INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Fast Computation of Minimum Distance Among Convex 3D Objects

by

David Kirk Fenger
B.A.Sc., University of Waterloo, 1991
M.A.Sc., University of Victoria, 1993

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming to the required standard

Dr. W.-S. Lu, Supervisor (Department of Electrical and Computer Engineering)

Dr. P. Agathoklis, Departmental Member (Electrical and Computer Engineering)

Dr. J. S. Collins, Departmental Member (Electrical and Computer Engineering)

Dr. R. P. Podhorodeski, Outside Member (Department of Mechanical Engineering)

Dr. C. M. Leung, External Examiner (Camosun College, Victoria, BC, Canada)

University of Victoria

Supervisor: Dr. Wu-Sheng Lu

ABSTRACT

A major issue in robotic control is the avoidance of collisions with the workspace or with other active robots in the case of cooperative systems. Many control and motion planning methods use some form of distance metric to determine how close the manipulator is to its environment. As a result, rapid calculation of distances between robot components and workspace obstacles is crucial to efforts in on-line collision avoidance. While efficient methods exist, further refinement is always desirable.

The computation of distances between convex polyhedra in 3D space can be presented as a Quadratic Programming problem. The DQP3 (Distance via Quadratic Programming, 3D-specific) algorithm for computing the distance between convex polyhedra is presented, based on the primal active set QP algorithm. The DQP3 algorithm enhances the primal active set method by using geometric solutions to its core calculations, instead of generalized matrix methods. Several variants of the DQP3 algorithm are presented with relative timing information. Comparison to algorithms from the literature is also performed, with CPU times scaled according to the Linpack benchmark of the CPU used. The most detailed comparison is to Cameron's Enhanced GJK (Gilbert, Johnson and Keerthi) algorithm, running his code on the same test machine as the DQP3 code.

In addition to robot components that typically have a simple polyhedral representation, ellipsoid representations of objects are also of interest, partly because of their simplicity. Three different methods of handling the ellipsoid distance are presented: a simple point to ellipsoid iteration, the use of high-definition polyhedral models (with DQP3), and an iterative process for refining the polyhedral model only in the area of interest. Comparison to the scarce results from the literature is done, but the main comparison of interest is among the proposed methods, which showed the point to ellipsoid method to be the most reliable, although the iterative refinement technique shows promise.

Examiners:

Dr. W.-S. Lu, Supervisor (Department of Electrical and Computer Engineering)

Dr. P. Agathoklis, Departmental Member (Electrical and Computer Engineering)

Dr. J. S. Collins, Departmental Member (Electrical and Computer Engineering)

Dr. R. P. Podhorodeski, Outside Member (Department of Mechanical Engineering)

Dr. C. M. Leung, External Examiner (Camosun College, Victoria, BC, Canada)

Table of Contents

List of Figures

List of Tables

Glossary

Coherence The tendency of subsequent references to an object to be proximate.

CSG Constructive Solid Geometry, assembling objects from geometric primitives.

Cycle An algorithm caught up in an infinite loop, always returning to a previous state.

DQP3 Distance via Quadratic Programming, 3D-specific

EGJK Enhanced Gilbert, Johnson and Keerthi algorithm. [Cameron97a]

Flops Floating point operations

GJK Gilbert, Johnson and Keerthi algorithm for computing the distance between convex polyhedra. [Gilbert88]

KKT Conditions Karush-Kuhn-Tucker conditions for determining the Lagrange multipliers for a minimization problem with equality and inequality constraints. [Fletcher87]

NURBS Non-Uniform Rational B-Splines

OBB Oriented Bounding Box

PD Positive definite

PSD Positive semidefinite

QP Quadratic programming

Surface radius of curvature Inverse of the second derivative of a surface.

TCSO Translational Configuration Space Obstacle [Cameron97a, Cameron97b]

Tracking problem Repeatedly computing the closest point between moving objects.

Voxel Discrete volume element, akin to 'pixel'.

Acknowledgments

I would like to thank my supervisor, Dr. Lu, for his assistance and advice during the research and preparation of this document.

I would also like to thank my parents and Aleta, both of whom helped encourage me to finish this. In addition, I would like to thank my employer, Precision MicroDynamics, Inc., for allowing me the time away from my duties to finish this document.

This work was supported in part by an NSERC PGS-B scholarship, and a grant under the GREAT program.

1. INTRODUCTION

The core of this thesis is the presentation of a convex polyhedra minimum distance calculation that is a hybrid of two approaches. The structure of the algorithm is derived from a QP optimization method, but the heart of the algorithm is intrinsically geometric. On simulated problems typical of those faced in robotics, it is the fastest to date in its field.

The algorithm is extended to the ellipsoid distance problem, a peculiarly difficult distance problem that has seen little activity in the literature. An approach using iteratively refined polyhedral models of the ellipse is described, with promising results.

1.1 General Introduction

The distance problem in robotics is a key component of collision avoidance control. Any manipulator or mobile robot that interacts with its environment must know where the obstacles in that environment are, and at either the supervisory or control level must take steps to avoid them. Efficient computation of the minimum distance problem is essential to on-line approaches to collision avoidance, and can serve to improve the speed of off-line planning as well.

A variety of mathematical models have been used to represent workspace obstacles and robot elements, from the simplicity of collections of points all the way to surfaces defined by parametric curves. One of the most popular of the more complex representations is to use a collection of convex objects to represent the manipulator and its workspace. Convex objects are particularly interesting because the distance problem between a pair of convex objects is itself a convex problem. As a result there are a variety of powerful algorithms for quickly finding the minimum.

A notable feature of the distance problem in robotics is that it does not just need to be computed once. In both on-line and off-line applications, the distance of the robot from its environment needs to be computed again and again, each time after tiny increments in

the position of the model objects in question. As a result, the closest point on the objects rarely changes dramatically, and this fact can be used to advantage. The case where the objects under consideration must be tested at many points along a trajectory is known as the *tracking problem*.

The Distance Quadratic Programming, 3D-specific or DQP3 algorithm presented here works on convex objects represented by a collection of facets. A variety of results on similar structures from the literature are used as a basis for comparison. Some results for the problem of finding the minimum distance between ellipsoids are also forthcoming.

1.2 Prior Work

Gilbert's work [Gilbert88] on the distance problem between convex polyhedra is widely regarded as the foundation of the topic. Many of the other results current in the literature are derivative of his original algorithm. While there is a convex distance computation literature that predates Gilbert (such as Lozano-Pérez and Wesley's initial development of the configuration space in 1979 [Lozano79], key to Gilbert's later work) his method is seen as the first truly efficient one.

Lin and Canny's approach to the problem [Lin91] paralleled Gilbert's work, also using vertices as a fundamental unit. The edges and facets of the object were also used extensively. Lin and Canny were the first to introduce a database of adjacency information, allowing the algorithm to run in near constant time in increasing numbers of facets. The constant time result only applies to the tracking problem, but the adjacency database greatly improves the calculation time in all circumstances.

Extensions of Gilbert's work include Sato and Cameron's efforts to provide a similar near constant time result [Sato96, Cameron97a, Cameron97b]. They use a similar approach, preprocessing the object data so that a list of the adjacent vertices for any given vertex is available.

Somewhat disregarded due to the focus Gilbert and Lin have thrown on geometric models are the optimization based results [Bobrow89, Ma92]. These use a different

representation of the objects, more facet oriented, defining the object by the intersection of a set of half-spaces. While relatively efficient, their methods were never updated to provide the near constant time result of the geometric, vertex-based methods.

1.3 Contributions Made by this Dissertation

The most significant development presented in this thesis is the DQP3 algorithm, a hybrid of the optimization and geometry based convex distance problem algorithms that exist to date. While the algorithm is not as fast as the constant-time algorithms on large facet counts, it is very competitive at the smaller facet counts typical to robotic problems. In addition, an adjacency database method is proposed, which should provide a similar speedup. In any case, it is good to see the optimization methods return to the ring.

As part of an effort to apply the DQP3 algorithm to ellipsoids, an efficient solution to the point to ellipsoid distance problem is produced, approximately 3 times as fast as current algorithms. This algorithm may be applied to the ellipsoid distance problem in an iterative manner, but the computation times show a strong inverse proportionality to the distance between the ellipsoids, making it quite impractical for collision detection.

Modelling true ellipsoids with high order polyhedra is somewhat impractical due to the scaling of the error with facet counts. An alternative to starting with a high density of facets, a method of iteratively refining the polyhedral model of the ellipsoid is introduced. While currently less efficient than the point to ellipse iteration, it shows much better linearity of computation time with distance, and has promise for future development.

1.4 Organization of Dissertation

This thesis is divided into four main chapters apart from this introduction; a chapter surveying the distance computation literature, a chapter detailing the DQP3 algorithm and its variants, a chapter comparing the DQP3 algorithm with its peers, and a chapter on the ellipsoid problem. Following these is a brief conclusion with suggestions for future work.

The second chapter, 'The Distance Problem in Robotics' presents a broad overview of the variety of distance computations between objects. The state of affairs in the robot control literature is first examined, followed by the work in convex models. Non-convex and dissociated models of objects are also looked at.

The third chapter, 'Fast Distance Computation using Quadratic Programming' details the quadratic programming (QP) formulation of the distance problem. The primal active set method for solving QP problems is described. It then goes on to find ways to enhance the computation of that method at various points, using the geometric nature of the distance problem. The result is the Distance Quadratic Programming, 3D-specific or DQP3 algorithm. A number of variants on the core algorithm are suggested to enhance its speed, and a method for handling the translation and rotation of object descriptions internally to the DQP3 algorithm is described.

The fourth chapter, 'Comparison of DQP3 to Other Algorithms' does just that. Cameron's Enhanced GJK [Cameron97a] is compiled on the same machine as DQP3 and the two are tested head to head. The results are mixed, with the DQP3 algorithm somewhat faster on the tracking problem, but slower elsewhere. Comparison to other algorithms in the field is undertaken by scaling their results by the Linpack benchmark for the computer in question.

The fifth chapter, 'Ellipsoid Distance Computation Methods' starts by providing the details of the ellipsoid distance problem, and what few references exist. A fast algorithm for point to ellipsoid distance computation is presented, followed by results for simply iterating that algorithm to solve the ellipsoid to ellipsoid problem. While the attempt to use DQP3 on high order polyhedral models of ellipsoids is rather ineffective, it does lead into an iterative algorithm that only adds facet coverage in the area needed. The iterative algorithm shows some promising results.

The thesis is rounded out with some conclusions and a long list of possible directions to take future work on the topic. An appendix of Matlab and C code implementing many of the algorithms described herein is also included.

2. The Distance Problem in Robotics

Any practical application of robotics must involve the robot interacting with its environment. In the process of doing so, obstacles in the workspace may be encountered, and at the very least it is undesirable for a manipulator arm to impact its target object with any part of its structure other than the end effector. In the complex environments seen in industrial robotics, this is an even more significant issue. As a result, various forms of modelling of obstacles have been applied to robotic control algorithms.

There are two views on the problem worth considering. The control theorists are interested in how the robot moves in the presence of obstacles, and are less concerned with the realism of those obstacles. This tends to lead to the use of simpler models, for ease of implementation and computation.

The collision detection and distance measure literature is more concerned with modelling the workspace and manipulator in as much detail as possible. Computational costs are an issue of course, but as the focus is on modelling, more complex models tend to be proposed and used. Recently, there has been a move to making the algorithms and software more accessible, through release of public domain software libraries. Lin [Lin96] lists several, and Cameron's work [Cameron97a] has been integrated into a full path planning package, OxSim [Cameron01].

This survey will attempt to sketch the types of distance computations in use in robotics control applications, then go on to a discussion of the various models in use in the collision detection literature. These are divided into the classes of convex methods, non-convex methods and dissociated models.

Some excellent survey papers on the subject of collision detection are Lin's work [Lin96, Lin98] and Jiménez's survey [Jiménez01].

2.1 Robotics Control Models

The author's original thesis [Fenger93] surveys the then current papers in the field of collision avoidance. A handful of more current results from the literature are also considered here.

The simplest of models seen in the literature is the use of spheres or circles. This reduces to measuring the distance between points, a trivial computation indeed. An example of this is Hopcroft's use of spheres [Hopcroft83] when looking at complex collections of objects.

Next we have line models, which are little more complex. Simple projective geometry can be used to find the closest point between a line and a point, and only more slightly complex computations are required to find the closest point between two lines. Line and point methods are used by Khadem [Khadem88], Cheng [Cheng93] and Fenger [Fenger93] for handling redundant manipulators, where the control issues become quite complex. Line to line methods are considered by Zchal [Zchal92] and Glass [Glass93] for handling a redundant manipulator working through a window-like opening.

A rather different sort of approach, coming closer to general geometry methods, is the use of potential fields. It models the workspace obstacles not as hard limits, but as hills in a potential field. Using these hills to dynamically 'push' the robot away from obstacles produces useful results. Kalaycioglu [Kalaycioglu93] uses these sorts of models, but only models the robot links as a series of points, not as discrete objects. McLean [McLean93] uses a similar method for handling a snakelike manipulator.

While general convex hulls are of interest at a theoretical level, few papers have tried to use them for actual control applications. McIntyre [McIntyre92] is one of these select few, actually implementing convex hull models for an industrial robot in an online control application. His update rate was only 352ms (on a 68030), however.

There is even a collision avoidance method that does not compute distances at all. Both Nakamura [Nakamura91] and Yoshikawa [Yoshikawa90] use a 'reference position'

selected by the machine operator, and attempt to stay as close to that configuration as possible.

More recent work such as Chuang's potential field paper [Chuang98] still use point (manipulator) to polyhedra (obstacle) distance measures. Seraji [Seraji99] implements a full robotic control system, but only on line to line models.

Ahuactzin and Gupta [Ahuactzin99] use polyhedral models of manipulator and workspace in a high-level path planner, as does Chen's SANDROS algorithm [Chen98]. Both consider running times in the minutes to be good, though it is difficult to determine how many distance computations are done in that time. The latter admits that it could use a better distance algorithm. Advances in convex hull computation speeds would be useful to both.

2.2 Convex Model Distance Algorithms

A convex object is one whose surface radius of curvature is always zero or positive, never negative. The points making up the object form a convex set. This is particularly useful to the minimum distance problem between convex objects, as it makes the problem a convex problem, amenable to directional search to find a global minimum.

The most common type of convex model used in distance computations is polyhedral, either defined as a set of polygonal facets, or as the convex hull of a collection of points. The seminal paper in the field was written by Gilbert [Gilbert88], the first of the truly efficient algorithms for the distance problem, using a convex hull model. Other contemporaries in the field include Bobrow [Bobrow89] and Ma and Nahon [Ma92], both working on polyhedra defined by the half-spaces of the facets. Another contemporary result, by Lin and Canny [Lin91], used a somewhat different vertex based approach. Lin and Canny were the first to use a database of adjacencies to accelerate the processing of the distance problem, and were the first to claim an O(1) time for what is known as the 'tracking problem', where the position calculation is repeated for small increments of relative motion between pairs of objects.

More recent work on convex polyhedra include Sato's [Sato96] and Cameron's [Cameron97a, Cameron97b] attempts to enhance the original Gilbert approach. Both also manage to achieve O(1) time on the tracking problem, using adjacency data.

Another interesting family of convex polyhedra distance computations creates a nested set of polyhedra of decreasing complexity. The biggest advantage of this is that when crossing large numbers of facets in a tracking type computation, it is possible to jump up to less complex levels to simplify the transition. The H-Tree work of Guibas [Guibas99] is one such example.

Another type of convex body is the ellipsoid. Rimon and Boyd [Rimon97] look at a distance-like measure between ellipsoids that can be computed more easily than the actual Cartesian distance. It is a useful measure in that it goes to zero as the ellipsoids touch. However, modelling non-ellipsoidal objects with ellipsoids is fairly ineffective due to the large error bounds, and ellipsoid models are thus only of real interest when there are objects fitting that description in the workspace.

2.3 Non-Convex Model Distance Algorithms

Convex is easy. Conversely, non-convex problems are rather harder. Unfortunately, the world is largely made up of things that are not convex. A variety of approaches have been used to deal with this issue.

The simplest approach would be to partition the non-convex object in question into a series of discrete convex parts, and test the distance from each of the parts to a target object. While conceptually simple it leads to large numbers of pairs of objects, each of which need to have their distance computed. Finding the optimal partitioning is a NP-hard problem [Bajaj92], but heuristic methods exist [Chazelle84].

Another partition based approach is *Constructive Solid Geometry* or CSG. This entails building objects out of geometric primitives such as rectangular prisms, spheres, cones and the like. While it is efficient at finding collision points [Cameron91], it is difficult to build a surface representation [Keyser97].

Parametric surface definitions are also used. The most popular is Non-Uniform Rational B-Splines or NURBS [Farin93], which have some nice properties in terms of smoothness and ease of rendering. One approach to handling NURBS defined surfaces is a hierarchical division into bounding volumes of high enough order to match their curvature [Krishnan98].

Hierarchical representations are a common approach to handling complex non-convex objects. Spheres are the simplest nodes to use, [Hubbard93, Quinlan94] as the distances between them are easy to compute. However, they do a poor job of mapping to general objects. Axis-aligned boxes are a slight improvement in terms of mappings, and are also easy to compute. The next logical step is oriented bounding boxes or OBB, which more efficiently tile arbitrary objects [Gottschalk96]. They have seen extensive use in the literature, initially in graphics applications [Avro89]. Higher order representations have been suggested [Klosowski96] but the OBB seems to be the most enduring.

Efficiently determining which subsets to test at any given time is the most critical aspect of these representations. They are more efficient at collision detection than finding a minimum distance, as the model only needs to be descended if two higher-level objects are in contact. Examples include Gottschalk's RAPID [Gottschalk96], Barequet's work [Barequet96] and I-COLLIDE [Lin96].

2.4 Dissociated Models

There are some models that have no physical coherence at all. Two such examples are so-called 'polygon soups', and voxels (discrete volume elements).

The polygon soup approaches simply take a database of arbitrary planar polygons, and tests each one for penetration by another. These polygons may represent arbitrarily shaped solid objects, or simply be free in space. The simples test for intersection between polygons determines whether the edge of one penetrates the area defined by the other [Canny87].

Voxels divided the workspace into discrete volume elements, typically of a fixed size. Fixed obstacles can be marked by the cells they occupy. Moving objects determine which cells they penetrate, and test those cells to see if any other static or moving objects occupy them [Overmars92]. The biggest problem is deciding on an efficient voxel size. It can work quite well for uniform objects [Turk89], and is amenable to parallelization and the use of hash tables for improved performance.

3. Fast Distance Computation using Quadratic Programming

3.1 Introduction

The distance problem is a central problem in the field of robotics. The distance problem has been solved in many ways, the most popular of which is Gilbert's method [Gilbert88] [Gilbert90]. It is not the solution of the problem that is of interest here, rather the speed with which it can be computed.

A number of approaches were listed by Gilbert, most of which were dismissed as impractical due to excessive attention to asymptotic performance. The Gilbert algorithm was comparable with other algorithms of its day, in that it ran in $O(n)$ time, where n is the number of facets of the two objects whose minimum separation is being computed. More recently, practical algorithms have been proposed with $O(\log n)$ time, and even some that with proper initialization and data storage can run in approximately constant time for tracking problems [Lin91, Sato96].

To get these results, an important assumption is made - that the distance between the objects is to be computed routinely, and that the objects are undergoing very small amounts of relative motion between the computations. Thus, on many evaluation cycles, the algorithm only needs to test whether or not the currently chosen points are still OK - which can be done in constant time.

Scalability, however, does not tell the whole story. Scalability is desirable, but the typical objects of interest in robotics applications are quite simple, and can be adequately modelled with a very small number of facets. For example, in [McIntyre92], a working robot control system using polyhedral distance computation, no object has more than 10 facets. Thus there is a need to solve small distance problems very quickly - the robot is represented by 14 such polyhedra, and each must be tested against the obstacles in the workspace. In the work in question, 25 polyhedra distance comparisons per cycle were deemed necessary, and the cycle time on a 68030 CPU was 352ms. Given that the Reis

robot they were controlling could move 2.5 meters in this time, a faster computation method is obviously needed, although modern CPUs also help.

Increases in processing power can make up some of the performance required for online collision avoidance and path planning tasks. However, what is needed is an algorithm with modest scaling, and a low constant cost.

The quadratic programming approach to distance calculation has seen little interest since 1992 [Ma92,Zeghloul92]. In this thesis, the almost-forgotten QP approach is revived, given a new coat of paint, and a thorough tune-up. Several accelerated methods for calculating core elements the QP problem are presented, along with timing results showing the performance of the new algorithm.

3.2 Quadratic Programming Formulation of the Distance Problem

In general, the class of problems known as quadratic programming (QP) problems are attempts to find the minimum of a quadratic function of some vector $\mathbf{x}$ subject to a set of linear constraints. In other words, to solve:

$$\underset{\mathbf{x}}{\text{minimize}} \quad F(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{x}^T\mathbf{q} \qquad (3.2.1)$$

$$\text{subject to} \quad \mathbf{a}_j^T\mathbf{x} - b_j \geq 0,\ j \in I \qquad (3.2.2)$$

where I is the set of inequality constraints. Formulations where equality constraints are allowed also exist, but are not required here.

The problem of finding the distance between two convex polyhedra may easily be formulated as a QP problem, in the following manner. The polyhedra may be defined as a set of linear constraints, each dividing the space (here presumed to be $\mathbb{R}^3$) into half-spaces, each half-space representing one facet of the polyhedra:

$$\mathbf{A}_i^T\mathbf{x}_i - \mathbf{b}_i \geq 0, i \in \{1,2\} \qquad (3.2.3)$$

where the columns of the $3\times n$ matrix $\mathbf{A}_i$ are the vectors defining the dividing plane of the half-space, the elements of the column vector $\mathbf{b}_i$ are the distances of the boundaries from the origin, measured normal to the plane, and i is the object number.

One set of these constraints defines each object, providing a nicely partitioned set of constraints on

$$\mathbf{x}=\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix} \tag{3.2.4}$$

These constraints can be expressed in a form fitting the QP problem

$$\begin{bmatrix}\mathbf{A}_1 & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_2\end{bmatrix}^T\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix}-\begin{bmatrix}\mathbf{b}_1\\ \mathbf{b}_2\end{bmatrix}\geq\mathbf{0} \tag{3.2.5}$$

One appropriate cost function is the square of the distance measure between the points $\mathbf{x}_1$ and $\mathbf{x}_2$, vis:

$$F(\mathbf{x})=\|\mathbf{x}_1-\mathbf{x}_2\|^2 \tag{3.2.6}$$

This may also be written as

$$F(\mathbf{x})=(\mathbf{x}_1-\mathbf{x}_2)^T(\mathbf{x}_1-\mathbf{x}_2) \tag{3.2.7}$$

or

$$F(\mathbf{x})=\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix}^T\begin{bmatrix}\mathbf{I} & -\mathbf{I}\\ -\mathbf{I} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix} \tag{3.2.8}$$

where the 4-block matrix is the Hessian matrix of the problem, H.

The structure of the Hessian is quite regular, and suggests that there will be more efficient methods of approaching this problem than using a general-purpose QP algorithm.

3.2.1 Numerical Stability

The Hessian of the distance QP problem exhibits characteristics that make it quite unsuitable for most conventional QP approaches.

In particular, the matrix is not *positive definite* (PD) - it is only *positive semidefinite* (PSD). While QP algorithms that handle PSD matrices exist, they still have difficulty with the problem above, as fully half of the singular values of the Hessian are zero.

None of the QP algorithms tested in 1995 (from the Matlab toolkit, mostly) worked when no constraints were active - by which it is meant that the inequality constraint in (3.2.3) is exactly satisfied. All of the algorithms required that at least a few, typically 3, of the constraints to be active (exactly equal, allowing them to be treated as equality constraints). This would allow the algorithm to use those constraints to form a stable extended problem.

In general, this restriction would not be a problem - it could be solved by picking a starting x that satisfies at least 3 constraints. However, efficiency again rears its ugly head - if the point is picked badly, it will take many more iterations to converge on complex objects. It turns out that for speed in a situation where a 'best starting point' is not known and cannot be guessed at, it is better to start at an internal point - one where no constraints are at equality. This result will be discussed later.

More modern approaches, such as those found in v5.3 of Matlab (1999) are significantly more robust, and both active set (qp) and interior point (quadprog) methods solve the zero initial constraint problem without difficulty.

3.2.2 Facets vs. Vertices

Gilbert's seminal work [Gilbert88] established a method of distance computation for convex polyhedra that has formed the basis of a decade's research. The Gilbert method uses the vertices of the object as its basic working elements. In contrast, the QP based approach suggested above uses the facets of the object as its basic working element.

It is worth examining the difference in computational cost between the two for typical operations, and the variations in working element count required to represent a given object. Overall, the results are fairly similar.

Other than the computational cost of the algorithm itself, which is addressed below, there are costs associated with rotating and translating the objects under consideration.

For rotation by a rotation matrix T:

- Vertex: each vertex has to be multiplied by T: 9 multiply, 6 add
- Facet: each facet vector has to be multiplied by T: 9 multiply, 6 add. The offset distances ($\mathbf{b}_f$ in (3.2.3)) do not change under rotation around the origin.

For displacement by a vector **d**:

- Vertex: each vertex adds **d**: 3 add
- Facet: each offset distance adds the inner product of **d** and its facet vector: 3 multiply, 3 add.

The computational cost for facets is higher for displacement, although this is somewhat offset by the fact that modern processors are capable of running floating point multiplication and addition in parallel.

Additionally, the costs above may be completely ignored if the translation and rotation of the objects are incorporated directly into the algorithm. Numerical results below suggest that the cost involved is minimal.

More significantly, choice of working element can affect the number of elements required to represent given objects. Two cases are considered here - a sphere, and a typical robot arm.

A sphere can be approximated by a set of evenly spaced polyhedra, in the manner of a soccer ball. The simplest form is to tile the sphere with triangles like a geodesic dome [Fuller75]. An evenly spaced set of facets on the sphere produces the former, while an

evenly spaced set of vertices produces the latter. If the number of elements is sufficiently large, the facets become hexagons. For a matching number of vertices the resulting triangles have approximately 1/3 the area of the hexagons. So, for an equal number of elements, vertices represent a sphere better than facets. Alternately, one can see that the average number of triangles meeting at each vertex in the triangularization is between 5 and 6 (from the example of the icosahedron), thus there will be almost twice as many facets as vertices in that representation. Either way, vertex representation is more efficient.

However, for a typical robot arm (Figure 3.1), the situation is reversed. The manipulator arm has two large facets on its sides, and a number of small rectangular facets to provide the curved profile of the edges. If the curved edge is represented in N facets, the total number of facets of the figure is N+2, while the vertex count is $2N$. Given the prevalence of similar figures in robotics, it appears that facet based methods are not unreasonable.

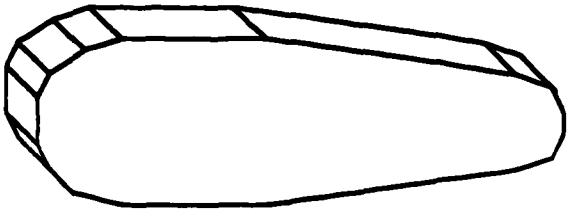


Figure 3.1 – 3D view of a typical robot arm

Another notable use of facets is in fitting an arbitrary convex object with smooth curves instead of obvious facets. When picking vertices for modelling the object, one must be careful to select them so that the facets so defined do not intersect the object. However, facets used to model the object can be defined by picking a nicely distributed set of points over its surface, and taking the normal planes at these points. The only problem is in deciding what a nice set of points is.

While the author cannot claim that the use of facets is superior to the use of vertices as working elements, for typical robotics tasks they are comparable.

3.3 The Primal Active Set QP algorithm

One common solution for QP problems is the *primal active set* method. [Fletcher87]

The primal active set method divides the constraints into two sets:

- Active constraints - these constraints are exactly satisfied, i.e.

$$\mathbf{a}_j^T\mathbf{x}-b_j=0 \qquad (3.3.1)$$

  if constraint j is an active constraint. Here, $\mathbf{a}_j$ is the j^{th} column of the constraint matrix **A**, and b_j the j^{th} entry of **b**.

- Inactive constraints - constraints that are more than satisfied, i.e.

$$\mathbf{a}_j^T\mathbf{x}-b_j>0. \qquad (3.3.2)$$

These two sets (the Active Set and the Inactive Set) are updated as the Primal Active Set algorithm iterates to an optimal solution. Generally speaking, the algorithm progresses by finding the optimal solution given the active constraints, then either adding or removing constraints as necessary to improve the overall solution.

3.3.1 Computations of the Primal Active Set Algorithm

3.3.1.1 Delta

As the algorithm progresses the active set changes as constraints are added and subtracted. At the core of the algorithm is the solution of the Equality problem,

$$\underset{\mathbf{x}}{\text{minimize}} \quad F(\mathbf{x})=\tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x}+\mathbf{x}^T\mathbf{q} \qquad (3.3.3)$$

$$\text{subject to} \quad \mathbf{a}_j^T\mathbf{x}-b_j=0,\ j\in\mathcal{A} \qquad (3.3.4)$$

where $\mathcal{A}$ is the active set.

On any particular iteration k with interim solution $\mathbf{x}^{(k)}$ this may be further simplified to the calculation of δ, such that $\mathbf{x}^{(k)}+\delta$ is the solution to problem (3.3.3-4):

$$\underset{\delta}{\text{minimize}} \quad F(\delta)=\tfrac{1}{2}\delta^T\mathbf{H}\delta+\delta^T\mathbf{q}^{(k)} \tag{3.3.5}$$

$$\text{subject to} \quad \mathbf{a}_j^T\delta=0,\ j\in\mathcal{A} \tag{3.3.6}$$

where $\mathbf{q}^{(k)}=\mathbf{H}\mathbf{x}^{(k)}+\mathbf{q}$, the gradient of $F(\mathbf{x})$ at $\mathbf{x}^{(k)}$ from (3.3.3) above.

Solving this QP problem with equality constraints is routine. Two feasible approaches include a generalized elimination of the constraints and a Lagrangian approach, with the latter being the most useful [Antoniou02, Fletcher87].

The equality QP problem from (3.3.5-6) has a Lagrangian function of

$$\mathcal{L}(\delta,\lambda)=\tfrac{1}{2}\delta^T\mathbf{H}\delta+\delta^T\mathbf{q}^{(k)}-\lambda^T\mathbf{A}_{act}\delta \tag{3.3.7}$$

where λ is a column vector of Lagrange multipliers and $\mathbf{A}_{act}$ is the set of active constraints from $\mathbf{A}$.

Applying the stationary point condition yields

$$\begin{aligned} \nabla_\delta\mathcal{L}=0:&\quad \mathbf{H}\delta+\mathbf{q}^{(k)}-\mathbf{A}_{act}\lambda=0 \\ \nabla_\lambda\mathcal{L}=0:&\quad \mathbf{A}_{act}^T\delta=0 \end{aligned} \tag{3.3.8}$$

This can also be expressed as the linear system

$$\begin{bmatrix} \mathbf{H} & -\mathbf{A}_{act} \\ -\mathbf{A}_{act}^T & \mathbf{0} \end{bmatrix}\begin{bmatrix} \delta \\ \lambda \end{bmatrix}=-\begin{bmatrix} \mathbf{q}^{(k)} \\ \mathbf{0} \end{bmatrix} \tag{3.3.9}$$

The coefficient matrix is symmetric but not positive definite. A variety of techniques are available to solve this problem, and will not be detailed here. Suffice to say that the computation is not cheap.

