MINRHO, *rho);


    /*  Step 0  --  get initial corral   */

    for (i=nn; i--; )   /* initial corral is given point in set */
        corral[0][i] = x[i];

    ncpts = 1;            /* one point in the corral */
    w[0] = 1.;

    /* initial R matrix (1 by 1) and set max squared norm */

    r(0,0) = sqrt(1. + norm2(x,nn));


    /* main loop of algorithm */

    for (k=0; ; k++) {


#ifdef DEBUG

        printf("x[%d]:  ",k);
        for (i=0; i<nn; i++)
            printf("%g  ",x[i]);
        printf("\n");
#endif

        /** Step 1 ** --  check for small x */

        if ((norm2x = norm2(x,nn)) < eps2)  {
            *imax = k;
            *ierr = 1;
            return (sqrt(norm2x));
            }


        /** Step 2 ** --  pick up tangency point */
        /* get point from free points */

        corral[ncpts++] = freepts[--nfpts];

        (*tanpt)(x, corral[ncpts-1], param);

        xdoty = dot(x, corral[ncpts-1], nn);


#ifdef DEBUG

/*
        printf("y[%d]:  ",k);
        for (i=0; i<nn; i++)
            printf("%g  ",corral[ncpts-1][i]);
        printf("\n");
*/

        c = sqrt(norm2x);
        if (norm2x > 1e-24)
            printf("%d: [%g,%g], e = %g, le = %g\n",
                k,xdoty/c,c,1.-xdoty/norm2x,llog(1.-xdoty/norm2x));
#endif

        /** Step 3 **  --  test for ratio stopping condition */

        if (norm2x - xdoty < rho_*norm2x) {
            /* stop - we have a solution */

            *imax = k;
            *ierr = 0;                          /* normal return */
            *rho = (norm2x - xdoty)/norm2x;     /* actual rho achieved*/

            return (sqrt(norm2x));              /* return distance */
            }

        /** stop if more iterations than desired */

        if (*imax && k >= *imax)  {     /* if *imax = 0, don't check */
            *imax = k;
            *ierr = 4;
            *rho = (norm2x - xdoty)/norm2x;
            return(sqrt(norm2x));
            }

        /* save x and normx to check for decrease in norm */

        for (i=nn; i--; )
            freepts[0][i] = x[i];
        onorm2x = norm2x;

        /**  Step 4  --  find nearest point in Q U {y}  */

        /**  Step 4(a) is handled implicitly  */
        /*  loop for Steps 4(b) through (e)   */

        for (;;)  {

            /**  Step 4(b)  --  add new point to corral */

            /* point is corral from either Step 2 or Step 4(e) */

            ncpts--;              /* subtract one for convenience */

            w[ncpts] = 0.;        /* zero weight */

            /* add new column to R */

            c = norm2(corral[ncpts],nn);    /* norm2 of tangency point */
            for (i = ncpts; i--; )
                bb[i] = 1. + dot(corral[i], corral[ncpts], nn);
            solve_t_upper(rr, &r(0,ncpts), bb, ncpts, qq);

            /* check that points are affinely independent in machine */
            /*   i.e., that system matrix is positive definite       */

            if ((a = 1. + c - norm2(&r(0,ncpts),ncpts)) <= MINGAMMA2)  {
                *imax = k;
                *ierr = 2;
                *rho = (norm2x - xdoty)/norm2x;
                return(sqrt(norm2x));
                }
            else  {
                r(ncpts, ncpts) = sqrt(a);
                ncpts++;
                }

            /* loop for Steps 4(c) and (d) */

            for (;;) {

                /**  Step 4(c)  --  solve system for new v */

                for (i = ncpts; i-- ; ) /* init vector of ones (e) */
                    v[i] = 1.;

                solve_t_upper(rr, bb, v, ncpts, qq);
                solve_upper(rr, v, bb, ncpts, qq);

                /* normalize v & check for small values */
                /* and compute lambda for Step 4(d) and */
                /*   element acheiving lambda */

                a = 0.;
                for (i = ncpts; i--; )
                    a += v[i];
                small = FALSE;
                lambda = 0.;              /* min is zero */
                for (i = ncpts; i--; )  {
                    if ((v[i] /= a) <= Z2)  {
```

```
                    if (!small)  {
                        small = TRUE;
                        zeroi = i;
                        }
                    if ((b = w[i] - v[i]) > Z3 &&
                          (c = -v[i]/b) > lambda)  {
                        lambda = c;
                        zeroi = i;
                        }
                    }
                }

            if (! small) {  /* good positive v - put in w */

                for (i = ncpts; i--; )
                    w[i] = v[i];

                /* update x and norm2x */

                for (i = nn; i-- ; ) {
                    a = 0.;
                    for (j = ncpts; j-- ; )
                        a += corral[j][i]*w[j];
                    x[i] = a;
                    }

                norm2x = norm2(x, nn);

                /* check for decrease in norm */

                if (norm2x >= onorm2x)  {  /* no decrease -- stop */

                    *imax = k;
                    *ierr = 3;

                    /* restore previous x */

                    for (i = nn; i--; )
                        x[i] = freepts[0][i];

                    norm2x = onorm2x;
                    *rho = (norm2x - xdoty)/norm2x;

                    return (sqrt(norm2x));
                    }

                break;      /* break out (go to Step 4(e)) */
                }

            /* Step 4(d)  -- project and delete point not in face */

            /* lambda and zeroi computed above */

            /*      w  =  lambda * w  +  (1 - lambda) * v   */

            for ( i = ncpts; i--; )
```

```
                w[i] = lambda*w[i] + (1.-lambda)*v[i];

            /* delete zeroi-th element from Q, w, & R  */

            ncpts--;                    /* one less point in corral */

            /* put zeroi-th element of corral on discard list */

            discard[ndpts++] = corral[zeroi];

            for (il = (i = zeroi) + 1; i < ncpts; i++, il++) {
                w[i] = w[il];
                for (j = il + 1; j--; )
                    r(j,i) = r(j,il);
                corral[i] = corral[il];
                }

            /* use plane rot to restore R to upper triangular form*/

            for (il = (i = zeroi) +1; i < ncpts; i++, il++) {
                a = r(i,i);
                b = r(il,i);
                a /= (c = sqrt(a*a + b*b));
                b /= c;
                for (j = i; j < ncpts; j++) {
                    r(i,j) = a*(c = r(i,j)) + b*(d = r(il,j));
                    r(il,j) = -b*c + a*d;
                    }
                }

            }       /* end of Step 4(c) & (d) loop (i.e., go (c)) */

        /**  Step 4(e)  --  check discarded points for separation */

        if (ndpts < 1)  break;      /* no points to check */

        /* find min dot in discard set */

        j = i = ndpts - 1;
        a = dot(x, discard[i], nn);
        for (; i-- > 0; )
            if ((b = dot(x, discard[i], nn)) < a)  {
                a = b;      /* a contains min dot */
                j = i;      /* j contains its index */
                }

        if (norm2x < eps2 || norm2x - a <= Z1*norm2x)  {
            /* all discarded points are separated */

            /* place discard points on free list */

            for (; ndpts--; )
                freepts[nfpts++] = discard[ndpts];

            ndpts++;
```

```
                    break;  /* i.e., go to Step 1 */
                    }

                /* else */

                /* place j-th point back in corral for further decrease */

                corral[ncpts++] = discard[j];

                ndpts--;

                /* shuffle discard pile */

                for (; j < ndpts; j++)
                    discard[j] = discard[j+1];

                }   /*  end of step 4(b) - (e) loop  (i.e., go to (b))  */


            }       /* end of main loop  (i.e., go to Step 1)  */

#undef r
    }           /* end of wolfeg_ function */


/*      solve_upper     solve R*x = b w/ R upper triangular
**
**              note:  do not called from fortran
*/

static solve_upper(R,  x,  b, n, q)
double            *R, *x, *b;
int                         n, q;
{
#define RR(i,j) R[(i) + (j)*q]          /* stored columnwise */

    int i, j;
    double z;

    for (i=n; i--; )  {
        z = 0.;
        for (j=i+1; j<n; j++)
            z += RR(i,j)*x[j];
        x[i] = (b[i] - z)/RR(i,i);
        }

#undef RR
    }  /* end of solve_upper */

/*      solve_t_upper   solve R-tr*x = b w/ R upper triangular
**
**              note:  do not called from fortran
*/

static solve_t_upper(R,  x,  b, n, q)
double            *R, *x, *b;
```

```
int                               n, q;
{
#define RR(i,j) R[(i) + (j)*q]

    int i, j;
    double z;

    for (i=0; i<n; i++)  {
        z = 0.;
        for (j=i; j--; )
            z += RR(j,i)*x[j];
        x[i] = (b[i] - z)/RR(i,i);
        }