3.3.1.2 Lambda

Removing a constraint requires some measure of how 'active' a given constraint is. It turns out that when the equation (3.3.5-6) is solved by $\delta = 0$, the Lagrange multipliers of the problem may be used for this purpose, in addition to their function in converting a constrained problem to an unconstrained one.

There will be as many Lagrangian multipliers in λ as there are active constraints, and each one matches an entry in $\mathcal{A}$. Any negative Lagrangian multiplier indicates a constraint where the search direction towards the solution is away from the constraint, and it is among this subset of the active constraints that we must choose one to remove. The constraint with the most negative associated Lagrangian multiplier is the conventional choice, and tends to work quite well [Fletcher87].

While it might seem efficient to remove more than one constraint at a time [Goldfarb72] it can increase the chance of causing the algorithm to *cycle* by returning to a previous active set in the sequence.

3.3.1.3 Alpha

Adding a new constraint requires that all of the inactive constraints be examined to see which one provides the greatest restriction on the solution of the current active set problem, $\delta^{(k)}$.

The new value of $\mathbf{x}$ found by the k^{th} step is:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k+1)} + \alpha^{(k)} \delta^{(k)} \qquad (3.3.10)$$

The $\alpha^{(k)}$ term is found by treating $\delta^{(k)}$ term as a search vector and finding the constraint which it reaches first:

$$\alpha^{(k)} = \min_{j \notin \mathcal{A}} \frac{\left(b_j - \mathbf{a}_j^T \mathbf{x}^{(k)}\right)}{\alpha_j^{(k)}} \qquad (3.3.11)$$

where $\alpha_j^{(k)}$ is

$$\alpha_j^{(k)} = \mathbf{a}_j^T \delta^{(k)} \qquad (3.3.12)$$

Hence, the search for an additional constraint will be referred to as the 'alpha' step. Note that equation (3.3.11) only needs to be computed if α_j is less than zero.

3.3.2 Steps of the Primal Active Set Algorithm

Fletcher [Fletcher87] describes the primal active set method as follows, starting with an initial feasible point $\mathbf{x}^{(1)}$ and its corresponding active set $\mathcal{A}^{(1)}$:

1) Given $\mathbf{x}^{(1)}$ and $\mathcal{A}^{(1)}$, set $k = 1$.
2) If $\delta = 0$ does not solve (3.3.5-6) go to step 4.
3) Compute Lagrange multipliers $\lambda^{(k)}$ and find the minimum, $\lambda^{(k)}_{min}$; if $\lambda^{(k)}_{min} \geq 0$ terminate with $\mathbf{x}^* = \mathbf{x}^{(k)}$, otherwise remove the matching constraint from $\mathcal{A}$.
4) Solve (3.3.5-6) for $\delta^{(k)}$.
5) Find $\alpha^{(k)}$ that solves (3.3.11) and if $\alpha^{(k)} > 1$ set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \delta^{(k)}$, else set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta^{(k)}$.
6) If $\alpha^{(k)} < 1$, add the corresponding constraint to $\mathcal{A}$.
7) Set $k = k + 1$ and go to step 2.

3.4 Problem-specific QP Algorithm for the 3D Distance Problem

3.4.1 Sources of Computational Cost

The computational cost of an algorithm can be viewed as the product of two terms, the computational cost per iteration, and the number of iterations. The latter tends to be difficult to improve without dramatic changes in the algorithm, leaving the former as the easiest to address. In the case of the primal active set method for solving a QP problem, the 'delta' computation is the dominant per-iteration cost for small numbers of constraints, and the 'alpha' computation is dominant for large numbers. For robotics control applications we are interested in simple object representations, so the initial focus will be on the 'delta' computation.

There are some structural changes to the QP algorithm itself that are worth considering. Recall that the distance problem formulation from Section 3.2 has a large amount of zeros that may be exploited. The key elements of the QP problem **H**, **q**, **A** and **b** are:

$$\mathbf{H}=\begin{bmatrix}\mathbf{I} & -\mathbf{I}\\ -\mathbf{I} & \mathbf{I}\end{bmatrix}, \mathbf{q}=\mathbf{0}, \mathbf{A}=\begin{bmatrix}\mathbf{A}_1 & \mathbf{0}\\ \mathbf{0} & \mathbf{A}_2\end{bmatrix}, \mathbf{b}=\begin{bmatrix}\mathbf{b}_1\\ \mathbf{b}_2\end{bmatrix} \qquad (3.4.1)$$

Simply plugging these values into standard QP problem will produce an answer, but it will not do so quickly. Each zero and each one will be computed as if it were a full precision floating point value, wasting countless *flops* (floating point operations).

The easiest way to save a few flops is to partition the problem, treating the QP problem as two separate problems, one on $\mathbf{x}_1$ and one on $\mathbf{x}_2$, as much as possible. The active set for each problem is maintained separately. When computing the 'alpha' phase of the primal active set method, determining which constraints to add on $\mathbf{x}_1$ and $\mathbf{x}_2$ separately cuts the computational cost in half, due to all the zeros that aren't being treated as necessary computations. In addition, since $\alpha^{(k)}$ is determined separately for each object, two constraints will generally get added each 'alpha' cycle rather than one, one to each object. This can yield a substantial reduction in the number of iterations required.

To fully take advantage of the improvement in total iterations caused by treating the two objects as two mostly-separate QP problems, it also helps to release one constraint from each object during the 'lambda' phase, if both have constraints with some $\lambda_j < 0$. While the standard texts suggest that the gains will be small [Fletcher87] and come at the risk of cycling, empirical evidence shows both significant gains in total iteration count, and no cases of cycling. For extra insurance, a small ε (10^{-12}) tolerance was added to the 'alpha' test, so constraints were only added if $\alpha_{min} < 1-\varepsilon$.

To make the partition of the problem into one QP problem per object really pay off, a method to compute the 'delta' problem is needed.

$$\underset{\delta = \begin{bmatrix}\delta_1 \\ \delta_2\end{bmatrix}}{\text{minimize}} \quad F(\delta) = \left\| (\mathbf{x}_1^{(k-1)} + \delta_1) - (\mathbf{x}_2^{(k-1)} + \delta_2) \right\|^2 \tag{3.4.2}$$

$$\text{subject to} \quad \begin{aligned} \mathbf{A}_{1act}^T \delta_1 &= \mathbf{0} \\ \mathbf{A}_{2act}^T \delta_2 &= \mathbf{0} \end{aligned} \tag{3.4.3}$$

where $\mathbf{A}_{1act}$ and $\mathbf{A}_{2act}$ are arrays of the constraint vectors on objects 1 and 2 respectively.

$$\begin{aligned} \mathbf{A}_{1act} &= \left[\mathbf{a}_{1j_1} \quad \cdots \quad \mathbf{a}_{1j_n} \right] j_1 \cdots j_n \in \mathcal{A}_1 \\ \mathbf{A}_{2act} &= \left[\mathbf{a}_{2j_1} \quad \cdots \quad \mathbf{a}_{2j_m} \right] j_1 \cdots j_m \in \mathcal{A}_2 \end{aligned} \tag{3.4.4}$$

where $\mathcal{A}_1$ is the active set of object 1, and $\mathcal{A}_2$ is the active set of object 2.

Solving equation (3.4.2-3) can be seen as solving the closest-point problem between elementary geometric figures such as infinite lines and infinite planes.

3.4.2 Physical Interpretation of the Distance Problem

Bobrow [Bobrow89] uses a very similar object representation to the one suggested in Section 3.3. He used a direct minimization approach that can be seen as:

1) Start with two points, x_1 and x_2, one on the surface of each object.
2) Determine the active constraints at x_1 and x_2.
3) Test for completion.
4) Compute new search direction by projecting $v = x_1 - x_2$ onto the most appropriate facets of the objects (as determined by the constraints and the Kuhn-Tucker conditions).
5) Find minimum along these lines, limit x_1 and x_2 by the constraints, go to step 2.

The problem with Bobrow's approach is that it will tend to zigzag across a facet, rather than proceeding directly to the minimum point. If the search starts from a point that is not on the projection of the edge onto the facet, and the projection of v onto the facet is not perpendicular to the edge projection, the resulting point will also not be on the edge projection, nor will it be at the true minimum point – where the vertex would pierce the facet if both were infinite. Each iteration will repeat this error, only slowly converging to the correct point. This is illustrated in figure 3.2.

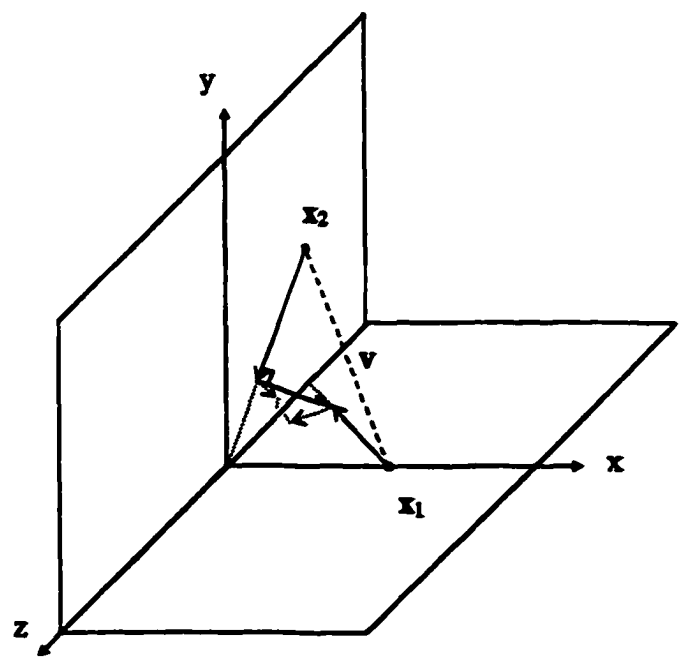


Figure 3.2 – Convergence of Bobrow method on facet-facet case

Essentially, the weakness of Bobrow's approach is its generality. The solution for any combination of facet, edge and vertex is treated the same way – reduce to lines in the 'most active' facet, then solve that problem. The DQP3 (Distance via Quadratic Programming, 3D-specific) method approaches the problem with less generality, treating each combination as a specific sub-case and solving that sub-case directly, as an individual geometric problem. The change in x_1 and x_2 required to satisfy this closest point problem for the sub-case in question is exactly the δ required to solve the 'delta' step of the QP primal active set method.

The cases can be enumerated by considering the number of possible constraints active on each object. We are dealing with at most 3 dimensional objects, so at most 3 constraints may be treated as active at any one time without causing degeneracy. This corresponds to one of the vertices of the object. At the other extreme, the current x_i may not be on the surface of the object at all, and zero constraints will be active.

	Object 2 constraints			
Object 1 constraints	0	1	2	3
0	space-space	space-facet	space-edge	space-vertex
1	facet-space	facet-facet	facet-edge	facet-vertex
2	edge-space	edge-facet	edge-edge	edge-vertex
3	vertex-space	vertex-facet	vertex-edge	vertex-vertex

Table 3.1 – Combinations of constraints

Discarding symmetric cases, 10 cases need to be dealt with, and all but facet-facet, facet-edge and edge-edge are trivial.

All three of the above 'hard' cases will occur on a typical run with starting points in the middle, even if preprocessing is applied.

Note that for numerical stability, there is a ε tolerance allowed on measures of how parallel the facets or edges are.

Once the closest point for the two geometric objects has been found, δ can be computed as:

$$\boldsymbol{\delta} = \begin{bmatrix} \Delta \mathbf{x}_1 \\ \Delta \mathbf{x}_2 \end{bmatrix} \tag{3.4.5}$$

3.4.2.1 Trivial Cases

Many of the cases have trivial solutions. For example, if no constraints are active on any side, any arbitrary point in $\Re^3$ may be chosen as a solution to the closest point problem. In cases where an infinity of valid solutions exist, it becomes necessary to choose among them. As a rule of thumb, a point halfway between $\mathbf{x}_1$ and $\mathbf{x}_2$ is a good position to choose – it tends to produce the minimum δ.

- Vertex-vertex: No motion, $\boldsymbol{\delta} = \mathbf{0}$.
- Vertex-space: Move the unconstrained $\mathbf{x}_2$ to the constrained $\mathbf{x}_1$.
- Vertex-facet/edge: Project the vertex constrained $\mathbf{x}_1$ onto the facet or edge, move $\mathbf{x}_2$ to that position.
- Space-space: Move both $\mathbf{x}_1$ and $\mathbf{x}_2$ to the halfway point between both of them,

$$\mathbf{x}_1 = \mathbf{x}_2 = \tfrac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2) \tag{3.4.6}$$

- Space-facet/edge: Project the free $\mathbf{x}_1$ onto the facet or edge, move both $\mathbf{x}_1$ and $\mathbf{x}_2$ to that position.

To project $\mathbf{x}_1$ onto $\mathbf{x}_2$'s facet (defined by $\mathbf{x}_2$ and one constraint, $\mathbf{a}_{21}$), move along $\mathbf{a}_{21}$ to reach the plane:

$$\mathbf{x}_p = \mathbf{x}_1 + \mathbf{a}_{21}\left((\mathbf{x}_2 - \mathbf{x}_1)^T \mathbf{a}_{21}\right) \tag{3.4.7}$$

To project $\mathbf{x}_1$ onto $\mathbf{x}_2$'s edge (defined by $\mathbf{x}_2$ and two constraints, $\mathbf{a}_{21}$ and $\mathbf{a}_{22}$), first generate an orthonormal pair $\mathbf{a}_{2p}$, $\mathbf{a}_{22}$:

$$\mathbf{a}_{2p} = \frac{\mathbf{a}_{21} - \mathbf{a}_{22}\left(\mathbf{a}_{22}^T \mathbf{a}_{21}\right)}{\left\|\mathbf{a}_{21} - \mathbf{a}_{22}\left(\mathbf{a}_{22}^T \mathbf{a}_{21}\right)\right\|} \tag{3.4.8}$$

With an orthonormal pair of vectors defining the $\mathbf{x}_2$ edge, we can now project from $\mathbf{x}_1$ to the edge by taking the projections onto each of the so defined planes in turn.

$$\mathbf{x}_p = \mathbf{x}_1 + \mathbf{a}_{2p}\left(\left(\mathbf{x}_2 - \mathbf{x}_1\right)^T \mathbf{a}_{2p}\right) + \mathbf{a}_{22}\left(\left(\mathbf{x}_2 - \mathbf{x}_1\right)^T \mathbf{a}_{22}\right) \tag{3.4.9}$$

3.4.2.2 The Facet-Facet Case

If the facets are parallel, follow the half-way rule of thumb. In this case, move from $\mathbf{x}_1$ halfway to the projection of $\mathbf{x}_2$ onto $\mathbf{x}_1$'s plane, and then project this new point onto $\mathbf{x}_2$'s plane to get the new value of $\mathbf{x}_2$. See equation (3.4.7).

In the non-parallel case, let $\mathbf{a}_{11}$ be the active constraint on object 1, and $\mathbf{a}_{12}$ be the active constraint on object 2. The components of each constraint in the opposite object's plane are:

$$\begin{aligned} \mathbf{a}_{1p} &= \mathbf{a}_{11} - \mathbf{a}_{21}\left(\mathbf{a}_{11}^T \mathbf{a}_{21}\right) \\ \mathbf{a}_{2p} &= \mathbf{a}_{21} - \mathbf{a}_{11}\left(\mathbf{a}_{21}^T \mathbf{a}_{11}\right) \end{aligned} \tag{3.4.10}$$

Renormalization of these values is only required if they are small enough that numerical stability becomes an issue. As the two planes are not parallel, they intersect at a line. Projecting $\mathbf{x}_1$ along $\mathbf{a}_{2p}$ to the $\mathbf{x}_2$ plane produces a point on the intersection.

$$\begin{aligned} \mathbf{x}_{1p} &= \mathbf{x}_1 + \mathbf{a}_{2p}\left(\frac{\left(\mathbf{x}_2 - \mathbf{x}_1\right)^T \mathbf{a}_{21}}{\mathbf{a}_{2p}^T \mathbf{a}_{21}}\right) \\ \mathbf{x}_{2p} &= \mathbf{x}_2 + \mathbf{a}_{1p}\left(\frac{\left(\mathbf{x}_1 - \mathbf{x}_2\right)^T \mathbf{a}_{11}}{\mathbf{a}_{1p}^T \mathbf{a}_{11}}\right) \end{aligned} \tag{3.4.11}$$

The $1/\left(\mathbf{a}_{2p}^{T}\mathbf{a}_{21}\right)$ term is only required if $\mathbf{a}_{2p}$ is not renormalized to produce an orthonormal pair. Doing this saves the cost of computing a square root.

The final result can be any value on the line defined by $\mathbf{x}_{1p}$ and $\mathbf{x}_{2p}$. Using the half-way rule of thumb we split the difference between the two, yielding:

$$\boldsymbol{\delta} = \begin{bmatrix} \frac{1}{2}\left(\mathbf{x}_{1p} + \mathbf{x}_{2p}\right) - \mathbf{x}_1 \\ \frac{1}{2}\left(\mathbf{x}_{1p} + \mathbf{x}_{2p}\right) - \mathbf{x}_2 \end{bmatrix} \tag{3.4.12}$$

Figure 3.3 illustrates the DQP3 method for handling the same facet-facet case as was used in figure 3.2, for Bobrow's solution.

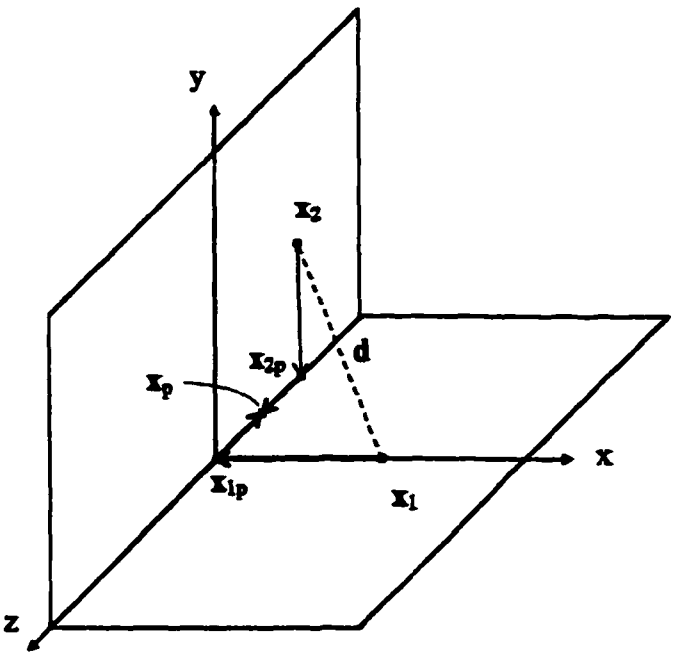


Figure 3.3 – Convergence of DQP3 method on facet-facet case

3.4.2.3 The Facet-Edge Case

If the $\mathbf{x}_2$ edge is parallel to the $\mathbf{x}_1$ facet, project $\mathbf{x}_1$ onto the edge, and move $\mathbf{x}_2$ halfway to this projected point. See equations (3.4.8-9). Then project this new point into $\mathbf{x}_1$'s plane to find the new value of $\mathbf{x}_1$.

In the non-parallel case the line and plane will intersect. Let $\mathbf{a}_{11}$ be the constraint on object 1, and let $\mathbf{a}_{21}$ and $\mathbf{a}_{22}$ be the constraints on object 2. The $\mathbf{x}_2$ line can be computed as the cross product of $\mathbf{a}_{21}$ and $\mathbf{a}_{22}$.

$$\mathbf{l}_2 = \mathbf{a}_{21} \times \mathbf{a}_{22} \tag{3.4.13}$$

The solution is now simply a matter of projecting $\mathbf{x}_2$ along $\mathbf{l}_2$ to the $\mathbf{x}_1$ plane.

$$\mathbf{x}_{2p} = \mathbf{x}_2 + \mathbf{l}_2 \left(\frac{(\mathbf{x}_1 - \mathbf{x}_2)^T \mathbf{a}_{11}}{\mathbf{l}_2^T \mathbf{a}_{11}} \right) \tag{3.4.14}$$

Move both $\mathbf{x}_1$ and $\mathbf{x}_2$ to $\mathbf{x}_{2p}$ to get the final positions at the intersection point.

$$\delta = \begin{bmatrix} \mathbf{x}_{2p} - \mathbf{x}_1 \\ \mathbf{x}_{2p} - \mathbf{x}_2 \end{bmatrix} \tag{3.4.15}$$

Figure 3.4 illustrates the DQP3 method for handling the facet-edge case.

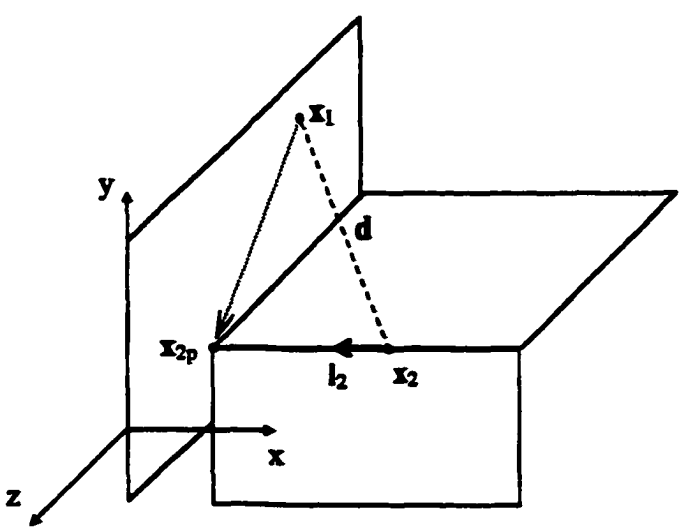


Figure 3.4 – Convergence of DQP3 method on facet-edge case

3.4.2.4 The Edge-Edge Case

If the edges are parallel, move x_1 to halfway to the projection of x_2 onto x_1's line, then project this new point onto x_2's line to get the new value of x_2. See equations (3.4.8-9) for these projections.

This is the most interesting case, in that no intersection may be assumed. Let a_{11} and a_{12} be the constraints on object 1, and let a_{21} and a_{22} be the constraints on object 2. The first step is to create an orthonormal pair defining the plane perpendicular to the x_1 line, a_{1p} and a_{12}.

$$a_{1p} = \frac{a_{11} - a_{12}\left(a_{11}^T a_{12}\right)}{\left\| a_{11} - a_{12}\left(a_{11}^T a_{12}\right) \right\|} \tag{3.4.16}$$

As in the facet-edge case, we can compute the x_2 line direction l_2 as the cross product of a_{21} and a_{22}. Projecting l_2 into the orthonormal coordinates $[a_{1p}\ a_{12}]$ produces

$$l_{2p} = \begin{bmatrix} a_{1p} & a_{12} \end{bmatrix}^T l_2 \tag{3.4.17}$$

Similarly, we can project $(x_2 - x_1)$ onto the $[a_{1p}\ a_{12}]$ plane, producing

$$d_p = \begin{bmatrix} a_{1p} & a_{12} \end{bmatrix}^T (x_2 - x_1) \tag{3.4.18}$$

Taking the component of d_p that is perpendicular to l_{2p}, we arrive at a solution for a new x_2 (which we'll call x_{2cp} in this projection, and x_{2c} on the original x_2 line). The unit vector perpendicular to l_{2p} is

$$l_{2i} = \frac{\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} l_{2p}}{\left\| l_{2p} \right\|} \tag{3.4.19}$$

Taking the component of d_p parallel to l_{2i} yields

$$x_{2cp} = l_{2i}\left(l_{2i}^T d_p\right) \tag{3.4.20}$$

To return this solution to the original frame of reference, it is possible to determine what distance along $\mathbf{l}_2$ corresponds to this solution, by scaling the distance from $\mathbf{x}_2$'s projection ($\mathbf{d}_p$) to $\mathbf{x}_{2cp}$ appropriately

$$t = \frac{\mathbf{l}_{2p}(\mathbf{x}_{2cp} - \mathbf{d}_p)}{\mathbf{l}_{2p}^T \mathbf{l}_{2p}} \tag{3.4.21}$$

Thus the closest position on object 2 to object 1 is

$$\mathbf{x}_{2p} = t\mathbf{l}_2 \tag{3.4.22}$$

The new value for $\mathbf{x}_1$ is simply the projection of this point onto the object 1 line, using the technique from equations (3.4.8-9).

Figure 3.5 illustrates the DQP3 method for handling the edge-edge case. Note that while $\mathbf{x}_{2p}$ falls on one of the object 1 facets, this is only due to the simple example shown (all planes are orthogonal to the x, y or z axis), and is not generally the case.

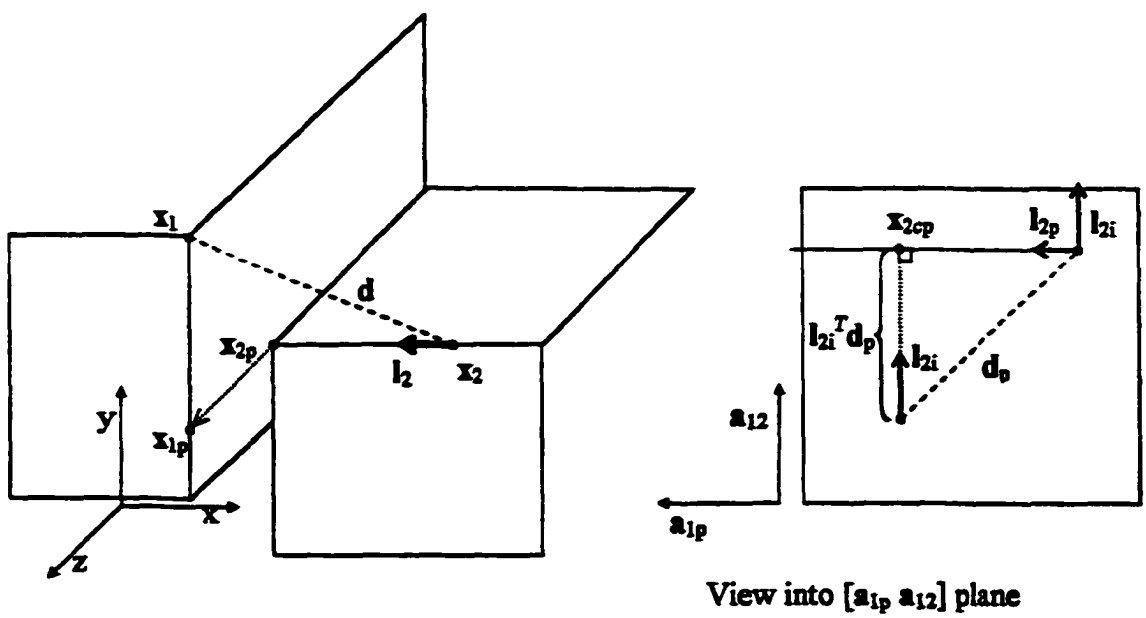


Figure 3.5 – Convergence of DQP3 method on edge-edge case

3.4.2.5 Computational Cost of Delta Step

The computation cost of the projective method of solving for δ varies depending on the number of constraints active on each object. Table 3.2 displays the cost of the various stages, using Matlab flops, based on the implementation 'dsolv.m' in Appendix A.

	Object 2 constraints			
Object 1 constraints	0 (space)	1 (facet)	2 (edge)	3 (vertex)
0 (space)	9	18	57	3
1 (facet)	18	51 / 87	89 / 39	15
2 (edge)	57	89 / 39	88 / 125	54
3 (vertex)	3	15	54	3

Table 3.2 – Computational Cost of Delta Step

Note that the entries with two values indicate the parallel and non-parallel cases, where the cases can occur. The parallel case is shown first. It is interesting to note that in the facet-edge case, the parallel case is more expensive to compute. Luckily, the parallel case is rare in general problems. For the tracking problem edge-vertex and vertex-vertex are the most common cases by far.

3.4.3 Calculating Lambda

The main drawback of using the above method of computing the 'delta' step of the QP algorithm is that there is no 'lambda' data provided. As a result, when it comes time to remove constraints, an analogue must be provided. What is needed is a physical criterion for when a constraint should be removed.

Lin [Lin91] addresses a similar consideration with her applicability criterion for the vertex-vertex, vertex-edge and vertex-face cases, defining a region around the specified feature such that if a point is inside that region, it is a closest point, and the search can terminate. Her method uses knowledge of the extent of the edge or facet, information that is not available to the DQP3 algorithm. As a result, some of the planes constraining

her applicability regions are not available. The general approach of taking each case separately is, however, a good one.

3.4.3.1 Lambda for a Facet

If there is only one constraint active on object 1, the test is simply whether or not $\mathbf{x}_2$ is outside the facet defined by constraint $\mathbf{a}_{11}$.

$$\lambda_1 = -\mathbf{a}_{11}^T(\mathbf{x}_2 - \mathbf{x}_1) \tag{3.4.23}$$

3.4.3.2 Lambda for an Edge

If there are two active constraints, then the edge on object 1 is defined by point $\mathbf{x}_1$ and constraints $\mathbf{a}_{11}$ and $\mathbf{a}_{12}$. We wish to release constraint $\mathbf{a}_{11}$ if the vector $\mathbf{d}=\mathbf{x}_2-\mathbf{x}_1$ when projected into the $\mathbf{a}_{12}$ plane points away from the edge. Thus,

$$\lambda_1 = -\mathbf{a}_{11}^T\mathbf{d} + \left(\mathbf{a}_{12}^T\mathbf{d}\right)\left(\mathbf{a}_{11}^T\mathbf{a}_{12}\right) \tag{3.4.24}$$

The test for $\mathbf{a}_{12}$ proceeds similarly. Figure 3.6 illustrates a case where λ_1 is negative, and shows the boundary plane past which removing $\mathbf{a}_{11}$ becomes necessary.

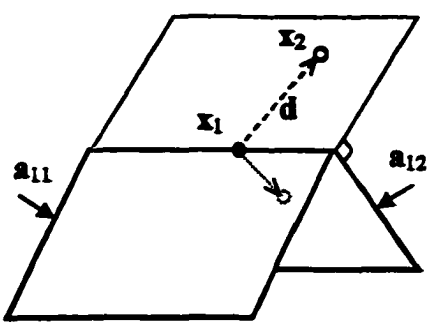


Figure 3.6 – Boundary Condition for Lambda, Edge Case

3.4.3.3 Lambda for an Vertex

When three constraints are active, the constraint release condition for $\mathbf{a}_{11}$ depends on the direction of the edge defined by the other two constraints, $\mathbf{a}_{12}$ and $\mathbf{a}_{13}$. If the vector $\mathbf{d} = \mathbf{x}_2-\mathbf{x}_1$ projected onto this line points away from $\mathbf{x}_1$, then lambda must be negative.

$$\lambda_1 = \frac{-\mathbf{d}^T(\mathbf{a}_{12} \times \mathbf{a}_{13})}{\mathbf{a}_{11}^T(\mathbf{a}_{12} \times \mathbf{a}_{13})} \qquad (3.4.25)$$

Note that the denominator can be used for all three λ_i. Figure 3.7 illustrates a case where λ_3 is negative, and shows the projection of d which indicates that removing $\mathbf{a}_{13}$ becomes necessary.

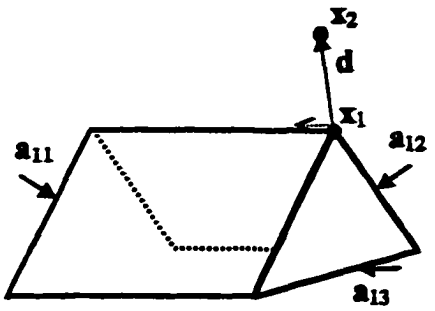


Figure 3.7 – Boundary Condition for Lambda, Vertex Case

3.4.3.4 Using Lambda in the DQP3 Algorithm

An advantage of this approach is that it can be computed whenever needed, it is not dependent on the 'delta' stage producing a result of $\delta = 0$. Thus it becomes possible to change the sequence of the QP algorithm somewhat, attempting to remove facets on every cycle. The new algorithm is:

1) Given $\mathbf{x}^{(1)}$ and $\mathcal{A}^{(1)}$, set $k = 1$.
2) If both objects have > 0 constraints, compute $\lambda^{(k)}$ and find the minimum, $\lambda^{(k)}_{min}$; if $\lambda^{(k)}_{min} \geq 0$ terminate with $\mathbf{x}^* = \mathbf{x}^{(k)}$, otherwise remove the matching constraint from $\mathcal{A}$
3) Solve (3.3.5-6) for $\delta^{(k)}$.

4) Find $\alpha^{(k)}$ that solves (3.3.11) and if $\alpha^{(k)} > 1$ set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \delta^{(k)}$, else set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \delta^{(k)}$.

5) If $\alpha^{(k)} < 1$, add the corresponding constraint to $\mathcal{A}$

6) Set $k = k + 1$ and go to step 2.

This version applied to DQP3 is referred to as the 'consistent reduction' variant.

3.4.4 Incorporating Translation and Rotation

While the above algorithm is adequate for finding the distance between arbitrarily defined convex polyhedra, it makes no provision for the motion of the objects. This is of particularly important in robotics applications, as the distance between the objects as they move through space is of interest.

The datasets representing the polyhedra may be manipulated directly to produce new versions objects in the new locations, and then the above algorithm may be applied. Rotation of the object can be accomplished by multiplying the vector constraints by the rotation matrix, **R**.

$$\mathbf{A}_{1R} = \mathbf{R}\mathbf{A}_1 \tag{3.4.26}$$

Displacement by vector **d** is simply a matter of translating the offsets for the vector constraints:

$$\mathbf{b}_{1d} = \mathbf{b}_1 + \mathbf{A}_{1R}^T \mathbf{d} \tag{3.4.27}$$

While this is reasonably efficient for small objects, as the number of facets grows larger the cost of the computation grows with it, linearly. For large objects it would be more useful if the translation and rotation of the objects were arguments to the algorithm, and did not have to be applied to the object data.

It is possible to maintain $\mathbf{x}_1$ and $\mathbf{x}_2$ in the object's frames of reference (i.e. in object coordinates) and only translate to absolute coordinates for the 'delta' and 'lambda' calculations. The active set is built and maintained in absolute coordinates via the same

sort of rotations as shown above, save that the rotations only need to be done as constraints are added, and are not applied to the entire set of constraints.

Given a displacement vector $\mathbf{p}_1$ for object 1, and rotation matrix $\mathbf{R}_1$, it is simple to transform the object coordinate $\mathbf{x}_{1o}$ to absolute coordinate $\mathbf{x}_{1a}$:

$$\mathbf{x}_{1a} = \mathbf{R}_1 \mathbf{x}_{1o} + \mathbf{p}_1 \tag{3.4.28}$$

Once δ has been found, the rotation each part must be reversed to establish the object version for the 'alpha' step. No displacement is required, as δ is relative to the object position in either frame of reference.

$$\delta_{1o} = \mathbf{R}_1^T \delta_{1a} \tag{3.4.29}$$

Adding a new active constraint to object 1:

$$\mathbf{a}_{11a} = \mathbf{R}_1 \mathbf{a}_{11o} \tag{3.4.30}$$

The above algorithm was implemented in C, as 'dqpt.c' (Appendix A). Some timing test results against the non-translating and rotating DQP3 algorithm can be found in the next chapter.

3.5 Comparison of QP and DQP3 Solutions

Note that most efficient starting point is the centroids of the respective objects, in the case where nothing is known about the problem. It is possible to pre-process the objects, projecting from one centroid to the other and stopping at the first triggered constraint. There is no need to take such a step, as the first iteration of the DQP3 algorithm using the centroids as starting point will in effect do just that.

A tactic first suggested by Lin [Lin91] was to exploit the *coherence* of the object data when the objects are undergoing relative motion, with the distance problem to be solved at each incremental time step. It stems from the basic observation that as the objects move with respect to each other, the closest features change slowly. Thus, the solution

computed for time step t_n is likely to be close to the solution for t_{n+1}, and usually makes a good starting point. This situation is often referred to as the *tracking problem*, where the closest point on a pair of objects must be tracked over motion of the objects.

The simplest exploitation of coherence is to save the solution x and active set A from one time step to the next, using the old solution (suitably translated for the ensuing motion of the objects) as the starting point of the next time step.

Tracking has also been used by others improving on Gilbert's original work [Sato96, Cameron97a].

Table 3.3 shows the results of running the core DQP3 algorithm in Matlab 5.3 against the two QP methods provided by Matlab, qp() and quadprog(). No tracking was done. The results are in Matlab flops, which count 1 for addition, subtraction, multiplication, division or transcendentals such as sqrt(). The objects used were spheroids of radius 1, at position $[5\ 5\ 5]^T$.

N_1+N_2	DQP3	QP	QuadProg
12	1487.5	8757.4	11627.1
44	3690.4	--*	21753.2**
92	6196.4	31399.7	27105.8
164	9361.0	49643.1	39521.0
256	14187.2	--*	61798.4**
576	29521.5	168294.1	119288.1
1020	58155.2	304370.8	214486.2
2296	110470.5	657131.5	449885.6
4084	199307.4	1149438.2	777696.1

* At least one point caused cycling, no termination condition.

** Outliers due to cycling (with termination) removed

Table 3.3 – Comparison of QP solutions, without tracking

Table 3.4 shows the results of comparing the three algorithms with tracking applied. The tracking algorithm used was simplistic, merely passing in the x that was returned by the

previous step, with no attempt to record or re-use the active set from the previous iteration.

N_1+N_2	DQP3	QP	QuadProg
12	768.3	9625.8	12095.3
44	2383.5	-- *	21450.8 **
92	2642.1	42120.9	31390.0
164	5752.7	81266.0	51303.2
256	7347.3	126798.2	77668.8
576	20617.9	263467.3	157053.0
1020	35502.4	468365.3	281196.4
2296	79397.3	1130476.1	636490.3
4084	157735.5	2070436.5	1159026.9

* At least one point caused cycling, no termination condition.

** Outliers due to cycling (with termination) removed

Table 3.4 – Comparison of QP solutions, with tracking

It is interesting to note that the QP methods provided by Matlab make poor use of the tracking data. No explanation for this behaviour is offered, save that the Matlab versions are not set up to initialize their active set (in the case of qp()) or otherwise to make good use of a 'good' initial point. Even when fed the exact solution to a problem, they will take almost as many flops to process as when they are given the centroid as a starting point.

The scaling of flops to total number of facets is close to linear for large numbers of facets under any of the 3 algorithms, as is to be expected when the 'alpha' term dominates. At the bottom end the curve is flatter, due to the contribution of the 'delta' term or equivalent. In any case, the DQP3 algorithm outperforms the Matlab 5.3 QP implementations by a factor of about 5 for large facet counts, and over 10 for small ones. However, note that DQP3 sacrifices the ability to handle general QP problems, and is only applicable to the 3D distance problem.

3.6 Large Objects

When dealing with objects with large numbers of facets, different considerations come into play. In particular, the dominant term of the computational cost for the DQP3 algorithm changes from the 'delta' step which was optimized in the previous sections to the 'alpha' step. The cost of searching through the entire list of facets to find which one most constrains δ is fairly obviously an $O(N)$ search, where N is the total number of facets in each object.

3.6.1 Adjacency Database

Another tactic used by Lin [Lin91] in exploiting the coherence of object data in the tracking problem was to create a database of neighbouring vertices for each vertex in the object. Lin simply refers to this as "preprocessing to reduce the size of the coboundary", Sato [Sato96] calls it the 'adjacent-vertex list' and Cameron [Cameron97a] calls it 'hill climbing'. In the end, it's all the same thing – reduce the size of the search set when checking to see where to go next.

In the context of the DQP3 algorithm, this would be a list of which facets are adjacent to each other. With this in hand, the 'alpha' phase of the algorithm can be reduced to searching a handful of facets, instead of the entire set. The evidence from [Lin91, Sato96, Cameron97a] strongly suggests that the computational cost for the tracking problem would be more or less constant in N, at approximately the cost of handling objects of 25 to 100 facets.

3.6.2 Generating the Adjacency List

An algorithm for generating the facet adjacency list was created serendipitously, as part of an effort to visualize the objects defined by half-space constraints. It started as an algorithm for finding the coordinates of the vertices of an arbitrary facet.

Let **A** and **b** be the constraint set defining an object

$$\left\{\mathbf{x}:\mathbf{A}^T\mathbf{x}-\mathbf{b}\geq\mathbf{0}\right\} \qquad (3.6.1)$$

To find the vertices of facet k, $0 < k \le N$, start by finding a pair of facets j_1 and j_2 such that the A columns corresponding to each facet ($\mathbf{a}_k$, $\mathbf{a}_{j1}$, $\mathbf{a}_{j2}$) form a matrix of rank 3. A simpler test, once $\mathbf{a}_k$ and a non-parallel $\mathbf{a}_{j1}$ have been found, is to test against their cross product:

$$(\mathbf{a}_k \times \mathbf{a}_{j1})^T \mathbf{a}_{j2} > 0 \qquad (3.6.2)$$

It was found to be more effective to pick an $\mathbf{a}_{j2}$ where the result of (3.6.2) was greater than 0.05, to avoid numerical difficulties later on.

Given this initial set and the corresponding b_k, b_{j1} and b_{j2}, the vertex at the meeting of the three facets can be found.

$$\mathbf{x} = \begin{bmatrix} \mathbf{a}_k & \mathbf{a}_{j1} & \mathbf{a}_{j2} \end{bmatrix}^{-T} \begin{bmatrix} b_k \\ b_{j1} \\ b_{j2} \end{bmatrix} \qquad (3.6.3)$$

This value must be tested against the full set of constraints, and likely will violate many of them. Choose the most violated constraint by picking j_3 that corresponds to the most negative result from the QP constraint inequality

$$\mathbf{A}^T \mathbf{x} - \mathbf{b} \ge 0 \qquad (3.6.4)$$

Replace either j_1 or j_2 with j_3, and iterate until $\mathbf{x}$ does not violate the constraints. Maintain a list of j_i that have been visited, to prevent cycling.

Once an x has been found that does not violate the constraints, record that as the first vertex, and set f_1 and f_2 to the corresponding values of j_i. These are the first two adjacent facets.

To find the next adjacent facet, project along the edge $\mathbf{a}_k \times \mathbf{a}_{j2}$ in the direction indicated by $\mathbf{a}_{j1}$. Find the first constraint that becomes active by the same method as used in the 'alpha' step. See equations (3.3.10-12). The coordinates for the new vertex can be found by applying equation (3.6.3).

This iteration is continued until the original vertex is reached again. To assist in making sure that the equation terminates properly, a list of all the visited facets, $\{f_1 \ldots f_n\}$ is kept. When the algorithm terminates, this list is the adjacent facet list for facet k.

Figures 3.8 and 3.9 illustrate the use of this algorithm to generate all the facets of two typical target objects for the DQP3 test algorithm. Figure 3.8 shows a 22 facet 4th-order facet-ball. Figure 3.9 shows a 288 facet 15th-order facet-ball.

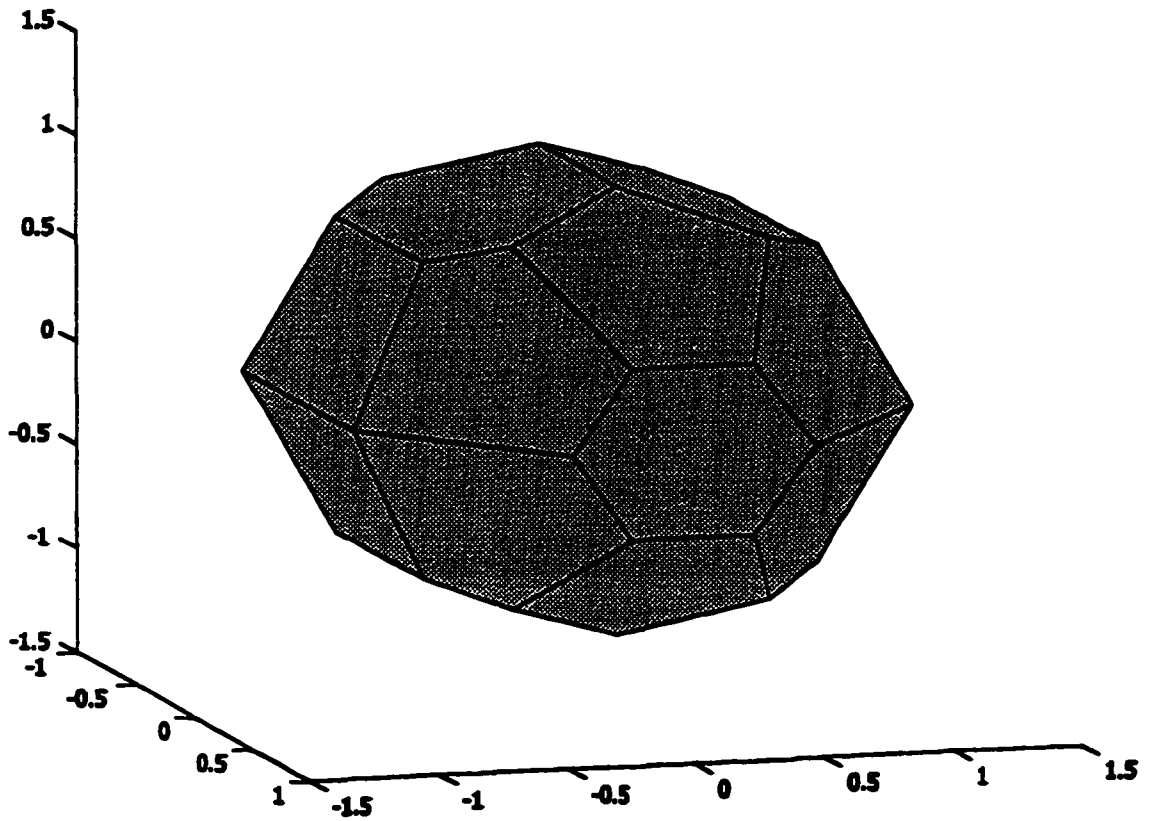


Figure 3.8 – Spheroid with 22 facets

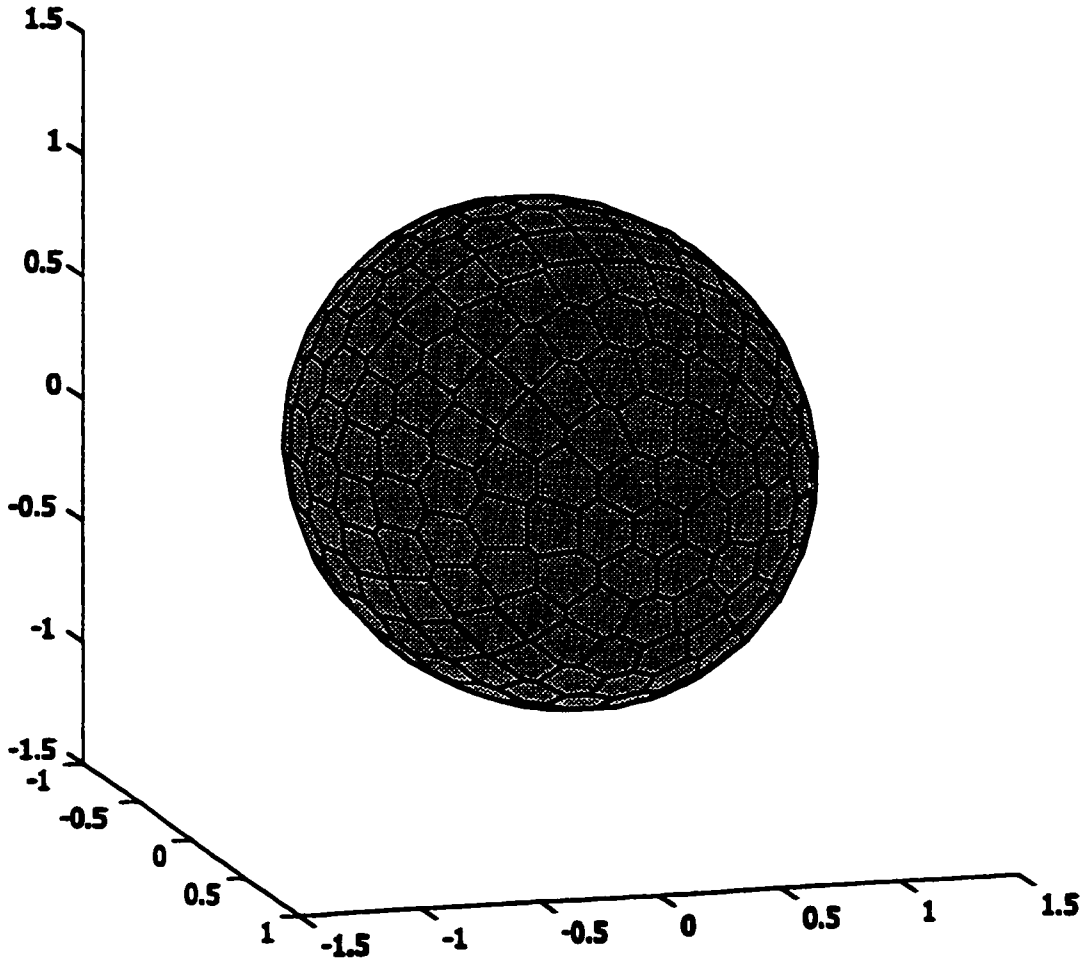


Figure 3.9 – Spheroid with 288 facets

4. COMPARISON OF DQP3 TO OTHER ALGORITHMS

Comparison between various algorithms presented in scattered papers [Bobrow89, Gilbert90, Lin91, Lin96, Sato96, Rimon97, Cameron97a] is a matter complicated by more than just the diversity of the algorithms themselves. Few authors make their source code easily available – Cameron is a notable exception. In addition, computational cost is generally reported in ms or μs to perform a minimum distance computation. The difficulty here is that the CPUs used in the above cited papers range from an Apollo DN3000 (12.5MHz 68020/68881) to an SGI Reality Engine with all its custom rendering hardware. The time of computation criteria will be used here, and where direct comparisons are not possible, benchmark comparisons will be used to approximate the scaling between the various machines.

Among the algorithms that use convex polyhedra to represent objects, the easiest to compare is Cameron's Enhanced GJK (EGJK) algorithm [Cameron97a], as he used hardware (Pentium running Linux) that is easily available. This case will be treated in some detail, running EGJK off against DQP3 on identical hardware with the same compiler and as similar as possible test vectors.

Where direct tests are impossible due to absence of source code, timing comparisons with published results will be given. However, some algorithms are sufficiently disparate that even that is infeasible, for no comparable test can be undertaken. For these cases, general arguments about structure and anticipated problem type will have to suffice.

However, before launching into comparisons of the DQP3 algorithm with its conceptual neighbours, it is first necessary to determine the performance of the suggested variants of the DQP3 algorithm itself, to find the best formulation with which to face the challenge of its peers.

4.1 Variations on DQP3

Several variations on the DQP3 algorithm have been suggested so far. The baseline algorithm takes a modified primal active set QP algorithm, and allows for the adding of two constraints (one per object) each cycle.

Variants:

- 'dual release' – Instead of removing only one constraint per cycle, allow the removal of one from each object.
- 'consistent release' – Evaluate λ every cycle, instead of just when $\delta=0$. Only one constraint per cycle is released.
- 'aggressive release' – Test to remove constraints every cycle, and remove all constraints with $\lambda < 0$.
- 'translation-rotation' – Incorporate rotation and displacement directly within the algorithm, bypassing the need to recompute the object constraints for object motion. This variant may be applied to any of the above.

The base DQP3 algorithm with active-set tracking (save active set from the previous iteration, in addition to $\mathbf{x}$) was implemented in C, and compiled on a 166MHz Pentium PC running Linux. The gcc compiler was used, with full optimization turned on. The test objects are the same facet-balls as used in section 3.5, generated by distributing facets evenly over the unit sphere.

Table 4.1 shows the results of these tests, on the base, aggressive release and dual release DQP3 algorithms. The consistent release code was unable to produce even the handful of stable results required to get into the chart, it was almost completely unusable. Tinkering with the structure of the QP algorithm must be done with care. The aggressive release results a very slight improvement over the base in the non-tracking case. However, it does not agree well with the tracking case, and tended to produce erroneous results from time to time, making data gathering more difficult. As a result, its use cannot be recommended. The dual release code is generally slightly slower than the baseline, with

the occasional improvement, confirming Fletcher's comment [Fletcher87] that changing the QP primal active set release conditions to release more constraints would have little effect on running time. None of the variants seem to be worth pursuing. All times are in microseconds, timed on a 166 MHz Pentium. Runs of 1000 to 10000 randomized initial positions were averaged to get useful timing resolution.

	non-tracking case			tracking case		
N_1+N_2	base	aggressive	dual	base	aggressive	dual
12	79.7	80.6	80.4	18.2	20.5	20.5
44	141.9	141.4	141.9	22.3	27*	25.7
92	236.4	234.3	236.7	28.8	34*	35.4
164	465.0	463.2	463.9	38.5	38*	41.6
256	659.9	654.6	658.3	56.5	58*	63.7
576	1640.1	1625.5	1758.4	143.4	154*	162.6
1020	2585.4	2501.3	2687.6	249.4	205*	320.2
2296	6668.8	6648.8	6482.0	643.3	835*	800.2

* Instability of algorithm forced limited data collection, accuracy is lower

Table 4.1 – Comparison of DQP3 variants

	without translate time		with translate time	
N_1+N_2	base	trans&rot	base	trans&rot
12	18.2	27.9	21.2	28.9
44	22.3	34.1	28.8	35.1
92	28.8	41.7	40.4	42.7
164	38.5	55.4	58.2	56.4
256	56.5	70.3	86.1	71.3
576	143.4	150.2	219.9	151.2
1020	249.4	267.1	409.9	268.1
2296	643.3	694.8	1014.6	695.8

Table 4.2 – Comparison of DQP3 with built-in translation and rotation

Table 4.2 shows the performance hit taken when using DQP3 with built-in translation and rotation code on tracking data. While the baseline speed is a little faster at all facet sizes, once one incorporates the time taking to set up the data, the story changes. The base DQP3 algorithm requires the vector b to be recomputed for new positions of the objects, and that carries an O(N) cost. On the other hand, the translation and rotation code handles it internally, without needing to recompute every facet in the object. Once the facet counts get above 120, the built-in translation and rotation code is more efficient.

One last chart shows the speed improvement when going from the Linux 166 MHz Pentium to a Windows 900 MHz Athlon. The Linpack results for these two CPUs are 15.7 MFlops and 418 MFlops respectively, leading to an expected speedup of about 26.6 times. One thing the table shows quite clearly is that using the Linpack benchmark to judge DQP3 performance scaling is only accurate to a fairly coarse degree. I suspect this is due to the large scale of the Linpack problem, and that it is math-intensive. The DQP3 code has a lot of decision-making and integer math to do as well, which skews the results. However, on problems of modest size, the results are usable.

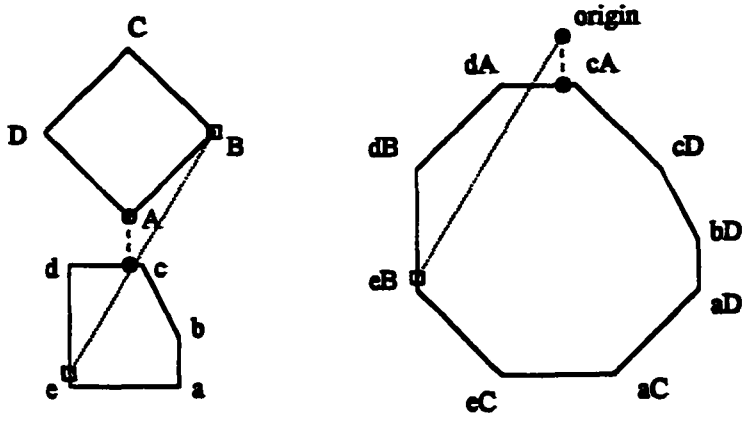
	non-tracking case			tracking case		
N_1+N_2	166MHz	900MHz	speedup	166MHz	900MHz	speedup
12	79.7	9.3	8.56	18.2	1.6	11.38
44	141.9	16.4	8.65	22.3	1.7	13.12
92	236.4	21.4	11.05	28.8	2.2	13.09
164	465.0	32.4	14.35	38.5	2.8	13.75
256	659.9	31.4	21.02	56.5	5.1	11.10
576	1640.1	79.1	20.73	143.4	11.6	12.36
1020	2585.4	127.4	20.29	249.4	14.9	16.74
2296	6668.8	329.5	20.24	643.3	47.3	13.60

Table 4.3 – Comparison of 166 MHz Pentium to 900 MHz Athlon

4.2 DQP3 vs. Enhanced GJK

Cameron's Enhanced GJK (EGJK) algorithm [Cameron97a] has source provided online, <http://web.comlab.ox.ac.uk/oucl/work/stephen.cameron/distances/>. Version 2.4 of the code was downloaded onto a 166MHz Pentium PC running Linux, and compiled with full optimization. The test problem was adjusted to more closely match the relative velocity and positioning of the DQP3 test problem.

The GJK algorithm, based on Gilbert's work [Gilbert88] uses a vertex based representation of the object. A concise description can be found in [Cameron97b]. Consider two objects P and Q, their convex hulls defined by sets of vertices. The distance between the two objects can be represented as a problem in *translational configuration* (TC) space, by combining the two polyhedra P and Q into a single convex object, the *translational configuration space obstacle* (TCSO) that represents the set of all translations that would bring the two objects into contact. Its structure is somewhat akin to a convolution of the two objects. In TC space, finding the distance between polyhedra P and Q is equivalent to finding the distance from the origin to the polyhedron.



(a) Original objects (b) Corresponding TCSO

Figure 4.1 – Two dimensional example of polyhedra and their equivalent TCSO

Figure 4.1 illustrates the equivalence of the two representations. The GJK algorithm works by taking some point within the TCSO, and iteratively choosing a better position

closer to the origin. This point in TC space represents a pair of features on the polyhedra, and as it gets closer to the origin, the estimate of the distance between the polyhedra improves. For example, if the initial 'witness points' were the squares in Figure 4.1, an iterative improvement might visit feature pairs (TCSO vertices) dB and dA on the way to the final solution.

Instead of just working with single points, the GJK algorithm works on sets of up to four points, defining the vertices of a *simplex* (i.e. a point, line, triangle or tetrahedron). The simplex will always be a subset of the TCSO. As the algorithm progresses points are added to and removed from the simplex to move it towards the origin.

Cameron's implementation of the Enhanced GJK algorithm never actually needs to generate the TCSO directly, only the current simplex. Handling of translation and rotation are incorporated directly, much like the dqpt.c implementation of the DQP3 algorithm. The main enhancement added by Cameron is what he refers to as 'hill-climbing', maintaining a map of neighbouring vertices. This greatly reduces the cost of searching for the best new vertex, especially during the tracking problem, as the new vertex will generally be a neighbour. Thus most of the time during the tracking problem, only one iteration is required.

The outputs of the EGJK algorithm include not only the distance between the objects (or zero if they intersect), but also a representation of the witness points on the objects, allowing the closest position between the objects to be easily determined.

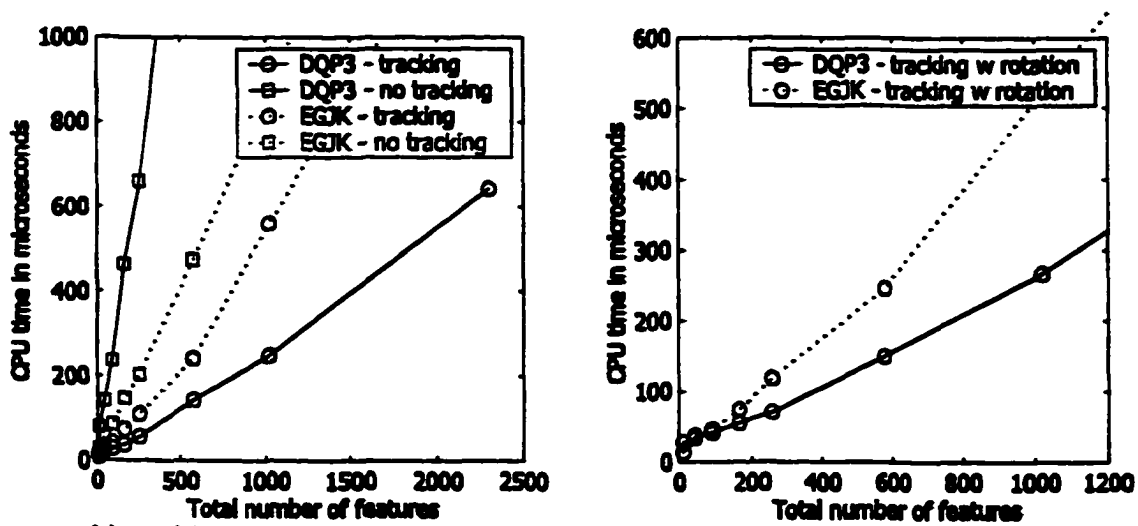
4.2.1 Direct Timing Comparison

Cameron's Enhanced GJK code was recompiled on the same 166MHz Pentium that the DQP3 code was tested on. In an attempt to make the comparison as even as possible, the code to randomly generate a starting point and trajectory was tweaked slightly to make it more closely resemble the trajectories used by the DQP3 tests. The main reason for this was that the EGJK code generated trajectories that caused intersection of the objects, and tended to run substantially faster for a series of intersecting positions than for non-intersecting ones.