#undef RR
    }   /* end of solve_t_upper */

/* dot  returns dot product of 2 n-vectors
**
**      note:  not to be called from fortran routines
*/

static double dot(x,  y, n)
double            *x, *y;
int                      n;
{
    double d;

    d = 0;
    for (; n--; x++, y++)
        d += *x * *y;
    return(d);
    }

/* norm2  returns sqared norm of n-vector
**
**      note:  not to be called from fortran routines
*/

static double norm2(x,  n)
double              *x;
int                     n;
{
    double d;

    d = 0;
    for (; n--; x++)
        d += *x * *x;
    return(d);
    }
```

```
#include <stdio.h>

#define N        20
#define max(a,b)        ((a)>(b)?(a):(b))
#define min(a,b)        ((a)<(b)?(a):(b))

struct parameter {
        int n;
        double lambda[N+1];
        };

main()
{
    int
        i,
        n,
        imax,
        ierr,
        quadtpt();

    double
        d,
        eps,
        rho,
        x[N],
        wolfeg_();

    struct parameter params;

    for (;;)  {

        printf("enter n (max %d): ", N);
        scanf("%d",&n);
        if (n == 0)  break;

        n = min(n,N);
        params.n = n;

        printf("enter v: ");
        scanf("%lf",&params.lambda[0]);

        printf("enter lambda[2 ... %d]: ", n);
        for (i=1; i<n; i++)
            scanf("%lf",&params.lambda[i]);

        printf("enter x: ");
        for (i=0; i<n; i++)
            scanf("%lf",&x[i]);

        printf("enter eps: ");
        scanf("%lf",&eps);

        printf("enter rho: ");
        scanf("%lf",&rho);

        printf("enter imax: ");
        scanf("%d",&imax);

        printf("\nWolfe algo:\n");
        printf("initial x:\n");
        for (i=0; i<n; i++)
            printf("  %g",x[i]);
        printf("\n");
        d = wolfeg_(x, &n, &eps, &rho, quadtpt,
                &imax, &ierr, (char *) &params);
        printf("min dist is %g + %g\n",
                params.lambda[0], d - params.lambda[0]);
        printf(" imax = %d, ierr = %d, rho = %g\n", imax, ierr, rho);
        printf("final x:\n");
        for (i=0; i<n; i++)
            printf("  %g",x[i]);
        printf("\n");

        }
    }

quadtpt(x,  y,  params)
double *x, *y;
char            *params;
{
    double
        x0, y0,
        *lambda,
        dot();

    int i,
        n;

    struct parameter *p;

    n = ((struct parameter *) params)->n;
    lambda = ((struct parameter *) params)->lambda;

    if ((x0 = *x) < (y0 = *lambda))  {
        /* x is not in set !! */
        for (i=n; i--; )
            *(y++) = 0.;
        }

    /* start from the last element of each vector */

    lambda += n - 1;
    y += n - 1;
    x += n - 1;

    for (i=n-1; i--; lambda--, x--, y--)  {
        *y = -(*lambda)*(*x)/x0;
        y0 += (*y)*(*y)/(*lambda)/2.;
        }
    *y = y0;

    }
```

```
      program main

      integer i, n, imax, ierr
      external quadtpt, wolfeg
      double precision d, eps, rho, x(20), params(21)

c     main loop (infinite loop)

 1    continue

      print *, "enter n (max 20): "
      read *, n
      if (n .eq. 0)  stop

      if (n .gt. 20) n = 20

      params(1) = n

      print *, "enter v: "
      read *, params(2)

      print *, "enter lambda(2 ... ", n, "): "
      read *, (params(i), i=3,n+1)

      print *, "enter x: "
      read *, (x(i), i=1,n)

      print *, "enter eps: "
      read *, eps

      print *, "enter rho: "
      read *, rho

      print *, "enter imax: "
      read *, imax

      print *, " "
      print *, "Wolfe algo:"
      print *, "initial x:"
      print *, (x(i), i=1,n)
      d = wolfeg(x,n,eps,rho,quadtpt,imax,ierr,params)
      print *, "min dist is ", params(2), " + ", d - params(2)
      print *, " imax = ", imax, ", ierr = ", ierr, ", rho = ", rho
      print *, "final x:"
      print *, (x(i), i=1,n)

      go to 1

      end


      subroutine quadtpt(x, y, params)
      double precision x(1), y(1), params(1)

      double precision x0, y0
      integer i, n
      n = params(1)

      x0 = x(1)
      y0 = params(2)

      if (x0 .lt. y0) then
c         /* x is not in set !! */
          do 1 i=1,n
 1            y(i) = 0.0
          return
          end if

      do 2 i=2,n
          y(i) = -params(i+1)*x(i)/x0
          y0 = y0 + y(i)*y(i)/params(i+1)/2.0
 2        continue

      y(1) = y0
      return

      end
```