A series of tests were run with objects of varying feature counts. It has been assumed that features are equal in value, whether they are facets or vertices. The results are below, for varying sampling densities along the trajectory and varying facet counts. An interesting phenomenon can be seen. While the EGJK algorithm is faster initially, when the cost of the uninitialized case is spread over larger and larger numbers of points, the DQP3 algorithm pulls ahead. This suggests that the per-iteration cost of an ongoing tracking problem will be about 25% lower using the DQP3 algorithm.

	non-tracking case		tracking case (100 pts)	
N_1+N_2	no rotation	rotation passed	no rotation	rotation passed
12	27.3	36.2	10.3	14.3
44	58.7	67.7	27.9	37.2
92	84.9	95.5	42.4	45.7
164	146.7	153.6	73.3	73.7
256	203.9	222.4	109.6	118.1
576	474.8	503.9	242.2	246.5
1020	872.2	984.0	560.9	520.8
2000	1801.5	1985.0	1215.5	1139.5

Table 4.4 – Timing results for Enhanced GJK



(a) – without rotation computations (b) – with rotation computations

Figure 4.2 – Comparing Enhanced GJK and DQP3

Figure 4.2 illustrates the relative performance of the two algorithms, on a variety of facet sizes. Figure 4.2 (a) compares the base DQP3 algorithm to EGJK without passing rotation matrices. Figure 4.2 (b) compares DQP3 with translation and rotation to EGJK with rotation matrices passed in. The DQP3 algorithms pull well ahead for the tracking case on 100 points, but the question remains – at what number of tracking points does this change take place?

In an attempt to answer this question, the data in Table 4.5 were gathered, showing the crossover happens when the tracking covers between 20 and 50 steps. The DQP3 algorithm is much more expensive on an uninitialized pair, and gains quite substantially by using tracking. As a result, it takes longer to amortize the initial cost. Since most robotic applications would continue tracking more or less indefinitely, the long-term tracking values are of the most interest. As always, times are in μs on a 166 MHz Pentium.

	$N_1+N_2=12$		$N_1+N_2=92$		$N_1+N_2=576$	
steps	EGJK	DQP3	EGJK	DQP3	EGJK	DQP3
1	27.3	79.7	84.9	236.4	474.8	1640.1
5	16.9	40.6	99.9	130.0	536.8	1165.8
20	11.7	22.6	66.8	52.3	332.3	387.2
50	11.1	19.3	48.1	34.6	271.8	204.2
100	10.3	18.2	42.4	28.8	242.2	143.4

Table 4.5 – DQP3 vs. EGJK over various tracking intervals

While the DQP3 algorithm is faster at the tracking problem than EGJK for medium and larger problems, and should maintain that lead on large problems when the adjacency list is available, there are other considerations when comparing the two algorithms.

4.2.2 Other Features

The biggest advantage the EGJK method claims is that it can provide a measure of the maximum penetration distance of a pair of overlapping objects. However, the EGJK code tested against DQP3 did not have that capability programmed. Examining the

structure of the TCSO representation at the heart of the EGJK method, it can be seen that finding the penetration distance is the same as finding the distance between an interior point and the surface of a convex polyhedron. This is not a convex problem, and the solution will require a comprehensive search.

The DQP3 algorithm does not yet have a maximum penetration depth measure, partly due to the fact that penetration depth is a non-convex problem and thus cannot be treated as a QP problem. Methods to extend DQP3 with penetration depth measures have been contemplated, but are not yet complete.

4.3 Other Convex Polyhedron Methods

A variety of methods for solving the convex polyhedron distance problem have been proposed over the last decade and a half. Working through the list chronologically, we will start with the original Gilbert paper [Gilbert88], proceeding up through to Sato's work on enhancing the Gilbert method. Most papers that use an adjacency list to enhance timing also show results without the enhancement, and those results will be used to compare with the DQP3. For the papers where tracking is not used, the DQP3 results without tracking will be used. If papers include results for colliding and non-colliding objects, the non-colliding results are reported. Reference points are 20 and 100 total features.

- Gilbert, Johnson and Keerthi, 1988 [Gilbert88]
  Basic feature: Vertices
  CPU: Harris 800. (Linpack: 0.23 MFlops)
  Best time for 20 features: approx. 1.3ms
  Best time for 100 features: approx. 7.8ms
  Gilbert's approach does not clearly state it, but it does appear to use tracking, and attempts a crude form of adjacency (by ordering the vertex list in some special way). Feature counts ranged from 4 to 200. Results were fairly linear. Gilbert does go on to show a tracking problem, but it cannot be easily analyzed in these terms. These

numbers are not derived from direct timing, but rather from 'equivalent flops' times the CPU time for division. The values are not considered realistic.

- Bobrow, 1989 [Bobrow89]

  Basic feature: Facets (defined by half-spaces)

  CPU: Apollo DN3000. (Linpack: 0.071 MFlops)

  Best time for 20 features: approx. 31ms

  Best time for 100 features: approx. 191ms

  Bobrow's approach does not include tracking or adjacency information. Feature counts ranged from 10 to 145, and objects were of varied shapes.

- Gilbert and Foo, 1990 [Gilbert90]

  Basic feature: Vertices (and some exact representations of curved objects)

  CPU: Apollo DN3000. (Linpack: 0.071 MFlops)

  Best time for 20 features: approx. 22ms

  Best time for 100 features: approx. 57ms

  Gilbert provided polyhedron results for comparison to exact representations of curved objects. In general, for vertex totals under 60, the polyhedron code was equivalent or superior. Feature counts ranged from 10 to 210. A close reading suggests that these are tracking results.

- Lin and Canny, 1991 [Lin91]

  Basic feature: Facets (with full description of extent)

  CPU: Sun4 SPARCstation. (Linpack: 1.4 MFlops)

  Best time for 20 features: approx. 23ms

  Best time for 100 features: approx. 52ms

  Lin and Canny present data for tracking and adjacency list, but not for the two cases separately. Just the non-tracking case is reported. Feature counts ranged from 16 to 144, with some interpolation needed to get the above numbers.

- Ma and Nahon, 1992 [Ma92]

  Basic feature: Facets (half-planes)

  CPU: Sparcstation 1+ (Linpack: 1.8 MFlops)

  Best time for 20 features: approx. 4ms

Best time for 100 features: approx. 12ms

Ma and Nahon formulate the distance problem as a quadratic programming problem, and use conventional solutions. Their code has been implemented for a full-scale working robot [McIntyre92]. Feature counts ranged from 6 to 150.

- Sato, Hirata, Maruyama and Arita, 1996 [Sato96]

  Basic feature: Vertices

  CPU: SGI Crimson (Linpack: 16 MFlops)

  Best time for 20 features: approx. 0.6ms (tracking)

  Best time for 100 features: approx. 1.6ms (tracking)

  This paper extends the Gilbert method by adding an adjacent vertex list. The above times are for their implementation of the original Gilbert method. The extended method managed to get 0.6ms times for all polyhedron sizes, ranging up to 15000 facets. Tracking was over 100 points.

The 166 MHz Pentium used for the DQP3 results was not in the Linpack benchmark database. However, a 133 MHz Pentium was, and comparisons using SPECfp between a 133 MHz Pentium and 166 MHz Pentium were available. The 133 MHz Pentium rates 14 MHz on the Linpack benchmark. A current mid-range PC such as the 1.2 GHz Athlon Thunderbird rates at 558MHz, so the beyond-linear effect of processor generations can be seen. The SPECfp95 ratings for 166MHz Pentiums average 1.12 times faster than 133 MHz Pentiums from the same manufacturer. Thus, the 166 MHz Pentium is placed at 15.7 MFlops.

Tables 4.6 and 4.7 show timing results for tracking (over 100 points) and non-tracking solutions respectively, and provide a scaling factor against the DQP3 result. The 20 and 100 facet results for the DQP3 code were interpolated from the results in Table 4.1, above.

Method	#features	time(ms)	MFlops	Normalized to P166	Performance ratio
Bobrow 1989	20	31	0.071	0.142	1.47
	100	191		0.864	3.30
Lin 1991	20	23	1.4	2.05	21.5
	100	52		4.64	17.7
Ma 1992	20	4	1.8	0.459	4.81
	100	12		1.38	5.27
DQP3	20	0.0955	15.7	0.0955	1
	100	0.262		0.262	1

Table 4.6 – Performance measure of various distance algorithms (non-tracking)

Method	#features	time(ms)	MFlops	Normalized to P166	Performance ratio
Gilbert 1990	20	22	0.071	0.0995	5.18
	100	57		0.258	8.63
Sato 1996	20	0.6	16	0.611	31.8
	100	1.6		1.63	54.5
Cameron 1997	20	0.0139	15.7	0.0139	0.724
	100	0.0426		0.0426	1.42
DQP3	20	0.0192	15.7	0.0192	1
	100	0.0299		0.0299	1

Table 4.7 – Performance measure of various distance algorithms (tracking)

The DQP3 outperforms all other algorithms on tracking tests of moderate size, and all but Cameron's EGJK for uninitialized cases and small feature count tracking tests. It is interesting to examine the total speedup since Gilbert's work in 1990. The tracking time for a 20 feature problem in 1990 was 22 ms. The same problem can be done on a 900 MHz Athlon, which is today a medium-range home PC, in less than 2 μs. That is over 4 orders of magnitude. Of course, a huge factor in this improvement is due to the change of CPU – the Athlon would get a Linpack benchmark of about 418 MFlops. While the Linpack results for the Apollo DN3000 seem perhaps a little low, over three of the orders

of magnitude in the speedup undeniably come from hardware. It's good to be alive in the 21st Century.

4.4 Other Methods

Two other methods of handling the minimum distance problem on convex objects deserve mention here. The first is Oriented Bounding Boxes (OBB) [Lin96, Gottschalk96], which is admittedly a more general algorithm, suited for concave and convex problems. A more curious entry is Rimon and Boyd's work on ellipsoid approximations [Rimon97].

The problems addressed by OBB methods tend to be much more complex, making use of the faster processors now available to treat more detailed models. The name is derived from the fundamental distance unit used – a rectangular prism, positioned and oriented in space. Computation of distance between these units is comparatively cheap due to their simplicity. The complexity comes in deciding which feature pairs need to be tested at a given point in time. The most common use is to generate a hierarchical tree of OBBs [Lin96, Gottschalk96] to represent non-convex polyhedra, and the bulk of the collision detection (rather than distance computation) problem becomes descending the tree to determine whether or not a collision has occurred. This allows fantastic efficiency, for example detecting collisions between two objects each of 143690 polygons in 4.2ms on average. (On an SGI Reality Engine R8000 CPU at 90MHz.) It is fairly obvious that these results will be difficult to compare with algorithms designed to handle smallish numbers of facets. In this case, it is notable that the distance measure is not provided, just a collide/no-collide decision. While the timings are impressive, they do not help the problem of control, as obstacle avoidance algorithms require a distance measure to optimize.

Other hierarchical representations exist. Guibas [Guibas99] improves on the Lin-Canny results [Lin91] by representing the polyhedra as nested or hierarchical sets of polyhedra. This allows faster transition between points with little coherence, by rising up the tree then descending it, rather than skirting all the way around the outside of the object. The

method seems promising for objects undergoing rapid relative motion, but direct comparison of the results is difficult, as no concrete measure of performance was cited.

An even simpler hierarchical representation is to tile a non-convex object with spheres [Quinlan94]. The advantage is that sphere to sphere distance calculations are dirt cheap, but spheres tend to map the object poorly, requiring large quantities to get reasonable resolution. Again, without a discussion of computational effort, comparisons on the basis of speed are difficult.

For a completely different take on the distance problem, one could try modelling arbitrary convex objects with their minimal enclosing ellipsoid [Rimon97]. There are two particularly significant problems with this approach. The first is that most objects do not look much like ellipsoids. The authors claim of 1:n best-fit enclosure is not heartening when one realizes that it means a 3D object will be mapped by an envelope that is 3 times as large as the object itself, in spots. In addition, the distance value that results is not in Cartesian coordinates. The authors point out that no closed form computation of the ellipse to ellipse distance is known, although it is a convex problem. Instead, they present the 'Free Margin Function' which only maps to distance in that it goes to zero for contact between the ellipses. On the positive side, the free margin function, based on the eigenvalues of the ellipsoid, can be computed in 2.5ms (on a DEC5000, Linpack: 5.3 MFlops). With tracking, the time can be brought down to 1ms. These times are equivalent to DQP3 processing of two 160 facet polyhedrons, or two 650 facet polyhedrons for the tracking case. That is without adjacency lists, which will improve the tracking case substantially.

4.5 Adjacency List Timing

Optimizing DQP3 with adjacency lists has not yet been done. Given the gains shown by Lin [Lin91], Sato [Sato96] and Cameron [Cameron97a], it is reasonable to expect that tracking times for objects of arbitrary facet count would be similar to those seen for objects of approximately 25-50 facets. Thus for problems that can be sufficiently well

represented by objects of this complexity, adjacency list methods are a needless complication.

Generating adjacency lists for the DQP3 dataset by the algorithm of section 3.6.2 takes a number of iterations per facet on the order of the number of edges. Each iteration requires a search through the entire facet set for the next adjacent facet, and thus the adjacency list for a given facet can be generated in $O(N)$ time. Thus, the facet list for the entire object requires $O(N^2)$ time.

facets	Matlab flops
6	3669
22	50640
46	664838
82	1653492
128	8645644
288	28257845
510	153121140
1148	511450114

Table 4.8 – Cost of computing adjacency list

Obviously, the facet adjacency algorithm isn't very efficient. However, it only needs to be run once on a given object, and can be reused indefinitely.

4.6 Multiple Object Example

McIntyre's application of a polyhedra based distance algorithm to a full robot control implementation [McIntyre92] had one major flaw: The collision detection loop took 352ms to run, in which time the robot, moving at the full base rotation rate of 180°/sec, could move up to 2.5 meters. Obviously, a faster result is needed for practical use. A large improvement may be made simply by moving to a modern (in 2001 terms) CPU – the 68030 CPU used by McIntyre has a Linpack benchmark of 0.55 Mflops, while an inexpensive modern CPU such as a 900MHz Athlon rates about 418 Mflops. Some of

that change is from compiler optimizations to inflate benchmarks, but much of the almost three orders of magnitude difference must be seen as genuine.

The application in question represented the robot arm with 14 separate polyhedra, none of more than 10 facets. The total facet count for the entire robot was only 96, in an effort to keep the distance computation costs down, no doubt. The workspace was represented by one polyhedral object, and a single plane to model the floor. Due to the robot geometry, 6 of the polyhedra could never impact the floor, and no self-collisions were considered. This left 25 pairs of objects that needed to be computed.

To handle this problem with the DQP3 algorithm, a fairly simple procedure can be followed. Start with a set of object data structures containing the base $\mathbf{A}_i$ and $\mathbf{b}_i$ information for each object i. Update them with the current orientation $\mathbf{R}_i$ and displacement $\mathbf{d}_i$ of the object. Given a list of object pairs of interest, $\mathcal{L}$, proceed as follows:

1) Take the first pair from list $\mathcal{L}$, set i and j to the values retrieved. Set d_{min}=+Inf.
2) Compute the distance between the objects by passing $\mathbf{A}_i$, $\mathbf{b}_i$, $\mathbf{R}_i$, $\mathbf{A}_j$, $\mathbf{b}_j$, $\mathbf{R}_j$ and $\mathbf{p}=[\mathbf{d}_i^T\ \mathbf{d}_j^T]^T$ to the dqpt.c function fnDistQPT, as well as the previous active sets and positions from the object data structures.
3) Save the active sets and final position back into the object data structures for reuse.
4) Check the returned distance d. If $d < d_{min}$, set d_{min} to d, and save whatever other data about the minimum distance position is of interest.
5) If we are at the end of list $\mathcal{L}$, terminate reporting d_{min} as the minimum distance, plus other data as required.
6) Take the next pair from list $\mathcal{L}$, set i and j to the values retrieved. Go to step 2.

Assuming that the above doesn't take too much overhead, the determining factor for the speed of the algorithm is how fast the polygon comparisons in step 2 can be completed. For simple polygons on a 900 MHz Athlon CPU, the time is under 2 μs for tracking

problems, which this is. As a result, we could expect to complete the list of 25 polyhedra pairs from the McIntyre problem in under 50 μs. This compares well to the 460 μs that direct scaling via Linpack benchmarks would suggest.

Assuming we have 1 ms to close the collision avoidance control loop, how many workspace obstacles could be considered, given a 14 polyhedra robot? Over 35 convex obstacles in the workspace could be handled. Increasing the robot and workspace object facet counts into the 20s would have no significant effect on this result, but as the facet counts get substantially larger, the total workspace objects computed in 1ms would have to decrease. For example, examining Table 4.3 suggests that average facet counts over 100 would drop the number of obstacles to 14 or less. Handling self-collision would also reduce the workspace object count.

In cases where there are too many objects to track every single pair every cycle, decisions must be made as to which object pairs get priority. If the list of object pairs is augmented with information on distance, closing velocity (from the current configuration information) and the last test time for each pair, the list could be sorted by priority of possible collision. Even something simple as a triage sort could be undertaken, sorting the pairs into bins by the number of cycle counts until a possible collision. The first few bins would have to be evaluated every cycle, of course. The remainder could be handled in a round-robin fashion, stalest first. Given that 500 pair-wise distance computations may be done in a 1ms cycle (less the triage overhead), lists of several thousand pairs to be processed quite promptly. Say, a 20-polyhedra robot among 100 or more objects, or a collection of up to 100 objects all requiring mutual distance computation.

More aggressive algorithms for handling this sort of sorting and processing can be found in the hierarchical distance computation literature. For example, Hopcroft's algorithm for handling collections of spheres [Hopcroft83] or Edelsbrunner's work on rectangular boxes [Edelsbrunner83] both suggest that object collections could be handled in approximately $O(N \log^2 N)$ pair-wise computations instead of $O(N^2)$.

5. Ellipsoid Distance Computation Methods

There are more ways to represent convex objects than just the polyhedral estimates so far examined in this thesis. One elegant representation is the ellipsoid, a 3 dimensional extension of the ellipse. The surface of a canonical or axis-oriented ellipsoid is defined as:

$$\frac{x_1^2}{a_1^2}+\frac{x_2^2}{a_2^2}+\frac{x_3^2}{a_3^2}=1 \qquad (5.0.1)$$

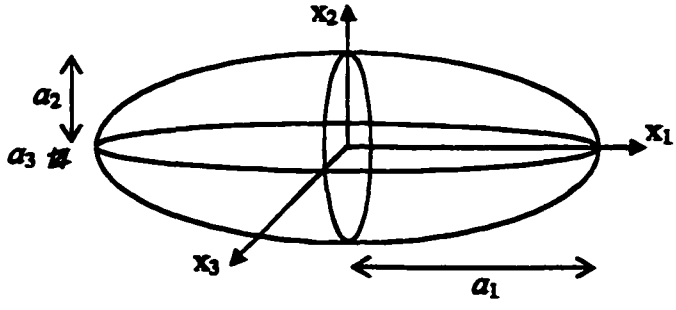


Figure 5.1 – Canonical Ellipsoid

The values a_1, a_2 and a_3 (often combined into a vector, **a**) represent the half-lengths or 'radius' of the ellipsoid along each of its three axes. If the axes are sorted by length, $a_1 > a_2 > a_3$, they are called the major, semi-major and minor axis, respectively. The ellipsoid is a disarmingly simple looking structure, looking like a squashed and stretched sphere. However, where the distance between spheres is trivial to compute, (merely taking the distance between the centers and deducting the radii) finding the distance between a pair of ellipsoids is far from trivial. No closed form is known for the computation [Rimon97].

Due to the computational difficulties, ellipsoids are rarely used in robotics. Rimon and Boyd [Rimon97] use ellipsoids to approximate arbitrary convex objects, and show some interesting results with respect to finding the minimum ellipsoid to enclose a convex object, and finding a distance-like measure between them. However, while their 'free margin function' does have the desirable property of going to zero when the ellipse touch

each other, and negative for interpenetration, the measure itself does not map in any useful way to the Cartesian distance. It is useful only for collision detection, not for collision avoidance control. In Gilbert's work on general convex objects [Gilbert90] he touches on ellipse sections (part of an ellipsoid with two of the axes having equal length). While he does provide support functions and mappings for the ellipse section, he does not actually show any results using these figures for distance computations, probably due to the complexity of the representation.

So why use ellipsoids? Certainly, few robots use anything even resembling an ellipsoid in their structure –far better to use circular sections, cylinders and polyhedra. Rimon and Boyd indirectly point out the difficulty in using an ellipsoid to model such an object, claiming that their best-fit ellipsoid only has an error factor of n, the dimensionality of the space in which the object resides. A polyhedral mapping should be able to do better than an error margin of 3 on almost any convex object using a very small number of facets. We must consider the workspace of the robot – it is entirely possible that the obstacles encountered will be ellipsoids, or the target objects themselves. Imagine, if you will, a robotic egg-sorter. (Granted, chicken eggs are more of a tapered ellipsoid-like shape, but an ellipsoid model would likely be quite suitable.) Note that spherical objects should be handled differently, as they can be much more simply modeled by a point and a distance offset.

The main reason to consider ellipsoids is that it is simply an interesting problem, easily defined but difficult to handle. In particular, ellipsoid based polyhedral models exercise the polyhedral distance algorithms of the previous chapter much more strongly than the sphere-like models typically used. Cases easily can be constructed where the centroid to centroid line is far away from the actual closest point. This may allow some of the DQP3 variants to show in a better light, as anything that allows faster transition across the surface of the ellipsoid will be useful.

Three different ways to compute the distance between a pair of ellipsoids are presented here. First and simplest is to iteratively use a 'closest point on the ellipsoid to a given point' solution to converge to the closest points. Another method is to model the

ellipsoid with a polyhedron at a high enough resolution that a polyhedra to polyhedra distance computation solves the distance problem to adequate resolution. Finally, there is an approach where the polyhedral models are iteratively refined in the area of interest.

5.1 Point to Ellipsoid Problem

As a stepping stone to working on the ellipsoid to ellipsoid problem, efficient code for finding the closest point from an arbitrary point to the surface of the ellipsoid is necessary. A search for existing solutions turned up NEARPT from the SPICE library from the Navigation and Ancillary Information Facility at JPL. The code was written in FORTRAN, and thus a non-trivial port. A cursory analysis of the code showed that it would take about 706 flops to make a decent estimate of the ellipsoid surface point closest to the point in question. Other references only noted that the problem could be reduced to finding the roots of a 4th order polynomial. Tests with Matlab show that its root-finding algorithm takes on average 1450 flops on a 4th order polynomial, and that doesn't take into account cost of re-casting the problem, naturally. So anything that reduces to finding roots on a 4th order polynomial is out.

The point to ellipsoid problem can be defined as a constrained optimization problem, trying to find $\mathbf{x}$ on the surface of the ellipsoid such that the distance to the target point $\mathbf{p}$ is minimized. Without loss of generality we can assume that the ellipsoid is centered at the origin.

$$\underset{\mathbf{x}}{\text{minimize}} \quad F(\mathbf{x}) = (\mathbf{x}-\mathbf{p})^T(\mathbf{x}-\mathbf{p}) \tag{5.1.1}$$

$$\text{subject to} \quad \mathbf{x}^T \begin{bmatrix} \frac{1}{a_1^2} & 0 & 0 \\ 0 & \frac{1}{a_2^2} & 0 \\ 0 & 0 & \frac{1}{a_3^2} \end{bmatrix} \mathbf{x} - 1 = 0 \tag{5.1.2}$$

We will call the diag($1/a_1^2$, $1/a_2^2$, $1/a_2^3$) array $\mathbf{A_e}$. Applying the KKT (Karush-Kuhn-Tucker) conditions [Fletcher87] to the problem we get the following equations.

$$(\mathbf{x}-\mathbf{p})+\lambda \mathbf{A}_e \mathbf{x}=0 \qquad (5.1.3)$$

$$\mathbf{x}^T \mathbf{A}_e \mathbf{x}-1=0 \qquad (5.1.4)$$

Rewriting (5.1.3) to solve for $\mathbf{x}$ in terms of λ and $\mathbf{p}$, we get

$$\mathbf{x}=\left[\frac{a_1^2 p_1}{a_1^2+\lambda} \quad \frac{a_2^2 p_2}{a_2^2+\lambda} \quad \frac{a_3^2 p_3}{a_3^2+\lambda}\right]^T \qquad (5.1.5)$$

Substituting into (5.1.4) and some algebraic juggling gives us

$$S(\lambda)=\frac{p_1^2}{a_1^2\left(1+\frac{\lambda}{a_1^2}\right)^2}+\frac{p_2^2}{a_2^2\left(1+\frac{\lambda}{a_2^2}\right)^2}+\frac{p_3^2}{a_3^2\left(1+\frac{\lambda}{a_3^2}\right)^2}=1 \qquad (5.1.6)$$

For obscure reasons we will call the three terms of the $S(\lambda)$ sum α_1, α_2 and α_3.

All that remains is to find a way of solving for λ. Direct solution is out, as (5.1.6) turns into a 6th order polynomial fairly quickly, and we've already established that we want something cheaper than root-finding.

It turns out that due to the shape of the $S(\lambda)$ function, λ has one unique solution, and it can be found by various convergent iterative methods.

The most effective iteration found utilized the observation that the relative values of α_1, α_2 and α_3. changed slowly. Thus, a suitable α_i would be picked, and on iteration k a new λ, $\lambda^{(k+1)}$ would be computed by setting:

$$\alpha_i^{(k+1)}=\frac{\alpha_i^{(k)}}{\alpha_1^{(k)}+\alpha_2^{(k)}+\alpha_3^{(k)}} \qquad (5.1.7)$$

Solving for the $\lambda^{(k+1)}$ that produces this $\alpha_i^{(k+1)}$ value is easy.

$$\lambda^{(k+1)}=\sqrt{\frac{a_i^2 p_i^2}{\alpha_i^{(k+1)}}}-a_i^2 \qquad (5.1.8)$$

While this iteration would produce convergence, at times the convergence rate was quite slow. Correct choice of α_i would, however, always lead to a convergent solution.

As the algorithm progresses $S(\lambda)$ converges asymptotically to 1 as λ converges to λ^*. This makes the value of $S(\lambda)$ an excellent termination test. The ratio of the convergences was observed to be highly consistent. This convergence would be monitored over three cycles of the iteration, the rate of the change in λ to change in $S(\lambda)$ would be determined, and the final value λ^* estimated using the infinite series sum. This 'ratio accelerator' produced a dramatic increase in the speed of convergence, at the cost of needing stability checking each time it was used.

The source code for the final algorithm, 'pptes.m', may be found in Appendix A.

Tests of pptes.m were done with random inputs

$$\mathbf{a}=\begin{bmatrix}\{1:100\}\\ 1\\ \{0.1:1\}\end{bmatrix},\ \mathbf{p}=\begin{bmatrix}\{-100:100\}\\ \{-100:100\}\\ \{-100:100\}\end{bmatrix} \qquad (5.1.9)$$

The average flops cost over 1000 trials was 236.74. This is a substantial improvement over the NEARPT algorithm, approximately a factor of 3.

5.2 Ellipsoid Distance via Point to Ellipsoid Iteration

A simple way to solve the ellipsoid distance problem is to iterate the point to ellipsoid solution. Due to the convex nature of the bodies, this will always converge toward the result.

The algorithm is simple. Pick a starting point on ellipsoid 1, typically the center. Call this $\mathbf{x}_1^{(0)}$. Iterate as follows:

1) Given ellipsoid parameters and k_{max}. Set k=0, choose initial $\mathbf{x}_1^{(0)}$.
2) Find closest point on ellipsoid 2 to $\mathbf{x}_1^{(k)}$, call this $\mathbf{x}_2^{(k+1)}$.
3) Find the closest point on ellipsoid 1 to $\mathbf{x}_2^{(k+1)}$, call this $\mathbf{x}_1^{(k+1)}$.
4) Increment k. If $k \leq k_{max}$, go to step 2.
5) Terminate with $\mathbf{x}_1^{(k)}$ and $\mathbf{x}_2^{(k)}$ as the solution.

A better termination condition than total cycle count is desirable. Testing the distance between $\mathbf{x}_1^{(k)}$ and $\mathbf{x}_1^{(k+1)}$ is not effective if the convergence rate is very slow, as tends to happen when the ellipsoids are very close together. The issue of convergence rate is in fact the main difficulty with this algorithm.

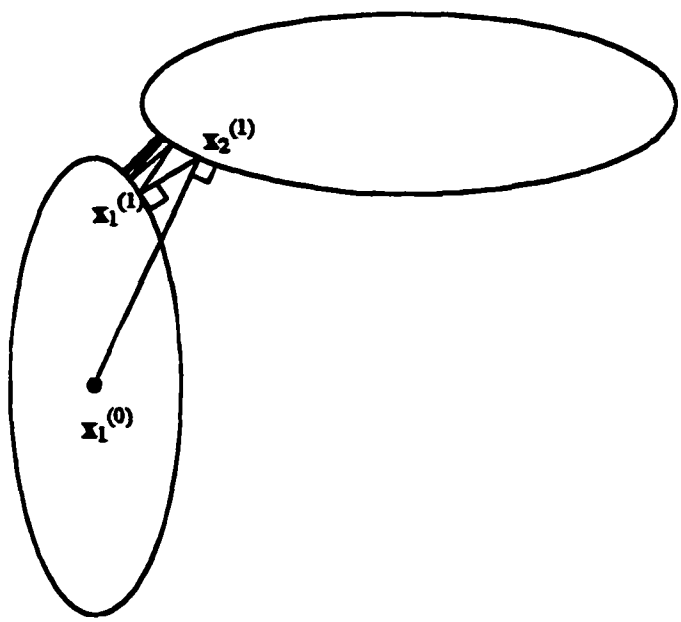


Figure 5.2 – Progress of point to ellipsoid iteration

Figure 5.2 illustrates the journey undertaken by the algorithm. The convergence rate slows dramatically as the solution point is approached, and the flatter the ellipses are at the solution point, the slower the convergence will be.

The algorithm is pretty cheap, however, and iterating until no further changes are seen gives a good measure of the solution of the ellipsoid distance problem. Having an easily determined gold standard for the problem solution makes the development of the following methods a lot easier to crosscheck. This method also provides a nice lower bound, a target to shoot for in terms of distance computation cost.

This algorithm was implemented in Matlab, and its source can be found in 'eepps.m' in Appendix A. The standard test use for all the algorithms is the 1:2:5 crossed ellipsoid test. The input parameters are $\mathbf{a}_1=[1\ 2\ 5]^T$, $\mathbf{a}_2=[5\ 2\ 1]^T$, $\mathbf{p}=[4\ 4\ 4]^T$. Running the algorithm for 100 iterations is more than enough to ensure convergence. The error drops to zero after 51 iterations. The error on the k^{th} iteration is measured with respect to the $\mathbf{x}^*$ produced after the iteration converges.

$$e^{(k)} = \left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \tag{5.2.1}$$

The result of the standard 1:2:5 test was

$$\mathbf{x}^* = \begin{bmatrix} 0.13705777781256 \\ 1.43051641611194 \\ 3.42644444486425 \\ 0.57355555513575 \\ 2.56948358388806 \\ 3.86294222218744 \end{bmatrix} \tag{5.2.2}$$

Convergence to an error less than 10^{-4} happens after 14 iterations, or 6818 flops.
Convergence to an error less than 10^{-8} happens after 21 iterations, or 10227 flops.

If the second ellipse is displaced along the solution vector to a distance of 0.1 units from the first, the convergences become much worse.

Convergence to an error less than 10^{-4} happens after 78 iterations, or 37693 flops.
Convergence to an error less than 10^{-8} happens after 163 iterations, or 79088 flops.

Obviously, there is room for improvement here.

5.3 Ellipsoid Distance via DQP3 on Polyhedral Approximation

Another possible approach is to solving the ellipsoid distance problem is to approximate the ellipsoids with sufficiently high resolution polyhedra, then solve with a convex polyhedral distance problem algorithm such as DQP3. It turns out that generating the polyhedral enclosure of an ellipsoid is fairly simple. The most important issue is the number of facets required to get a good enough coverage of the ellipsoid to meet the error requirements.

The facet-balls used in Section 4 for testing were generated by a Matlab program called 'genball.m' (see Appendix A). It generates the facet normals, A for a set of vectors more or less evenly spanning the sphere. The 'order' of the facet-ball is the number of rings of facets from top to bottom, counting the top but not the bottom. A facet-ball of order 4 has 8 facets around the equator, etc. The facet distances are all 1, making generating the **b** vector very easy.

To generate an ellipsoid with this code, only the **b** vector needs to be scaled. This is handled for a full ellipse by 'b2ell.m', or for a single facet by 'elldn.m'. To compute the distance of an ellipsoid facet, it suffices to find the point $\mathbf{x}_n$ that corresponds to the ellipsoid normal, **n**. The dot product of these two is the facet distance b_i.

The ellipsoid normal of $\mathbf{x}_n$ of an ellipsoid defined by its axes **a** is

$$\mathbf{n} = \left[\frac{x_{n1}}{a_1^2} \quad \frac{x_{n2}}{a_2^2} \quad \frac{x_{n3}}{a_3^2} \right]^T \tag{5.3.1}$$

Given that we already have a normalized **n**, the distance $\mathbf{n}^T\mathbf{x}_n$ simplifies to

$$d = \sqrt{\sum_{i=1}^{3} n_i^2 a_i^2} \tag{5.3.2}$$

Polyhedral ellipsoid models generated by the above method provide an interesting set of objects for use with DQP3 as a performance measure. The problem with the spherical

test shapes typically used here and elsewhere is that using the centroid as a starting point makes the solution much easier. The solution point, after all, lies very near the line between the two centroids. However, with the ellipsoid models it is easy to construct cases where the solution lies far away from the centroid line, making the algorithm work for its result.

The error caused by the tiling of the polyhedral model is largely due to the fact that the polyhedral distance algorithms tend to converge on the vertices. Starting with that assumption, one can see that the best error margin one can hope for is half the facet width. Since the facets must tile the two-dimensional ellipsoid surface, it is obvious that the facet width is approximately proportional to the inverse square of the facet count.

Assuming an even tiling, to get an error factor of 0.1% of the ellipsoid's dimensions, one would need a tiling of facets that can produce an equatorial band of 1000 facets. That would be a facet-ball of order 500, which would have approximately 318,000 facets. This is far beyond the values tested for the DQP3 algorithm, and would take prohibitive amounts of time. The best this algorithm can hope for is a coarse result on a moderately dense tiling. This analysis assumes that one is interested in the closest point on the ellipsoid, not the distance approximation.

If only the error in the distance measure is of interest, the results are better. The distance from the vertex to the surface of an ellipsoid (or the unit sphere for simplicity of argument) is proportional to $\sqrt{r-w^2}-1$ where r is the local radius, and w is half the width of the facet. For small w, this is approximately $0.5w^2$. Thus for the 3D case the error is inversely proportional to the facet count. Even so, Table 5.1 illustrates how expensive even a modest distance error is in facet count and computation time.

While having adjacent facet information would make the DQP3 algorithm more efficient, it is doubtful that it would be able to handle millions of facets sufficiently efficiently.

Still it is interesting to test the DQP3 solutions on this problem, to see how changing the target object shape affects the results.

A series of tests were run in both Matlab and in complied C to see how well DQP3 would handle ellipsoid models. The standard 1:2:5 problem far from contact was used. The results were mixed, especially as the C code started to exhibit some instabilities with the ellipsoid models, necessitating the tweaking of some of the error thresholds.

order	N_1+N_2	x error	d error	flops	time (μs) non-tracking	time (μs) tracking
4	44	0.651	0.564	4231	205.6	36.9
6	92	0.417	0.301	11817	396.8	41.1
8	164	0.242	0.0961	28058	790	61.7
10	256	0.123	0.0338	27319	1130	84.5
15	576	0.148	0.00395	108940	3160	227
20	1020	0.127	0.00843	271718		
30	2296	0.0478	0.00440	923458		

Table 5.1 – Performance results for DQP3 on ellipsoid models

The results in Table 5.1 are less than stellar. The flops counts for even the smallest models far exceed those seen in the point to ellipse iteration, and the error thresholds due to facet size are quite dreadful. The slower C code times are largely due to the increase in the number of iterations required. The number of 'alpha' evaluations per trajectory with the ellipsoids is nearly double that of the spheroid case.

The above times suggest that DQP3 on ellipsoids models of approximately 300 facets each would take the same time to compute as Rimon and Boyd's result. It is undeniable that the error would be rather worse.

The only strong point in the result is that when the 0.1 unit clearance case is tested, the flops counts do not rise significantly. Thus the solution is close to constant time as the distance between the ellipses varies, a very useful result. Note the inverse-linear relationship between the d error and the facet count.

No special code is required, the standard DQP3 algorithms are used, on problem sets generated by 'genball.m' and 'b2ell.m'.

5.4 Ellipsoid Distance via Iterative Polyhedral Refinement

Since the facet densities to get a good ellipsoid distance estimate from the DQP3 algorithm are prohibitively high, it would be advantageous to only have high density in the area that needed it. The iterative polyhedral refinement method attempts to do just that, by starting with a very simple representation of the ellipsoid, and adding facets only as they are needed to improve the model.

Start with a pair of ellipsoids represented by their axis vectors $\mathbf{a}_1$ and $\mathbf{a}_2$, and a position vector $\mathbf{p}$ for the second ellipse. (Relative rotation can be handled with the dqpt.c algorithm, and can be neglected here.) Generate the initial ellipsoid models as 6-facet rectangular prisms, with axis lengths corresponding to the ellipsoid axes.

$$\mathbf{A}_1 = \begin{bmatrix} \mathbf{I}_3 & -\mathbf{I}_3 \end{bmatrix} \mathbf{b}_1 = \begin{bmatrix} -\mathbf{a}_1 \\ -\mathbf{a}_1 \end{bmatrix}, \mathbf{A}_2 = \begin{bmatrix} \mathbf{I}_3 & -\mathbf{I}_3 \end{bmatrix} \mathbf{b}_2 = \begin{bmatrix} -\mathbf{a}_2 \\ -\mathbf{a}_2 \end{bmatrix} \qquad (5.4.1)$$

1) Initialize $\mathbf{A}_1$, $\mathbf{b}_1$, $\mathbf{A}_2$ and $\mathbf{b}_2$. Set k=0.
2) Compute the distance between the polyhedra with DQP3.
3) Compute the closest point, $\mathbf{x}_{p1}$ on ellipse $\mathbf{a}_1$ from the point found on object 1.
4) Add a facet to $\mathbf{A}_1$ with normal equal to the negative ellipse normal at $\mathbf{x}_{p1}$.
5) Add the corresponding facet distance to $\mathbf{b}_1$ by using equation (5.3.2).
6) Add several more facets near $\mathbf{x}_{p1}$, and their facet distances.
7) Discard excess facets from $\mathbf{A}_1$ and $\mathbf{b}_1$ to keep the total count reasonable.
8) Repeat steps 3-7 for object 2.
9) Increment k. If $k \geq k_{max}$, terminate with $[\mathbf{x}_{p1}{}^T \mathbf{x}_{p2}{}^T]^T$ as the solution.
10) Go to step 2.

In the interest of accelerating the algorithm clusters of facets are added instead of one at a time, as adding more facets is cheap compared to the cost of an iteration. The first estimate as to where to put a second facet is at the point with a normal equal to the current distance vector between the two ellipsoids ($\mathbf{x}_{p2}$-$\mathbf{x}_{p1}$). This can be computed quite cheaply, and is seen as something of a worst case – if it were a point to ellipse problem

the final solution would be somewhere roughly between these two locations. Of course, since both points are moving this estimate is poor.

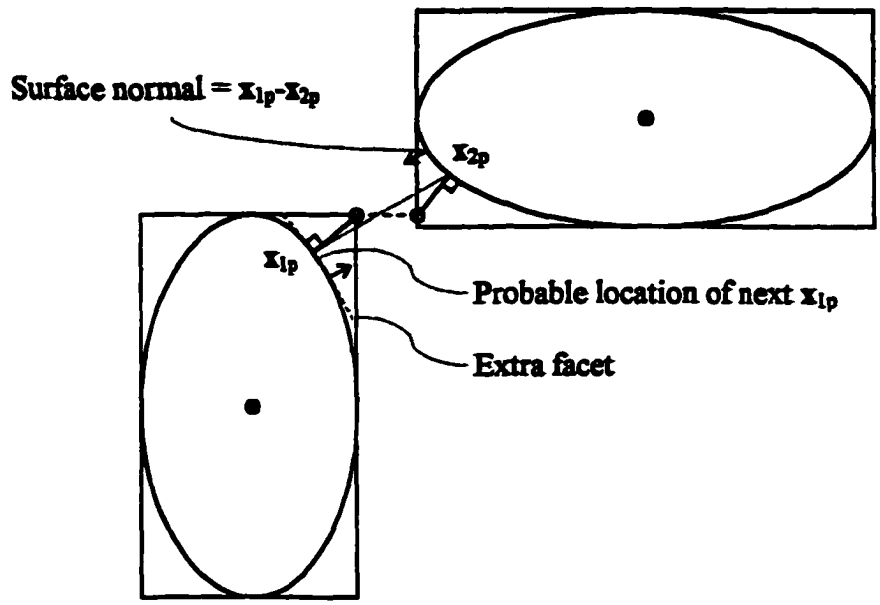


Figure 5.3 – First pass of Iterative Polyhedral Refinement

Figure 5.4 illustrates the progress of the first iteration of the algorithm, showing where the first DQP3 evaluation ends (at the open circles), the surface points that get projected, and the facets generated by those points and the surface normal of x_{1p}-x_{2p}. Without that extra facet, the next x_{1p} in the iteration would probably be at the top end of the new facet. The new facet greatly improves the results of the next iteration with quite a modest computational cost.

It is inefficient to carry too many facets, so old facets that haven't been used in a few iterations should be discarded as no longer relevant. To ensure that the algorithm doesn't blow up, it is best to keep the first 6 facets, and obviously the active set from the previous iteration should be kept as well. Pick the most recently generated of the remainder to fill out the quota.

Note that the adjacent facet list would not be useful to accelerate this method, as it would need to be regenerated every cycle. It is better to just keep the total facet count low enough that the DQP3 algorithm runs quickly.

Two different algorithms were tested, varying only on how they added auxiliary facets. The first adds a total of 4 facets each cycle, one at x_{lp}, one at the normal, and two more evenly spaced around the circle that has x_{lp} as its center and includes the normal. The second adds another three facets in a second ring around this first one. The Matlab programs implementing these two cases are 'eedist42.m' and 'eedist7.m', both of which can be found in Appendix A.

The two different approaches were tested on the standard 1:2:5 problem at distance 4. Convergence beyond an error of 10^{-7} was erratic at best, and ran into an instability in the DQP3 algorithm that caused it to start cycling. It is possible that the facet constraints became somewhat inconsistent, or at the least some of the facets could have become too parallel for the resolution of DQP3.

Convergence to an error of 10^{-4} took 21 cycles of eedist42.m, but only 15 cycles of eedist7.m. The flops counts were 100107 and 62943, respectively. This is about 10 times slower than the point to ellipse iteration. Convergence to 10^{-7} took 41 cycles, and 183083 and 184980 flops respectively. The convergence was mostly a smooth exponential decay, with the occasional bobble upwards for unknown reasons.

Tests on the 0.1 unit clearance case ran into the instability problems rather more quickly, but the results gathered suggested a similar convergence rate to the previous, at a similar total flops cost, showing no tendency to increased cost for a given error tolerance.

If the issues with instability and ultimate convergence can be worked out, and perhaps some improvements in efficiency added (such as passing starting active set to the DQP3 algorithm, not just the starting x), this algorithm could show some distinct promise.

5.5 Other Possibilities

During the course of development of the above methods, a few other ideas came up. Many attempts were made to reduce the ellipsoid problem to a simpler form through a variety of projections. A curious third-order relationship between alpha and lambda was explored, and while interesting, did not become a viable algorithm. One of the more interesting directions considered modifying the DQP3 algorithm to find the closest point between a polyhedron and an ellipsoid. The hope was that the polyhedron could then be iteratively refined in a way similar to the previous section. A more promising approach was to hybridize the DQP3 solution with the point to ellipsoid solution, using the former to get a good coarse estimate, then refining it with the latter.

5.5.1 Polyhedron to Ellipsoid DQP3 Analogue

Much like the DQP3 algorithm, the polyhedron to ellipsoid closest point algorithm would use a QP core to search over the polyhedron, using specialized point to ellipse (already done above), line to ellipse, and facet to ellipse computations to handle the 'delta' step.

Facet to ellipsoid is quite simple. Take the facet normal, find the point on the ellipsoid with that normal, and you're done.

Line to ellipsoid is nontrivial. A method was developed to project the line to ellipsoid problem into a plane perpendicular to the line, making it a 2D ellipse to point problem. However, despite repeated attempts, no solution to the ellipse to point problem was found that did not involve solving a fourth order polynomial. The point to ellipsoid method from section 5.1 was written generally enough to handle n-dimensional ellipsoids (where $n \geq 2$). This allows an efficient solution of the line to ellipsoid case.

Start with an infinite line in direction $\mathbf{l}_1$ from point $\mathbf{x}_1$, and an axis-oriented ellipsoid at the origin defined by its axis half-lengths $\mathbf{a}$. Transforming this distance problem into a point to ellipse problem starts easily enough. We note that the closest point to the line will have a surface normal perpendicular to that line. If we take the 1-dimensional set of normals that are perpendicular to $\mathbf{l}_1$, then the mapping from normals to ellipsoid surface

points will define an ellipse, and a point on that ellipse will be the closest point to the line. This ellipse can then be projected into the plane perpendicular to the line, and a 2D point to ellipse problem is the result. Once that is solved, reverse the projection to find the closest point on the ellipsoid, then project that onto the line to get the other half of the solution. All that needs to be done is find the parameterization of the ellipse in the plane.

Two points $\mathbf{x}_a$ and $\mathbf{x}_b$ on the surface of the ellipsoid can be used to parametrically define an ellipse $\mathbf{f}(\theta)$ on that surface.

$$\mathbf{f}(\theta) = \mathbf{x}_a \sin(\theta) + \mathbf{x}_b \cos(\theta) \tag{5.5.1}$$

While $\mathbf{x}_a$ may be an arbitrary surface point on the ellipse, the $\mathbf{x}_b$ surface point must be restricted so that the ellipse equation holds.

$$\sum_{i=1}^{3} \frac{(x_{ai} \sin(\theta) + x_{bi} \cos(\theta))^2}{a_i^2} = 1 \tag{5.5.2}$$

Multiplying this out the $\sin^2$ and $\cos^2$ terms cancel to 1, which means that

$$\mathbf{x}_a^T \begin{bmatrix} x_{b1}/a_1^2 \\ x_{b2}/a_2^2 \\ x_{b3}/a_3^2 \end{bmatrix} = \mathbf{x}_a^T \mathbf{n}_b = 0 \tag{5.5.3}$$

In other words, $\mathbf{x}_b$ must a point whose normal, $\mathbf{n}_b$ is perpendicular to $\mathbf{x}_a$.

If we choose $\mathbf{x}_a$ as the surface point with a normal equal to the projection of $\mathbf{x}_1$ (the line origin) onto the $\mathbf{l}_1$ plane, this provides us a convenient origin. Call $\mathbf{n}_a$ the normal at $\mathbf{x}_a$. Thus $\mathbf{n}_b$ must be perpendicular to $\mathbf{x}_a$.

Going back to the original problem, we wish to create a parameterization with xa and xb such that all the points generated have normals perpendicular to $\mathbf{l}_1$. To do that, the

normals of $\mathbf{x}_a$ and $\mathbf{x}_b$, $\mathbf{n}_a$ and $\mathbf{n}_b$, must be perpendicular to $\mathbf{l}_1$. We have already chosen a suitable $\mathbf{x}_a$. Since $\mathbf{n}_b$ must be perpendicular to both $\mathbf{x}_a$ and $\mathbf{l}_1$,

$$\mathbf{n}_b = \mathbf{x}_a \times \mathbf{l}_1 \qquad (5.5.4)$$

The rest is a matter of projecting $\mathbf{x}_a$ and $\mathbf{x}_b$ onto the $\mathbf{l}_1$ plane, finding the major and minor axes of the parameterized ellipse, and then solving the 2D point-ellipse problem. Since the 'pptes.m' code was generalized to *n* dimensions, it will prove handy here.

While this method does not directly solve the ellipsoid to ellipsoid problem, it does nicely bridge the gap between polyhedral representations and ellipsoid ones. The example of a polyhedral robot in a workspace with some ellipsoids illustrates the direct usefulness of such an algorithm. In addition, the iterative refinement approach could be used, refining only one ellipsoid model, instead of both in parallel.

This method was never fully implemented, but may be tested in the future.

5.5.2 DQP3 Hybrid with Point to Ellipsoid

Using a high facet count model of an ellipsoid with the DQP3 algorithm provides a moderately accurate estimate of the closest point between a pair of ellipsoids. It is a fairly obvious extension to use the point to ellipse iteration to refine that estimate further.

The difficulty comes in balancing the cost of various degrees of polyhedral complexity against the amount of effort the subsequent iterative refinement will require. This decision is further complicated by the possibility of using tracking data from cycle to cycle, which makes the DQP3 code substantially more efficient. The future addition of adjacent facet lists will shift the balance point even further towards high counts.

This method is not terribly effective, however. On the standard problem with 0.1 unit separation, using a 6^{th} order facet-ball costs 11817 flops to solve. It saves 6 steps in the iteration to 10^{-4} error, from 78 iterations to 72, saving 2680 flops in the pptes.m convergence, for a net loss of 9137 flops. This is not worth pursuing.

5.6 Comparison of Ellipsoid Distance Computations

Only one result from the literature was found, that of Rimon and Boyd [Rimon97], and they didn't produce an actual Cartesian distance. Their 'free margin function' could be computed in 2.5 ms from an uninitialized start, or 1 ms if tracking moving objects. (DEC5000, Linpack: 5.3 MFlops). The naïve calculation of Chapter 4 based on spherical objects suggests that these times are equivalent to DQP3 processing of two 160 facet polyhedra, or two 650 facet polyhedra for the tracking case. More realistic calculations using polyhedral ellipsoid models suggest figures about half of that value.

The error in $\mathbf{x}$ from the ellipsoid models is quite high, though the error in the distance measure d is somewhat reasonable for modestly large facet counts, as it scales inversely to the number of facets, instead of the square root.

The point to ellipsoid iteration was the most reliable algorithm for the ellipsoid distance problem. Its cost at modest distances were typically under 10000 flops. The scaling with respect to object separation very poor – see Table 5.2. An algorithm that becomes more expensive the closer the objects are to touching is highly undesirable for control work.

Examining Table 5.2 we can see that the iterative polyhedral refinement methods did improve on the consistency of cost, but were an order of magnitude more computationally intensive than the point to ellipsoid.

Method	Separation (units)				Error Tolerance
	0.1	0.5	1.2955	5	
point-to-ellipsoid iteration	37693	11688	6818	3422	$\mathbf{x}_{err}<10^{-4}$
polyhedral model (288 facets)	119034	121038	108940	62899	$\mathbf{d}_{err}<4\times10^{-3}$
iterative polyhedral (4 facets)	X	86651	100107	68300	$\mathbf{x}_{err}<10^{-4}$
iterative polyhedral (7 facets)	X	X	62943	86534	$\mathbf{x}_{err}<10^{-4}$
hybrid (Section 5.5.2)	46830				$\mathbf{x}_{err}<10^{-4}$

Table 5.2 – Comparison of separation scaling for proposed algorithms

6. Conclusions and Future Work

A QP based algorithm (DQP3) for computing the minimum distance between convex polyhedra was presented, using geometrically derived computations to replace the standard ones for greater efficiency. Several variations of the QP part of the DQP3 algorithm were examined. The variants proved either unstable, or not consistently faster than the baseline DQP3 algorithm and were thus discarded. A method of handling translation and rotation of the objects intrinsically within the DQP3 algorithm was added, and performance results with respect to the primary algorithm were promising, when the savings in object transformation time were considered.

Direct tests against the most recent algorithm from the literature, Cameron's Enhanced GJK algorithm [Cameron97a] were run, with mixed results. The DQP3 algorithm is 25%-40% faster during tracking, better exploiting the coherence of the tracking problem, but only for total facet counts above 120. It is significantly (up to 50%) slower on small problems where tracking is not applicable, and substantially slower (270%) on large problems. However, it is felt that the tracking results are the most relevant to the intended application in robotics, so the DQP3 algorithm can be considered a success.

Many of the examined methods used feature adjacency lists to improve their speed. Adding such a method to the DQP3 algorithm is feasible, and an algorithm to generate the list is available. The results of such a test could be expected to follow the pattern seen in the literature of approximately O(1) time in facet count for tracking problems.

The ellipsoid distance problem was considered next, and a variety of methods were proposed to handle it. Iterating a point to ellipse distance algorithm works well, converging consistently to a result. The convergence rate can be slow, however. Using DQP3 on a high facet count polyhedral model of the ellipsoid can work for coarse results, but for any sort of fine error tolerance the facet count needs to be quite high, making the cost prohibitive. Iteratively refining a polyhedral model of the ellipsoid by adding facets where they are most needed worked quite well, to a point. The convergence while smooth, ran into some instability issues at error thresholds around 10^{-7} and ceased to

converge. The method by which additional facets are added could use more work to address this.

In the end, the point to ellipsoid iteration was the most reliable algorithm, but did not scale well with object separation. The iterative refinement algorithm shows excellent scaling with separation, but is substantially more expensive.

6.1 Future work

- Enhancing the DQP3 algorithm with adjacent facet data, and comparison to others in the field.
- Optimization of the DQP3 core, improving lambda calculation. Re-examine the various projections to try to find more efficient ways of computing them.
- Examine stability considerations within the DQP3 algorithm, finding ways to improve the result other than *a posteriori*.
- Polyhedron to ellipsoid algorithm, and comparison to other results with and without iterative refinement.
- Examine other criteria for adding facets to the iterative ellipsoid approximation method, such as halving the distance between the current solution facet and the facets of the current active set.
- Application of the ratio accelerator concept to the point to ellipse iteration.
- Build an integrated system that can handle a collection of polyhedra, and attempt an analogue of the McIntyre collision detection model. Implement the algorithm of section 4.6, including the 'triage' aspects.
- Use the above model in an integrated redundant robot path planning example.
- A distant possibility is an algorithm to determine the point of closest approach between two polyhedra, given a starting and ending position, without iterating through the positions in between. A reliable lower bound on the closest distance using only the two sets of active set data seems in principle possible. The translation only case is simplest, but the effects of rotation could be modeled as well.

References

[Ahuactzin99] J. M. Ahuactzin, K. K. Gupta, "The Kinematic Roadmap: A Motion Planning Based Global Approach for Inverse Kinematics of Redundant Robots," *IEEE Transactions on Robotics and Automation*, 15(4), 653-669, 1999.

[Antoniou02] A. Antoniou, W.-S. Lu, *Optimization: Methods, Algorithms and Applications*, New York: Kluwer Academic, 2002. (Preprint)

[Arvo89] J. Arvo, D. Kirk, "A survey of ray tracing acceleration techniques," *An Introduction to Ray Tracing*, 201-262, 1989.

[Bajaj92] C. Bajaj, T. Dey, "Convex decomposition of polyhedra and robustness," *SIAM J. Comput.*, 21(2), 339-364, 1992.

[Barequet96] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, A. Tal, "Box-tree: A hierarchical representation of surfaces in 3D," *Proc. of Eurographics '96*, 1996.

[Bobrow89] J. E. Bobrow, "A Direct Minimization Approach for Obtaining the Distance between Convex Polyhedra," *International Journal of Robotics Research*, 8(3), 65-76, 1989.

[Cameron91] S. Cameron, "Approximation hierarchies and s-bounds," *Proc. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, 129-137, 1991.

[Cameron97a] S. Cameron, "Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra," *Proc. 1997 IEEE Int. Conf. on Robotics and Automation*, 3112-7, 1997.

[Cameron97b] S. Cameron, "A Comparison of Two Fast Algorithms for Computing the Distance Between Convex Polyhedra," *IEEE Transactions on Robotics and Automation*, 13(6), 915-920, 1997.

[Cameron01] S. Cameron, J. Pitt-Francis, "Using OxSim for Path Planning," *Journal of Robotic Systems*, 18(8), 421-431, 2001.

[Canny87] J. Canny, *The Complexity of Robot Motion Planning*, Cambridge MA: MIT Press, 1987.

[Chazelle84] B. Chazelle, "Convex partitions of polyhedra: A lower bound and a worst-case optimal algorithm," *SIAM J. Comput.*, 13, 488-507, 1984.

[Chen93] F.-T. Chen, T.-H. Chen, Y.-S. Wang, Y.-Y. Sun, "Obstacle Avoidance for Redundant Manipulators Using the Compact QP Method," *Proc. 1993 IEEE Conference on Robotics and Automation*, v.3, 113-119, 1993.

[Chen98] P. C. Chen, Y. K. Hwang, "SANDROS: A Dynamic Graph Search Algorithm for Motion Planning," IEEE Transactions on Robotics and Automation, 14(3), 390-403, 1998.

[Chuang98] J.-H. Chuang, "Potential-Based Modeling of Three-Dimensional Workspace for Obstacle Avoidance," IEEE Transactions on Robotics and Automation, 14(5), 778-785, 1998.

[Edelsbrunner83] H. Edelsbrunner, "A new approach to rectangle intersections," *International Journal of Computational Mathematics*, 13, 209-219, 1983.

[Farin93] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, Academic Press Inc., 1993.

[Fenger93] D. K. Fenger, *Global Resolution of Kinematic Redundancy Using Parameterized Self-Motion Optimization*, M.A.Sc Thesis, Electrical Engineering Department, University of Victoria, 1993.

[Fletcher87] R. Fletcher, *Practical Methods of Optimization*, 2nd ed, Chichester: John Wiley & Sons Ltd., 1987.

[Fuller75] R. B. Fuller, E. J. Applewhite, *Synergetics: Explorations in the Geometry of Thinking*, New York: Macmillan, 1975.

[Gilbert88] E. G. Gilbert, D. W. Johnson, S. S. Keerthi, "A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space," *IEEE Journal of Robotics and Automation*, 4(2), 193-203, 1988.

[Gilbert90] E. G. Gilbert, C.-P. Foo, "Computing the Distance Between General Convex Objects in Three-Dimensional Space," *IEEE Transactions on Robotics and Automation*, 6(1), 53-61, 1990.

[Glass93] K. Glass, R. Colbaugh, D. Lim, H. Seraji, "On-Line Collision Avoidance for Redundant Manipulators," *Proc. 1993 IEEE Conference on Robotics and Automation*, v.1, 36-43, 1993.

[Goldfarb72] D. Goldfarb, "Extensions of Newton's method and simplex methods for solving quadratic programs," in *Numerical Methods for Nonlinear Optimization* (Ed. F. A. Lootsma), London: Academic Press, 1972.

[Gottschalk96] S. Gottschalk, M. C. Lin, D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," *Computer Graphics (SIGGRAPH '96 Proceedings)*, 171-180, 1996.

[Guibas99] L. J. Guibas, D. Hsu, L. Zhang, "A Hierarchical Method for Real-Time Distance Computation among Moving Convex Bodies," *Computational Geometry: Theory and Applications, Special Issue on Virtual Reality*, 15 (1-3), 51-68, 2000.

[Hopcroft83] J. E. Hopcroft, J. T. Schwartz, M. Sharir, "Efficient detection of intersections among spheres," *International Journal of Robotics Research*, 2(4), 77-80, 1983.

[Hubbard93] P. M. Hubbard, "Interactive collision detection," *Proc. IEEE Symposium on Research Frontiers in Virtual Reality*, 1993.

[Jiménez01] P. Jiménez, F. Thomas, C. Torras, "3D Collision Detection: A Survey," *Computers and Graphics*, 25(2), 269-285, 2001.

[Kalaycioglu93] S. Kalaycioglu, M. Tandirici, D. S. Necsulescu, "Real-Time Collision Avoidance of Robot Manipulators for Unstructured Environments," *Proc. 1993 IEEE Conference on Robotics and Automation*, v.1, 44-51, 1993.

[Khadem88] S. E. Khadem, R. Dubey, "A Global Redundant Robot Control Scheme for Obstacle Avoidance," *Pro. 1988 IEEE Southeastcon*, 397-402, 1998.

[Klosowski96] J. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *SIGGRAPH'96 Visual Proceedings*, p151, 1996.

[Krishnan98] S. Krishnan, A. Pattekar, M. Lin, D. Manocha, "Spherical shell: A higher order bounding volume for fast proximity queries," *Proc. of Third International Workshop on Algorithmic Foundations of Robotics*, 1998.

[Lin91] M. C. Lin, J. F. Canny, "A Fast Algorithm for Incremental Distance Calculation," *Proc. 1991 IEEE Int. Conf. on Robotics and Automation*, 1008-14, 1991.

[Lin96] M. C. Lin, D. Manocha, J. Cohen, S. Gotschalk, "Collision Detection: Algorithms and Applications," *Algorithms for Robotics Motion and Manipulation*, (eds. Jean-Paul Laumond and M. Overmars, A.K. Peters), 129-142, 1996.

[Lin98] M. C. Lin, S. Gottschalk, "Collision detection between geometric models: a survey," *IMA Conference on Mathematics of Surfaces*, 602-8, 1998.

[Lozano79] T. Lozano-Pérez, M. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *Comm. ACM*, 22(10), 560-570, 1979.

[Ma92] O. Ma, M. Nahon, "A General Method for Computing the Distance Between Two Moving Objects Using Optimization Techniques," *Proc. 1992 ASME Design Automation Conf.*, 1992.

[McIntyre92] L. McIntyre, M. Nahon, "Implementation of a Collision Detection Algorithm for Robotic Manipulators," *Proc. 7th CASI Conf. on Astronautics*, 1992.

[McLean93] A. McLean, S. Cameron, "Snake-Based Path Planning for Redundant Manipulators," *Proc. 1993 IEEE Conference on Robotics and Automation*, v.2, 275-282, 1993.

[Nakamura91] Y. Nakamura, *Advanced Robotics: Redundancy and Optimization*, Reading MA: Addison-Wesley, 1991.

[Overmars92] M. H. Overmars, "Point location in fat subdivisions," *Inform. Proc. Lett.*, 44, 261-265, 1992.

[Quinlan94] S. Quinlan, "Efficient distance computation between non-convex objects," *Proc. IEEE Int. Conf. on Robotics and Automation*, 3324-3329, 1994

[Rimon97] E. Rimon, S. P. Boyd, "Obstacle Collision Avoidance Using Best Ellipsoid Fit," *Journal of Intelligent and Robotic Systems*, 18(2), 105-126, 1997

[Sato96] Y. Sato et al, "Efficient Collision Detection using Fast Distance-Calculation Algorithms for Convex and Non-Convex Objects," *Proc. 1996 IEEE Int. Conf. on Robotics and Automation*, 771-8, 1996.

[Seraji1999] H. Seraji, "Real-Time Collision Avoidance for Position-Controlled Manipulators," *IEEE Transactions on Robotics and Automation*, 15(4), 1999.

[Turk89] G. Turk, *Interactive collision detection for molecular graphics*, Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1989.

[Yoshikawa90] T. Yoshikawa, *Foundations of Robotics*, Cambridge, MA: The MIT Press, 1990.

[Zchal92] H. Zchal, R. V. Dubey, J. A. Euler, "Collision Avoidance of a Multiple Degree of Redundancy Manipulator Operating Through a Window," *Journal of Dynamic Systems, Measurement and Control*, v.114, 717-721, 1992.

[Zeghloul92] S. Zeghloul, P. Rambeaud, J. P. Lallemand, "A fast distance calculation between convex objects by optimization approach," *Proc. IEEE Int. Conf on Robotics and Automation* (Nice, May 1992) 3, 2520-25, 1992.

Appendix A – Source Code

Matlab files:

dqp3l.m	DQP3 solution
dsolv.m	delta step of DQP3
facet.m	build adjacency list (and plotting data) for one facet.
genlambda.m	Calculate lambda step.
genball.m	Generate a facet-ball of order *n*.

Matlab ellipse code:

b2ell.m	Convert facet-ball to ellipse by changing b.
elldn.m	Generate facet distance (b) for an ellipse, given the normal.
pptes.m	point to ellipse code
eepps.m	Ellipse distance by iterating point-to-ellipse code.
eedist42.m	Ellipse distance by refining polyhedral estimate (+4 facets/cycle).
eedist7.m	Ellipse distance by refining polyhedral estimate (+7 facets/cycle).

C code:

dqpt.c	DQP3 with translation and rotation (Cleaned up so it's <30 pages.)
dsolvqpe.c	Delta solver for DQP3. (Clean up the code. No need for MEX stuff.)

Matlab file: dqp3l.m

```
% dqp3.m - David's Quadratic Programming algm.
%         derived from Assignment #4, ELEC 603
%
%       - based on Active set method, and some insights from class
%       - '2' - maintains two separate Active sets of constraints, one
%         for the first 3 variables, one for the last 3.  This is specifically
%         optimized for the distance-measurement calculation.
%       - '3' - Uses a physical-problem inspired algorithm for the core, equality
%          QP problem.
%       - '1' - uses genlambda to get lambda, not a simplistic calculation.

function [z,Act1,Act2]=dqp3(G,q,A1,b1,A2,b2,x0)

epsilon = 1e-12;        % Cutoff for 'zero' delta
epsLambda = 1e-12;
epsAlpha = 1e-10;      % Cutoff for 'zero' error in Ax-b>0
n=length(x0);
if n ~= 6,
  error('Invalid starting vector size - must be 6x1');
end

% Move x0(1:3) to a feasible location
for i1=1:4,
  r1 = A1'*x0(1:3)-b1;
  [minr,imin]=min(r1);
  if (minr>=-epsAlpha | i1>3)       % Reduce recomputations required
    break
  end
  % Move x0 so that it *is* feasible on this specific constraint
  x0(1:3)=x0(1:3)-minr*A1(:,imin);
end %for i1
if minr < -epsAlpha,
  % Oh well.
  x0(1:3)
  error('x0(1:3) is not a feasible point')
end

% Move x0(4:6) to a feasible location
for i1=1:4,
  r2 = A2'*x0(4:6)-b2;
  [minr,imin]=min(r2);
  if (minr>=-epsAlpha | i1>3)       % Reduce recomputations required
    break
  end
  % Move x0 so that it *is* feasible on this specific constraint
  x0(4:6)=x0(4:6)-minr*A2(:,imin);
end %for i1
if minr < -epsAlpha,
  % Oh well.
  x0(4:6)
  error('x0(4:6) is not a feasible point')
end

r = [r1;r2];

% Make feasible x the user's responsibility
if min(r) < -epsAlpha,
  error('x0 is not a feasible point.');
end

% Generate initial active set
m1=length(b1);          % Number of constraints
Act1=[]; nAct1=[];
for i1=1:m1,
  if (r(i1) == 0 & length(Act1)<3),
    Act1=[Act1 i1];
  else
```

```
    nAct1=[nAct1 i1];
  end
end

% Initial active set for constraint set 2.
m2=length(b2);          % Number of constraints
Act2=[]; nAct2=[];
for i1=1:m2,
  if (r(m1+i1) == 0 & length(Act2)<3),
    Act2=[Act2 i1];
  else
    nAct2=[nAct2 i1];
  end
end
% nAct is non-active terms, Act is active terms.

k=1;
x=x0;
cNoFac = 0;  % Count of step 6 'no facets added' => termination cond'n

while finite(x),      % Loop while x is still acceptable - Step 3 also breaks out.
  % set up the optimization  - Step 2/4
  %qk=G*x+q;
  bk1 = b1(Act1);
  bk2 = b2(Act2);
  bk = [bk1;bk2];
  mk1 = length(bk1);      % Number of active constraints
  mk2 = length(bk2);      % Number of active constraints
  mk = mk1+mk2;
  Ak1 = A1(:,Act1);
  Ak2 = A2(:,Act2);
%  Ak = [Ak1 zeros(3,mk2);zeros(3,mk1) Ak2];

  % Solve .5*delta'*G*delta + delta'*qk subject to Ak'*delta = bk
  % New approach: Use dsolv(x, Ak1, Ak2) -> delta.
  %delta = dsolvqpe(x, Ak1, Ak2);    % C version
  delta = dsolv(x,Ak1,Ak2);
%  disp('Step 2/4:');

  % Force delta to be within constraints.
  if (mk1 > 0),
    delta(1:3) = delta(1:3)-Ak1*(Ak1'*delta(1:3));
  end
  if (mk2>0),
    delta(4:6) = delta(4:6)-Ak2*(Ak2'*delta(4:6));
  end
  % Test delta - if it is of zero magnitude, do step 3.
  if(norm(delta)) < epsilon,
    % Step 3:  lambda needs to be calculated:
    lambda = genlambda(x,Ak1,Ak2);
    [min1,ind1] = min(lambda);        % ind1 is index of minimum
    if(min1 < -epsLambda),
      % Remove a constraint.
      if ind1 <= mk1,
        % Remove constraint from set 1:
        nAct1=[nAct1 Act1(ind1)];
        Act1=Act1([1:(ind1-1) (ind1+1):mk1]);       % Extract ind1 from Active-1
      else
        % Remove constraint from set 2:
        ind1=ind1-mk1;
        nAct2=[nAct2 Act2(ind1)];
        Act2=Act2([1:(ind1-1) (ind1+1):mk2]);       % Extract ind1 from Active-1
      end
    else
      % Done!
      xstar = x;
      break;
    end
  else
```

```
    % Steps 5-7: Solution of problem produced non-zero delta
    % Step 5:
    all = [];
    for il=nAct1,     % Test all non-active indices - set 1
      ad = A1(:,il)'*delta(1:3);
      if ad < 0,
        all = [all (b1(il) - A1(:,il)'*x(1:3))/ad];
      else
        all=[all 1];
      end
    end
    [alpha1,inda1]=min(all);     % Find minimum alpha, and corresponding index - set 1

    al2 = [];
    for il=nAct2,     % Test all non-active indices - set 2
      ad = A2(:,il)'*delta(4:6);
      if ad < 0,
        al2 = [al2 (b2(il) - A2(:,il)'*x(4:6))/ad];
      else
        al2=[al2 1];
      end
    end
    [alpha2,inda2]=min(al2);     % Find minimum alpha, and corresponding index - set 2

    % Apply minimum alpha (add constraint if <1) to each set independently - fast
converge
    cAdd = 0;  % Count of added facets...
    if (alpha1 < 1-epsAlpha & length(Act1)<3),          % epsAlpha helps reduce cycling.
      % Add a constraint to set 1:
      Act1=[Act1 nAct1(inda1)];                              % Add to Active - 1
      nAct1=nAct1([1:(inda1-1) (inda1+1):length(nAct1)]);    % Extract inda1 from nAct1
      cAdd=cAdd+1;
    end
    if (alpha2 < 1-epsAlpha & length(Act2)<3),
      % Add a constraint to set 2:
      Act2=[Act2 nAct2(inda2)];                              % Add to Active - 2
      nAct2=nAct2([1:(inda2-1) (inda2+1):length(nAct2)]);    % Extract inda2 from nAct2
      cAdd=cAdd+1;
    end

    % Check - if there were no facets added, we're done.
    % (Numerical instability - norm(delta)>epsilon, but only just)
    if(cAdd<1),
      if(cNoFac > 1),
        disp('Terminated from step 6 - no facets could be added, but delta>0');
        disp(delta');
        xstar = x;
        break;
      end
      cNoFac = cNoFac+1;
    else
      cNoFac = 0;
    end

    % Step 7
    k=k+1;
    if(k>200),
      disp([Act1,-1,Act2]);
    end
    % Move by alpha1 along delta1, alpha2 along delta2.
    x=x+[alpha1*delta(1:3);alpha2*delta(4:6)];
  end
end

z=xstar;  % Return result
```

Matlab file: dsolv.m

```
% dsolv.m - solve a 'standard distance problem', used as the
%           basis of a QP distance algorithm.
%           The problem is a QP problem with equality constraints,
%           physically represented by the problem of finding the
%           closest point between a point/line/plane and another
%           point, line  or plane.
%   Input: x, A1, A2
%          x(1:3) is point P1, x(4:6) is point P2
%          A1 is the constraints on P1 - directions it may not move,
%             defining planes of constraint.  (Must be unit, cannot be
%             presumed orthogonal.)
%          A2 - as A1, for P2.
%   Output: dx
%           Change in x to produce solution: dx+x = x2

function dx = dsolv(x, A1, A2)

eps_par = 1e-40;  % Make this a global, later
[n1, ca1] = size(A1);
[n2, ca2] = size(A2);

d = x(4:6) - x(1:3);  % Distance vector
dx = zeros(6,1);      % Preload result vector

if ca1==2,
  a1 = A1(:,1); a2 = A1(:,2);   % Useful terms - P1 constraint vectors
  if ca2==2,
    % Line-Line case is most costly, so do it first.
    alp = (a1-a2*(a2'*a1));
    alp = alp/norm(alp);              % Generate orthonormal pair, [alp a2]
    Ap = [alp';a2'];                  % projector onto the plane perp to Line1
    l2 = cross(A2(:,1),A2(:,2));      % Line-constraint for P2
    l2p = Ap*l2;                      % Projection of l2 onto plane
    l2n = l2p'*l2p;                   % Norm of vector.
    if l2n < eps_par,
      % Lines are close to parallel, so average the positions
      l1 = cross(a1,a2);
      t1 = d'*l1/2;
      t2 = d'*l2/2;
      dx = [l1*t1;-l2*t2];
      return;
    end
    l2i = [-l2p(2);l2p(1)];     % 'Inverse' of l2 - perpendicular
    l2i = l2i/norm(l2i);        % Need norm for next step...
    dp = Ap*d;                  % d projected into plane (useful interim result)
    p2cp = l2i*(l2i'*dp);       % Solution in plane - compenent perp to line
    t = l2p'*(p2cp - dp)/l2n;   % t*l2p = solution in plane, thus
    dx(4:6) = l2*t;             % t*l2 is solution in 3-space, by projectionA
    p2c = x(4:6) + dx(4:6);     % New P2, needed to find new P1
    dc = p2c - x(1:3);          % Distance from old P1 to new P2.

    dx(1:3) = dc - alp*(alp'*dc) - a2*(a2'*dc);
     % Solution of point-line problem to get new P1.
    return;
  end
  if ca2==3 | ca2==0,
    alp = (a1-a2*(a2'*a1));
    alp = alp/norm(alp);                % Generate orthonormal pair, [alp a2]
    dx(1:3) = d - alp*(alp'*d) - a2*(a2'*d);
    if ca2==0,
      % move free point to new location
      dx(4:6) = dx(1:3) - d;
    end
    return
  end
  % Only case left is ca2 == 1 (line-face)
  l1 = cross(a1,a2);
```

```
  if abs(A2' * l1) < eps_par,
    % Less than eps_par - parallel case, go 1/2 way
    % along line towards closest point to P2, then move
    % P2 to closest point to the new P1.
    a1p = (a1-a2*(a2'*a1));
    a1p = a1p/norm(a1p);        % Generate orthonormal pair, [a1p a2]
    dx(1:3) = (d - (a1p*(a1p'*d) + a2*(a2'*d))) * 0.5;   % Half normal dx
    d2 = d-dx(1:3);     % Distance vector from new point
    dx(4:6) = -d2 + A2*(A2'*d2);
    return
  end
  % Non-parallel case - project along l1 a distance corresponding to
  % the ratio a3'*d / a3'*l1 (distance to project along l1 to reach plane)
  dx(1:3) = l1 * ((A2'*d)/(A2'*l1));
  % Then, move point in plane to this intersection point.
  dx(4:6) = dx(1:3) - d;
  return
end

% End of first case, ca1 == 2.  Next: ca1 == 1 (face-?)

if ca1 == 1,
  if ca2 == 2,
    % Same as the 2/1 case above, just reversing use of some terms.
    a2 = A2(:,1); a3 = A2(:,2);        % a1 = A1, so no need...
    l2 = cross(a2,a3);
    if abs(A1' * l2) < eps_par,
      % Less than eps_par - parallel case, go 1/2 way
      % along line towards closest point to P1, then move
      % P1 to closest point to the new P2.
      a2p = (a2-a3*(a3'*a2));
      a2p = a2p/norm(a2p);              % Generate orthonormal pair, [a2p a3]

      dx(4:6) = (- d + (a2p*(a2p'*d) + a3*(a3'*d))) * 0.5;   % Half normal dx
      d2 = d+dx(4:6);     % Distance vector from new point
      dx(1:3) = d2 - A1*(A1'*d2);
      return
    end
    % Non-parallel case - project along l2 a distance corresponding to
    % the ratio a1'*d / a1'*l2 (distance to project along l1 to reach plane)
    dx(4:6) = - l2 * ((A1'*d)/(A1'*l2));
    % Then, move point in plane to this intersection point.
    dx(1:3) = dx(4:6) + d;
    return
  end
  if ca2 == 0 | ca2 == 3,
    % Treat P2 as a fixed point (and if it's actually free, move it later)
    dx(1:3) = d - A1*(A1'*d);     % Project d in P1 plane (defined by A1)
    if ca2==0,
      % move free point to new location
      dx(4:6) = dx(1:3) - d;
    end
    return
  end

  % Case ca2 == 1 - face-face
  a12 = A2'*A1;
  a1p = A1 - A2*a12;                        % Component of A1 in A2 plane
  if (a1p'*a1p) < eps_par,
    % Planes are (nearly) parallel, take average.
    dx(1:3) = (d - A1*(A1'*d)) * 0.5;       % Half normal dx.
    d2 = d-dx(1:3);                         % Distance vector from new point
    dx(4:6) = -d2 + A2*(A2'*d2);            % Closest point to P1+dx(1:3)
    return
  end
  % Planes intersect.  Project from P1 along a2p, and P2 along a1p
  a2p = A2 - A1*a12;                        % Generate comp of A2 perpend. to A1
  if (a1p'*a1p) < 1e-4,
    % Renormalize and restore a1p and a2p.
```

```
    a1p=a1p/norm(a1p);
    a1p = a1p - A2*(A2'*a1p);              % Component of a1p perp to A2
    a2p=a2p/norm(a2p);
    a2p = a2p - A1*(A1'*a2p);              % Component of a2p perp to A1
  end
    % d'*A1 => distance from P2 to the plane defined by (P1,a1)
    % 1/(a1p*A1) - scale factor to get right projection
  p = ((x(4:6) - a1p*(d'*A1)/(a1p'*A1)) + ...
       (x(1:3) + a2p*(d'*A2)/(a2p'*A2)) ) * 0.5;
    % p is the average of the projected points.  Resulting dx is obvious:
  dx(1:3) = p - x(1:3);
  dx(4:6) = p - x(4:6);
  return
end
% end of ca1 == 1
% Next: ca1 == 0 or 3 - in most cases, these are similar

if ca2 == 2,
  % 3,2 is going to be common - handle first.
  % Point-line or volume-line case:
  % Generate closest point on line:
  a2p = A2(:,1) - A2(:,2)*(A2(:,2)'*A2(:,1));
  a2p = a2p/norm(a2p);          % Generate normal pair: [a2p a3=A2(:,2)]
  dx(4:6) = (-d + (a2p*(a2p'*d) + A2(:,2)*(A2(:,2)'*d)));
  if ca1 == 3,
    % Point-line case: dx1 == 0.
    return
  end
  % Volume-line: Move P1 to the newly generated point.
  dx(1:3) = d+dx(4:6);
  return
end

if ca2 == 3,
  % Point-point or volume-point case:
  if ca1 == 3,
    % Point-point case: No possible change.
    return
  end
  % Volume-point: Move P1 to P2
  dx(1:3) = d;
  return
end

if ca2 == 0,
  % Point-volume or volume-volume
  if ca1 == 3,
    % Point-volume: Move P2 to P1.
    dx(4:6) = -d;
    return
  end
  % Volume-volume: take average.
  dx(1:3) = d * 0.5;
  dx(4:6) = -d * 0.5;
  return
end

if ca2 == 1,
  % Point-plane or volume-plane
  % Generate closest point:
  dx(4:6) = (-d + (A2*(A2'*d)));
  if ca1 == 3,
    % Point-plane case: dx1 == 0.
    return
  end
  % Volume-plane: Move P1 to the newly generated point.
  dx(1:3) = d+dx(4:6);
  return
end
```

Matlab file: facet.m

```
% facet.m - returns an array of XYZ triples that correspond to the
%           vertices of a specified facet of an A-b (DPQ3) dataset.
%           ie, all x (column 3-vector) st A'*x-b>0 are inside the polytope
%
% [X,Y,Z] = facet(A,b,N);
%
%   Inputs: A - constraint vector matrix from DQP3
%           b - constraint vector from DQP3
%           N - facet number to compute.
%  Outputs: Depends on # of arguments.  If X is only output, X=[X Y Z].
%           X - column vector of vertex positions in X
%           Y - column vector of vertex positions in Y
%           Z - column vector of vertex positions in Z

function [X,Y,Z,F] = facet(A,b,N);

n=prod(size(b));  % Total number of facets

% Error tolerance for 'near zero' and such
eps = 1e-12;

% Some basic tests.
if (nargin<3),
  error('Insufficient arguments (A,b,N are required)');
end
if (N<1 | N>n),
  N
  error('Must specify a valid facet, N');
end
if (n<3),
  error('Data set too small to produce facets');
end

% Generate an active set where N is first element.
if (N==1),
  Act=[1 2 3];
elseif (N==2),
  Act=[2 1 3];
else
  Act=[N 1 2];
end

% Generate an active set where N is first element, and
% the next two elements are not parallel to N  (Note, A is assumed
% to be normalized...)
Act=[N 0 0];
for(i1=1:n),
  if(i1~=N) % Don't match vector N with itself...
    if(abs(A(:,N)'*A(:,i1)) < .95)
      Act(2) = i1;
      break;
    end
  end
end
% Find the third active set element, !par to first and second.
% No, it has to produce a rank 3 matrix - much more demanding.
% Equivalently, (A(:,a)xA(:,b)).(A(:,c)) > 0 (eps)
% Generate unit cross vector of Act(1) and Act(2) elements of A
Ax=cross(A(:,N),A(:,Act(2)));
Ax=Ax/norm(Ax);   % Normalize
for i1=1:n,
  if(i1~=N & i1~=Act(2))
    % Don't repeat vectors
    if(abs(Ax'*A(:,i1)) > .05),   % Pick something with *some* perpendicular
      Act(3) = i1;
      break;
    end
```

```
  end
end
if(Act(2) < 1 | Act(3) < 1),
  Act
  error('Could not find an appropriate active set of nonparallel vectors');
end

% OK, now for the fun stuff - where do these three facets meet?
% A'*x-b=0 is where.  x=b/A
Aprev=0;iprev=0;
while 1,
  Aa = A(:,Act);
  ba = b(Act);
  x = (ba'/Aa)';    % Aa is invertible thanks to above conditions.

  % Now... let's see how our constraints measure up.
  c = A'*x-b;   % Any terms < 0 means x is outside object.

  % Find most -ve c   (Also get imin, index of cmin)
  [cmin,imin]=min(c);

  if(cmin<-eps),
    % Time to swap constraints, either 2 or 3.
    Ax2=cross(A(:,Act(1)),A(:,Act(3)));
    Ax2=Ax2/norm(Ax2);  % Normalize
    Ax3=cross(A(:,Act(1)),A(:,Act(2)));
    Ax3=Ax3/norm(Ax3);  % Normalize

    % Which Active set value to swap?
    if(Aprev == imin),
      % Ooops, we're repeating ourselves.  Swap the opposite of last time
      if(iprev==2),
        Act(3) = imin;
      else
        Act(2) = imin;
      end
      Aprev = 0;  % Prevent 'anti-loop' test next cycle.
    else
      % Pick whichever produces most perpendicular result
      if(abs(Ax2'*A(:,imin)) > abs(Ax3'*A(:,imin))),
        % Swapping Act(2) produces most 'robust' result
        Aprev = Act(2); iprev=2;    % Save prev val to prevent loops
        Act(2) = imin;
      else
        % Swapping Act(3) produces most 'robust' result
        Aprev = Act(3); iprev=3;    % Save prev val to prevent loops
        Act(3) = imin;
      end
    end
  else
    % No more constraints, so we're done.  x is valid.
    break;
  end
end   % while 1

X = x';               % Save the first vertex
F = Act(3);           % Save the facets, too

% Once we have *one* vertex, we release Act(2), move Act(3) to Act(2),
% and move along the edge defined by Act(1) and Act(2), away from
% the old facet dir'n.  Pick up next constraint, and carry on until
% we're back at the first facet agian.  Oh, and don't forget to
% save the positions in X.

Acti=Act;  % Save 'initial' active set.

% Loop, releasing one facet, picking up next along the edge...
k=0;
while 1,
```

```
  Acto=Act(2);
  Act(2)=Act(3);                % Release Act(2)
  % Determine vector from x along Act(1),Act(2)-defined edge
  v = cross(A(:,Act(1)),A(:,Act(2)));
  v = v/norm(v);                % Normalize v
  % Make sure v points 'away' from old facet  (Facets point 'in')
  % (Facets point 'in' so v dot A(:,i) should be +ve)
  if(v'*A(:,Acto) < 0)
    v=-v;
  end

  % Hm, best way to find that next facet?  Dig through DQP3 for it...
  % Find facet s.t. current A(:,il)'*x-b * (A(:,il)'*v) is minimized,
  % and A(:,il)'*v is negative (v points towards the facet)
  %imin=0;
  %amin=Inf;
  ad = A'*v;      % Precompute all ad values quickly (vectorized)
  %for il=1:n,
  %  if(il ~= Act(1) & il ~= Act(2))    % Don't check against active
  %    % Compute divergence between v and this facet
  %    %ad = A(:,il)'*v;                    % A bit like 'delta' in DQP3
  %    if(ad(il) < -eps),                % Use of 'eps' prevents n/0
  %      % v points 'toward' this facet, so check for minimum...
  %      % Note that we divide by ad: as v becomes parallel to facet norm,
  %      % the distance along v to the facet increases...
  %      av = (A(:,il)'*x-b(il)) / (-ad(il));
  %      if(av<amin),
  %        amin=av;
  %        imin=il;
  %      end
  %    end
  %  end
  %end  % for
  ad(Act(1)) = 1; ad(Act(2)) = 1;  % Keep Act(1:2) out of index
  ind = find(ad < -eps);           % Index of all (ad(il) < -eps) - Cool!
  av = (A(:,ind)'*x-b(ind)) ./ (-ad(ind));
  [amin,imin]=min(av);             % It's just that easy...
  imin=ind(imin);                  % Uh, not quite - need the *right* imin...

  Act(3) = imin;

  % Now that we have the next facet, compute its vertex
  x = (b(Act)'/A(:,Act)')';     % Active set should always be consistant.
  X = [X;x'];                   % Save the vertex!
  F = [F;Act(3)];

  % If we're back to the original active set (or back to the old
  % Act(2), anyway), we're done.  Also done if we hit n iterations.
  if(ismember(Act(3),Acti))
    % We're back where we started.
    break;
  end

  % Test iteration count
  k=k+1;
  if(k>n),
    % Ooops
    break;
  end
end

%...
% Done, just handle outputs...
if(nargout>2)
  Y=X(:,2);
  Z=X(:,3);
  X=X(:,1);
end
```

Matlab file: genlambda.m

```
% genlambda.m - generates lambda, given A1k, A2k and x.

function [lambda] = genlambda(x,A1,A2)

% Multiple cases: n=1,2 or 3 for A1 and A2.
lambda = [];

d = x(4:6)-x(1:3);

[m1 n1]=size(A1);

if(n1 == 1),
  lambda = -A1(:,1)'*d;
end
if(n1 == 2),
  % 'Release condition' is perpendicular to other facet
  a12 = A1(:,1)'*A1(:,2);
  d1 = -d'*A1(:,1);
  d2 = -d'*A1(:,2);
  lambda = [d1-d2*a12; d2-d1*a12];
  % For true lambda, scale by 1/(1-a12^2)...  Always +ve, no need
end
if(n1 == 3),
  % 'Release condition' is perpendicular to the line that is the
  % intersection of the other two facets.
  ax1 = cross(A1(:,2),A1(:,3));
  ax2 = cross(A1(:,3),A1(:,1));
  ax3 = cross(A1(:,1),A1(:,2));
  c3 = 1/(A1(:,1)'*ax1);  % 1/det(A) - scaling factor - may be + or -
  lambda = [-d'*ax1*c3; -d'*ax2*c3; -d'*ax3*c3];
end


[m2 n2]=size(A2);

if(n2 == 1),
  lambda = [lambda; A2(:,1)'*d];
end
if(n2 == 2),
  % 'Release condition' is perpendicular to other facet
  a12 = A2(:,1)'*A2(:,2);
  d1 = d'*A2(:,1);
  d2 = d'*A2(:,2);
  lambda = [lambda; d1-d2*a12; d2-d1*a12];
  % For true lambda, scale by 1/(1-a12^2)...  Always +ve, no need
end
if(n2 == 3),
  % 'Release condition' is perpendicular to the line that is the
  % intersection of the other two facets.
  ax1 = cross(A2(:,2),A2(:,3));
  ax2 = cross(A2(:,3),A2(:,1));
  ax3 = cross(A2(:,1),A2(:,2));
  c3 = 1/(A2(:,1)'*ax1);   % 1/det(A) - scaling factor - may be + or -
  lambda = [lambda; d'*ax1*c3; d'*ax2*c3; d'*ax3*c3];
end
```

Matlab file: genball.m

```
% genball.m  - generates a 'sphere' with 'n' rows of facets.  Facets are
%              generated in evenly radially spaced rows, and the number per
%              row is scaled to sin(theta) of the equatorial count.
%
%              The equator has 2n facets - a cube is genball(2)
%              genball(3) is something like a dodecahedron.
%
%              To get good packing, alternating rows start their points
%              at 0 or pi/(count in row).

% Tests show this works quite well - average proximity of faces is quite
% uniform.  (ie, look at the 'closest face' (dot product max) of each
% facet, see what the numbers are like for all facets.

function [A,b]=genball(n)

% Test: Need n at least 2.
if (n<2),
   error('Invalid order.  N>=2 required');
end

A1 = [1 -1 0  0 0  0;
      0  0 1 -1 0  0;
      0  0 0  0 1 -1];

% First point: the top.
A = [1;0;0];

% Next, we go down the rows, computing thetaRow, nRow, and iterating over phiRow
for i1=1:n,
  thetaRow = i1/n*pi;
  nRow = round(sin(thetaRow)*2*n);
  if nRow<1,
    nRow=1;
  end
  if floor(i1/2)*2 == i1,
    phi1 = pi/nRow;
  else
    phi1 = 0;
  end
  for i2=0:nRow-1,
    phiRow = phi1 + i2*2*pi/nRow;
    A2 = [cos(thetaRow);sin(thetaRow)*sin(phiRow);sin(thetaRow)*cos(phiRow)];
    A=[A A2];
  end
end

[An,Am] = size(A);

% Compute b for each vector such that all planes' closest approach to 0,0,0
% is 1 unit.

b=-ones(Am,1);    % Valid only if all the Ai are normalized.  They are.

% Done!
```

Matlab file: b2ell.m

```
% b2ell.m - given a matrix of normal vectors, generate facets for an
%           ellipse at those normals, producing a b-vector for DQP3.

function [b] = b2ell(A,a)
[n,m] = size(A);
b=ones(m,1);
for il=1:m,
  b(il)=-elldn(a,A(:,il));
end
```

Matlab file: elldn.m

```
% elldn.m - Generate facet dist for an ellipse
%            (canonical form) for a given normal vector.
% [nd] = elldn(a,n);
%       a - 3-vector of ellipse's axes
%       n - normal vector at which to compute nd
%      nd - normal distance (negate for dqp3 use)

function [nd] = elldn(a,n)

% Normal of ellipse a at x
%n = x./(a.^2);
%n = n/norm(n);  % Normalize
% x given n:
%x = n.*a.^2;
%alpha = sum(x.^2./a.^2);
%nd = norm(x)./sqrt(alpha);
% nd Solves down to x'*n/sqrt(sum(x.^2./a.^2))
%    or sum(n.^2.*a.^2))/sqrt(sum(n.^2.*a.^2))   How nifty, it cancels...
nd = sqrt(sum(n.^2.*a.^2));
```

Matlab file: pptes.m

```
% pptes.m - fast point-to-ellipse algorithm, using dominant-alpha
% solutions.  Initial estimate of lambda is sk-1
%
% Given a canonical ellipsoid centered at the origin
% with semi-axes a1, a2, a3 along x, y, and z direction,
% respectively, and point p = [a1 a2 a3] which is a
% vertex of the bounding box for the ellipsoid,
% this function finds x = [x1 x2 x3]' which is exactly
% on the ellipsoid that reaches the minimum distance
% to point p.
% The equation describing the ellipsoid is given by
%
%   x1^2/a1^2 + x2^2/a2^2 + x3^2/a3^2 = 1
%
% Inputs:   a = [a1 a2 a3]' -- semi-axes of the ellipsoid.
%           p = [p1 p2 p3]' -- point to compute distance to
%           [K] -- number of iterations to be performed.    [Default:6]
%           [tol] -- tolerance on sk - if abs(sk-1)<=tol, we're done.
%                    [Default:1e-6, minimum 1e-10]
% Outputs:  x = [x1 x2 x3]' -- the point on the ellipsoid with
%                              minimum distance to point p.
%           FL -- number of flops used.
%           L  -- sequence of lambdas calculated
%           S  -- internal state data: sk;imax;atk;Kmax
% Created June 24, 2001 by DKF
%
% An exmaple [x,FL] = pptes([1 2 1.5],[3 3 3],4);

function [x,FL,L,S,qm] = pptes(a,p,K,tol,L0)

a = a(:);
n = prod(size(a));
p = p(:);
if nargin < 4,
  tol = 1e-6;
end
% Don't allow tolerance *too* small.  Just to be safe
if tol < 1e-10,
  tol = 1e-10;
end

FL = flops;
at = a.^2;
pt = p.^2;
af = pt.*at;
% Some initializations
atk=pt./at;
sk=sum(atk);
% Check sk - if it's too small, lmin will cripple us.
if(sk<1e-4),
  % We're at the center, or close enough.  Use one of the semi-minor points.
  amin=a(1); imin=1;
  for il=2:n,
    if a(il)<amin,
      imin=il;
      amin=a(il);
    end
  end
  % Prepare output x
  x = zeros(n,1);
  if(p(imin) < 0),
    % Use negative amin.
    x(imin) = -a(imin);
  else
    % Positive amin
    x(imin) = +a(imin);
  end
```

```
  FL = flops-FL;
  L = [];
  S = [];
  qm = zeros(n,1);
  return
end

% Minimum lambda:
amin=a(1);
for il=2:n,if(a(il)<amin),amin=a(il);end;end
lmin = -amin^2;
% Add a 'fudge factor
if(lmin<=-1),
  lmin = lmin+.001;
else
  lmin = lmin*0.999;
end

Kmin = 1;   % When to start evaluating for the accelerator (min 1)

% Use a lambda-estimator that works better close to ellipse: sk0-1
if (nargin>4),
  lk = L0;
else
  lk = (sk-1)*amin^2;
  if(lk<lmin*.5),  % Start somewhere safe...
    lk=lmin*.5;
  end
end

% Some initializations
S =[];
L = lk;
k = 0;
%Kmod = bitand(K+1,1);   % When to start the accelerator depends on # of cycles
Kmod=0;  % Give us enough time to generate a good curve.

while k < K,
  % Compute terms of alpha
  atk = af./(at+lk).^2;
  skt = sum(atk);

  stol = abs(skt-1);
  if stol <= tol,
    % Yay, we're done!
    sk=skt;
    break
  end

  if(k==0),
    % Find most variable lambda - only needed on first pass
    % Some slow-convergence runs can use a reorder, too.
    % Compute 'variability' (d/dlambda a^2*p^2/(a^2+lambda)^2)
    % No - pick best of n sk results

    % Prevent div-by-zero
    atk = atk+(atk<1e-12)*1e-12;
    lambda = sqrt(af.*skt./atk) - at;

    imax=0; smax=1e10;   % Find closest sk to 1
    for il=1:n,
      if(lambda(il) > lmin),
        % For greater efficiency (later) save the atk,skt values
        stmp = sum(af./(at+lambda(il)).^2);
        if(abs(stmp-1) < smax),
          smax = abs(stmp-1);
          imax=il;
        end
      end
```

```
  end
  %lkt = lambda(imax);  % Gets computed below, anyway.  <sigh>
end

% Check stability.
if(stol > abs(sk-1) & k>=Kmin),
  % We boo-booed.  Two options, first see if accelerator ran away...
  % No, just go for the linear best - assuming a crossover.
  if((skt>1 & sk<1) | (skt<1 & sk>1)),
    lkt2 = lk0+(lkt-lk0)*((1-sk)/(skt-sk));
    %stmp=sum(af./(at+lkt2).^2);
    %disp(sprintf('lkt interpolated to: %g (%g .. %g)\n\t(%g %g %g) r:%g %d', ...
    %              lkt2,lk0,lkt,sk,stmp,skt,(1-sk)/(skt-sk),k));
    lkt=lkt2;
    Kmin=k+1;
  else
    if(lknr ~= lkt),
      % Restore from acceleration.
      disp('Accel error');
      lkt=lknr;
      Kmin=k+2;  % Prevent inappropriate acceleration
    else
      % It wasn't a runaway, just instability.  Um...  Oh well.
      % Split the difference between last lambda and this one.
      % Note - this only works on oscillatory instability...
      % ??? I need a fix for sk and skt both on same side of 1.
      disp('Runaway.');
      lkt=(lkt+L(k))*0.5;
    end
  end
  % Recompute the key values. (Otherwise, we had *no* effect.)
  atk = af./(at+lkt).^2;
  skt = sum(atk);
  lk0=lkt;
else
  lk0=lk;
end


% Compute 'dominant-alpha' solution - find lambda such that
% the imax term of atk exactly equals atk(imax)/sum.
% Thus, if relative ratios stay similar, we get the solution...
if(atk(imax) ~= 0),  % Prevent divide by zero
  lkt = sqrt(at(imax)*pt(imax)*skt/atk(imax)) - at(imax);
else
  % Um, now what?  Ah, I know - let atk(imax) = .001
  lkt = sqrt(at(imax)*pt(imax)*skt*1000) - at(imax);
end

% Of course, the relative ratios never stay quite the same.

S = [S [skt;imax;atk;Kmin]];

% Ratio-accelerator
lknr = lkt;   % Save non-accelerated value
% Accelerator limited to alt cycles, and Kmin is used for stabilizing changes
if(k>=Kmin & bitand(k,1)==Kmod),
  % We are on an appropriate pass, L(k+1) is most recent lambda
  Lden=L(k+1)-L(k);
  if(Lden ~= 0),  % Don't allow div-by-zero
    ratio = (lkt-L(k+1))/(Lden);
    if (abs(ratio) < 1),
      % Project along this ratio to infinity
      lkt = lkt + (lkt-L(k+1))*ratio/(1-ratio);
      %disp(sprintf('Accelerated from %g - %g - %g to %g cycle %d',...
      %     L(k),L(k+1),lknr,lkt,k));
    end
  end
end
```

```
  if lkt<lmin,
    disp(sprintf('lmin triggered.   lkt=%g, k=%d',lkt,k));
    % We need to pick a new imax, this one is broken.
    % Don't allow div-by-zero.
    atk=atk+.001*(atk == 0);
    lambda = sqrt(at.*pt*skt./atk) - at;
    lambda(imax) = lmin-1;  % There better be something better out there...
    imax=1; lmax=lambda(1);
    for il=2:n,
      if(lambda(il) > lmax),
        imax=il; lmax=lambda(il);
      end
    end
    Kmin=k+2;  % Prevent accelerator for 2 cycles
    lkt = lambda(imax);
    % Check - try setting lkt such that atk(imax) >= 0.01
    if(af(imax)/(at(imax)+lkt)^2 < .01),
      lkt2 = sqrt(100)*a(imax)*abs(p(imax))-at(imax);
      if(lkt2>lmin),lkt=lkt2;end;
    end
    lk=lkt;

    if(lk<lmin),
      lk = lmin;
    end
  else
    lk = lkt;
  end

  sk=skt;
  L = [L lk];
  k=k+1;  % kind of important
end

% Compute estimate x given by lk, then refine to surface (towards ctr)
g = p./(at+lk);
x = at.*g;          % xi=ai^2*pi/(ai^2+lk)
alpha = sum(x.*g);  % ellipse equation on x

% Refine x to ellipse surface along x-c (scale by 1/sqrt(alpha))
x=x/sqrt(alpha);

FL = flops-FL;
% Use abs(sk-1) as quality measure.
qm = abs(sk-1);
```

Matlab file: eepps.m

```
% eepp.m - ellipse-ellipse distance via point-point iteration.
% [x,FL] = eepp(a1,a2,p,K,EP)
%
% a1, a2 - 3-vector of the ellipses' axes
%      p - displacement of ellipse a2
%      K - (optional: 100) # of iterations
%     EP - (optional: 1e-12) tolerance threshold,
%          bounding-polytope 'closest point' to ellipse closest point
%      x - [x1;x2] - surface points on a1 and a2
%     FL - flop count of the run

function [x,FL,L,xhist] = eepps(a1,a2,p,K,EP)

% Handle optional arguments
if (nargin<4),
  K=100;
end
if (nargin<5),
   EP=1e-12;
end

% Orient arguments
a1 = a1(:); a2 = a2(:); p = p(:);

% Flop counter
FL = flops;

% Generate initial values
x1 = zeros(3,1); x2 = p;
lam1 = 0; lam2 = 0;

% Tracking values
L = [lam1 lam2];
xhist = [x1;x2];

k=0;
while(k<K),
  % Do a few iterations of ppte3p, saving lambda.
  [xp1 f1 L1] = pptes(a1,x2,20);
  [xp2 f1 L2] = pptes(a2,x1-p,20);
  xp2 = xp2 + p;
  % Current lambda is last entry in Li, select for next lambda...
  [m n]=size(L1);
  lam1 = L1(n);
  [m n]=size(L2);
  lam2 = L2(n);
  % Copy this iteration to next xi...
  x1 = xp1;
  x2 = xp2;
  k = k+1;
  % Maintain tracking data
  L = [L;[lam1 lam2]];
  xhist = [xhist [x1;x2]];
end

% Return results
x=[x1;x2];
FL=flops-FL;

% Resolution tests - compute closest point on opposite ellipse
% for each of the results points.  (High-resolution runs)
xp1 = pptes(a2,x(4:6),40);  % 40 iterations
xp2 = pptes(a1,x(1:3)-p,40)+p;
xp = [xp1;xp2];
disp('Distance from computed pos to closest ellipse point');
norm(x-xp)
[x,xp]
```

Matlab file: eedist42.m

```
% eedist42.m - compute distance between two canonical ellipses,
%              using polytope approximations of the ellipses in question.
% [x,FL,XK] = eedist42(a1,a2,p,K,EP,nkeep)
%
% a1, a2 - 3-vector of the ellipses' axes
%      p - displacement of ellipse a2
%      K - (optional: 100) # of iterations
%     EP - (optional: 1e-12) tolerance threshold,
%          bounding-polytope 'closest point' to ellipse closest point
%  nkeep - (optional: 20) # of facets to keep from iter to iter (1-6+Act+MRGen)
%      x - [x1;x2] - surface points on a1 and a2
%     FL - flop count of the run
%     XK - array of all intermediate solutions (xk)

% eedist42 - 4 adds 4 facets/cycle, keeping nkeep facets from cycle to cycle

%  * Minimal precomputation - just the bounding box of each ellipse
%  * Future: add rotation matrices for each object
%            add 'reuse' parameters, probably 'previous tangent point' input
%            (Will probably have to vary to consider other options - perhaps
%             allowing multiple tangent points is enough.)
%            Incorporate reuse of lambda from one iteration to next
%
% eedist4: Add 4 facets per iteration - tangent, tangent at x2-x1,
%          two more at 120deg separation opposite the second.
% - also uses upgraded pptes.m point-ellipse distance code.

function [x,FL,XK] = eedist42(a1,a2,p,K,EP,nkeep)

% Handle optional arguments
if (nargin<4),
  K=100
end
if (nargin<5),
   EP=1e-12
end
if (nargin<6),
  nkeep = 20;% Default to keeping 20 facets
end
if (nkeep < 9),
  nkeep = 9; % Minimum to be somewhat effective
end

XK=[];

% Orient arguments
a1 = a1(:);
a2 = a2(:);
p = p(:);

% Flop counter
FL = flops;
% Generate initial bounding polytopes
% Rectangular prisms oriented with the axes
I = eye(3);
A1i=[I -I];  % array of facet-normal column vectors
A2i=[I -I];
% Length of prism corresponds to axis length
b1i=[-a1;-a1];
b2i=[-a2;-a2];

% I'll admit the notation is a bit backwards.  Each vector
% in A corresponds to an inward-pointed surface normal, and the
% value in b is its distance to the origin, projected on that vector, negated.
% To keep the calculations cheap, each vector in A must be normalized.

% Copy initial values to working values
```

```
A1=A1i;A2=A2i;b1=b1i;b2=b2i;
% Compute displaced b2
b2di = A2'*p+b2;
b2d = b2di;

% Generate some data objects used by the algorithm
G = [I -I;-I I];  % Not really used any more...
q = zeros(6,1);
%d = [zeros(3,1);p];  % Displacement of both objects
sin60 = sin(60*pi/180);        % sin(60) for use below.

% Main loop
k=0;
while(k<K),
  % Compute distance on current model
  if(k == 0),
    xest = [0;0;0;p]; % Initial estimate: center of ellipses
  end
  k
  [xk,Act1,Act2] = dqp3l(G,q,A1,b1,A2,b2d,xest);

  % Compute closest point on each ellipsoid in the ellipsoid's coordinates
  %
  % ??? Dynamic adjustment of tol from about 1e-4 down might help, later.
  xp1 = pptes(a1,xk(1:3),40,1e-8);
  xp2 = pptes(a2,xk(4:6)-p,40,1e-8);

  % Compute epsilon, test for termination  (Do it early, don't waste cycles)
  % Note: epsilon is distance beween xk (posn's on polytopes) and xp
  %       (posn's on ellipses).  Should represent error fairly well.
  %xp = [xp1*.9999;xp2*.9999+p];   % .9999 keeps us from being 'outside'
  xp = [xp1;xp2+p];
  curEP = norm([xp1;xp2+p]-xk);
  if(curEP<EP),
    break
  end

  % Generate normal vector and facet distance
  [n1,nd1] = ellnd(a1,xp1);
  [n2,nd2] = ellnd(a2,xp2);

  % Add facets to current set (need to invert to match sense used in A,b)
  % First, remove previous set by ...
  % keeping active, initial6, and most recent up to nkeep.
  if(length(A1)>nkeep)
    % Create active list...
    Act1u = unique([1:6,Act1]);
    At = setxor(1:length(A1),Act1u);    % Set of facets not in Act1u
    Act1u = [Act1u At(length(At)-(nkeep-length(Act1u)):length(At))];
  else
    Act1u = 1:length(A1);
  end
  if(length(A2)>nkeep)
    % Create new active list...
    Act2u = unique([1:6,Act2]);
    At = setxor(1:length(A2),Act2u);    % Set of facets not in Act2u
    Act2u = [Act2u At(length(At)-(nkeep-length(Act2u)):length(At))];
  else
    Act2u = 1:length(A2);
  end

  A1=A1(:,Act1u);A2=A2(:,Act2u);
  b1=b1(Act1u);b2d=b2d(Act2u);
  A1=[A1,-n1];
  b1=[b1;-nd1];
  A2=[A2,-n2];
  %b2=[b2;-nd2];
  b2d =[b2d;(-n2)'*p-nd2];   % Offset version of b2;
```

```
  % Three more facets: one at norm xp2-xp1 (vice versa on 2nd),
  % other two bracketing n at 120 degree increments.
  % Don't forget to add p to xp2 to get actual pos of piont on ellipse 2
  n1a = xp2+p-xp1;n1a=n1a/norm(n1a);   % Normal of current e-e distance vector
  d1a = n1-n1a;      % Distance vector from n1a to n1.  Conceptually.
  dx1a = cross(n1a,d1a);   % Perpendicular to d1a and n1 - other component
  % Now we can build n1b and n1c...
  n1b = n1+d1a*0.5+dx1a*sin60;   % sin(60), cos(30)
  n1b = n1b/norm(n1b);  % Normalize again - it'll be close, but not exact
  n1c = n1-d1a*0.5+dx1a*sin60;   % sin(60), cos(30)
  n1c = n1c/norm(n1c);  % Normalize again - it'll be close, but not exact

  % Add into A1:b1, using elldn to get distances to facets.
  A1 = [A1,-[n1a n1b n1c]];
  b1 = [b1;-[elldn(a1,n1a);elldn(a1,n1b);elldn(a1,n1c)]];

  % Second ellipse
  n2a = -n1a;
  d2a = n2-n2a;      % Distance vector from n2a to n2.  Conceptually.
  dx2a = cross(n2a,d2a);   % Perpendicular to d2a and n2 - other component
  % Now we can build n2b and n2c...
  n2b = n2+d2a*0.5+dx2a*sin60;      % sin(60), cos(30)
  n2b = n2b/norm(n2b);  % Normalize again - it'll be close, but not exact
  n2c = n2-d2a*0.5+dx2a*sin60;      % sin(60), cos(30)
  n2c = n2c/norm(n2c);  % Normalize again - it'll be close, but not exact

  % Add into A2:b2, using elldn to get distances to facets.
  A2 = [A2,-[n2a n2b n2c]];
  %b2 = [b2;-[elldn(a2,n2a);elldn(a2,n2b);elldn(a2,n2c)]];
  b2d =[b2d;(-n2a)'*p-elldn(a2,n2a)];   % Offset version of b2;
  b2d =[b2d;(-n2b)'*p-elldn(a2,n2b)];   % Offset version of b2;
  b2d =[b2d;(-n2c)'*p-elldn(a2,n2c)];   % Offset version of b2;

  norm_d1a=norm(d1a)
  XK = [XK,[xk;norm_d1a;n1;n1a]];

  % Next dqp3 run starts at xp, best guess for closest point
  xest = xp;
  k=k+1;
end

% Return results
x=xk;
FL=flops-FL;

% Resolution tests - compute closest point on opposite ellipse
% for each of the results points.  (High-resolution runs)

xp1 = pptes(a1,x(4:6),100,1e-10);            % 100 iter or 1e-10 error,
xp2 = pptes(a2,x(1:3)-p,100,1e-10)+p;        % whichever comes first.
xp = [xp1;xp2];
disp('Distance from computed pos to closest ellipse point');
norm(x-xp)
[x,xp,x-xp]
```

Matlab file: eedist7.m

(Excerpt only)

After adding facets 1a-1c:

```
% Add extra facets: Rotate 60 deg, and double the scale (wider catchment)
n1d = n1+d1a*2;      % Opposite n1a, *2 dist
n1d = n1d/norm(n1d);
n1e = n1-d1a-dx1a*sin60*2;   % Opposite n1b, *2 dist
n1e = n1e/norm(n1e);
n1f = n1+d1a-dx1a*sin60*2;   % Opposite n1c, *2 dist
n1f = n1f/norm(n1f);

% Add into A1:b1, using elldn to get distances to facets.
A1 = [A1,-[n1d n1e n1f]];
b1 = [b1;-[elldn(a1,n1d);elldn(a1,n1e);elldn(a1,n1f)]];
```

After adding facets 2a-2c:

```
% Add extra facets: Rotate 60 deg, and double the scale (wider catchment)
n2d = n2+d2a*2;      % Opposite n2a, *2 dist
n2d = n2d/norm(n2d);
n2e = n2-d2a-dx2a*sin60*2;   % Opposite n2b, *2 dist
n2e = n2e/norm(n2e);
n2f = n2+d2a-dx2a*sin60*2;   % Opposite n2c, *2 dist
n2f = n2f/norm(n2f);

% Add into A2:b2, using elldn to get distances to facets.
A2 = [A2,-[n2d n2e n2f]];
%b2 = [b2;-[elldn(a2,n2a);elldn(a2,n2b);elldn(a2,n2c)]];
b2d =[b2d;(-n2d)'*p-elldn(a2,n2d)];   % Offset version of b2;
b2d =[b2d;(-n2e)'*p-elldn(a2,n2e)];   % Offset version of b2;
b2d =[b2d;(-n2f)'*p-elldn(a2,n2f)];   % Offset version of b2;
```

C source file: dqpt.c

```
/* dqpc.c - Distance calculation via Quadratic Programming, C version */
/*        - Based on Active set QP algorithm, with an alternative solution */
/*          for the inner Equality-QP problem.                             */
/*        - maintains two sets of active constraints, one for each object  */
/*        - Constraints can be added in pairs, but only one is removed     */
/*          per iteration.                                                 */
/*        rgdZ - output: final position                                    */
/*        rgdA1, rgdA2 - constraints: orientations.                        */
/*        rgdB1, rgdB2 - constraints: offsets                              */
/*        cA1, cA2 - count of constraints (3*cA1 rgdA1's, cA1 rgdB1's)     */

/* Version2 : at C level takes two arrays: rgiAct1, rgiAct2...  Required.   */
/*            -1s are empty.  Get overwritten with new values               */

/* Version T: (dqpt.c): Incorporate rotation and translation                */
/*    Each object has a R matrix associated with it (3x3), and there is a   */
/*    combined 'd' (displacement) matrix.  Objects are rotated then         */
/*    displaced.                                                            */
/*    Handling: delta and lambda steps use absolute x, xa.                  */
/*              initialization and alpha steps use relative x, xo           */
/*              xa is always derived from xo at the end of the xo steps.    */
/*              xo is the input, and the output.  The final distance        */
/*              vector, d=xa2-xa1 is also output                            */

/*  Set up to be callable as a C function or from Matlab */

/* Matlab calling: [z,dv]=dqpt(A1,b1,R1,A2,b2,R2,p,x0)                      */
/*   Output: z  - new position, in object frame (final [xo1;xo2])           */
/*           dv - distance vector, absolute frame (xa2-xa1)                 */
/*   Input:  A1,A2 - face direction vectors, objects 1 and 2                */
/*           b1,b2 - face displacements, objects 1 and 2                    */
/*           R1,R2 - object rotation matrices, objects 1 and 2              */
/*           p     - displacement vector [p1;p2]                            */
/*           x0    - initial position, object frame [xo1;xo2]               */

/* C calling: int fnDistQPT(rgdZ,rgdD,rgdA1,rgdB1,rgdR1,rgdA2,rgdB2,rgdR2,  */
/*                          rgdP,rgdX0,rgiAct1,rgiAct2,cA1,cA2);            */
/*  As above, all rgds are arrays of doubles, plus...                       */
/*    Input: cA1,cA2 - count of working elements (faces) in A1/b1 and A2/b2 */
/*           rgiAct1,rgiAct2 - Active sets, -1 represents no entry          */
/*  Matrices are stored in column order: (a11,a21,a31,a12,a22,a32... etc    */

#include <math.h>
#ifndef __NOTMEX__
#include "mat.h"
#include "mex.h"
#endif /* Only include for MEX compiles */

#include <stdio.h>

#define __INCLUDED_C__
#include "dsolvqpe.c"
#undef __INCLUDED_C__

/* Standard Matlab header stuff */
#ifndef THINK_C
#define DOUBLE double
#define INT int
#else
#define DOUBLE short double
#define INT long
#endif

/* Macros for fast 3-vector work */
#define v3cross(A,B,C) (A)[0]=((B)[1]*(C)[2]-(B)[2]*(C)[1]);\
                       (A)[1]=((B)[2]*(C)[0]-(B)[0]*(C)[2]);\
                       (A)[2]=((B)[0]*(C)[1]-(B)[1]*(C)[0])
```

```
#define v3dot(A,B,C)  (A)=((B)[0]*(C)[0]+(B)[1]*(C)[1]+(B)[2]*(C)[2])
#define v3sum(A,B,C)  (A)[0]=((B)[0]+(C)[0]);(A)[1]=((B)[1]+(C)[1]);(A)[2]=((B)[2]+(C)[2])
#define v3diff(A,B,C)  (A)[0]=((B)[0]-(C)[0]);(A)[1]=((B)[1]-(C)[1]);(A)[2]=((B)[2]-
(C)[2])
#define v3scal(A,B,C)  (A)[0]=((B)[0]*(C));(A)[1]=((B)[1]*(C));(A)[2]=((B)[2]*(C))
#define v3norm(A,B)   (A)=sqrt((B)[0]*(B)[0]+(B)[1]*(B)[1]+(B)[2]*(B)[2])
#define v3copy(A,B)   (A)[0]=(B)[0];(A)[1]=(B)[1];(A)[2]=(B)[2]

/* Matrix * vector: B is 3x3 matrix, column order, A=B*C */
#define a3v3mult(A,B,C)  (A)[0]=(B)[0]*(C)[0]+(B)[3]*(C)[1]+(B)[6]*(C)[2]; \
                         (A)[1]=(B)[1]*(C)[0]+(B)[4]*(C)[1]+(B)[7]*(C)[2]; \
                         (A)[2]=(B)[2]*(C)[0]+(B)[5]*(C)[1]+(B)[8]*(C)[2]
/* And multiplication by transpose of B */
#define a3tv3mult(A,B,C)  (A)[0]=(B)[0]*(C)[0]+(B)[1]*(C)[1]+(B)[2]*(C)[2]; \
                          (A)[1]=(B)[3]*(C)[0]+(B)[4]*(C)[1]+(B)[5]*(C)[2]; \
                          (A)[2]=(B)[6]*(C)[0]+(B)[7]*(C)[1]+(B)[8]*(C)[2]

#define v3pr(A,B) printf("%s: %lg %lg %lg\n",(A),(B)[0],(B)[1],(B)[2])
#define v2pr(A,B) printf("%s: %lg %lg\n",(A),(B)[0],(B)[1])
#define v1pr(A,B) printf("%s: %lg\n",(A),(B))

#define v6pr(A,B) printf("%s: %lg %lg %lg %lg %lg
%lg\n",(A),(B)[0],(B)[1],(B)[2],(B)[3],(B)[4],(B)[5])
#define a3pr(A,B) printf("%s: %lg %lg %lg; %lg %lg %lg; %lg %lg %lg\n",\
                      (A),(B)[0],(B)[3],(B)[6],(B)[1],(B)[4],(B)[7],(B)[2],(B)[5],(B)[8])

#define v3pri(A,B) printf("%s: %d %d %d\n",(A),(B)[0],(B)[1],(B)[2])
#define v2pri(A,B) printf("%s: %d %d\n",(A),(B)[0],(B)[1])
#define v1pri(A,B) printf("%s: %d\n",(A),(B))

/* Useful values */
static double gdEpsTol=1.0e-12;
static double gdEpsLambda=1.0e-8;
static double gdEpsDelta=1.0e-10;
static double gdEpsD = 1e-8;
static double giWatchDogSet = 50;
#define RET_ERR -1
#define RET_OK   0

/* Functions defined: */
#ifdef __STDC__
int fnDistQPT(double rgdZ[], double rgdD[], double rgdA1[], double rgdB1[],
              double rgdR1[], double rgdA2[], double rgdB2[], double rgdR2[],
              double rgdP[], double rgdX0[],
              int rgiAct1[], int rgiAct2[], int cA1, int cA2);
int fnGenLambda(double *rgdLambda1, double *rgdLambda2, double *rgdX, double *rgdA1,
              double *rgdA2, int cA1, int cA2);
#else
int fnDistQPT();
int fnGenLambda();
#endif

/*************************************************************************/
/*************************************************************************/
/*************************************************************************/
/*
 * fnDistQPT() - QP algorithm for distance
 */
/*************************************************************************/
/*************************************************************************/

#ifdef __STDC__
int fnDistQPT(double rgdZ[], double rgdD[], double rgdA1[], double rgdB1[],
              double rgdR1[], double rgdA2[], double rgdB2[], double rgdR2[],
              double rgdP[], double rgdX0[],
              int rgiAct1[], int rgiAct2[], int cA1, int cA2)
#else
int fnDistQPT(rgdZ,rgdD,rgdA1,rgdB1,rgdR1,rgdA2,rgdB2,rgdR2,rgdP,rgdX0,
              rgiAct1,rgiAct2,cA1,cA2)
```

```
double rgdZ[],rgdD[],rgdA1[],rgdB1[],rgdR1[],rgdA2[],rgdB2[],rgdR2[];
double rgdP[],rgdX0[];
int rgiAct1[],rgiAct2[];
int cA1,cA2;
#endif
{
int i,iMin,iMin2,iN1,iN2,iN3;
int iAct1,iAct2;
double dDold;
double rgdXo[6],rgdXa[6];  /* Object frame, absolute frame */
double rgdTmp[3],dTmp,dTmp2,dMin,dMin2;
double rgdA1T[9],rgdA2T[9];                    /* Temporary A1, A2 - active sets for dsolv
*/
                                               /* Maintained in absolute frame
*/
double rgdDeltaa[6],rgdDeltao[6];        /* Absolute frame, object frame */
double rgdLambda1[3],rgdLambda2[3];      /* Absolute frame */
int iWatchDog;                                 /* Don't allow too many iterations */
int iAlpha;                                    /* Count of alpha steps */

/* Initialize output */
for(i=0;i<6;i++) {
    rgdZ[i]=0;
}

iAlpha=0;

/* Initialize watchdog counter - drop out if it hits 0. */
iWatchDog = giWatchDogSet;

/* Initialize active sets: Not any more - passed in */
/* X0 and Ai/Bi are in consistent frames (object) */

/* Generate active set 1 - if no initial active set given (rgiAct1[0]=-1) */
if(rgiAct1[0]<0) {
    for(i=0;i<3;i++) rgiAct1[i]=-1;  /* Init active set 1 */
    iAct1=0;
    for(i=0;i<cA1;i++) {
        v3dot(dTmp,rgdA1+i*3,rgdX0);
        dTmp=dTmp-rgdB1[i];                    /* A1(:,i).x0(1:3) - b1(i) */
        if(dTmp < -gdEpsTol) {
            /* Point is outside constraints - error! */
            printf("Out of bounds in 1 (%lf) at %d\n",dTmp,i);
            return(RET_ERR);
        } else if (dTmp < gdEpsTol) {
            /* Constraint is 'close enough' to zero - active! */
            if(iAct1<3) {
                /* Active set not yet full - add a constraint */
                rgiAct1[iAct1++]=i;
            } else {
                break;        /* Enough constraints found */
            }
        }
    }
} else {
    for(iAct1=0;iAct1<3;iAct1++) {
        if(rgiAct1[iAct1]<0) break;
        v3dot(dTmp,rgdA1+rgiAct1[iAct1]*3,rgdX0);
        dTmp=dTmp-rgdB1[rgiAct1[iAct1]];
        if(fabs(dTmp) > gdEpsTol) {
            /* Point is outside established constraints - error! */
            printf("Constraint set error in 1: (%lf) at %d (%d)\n",
                    dTmp,rgiAct1[iAct1],iAct1);
            return(RET_ERR);
        }
    }
}

/* Generate active set 2 - if no initial active set given (rgiAct2[0]=-1) */
```

```
if(rgiAct2[0]<0) {
    for(i=0;i<3;i++) rgiAct2[i]=-1;  /* Init active set 2 */
    iAct2=0;
    for(i=0;i<cA2;i++) {
        v3dot(dTmp,rgdA2+i*3,rgdX0+3);
        dTmp=dTmp-rgdB2[i];                /* A2(:,i).x0(4:6) - b2(i) */
        if(dTmp < -gdEpsTol) {
            /* Point is outside constraints - error! */
            printf("Out of bounds in 2 (%lf) at %d\n",dTmp,i);
            return(RET_ERR);
        } else if (dTmp < gdEpsTol) {
            /* Constraint is 'close enough' to zero - active! */
            if(iAct2<3) {
                /* Active set not yet full - add a constraint */
                rgiAct2[iAct2++]=i;
            } else {
                break;        /* Enough constraints found */
            }
        }
    }
} else {
    for(iAct2=0;iAct2<3;iAct2++) {
        if(rgiAct2[iAct2]<0) break;
        v3dot(dTmp,rgdA2+rgiAct2[iAct2]*3,rgdX0+3);
        dTmp=dTmp-rgdB2[rgiAct2[iAct2]];
        if(fabs(dTmp) > gdEpsTol) {
            /* Point is outside established constraints - error! */
            printf("Constraint set error in 2: (%lf) at %d (%d)\n",
                    dTmp,rgiAct2[iAct2],iAct2);
            v6pr("B",rgdB2);
            v6pr("x",rgdX0);
            v3pr("A2i",rgdA2+rgiAct2[iAct2]*3);
            return(RET_ERR);
        }
    }
}

/* Establish values for rgdA1T, rgdA2T - active constraints */
for(i=0;i<iAct1;i++) {
    /* Rotate the A1 columns by R1 */
    a3v3mult(rgdA1T+3*i,rgdR1,rgdA1+rgiAct1[i]*3);
}
for(i=0;i<iAct2;i++) {
    /* Rotate the A2 columns by R2 */
    a3v3mult(rgdA2T+3*i,rgdR2,rgdA2+rgiAct2[i]*3);
}

/* Copy X0 to Xo */
v3copy(rgdXo,rgdX0);
v3copy(rgdXo+3,rgdX0+3);

/* Generate Xa (absolute) from Xo (input, object frame) */
a3v3mult(rgdXa,rgdR1,rgdXo);        /* Rotate xo1 */
v3sum(rgdXa,rgdXa,rgdP);            /* then displace */

a3v3mult(rgdXa+3,rgdR2,rgdXo+3);    /* Rotate xo2 */
v3sum(rgdXa+3,rgdXa+3,rgdP+3);      /* then displace */

/* 'Old' distance - for auxiliary termination condition */
v3diff(rgdTmp,rgdXa+3,rgdXa);
v3norm(dDold,rgdTmp);               /* D = |xa2-xa1| */

/* Main processing loop: Active set QP */
while(1) {
    if(--iWatchDog <= 0) {
        /* Drop out here, with whatever results we have. */
        printf("Watchdog failure in dqpt.c  iAct1 %d iAct2 %d\n",iAct1,iAct2);
        v3copy(rgdZ,rgdXo);
        v3copy(rgdZ+3,rgdXo+3);             /* Output z=xo */
```

```
        v3diff(rgdD,rgdXa+3,rgdXa);         /* Output dv=xa2-xa1 */
        v3pri("rgiAct1",rgiAct1);
        a3pr("rgdA1T",rgdA1T);
        v3pri("rgiAct2",rgiAct2);
        a3pr("rgdA2T",rgdA2T);
        v6pr("delta-a",rgdDeltaa);
        v6pr("xo",rgdXo);
        v6pr("xa",rgdXa);
        printf("distance: %.10lg  Min1: %lg  Min2: %lg\n",dDold,dMin,dMin2);
       v3pr("lambda1",rgdLambda1);
        v3pr("lambda2",rgdLambda2);
        if(iWatchDog < -20) {
        v6pr("Input x0",rgdX0);
            return(RET_ERR);
        }
    }

    /* Test x, to make sure values are OK */
    for(i=0;i<6;i++) {
        if(!finite(rgdXo[i])) {
            /* Return error on non-finite x (Nan or Inf) */
            /* Return all zeros */
            rgdZ[0]=rgdZ[1]=rgdZ[2]=0;
            v3copy(rgdZ+3,rgdZ);
            v3copy(rgdD,rgdZ);

            v3pri("rgiAct1",rgiAct1);
            v3pri("rgiAct2",rgiAct2);
            v6pr("delta-a",rgdDeltaa);
            v6pr("xa",rgdXa);
            v6pr("xo",rgdXo);
            printf("distance: %.10lg  Min1: %lg  Min2: %lg\n",dDold,dMin,dMin2);
            v3pr("lambda1",rgdLambda1);
            v3pr("lambda2",rgdLambda2);
        v6pr("Input x0",rgdX0);
            v6pr("Infinite xo",rgdXo);
            return(RET_ERR);
        }
    }

    /* Step 2 or Step 4 - 'Delta' -  Absolute frame */
    /* Call fnDSolvQPE to solve the QP equality problem posed by taking the */
    /* active constraints as equality constraints */

    /* A1T and A2T are already created */

    /* Solve .5*delta'*G*delta + delta'*qk subject to Ak'*delta = bk */
    /* (G=[I,-I;-I,I], qk=0, etc.) */
    fnDSolvQPE(rgdDeltaa,rgdXa,rgdA1T,rgdA2T,iAct1,iAct2);
        /* Generate absolute delta */
if(iWatchDog<=0) v6pr("delta-a",rgdDeltaa);
    /* Re-assert the constraints on delta - QPE is of questionable numerical stability */
    for(i=0;i<iAct1;i++) {
        v3dot(dTmp,rgdDeltaa,rgdA1T+i*3);       /* A1(:,i) . delta(1:3) */
        v3scal(rgdTmp,rgdA1T+i*3,dTmp);             /* A1(:,i)*(A1(:,i) . delta) */
        v3diff(rgdDeltaa,rgdDeltaa,rgdTmp);     /* fix delta(1:3) */
    }
    for(i=0;i<iAct2;i++) {
        v3dot(dTmp,rgdDeltaa+3,rgdA2T+i*3);     /* A2(:,i) . delta(4:6) */
        v3scal(rgdTmp,rgdA2T+i*3,dTmp);             /* A2(:,i)*(A2(:,i) . delta(4:6)) */
        v3diff(rgdDeltaa+3,rgdDeltaa+3,rgdTmp); /* fix delta(4:6) */
    }
if(iWatchDog<=0) v6pr("delta-a-fixed",rgdDeltaa);

    /* Test delta - if delta is of zero magnitude, do step 3. */
    v3dot(dTmp,rgdDeltaa,rgdDeltaa);
    v3dot(dTmp2,rgdDeltaa+3,rgdDeltaa+3);
    if(dTmp+dTmp2 < gdEpsDelta*gdEpsDelta) {         /* Faster than sqrt... */
        /* Close enough to zero to count */
```

```
        /* Step 3 - 'Lambda' - absolute frame */
        /* Step 3 - Remove a constraint */
        /* First: generate requisite lambdas */
        fnGenLambda(rgdLambda1,rgdLambda2, rgdXa, rgdA1T,rgdA2T,iAct1,iAct2);
        dMin=1000;              /* Initialize 'min Lambda' */

        /* After experimentation with allowing simultaneous releases from */
        /* both constraint sets, instabilities have been found.  Single   */
        /* release code is required.                                      */

        /* Test lambdas in both active sets */
        for(i=0;i<iAct1;i++) {
            if(rgdLambda1[i] < dMin) {
                /* Found a new current smallest value */
                dMin=rgdLambda1[i];
                iMin=i;
            }
        }

        dMin2=1000;
        iMin2=-1;
        /* Test lambdas in active set 2 */
        for(i=0;i<iAct2;i++) {
            if(rgdLambda2[i] < dMin2) {
                /* Found a new current smallest value */
                dMin2=rgdLambda2[i];
                iMin2=i;
            }
        }

        /* If smallest lambda is less than 0, remove corresponding constraint */
        if(dMin2 < dMin) {
            if(dMin2 < -gdEpsLambda) {
                /* Remove a constraint from active group 2 */
/*  printf("-- Remove on lambda2 = %lg\n",dMin2);*/
                for(i=iMin2+1;i<iAct2;i++) {
                    /* Remove from index list */
                    rgiAct2[i-1]=rgiAct2[i];
                    /* Remove from Active constraint array */
                    v3copy(rgdA2T+(i-1)*3,rgdA2T+i*3);
                }
                iAct2--;
                rgiAct2[iAct2]=-1;              /* Keep -1's in place */
            } else {
                /* We are done - this is a solution point */
                v3copy(rgdZ,rgdXo);
                v3copy(rgdZ+3,rgdXo+3);             /* Output: z = xo */
                v3diff(rgdD,rgdXa+3,rgdXa);         /* Output: dv = xa2-xa1 */
                break;
            }
        } else {
            if(dMin < -gdEpsLambda) {
                /* Remove a constraint from active group 1 */
/*  printf("-- Remove on lambda1 = %lg\n",dMin);*/
                for(i=iMin+1;i<iAct1;i++) {
                    /* Remove from index list */
                    rgiAct1[i-1]=rgiAct1[i];
                    /* Remove from Active constraint array */
                    v3copy(rgdA1T+(i-1)*3,rgdA1T+i*3);
                }
                iAct1--;
                rgiAct1[iAct1]=-1;              /* Keep -1's in place */
            } else {
                /* We are done - this is a solution point */
                /* Output is the xo used to generate the current xa */
                v3copy(rgdZ,rgdXo);
                v3copy(rgdZ+3,rgdXo+3);             /* Output: z = xo */
                v3diff(rgdD,rgdXa+3,rgdXa);         /* Output: dv = xa2-xa1 */
                break;
```

```
            }
        } /* end if(dMin2 < dMin)..else */
    } /* end if( magnitude(delta)<gdEpsDelta ) -> Step 3 */
    else {
        /* Steps 5-7 - 'Alpha' - Object frame(s) */
        /* Steps 5-7: Solution of problem produced non-zero delta.  Move+test */
        /* Step 5: Test the non-active indices */
        /* Set 1 - only test if norm(delta(1:3)) > 0 */
        /* Note that the norm is frame-invariant */
        /* v3dot(dTmp,rgdDeltaa,rgdDeltaa);  - done above */

        dMin = 1000;      /* Initialize values for minimum-value search */
                    /* dMin is alpha1, if dMin < 1.  Otherwise, alpha1 = 1 */
        if (dTmp > 0) {
            /* delta(1:3) is non-zero, so we have to test the list */

            /* Generate object frame delta1 - now that we know we need it */
            /* Inverse rotation (mult by R1 transpose) */
            a3tv3mult(rgdDeltao,rgdR1,rgdDeltaa);
            /* Set up test values for not-active list */
            iN1 = (iAct1>0 ? rgiAct1[0] : -1);
            iN2 = (iAct1>1 ? rgiAct1[1] : -1);
            iN3 = (iAct1>2 ? rgiAct1[2] : -1);

            iMin = -1;

            for(i=0;i<cA1;i++) {
                if(i==iN1||i==iN2||i==iN3) continue;        /* Skip active elements */
                v3dot(dTmp,rgdA1+3*i,rgdDeltao);            /* ad = A1(:,i).deltaol */
                if(dTmp < 0) {
                    /* Test as a possible minimum value */
                    v3dot(dTmp2,rgdA1+3*i,rgdXo);       /* A1(:,i).xol */
                    dTmp = (rgdB1[i] - dTmp2)/dTmp;         /* (b1(i)-A1(:,i).xol / ad
*/
                    if(dTmp < dMin) {
                        /* New minimum value */
                        dMin = dTmp;
                        iMin = i;
                    }
                }
            }
            if(dMin < 1) {
                /* Add a constraint - we can't move as far as planned */
                /* Note: iAct1 must be <3, as delta(1:3) was nonzero */
                /* Add constraint to index list */
                rgiAct1[iAct1]=iMin;
                /* Add rotated constraint to Active constraint matrix A1T */
                a3v3mult(rgdA1T+iAct1*3,rgdR1,rgdA1+iMin*3);
                /* Increment index */
                iAct1++;

                /* Step 7, block 1: scale deltaol and deltaal by alpha1 (dMin) */
                v3scal(rgdDeltao,rgdDeltao,dMin);
                v3scal(rgdDeltaa,rgdDeltaa,dMin);
            }
        } /* end if (norm(delta(1:3)) > 0) */
        else {
            /* Generate zero deltaol - since deltaal is 0 */
            rgdDeltao[0]=rgdDeltao[1]=rgdDeltao[2]=0;
        }
        /* Set 2 - only test if norm(deltaa(4:6)) > 0 */
        /* Again, this is a frame invariant value */
        v3dot(dTmp,rgdDeltaa+3,rgdDeltaa+3);                /* norm(deltaa(4:6)) */

        dMin2 = 1000;     /* Initialize values for minimum-value search */
                    /* dMin2 is alpha2, if dMin2 < 1.  Otherwise, alpha2 = 1 */
        if (dTmp > 0) {
            /* delta(4:6) is non-zero, so we have to test the list */
```

```
        /* Generate object frame delta2 - now that we know we need it */
        /* Inverse rotation (mult by R2 transpose) */
        a3tv3mult(rgdDeltao+3,rgdR2,rgdDeltaa+3);

        /* Set up test values for not-active list */
        iN1 = (iAct2>0 ? rgiAct2[0] : -1);
        iN2 = (iAct2>1 ? rgiAct2[1] : -1);
        iN3 = (iAct2>2 ? rgiAct2[2] : -1);

        iMin = -1;

        for(i=0;i<cA2;i++) {
            if(i==iN1||i==iN2||i==iN3) continue;        /* Skip active elements */
            v3dot(dTmp,rgdA2+3*i,rgdDeltao+3);      /* ad = A2(:,i).deltao2 */
            if(dTmp < 0) {
                /* Test as a possible minimum value */
                v3dot(dTmp2,rgdA2+3*i,rgdXo+3);         /* A2(:,i).xo2 */
                dTmp = (rgdB2[i] - dTmp2)/dTmp;         /* (b2(i)-A2(:,i).xo2 / ad
*/
                if(dTmp < dMin2) {
                    /* New minimum value */
                    dMin2 = dTmp;
                    iMin = i;
                }
            }
        }
        if(dMin2 < 1) {
            /* Add a constraint - we can't move as far as planned */
            /* Note: iAct2 must be <3, as delta2 was nonzero */
            /* Add constraint to index list */
            rgiAct2[iAct2]=iMin;
            /* Add rotated constraint to Active constraint matrix A2T */
            a3v3mult(rgdA2T+iAct2*3,rgdR2,rgdA2+iMin*3);
            /* Increment index */
            iAct2++;

            /* Step 7, block 2: scale deltaa2 and deltao2 by alpha2 (dMin2) */
            v3scal(rgdDeltaa+3,rgdDeltaa+3,dMin2);
            v3scal(rgdDeltao+3,rgdDeltao+3,dMin2);
        }
    } /* end if (norm(delta(4:6)) > 0) */
    else {
        /* Generate zero deltao2 - since deltaa2 is 0 */
        rgdDeltao[3]=rgdDeltao[4]=rgdDeltao[5]=0;
    }

    iAlpha++;         /* Increment count of alpha steps */

    /* Step 7: end - move to new x */
    /* But first - will the new x be closer to a minima than the current one? */
    /* (It should be, but due to numerical instability, it may not...) */
    /* Use absolute frame values - easier.  This is why deltaa was scaled by alpha */

    v3diff(rgdTmp,rgdXa+3,rgdXa);       /* x2a-x1a */
    v3sum(rgdTmp,rgdTmp,rgdDeltaa+3);   /* x2+deltaa2-x1 */
    v3diff(rgdTmp,rgdTmp,rgdDeltaa);    /* x2+deltaa2-x1-deltaa1 ie, dnew */
    v3norm(dTmp,rgdTmp);                /* Dnew == ||dnew|| */

    /* Test distance */
    if(dTmp > dDold+gdEpsD) {
        /* Test whether or not either set had a constraint added */
        /* If neither did, then there is an error... */
        if(dMin < 1 || dMin2 < 1) {
            /* Restrict alphas: Greater alpha must be reduced to lesser. */

            if(dMin < dMin2) {
                /* Restrict dMin2 to be the same as dMin */
                if(dMin2 < 1) {
                    /* alpha2 is < 1, so a constraint was added */
```

```
                        /* Rescale deltao2 appropriately, remove new constraint */
                        dTmp = dMin/dMin2;
                        v3scal(rgdDeltao+3,rgdDeltao+3,dTmp);

                        iAct2--;
                        rgiAct2[iAct2]=-1;          /* Unused entries get -1's */
                    } else {
                        /* if dMin2 >= 1, take it as equal to 1, and scale by alpha1 */
                        /* Rescale deltao2 */
                        v3scal(rgdDeltao+3,rgdDeltao+3,dMin);
                    }
                } else {
                    /* Restrict dMin to be the same as dMin2 */
                    if(dMin < 1) {
                        /* alpha1 is < 1, so a constraint was added */
                        /* Rescale deltao1 appropriately, remove new constraint */
                        dTmp = dMin2/dMin;
                        v3scal(rgdDeltao,rgdDeltao,dTmp);

                        iAct1--;
                        rgiAct1[iAct1]=-1;          /* Unused entries get -1's */
                    } else {
                        /* if dMin >=1, take as equal to 1, and scale by alpha2 */
                        /* Rescale deltao1 */
                        v3scal(rgdDeltao,rgdDeltao,dMin2);
                    }
                }

                if(iWatchDog<=0) {
                    printf("Alpha-limit: alpha1: %lg alpha2: %lg\n",dMin,dMin2);
                    v6pr("deltao",rgdDeltao);
                }
            } else {
                /* Error - return old Xo value */
                v3copy(rgdZ,rgdXo);
                v3copy(rgdZ+3,rgdXo+3);
                v3diff(rgdD,rgdXa+3,rgdXa);          /* Output: d = xa2-xa1 */
                printf("Retreat error: old dist: %lg new dist: %lg  diff: %lg\n",
                        dDold,dTmp,dDold-dTmp);
                return(RET_ERR);
            }
        }
        dDold = dTmp;

        /* Generate the new Xo using deltao */
if(iWatchDog<=0){
    v6pr("Xo",rgdXo);
}
        v3sum(rgdXo,rgdXo,rgdDeltao);
        v3sum(rgdXo+3,rgdXo+3,rgdDeltao+3);

if(iWatchDog<=0){
    v6pr("deltao",rgdDeltao);
    v6pr("Xo+deltao",rgdXo);
}

        /* Generate Xa from new Xo */
        a3v3mult(rgdXa,rgdR1,rgdXo);          /* Rotate xo1 */
        v3sum(rgdXa,rgdXa,rgdP);              /* then displace */

        a3v3mult(rgdXa+3,rgdR2,rgdXo+3);      /* Rotate xo2 */
        v3sum(rgdXa+3,rgdXa+3,rgdP+3);        /* then displace */

        /* Generate distance - save as new 'old' distance for next time */
        v3diff(rgdTmp,rgdXa+3,rgdXa);
        v3norm(dDold,rgdTmp);                 /* D = |xa2-xa1| */

    } /* end steps 5-7 */
} /* end while(1) - main loop */
```

```
/* Return value is already in rgdZ */

return(iAlpha);
} /* end fnDSolvQPE */

#ifdef __STDC__
int fnGenLambda(double *rgdLambda1, double *rgdLambda2, double *rgdX,
            double *rgdA1, double *rgdA2, int cA1, int cA2)
#else
int fnGenLambda(rgdLambda1, rgdLambda2, rgdX, rgdA1, rgdA2, cA1, cA2)
double *rgdLambda1, *rgdLambda2, *rgdX, *rgdA1, *rgdA2;
int cA1, cA2;
#endif
{
double dC3,dA12,dA13,dA23,dD1,dD2,dTmp;
double rgdAx1[3],rgdAx2[3],rgdAx3[3];
double rgdD[3];

/* Multiple cases for each sub-problem.  This should really be 2 calls */
/* Lambda1 - subproblem 1: -d=-(X2-X1) */
v3diff(rgdD,rgdX,rgdX+3);

/* One constraint: release if point is on the 'inner' side of the plane */
if(cA1 == 1) {
    v3dot(rgdLambda1[0],rgdA1,rgdD);        /* lambda = A1(:,1) . (-d) */
}

/* Two constraints: release if the point is past the perpendicular of the
   opposite plane */
if(cA1 == 2) {
    v3dot(dA12,rgdA1,rgdA1+3);          /* a12 = A1(:,1) . A1(:,2) */
    v3dot(dD1,rgdA1,rgdD);              /* d1 = A1(:,1) . (-d) */
    v3dot(dD2,rgdA1+3,rgdD);            /* d2 = A1(:,2) . (-d) */

    dC3 = 1.0/(1.0 - dA12*dA12);
    /* lambda1 = d.(a1 - a2(a1.a2)), etc. */
    rgdLambda1[0] = (dD1-dD2*dA12)*dC3;
    rgdLambda1[1] = (dD2-dD1*dA12)*dC3;
    /* Neglect the scaling factor of 1/(1-a12^2) - doesn't affect sign. */
    /* But it *does* affect comparison of lambdas - A1 vs A2.  Put in. */
}

/* Three constraints: release if the point is past the perpendicular of
   the line formed by the two opposing planes */
if(cA1 == 3) {
    v3cross(rgdAx1,rgdA1+3,rgdA1+6);    /* ax1 = A1(:,2)xA1(:,3) */
    v3cross(rgdAx2,rgdA1+6,rgdA1);      /* ax2 = A1(:,3)xA1(:,1) - order is important! */
    v3cross(rgdAx3,rgdA1,rgdA1+3);      /* ax3 = A1(:,1)xA1(:,2) */
    v3dot(dC3,rgdA1,rgdAx1);        /* det(A1) = A1(:,1).(A1(:,2)xA1(:,3)) */

    /* Scale factor of 1/det(A1) used to recover 'actual' lambda */
    /* Necessary so that sign is correct. */
    if(fabs(dC3) > gdEpsLambda) {
        dC3 = 1/dC3;
    } else {
        /* Don't let things get too out of hand if the determinant is near 0 */
        dC3 = (dC3 > 0 ? 1.0/gdEpsLambda : -1.0/gdEpsLambda);
    }

    /* lambda1 = d.(a2xa3)/det(A1), etc. */
    v3dot(dTmp,rgdD,rgdAx1);
    rgdLambda1[0] = dTmp*dC3;
    v3dot(dTmp,rgdD,rgdAx2);
    rgdLambda1[1] = dTmp*dC3;
    v3dot(dTmp,rgdD,rgdAx3);
    rgdLambda1[2] = dTmp*dC3;
}
```

```
/* Lambda2 - subproblem 2: d=(X2-X1) */
v3scal(rgdD,rgdD,-1);

/* One constraint: release if point is on the 'inner' side of the plane */
if(cA2 == 1) {
    v3dot(rgdLambda2[0],rgdA2,rgdD);        /* lambda = A2(:,1) . (-d) */
}

/* Two constraints: release if the point is past the perpendicular of the
   opposite plane */
if(cA2 == 2) {
    v3dot(dA12,rgdA2,rgdA2+3);          /* a12 = A2(:,1) . A2(:,2) */
    v3dot(dD1,rgdA2,rgdD);              /* d1 = A2(:,1) . (-d) */
    v3dot(dD2,rgdA2+3,rgdD);            /* d2 = A2(:,2) . (-d) */

    dC3 = 1.0/(1.0 - dA12*dA12);
    /* lambda1 = d.(a1 - a2(a1.a2)), etc. */
    rgdLambda2[0] = (dD1-dD2*dA12)*dC3;
    rgdLambda2[1] = (dD2-dD1*dA12)*dC3;
    /* Neglect the scaling factor of 1/(1-a12^2) - doesn't affect sign. */
    /* But it *does* affect comparison of lambdas - A1 vs A2.  Put in. */
}

/* Three constraints: release if the point is past the perpendicular of
   the line formed by the two opposing planes */
if(cA2 == 3) {
    v3cross(rgdAx1,rgdA2+3,rgdA2+6);    /* ax1 = A2(:,2)xA2(:,3) */
    v3cross(rgdAx2,rgdA2+6,rgdA2);      /* ax2 = A2(:,3)xA2(:,1) - order is important! */
    v3cross(rgdAx3,rgdA2,rgdA2+3);      /* ax3 = A2(:,1)xA2(:,2) */
    v3dot(dC3,rgdA2,rgdAx1);        /* det(A2) = A2(:,1).(A2(:,2)xA2(:,3)) */

    /* Scale factor of 1/det(A) used to recover 'actual' lambda */
    /* Necessary so that sign is correct. */
    if(fabs(dC3) > gdEpsLambda) {
        dC3 = 1/dC3;
    } else {
        /* Don't let things get too out of hand if the determinant is near 0 */
        dC3 = (dC3 > 0 ? 1.0/gdEpsLambda : -1.0/gdEpsLambda);
    }

    /* lambda1 = d.(a2xa3)/det(A2), etc. */
    v3dot(dTmp,rgdD,rgdAx1);
    rgdLambda2[0] = dTmp*dC3;
    v3dot(dTmp,rgdD,rgdAx2);
    rgdLambda2[1] = dTmp*dC3;
    v3dot(dTmp,rgdD,rgdAx3);
    rgdLambda2[2] = dTmp*dC3;
}

return(RET_OK);
}  /* End of fnGenLambda() */
```

C source file: dsolvqpe.c

```
/* dsolvqpe.c - David's solution for the Equality QP problem at the core of the
                distance problem.  Treats the problem as a physical problem.
*/

/* To include into another matlab-mexfile, define __INCLUDED_C__ before #including */

#include <math.h>
#ifndef __NOTMEX__
#include "mat.h"
#include "mex.h"
#endif  /* Only include for MEX compiles */

#include <stdio.h>

#ifndef __INCLUDED_C__
/* Standard Matlab header stuff */
#ifndef THINK_C
#define DOUBLE double
#define INT int
#else
#define DOUBLE short double
#define INT long
#endif

#endif

/* Macros for fast 3-vector work */
#define v3cross(A,B,C)  (A)[0]=((B)[1]*(C)[2]-(B)[2]*(C)[1]);\
                        (A)[1]=((B)[2]*(C)[0]-(B)[0]*(C)[2]);\
                        (A)[2]=((B)[0]*(C)[1]-(B)[1]*(C)[0])
#define v3dot(A,B,C)  (A)=((B)[0]*(C)[0]+(B)[1]*(C)[1]+(B)[2]*(C)[2])
#define v3sum(A,B,C)  (A)[0]=((B)[0]+(C)[0]);(A)[1]=((B)[1]+(C)[1]);(A)[2]=((B)[2]+(C)[2])
#define v3diff(A,B,C)  (A)[0]=((B)[0]-(C)[0]);(A)[1]=((B)[1]-(C)[1]);(A)[2]=((B)[2]-
(C)[2])
#define v3scal(A,B,C)  (A)[0]=((B)[0]*(C));(A)[1]=((B)[1]*(C));(A)[2]=((B)[2]*(C))
#define v3norm(A,B)   (A)=sqrt((B)[0]*(B)[0]+(B)[1]*(B)[1]+(B)[2]*(B)[2])
#define v3copy(A,B)   (A)[0]=(B)[0];(A)[1]=(B)[1];(A)[2]=(B)[2]

#define v3pr(A,B) printf("%s: %lg %lg %lg\n",(A),(B)[0],(B)[1],(B)[2])
#define v2pr(A,B) printf("%s: %lg %lg\n",(A),(B)[0],(B)[1])
#define v1pr(A,B) printf("%s: %lg\n",(A),(B))

/* Macros for handling Matlab matrix structures. */

/*static double gdEpsPar=1.0e-40;           / Always used to test dot products */
static double gdEpsPar22=1.0e-20;           /* line-line parallelness test (dot) */
static double gdEpsPar12=1.0e-10;           /* line-plane parallelness test (norm) */
static double gdEpsPar11=1.0e-4;            /* plane-plane parallelness test */
                                              /*        (just plain sensitive) */

/* Functions defined: */
#ifdef __STDC__
void fnDSolvQPE(double rgdDX[], double rgdX[], double rgdA1[], double rgdA2[],
             int cA1, int cA2);
#else
void fnDSolvQPE();
#endif

/*************************************************************************/
/*************************************************************************/
/*************************************************************************/
/*
 * fnDSolvQPE() - Solves the QP Equality problem from the QP distance problem.
 */
/*************************************************************************/
/*************************************************************************/
```

```
#ifdef __STDC__
void fnDSolvQPE(double rgdDX[], double rgdX[], double rgdA1[], double rgdA2[],
            int cA1, int cA2)
#else
void fnDSolvQPE(rgdDX,rgdX,rgdA1,rgdA2,cA1,cA2)
double rgdDX[],rgdX[],rgdA1[],rgdA2[];
int cA1,cA2;
#endif
{
int i;
double rgdD[3],rgdTmp[3],dTmp;
double *pda1,*pda2,*pda3;
double rgda1p[3],rgda2p[3],rgdl2p[2];
double dl2n,dTmp2,da12;
double rgdl2[3],rgdl2i[3],rgdDp[3],rgdTmp2[3];
double rgdl1[3];

/* Initialize output */
for(i=0;i<6;i++) {
    rgdDX[i]=0;
}

/* Generate distance vector */
v3diff(rgdD,rgdX+3,rgdX);      /* d = x(4:6)-x(1:3) */

/* Main processing: break into cases */

if(cA1==2) {
    pda1 = rgdA1;                  /* P1 constraint vector 'a1' */
    pda2 = rgdA1+3;                    /* P1 constraint vector 'a2' */
    if(cA2==2) {
        /* Line-line case is the most costly, so we get to it first */
        v3dot(dTmp,pda1,pda2);
        v3scal(rgdTmp,pda2,dTmp);
        v3diff(rgda1p,pda1,rgdTmp);        /* a1p = a1-a2(a1.a2) */
        v3norm(dTmp,rgda1p);
        dTmp=1/dTmp;
        v3scal(rgda1p,rgda1p,dTmp);        /* Normalize a1p: [a2, a1p] orthonormal */
                                           /* Skip: Ap=[a1p';a2'] */
        v3cross(rgdl2,rgdA2,rgdA2+3);  /* l2=cross(A2(:,1),A2(:,2)) */
        v3dot(rgdl2p[0],rgda1p,rgdl2);
        v3dot(rgdl2p[1],pda2,rgdl2);   /* l2p = Ap*l2; - Line2 in plane perp to line1 */

        dl2n=rgdl2p[0]*rgdl2p[0]+rgdl2p[1]*rgdl2p[1];
                                           /* l2n = l2p'*l2p  -  Norm of vector*/
/*printf("------ dl2n: %lg\n",dl2n);*/
        if(dl2n < gdEpsPar22) {
            /* Lines are close to parallel, treat as so.  Average the posn's */
            /* Move point 2 halfway along L2 towards point 1 */
            v3dot(dTmp,rgdD,rgdl2);
            v3norm(dl2n,rgdl2);                /* Norm of l2 - get scale right! */
            dTmp=dTmp/(dl2n*dl2n*2);       /* t2 = d'*l2/(2*|l2|) */
            v3scal(rgdDX+3,rgdl2,-dTmp);   /* dx(4:6) = -l2/|l2|*t2 */
                                               /* the extra |l2| term is in prev. line
*/

            /* Move point 1 to closest point on L1 to new point 2 */
            v3cross(rgdl1,pda1,pda2);
            v3sum(rgdD,rgdD,rgdDX+3);      /* new d = d+dx2 */
            v3dot(dTmp,rgdD,rgdl1);
            v3norm(dl2n,rgdl1);                /* |l2| cannot be borrowed from |l1| */
            dTmp=dTmp/(dl2n*dl2n);         /* t1 = d2'*l1/(|l1|); */

            v3scal(rgdDX,rgdl1,dTmp);      /* dx(1:3) -= l1*t1 */
                                               /* the extra |l1| term is in prev. line
*/
            return;
        }
        /* Lines are not parallel.  Find closest point in perp. plane, project back */
```

```
        /* Find perpendicular unit vector to l2p */
        dTmp = sqrt(dl2n);
        rgdl2i[0]=-rgdl2p[1]/dTmp;
        rgdl2i[1]=rgdl2p[0]/dTmp;       /* l2i=[-l2p(2);l2p(1)] - perpendicular in 2-d */
                                            /* dTmp makes l2i a unit vector */
        v3dot(rgdDp[0],rgdalp,rgdD);
        v3dot(rgdDp[1],pda2,rgdD);      /* Component of D in the plane */
        dTmp = rgdl2i[0]*rgdDp[0] + rgdl2i[1]*rgdDp[1];      /* l2i . Dp */
        rgdTmp[0] = rgdl2i[0]*dTmp;
        rgdTmp[1] = rgdl2i[1]*dTmp;         /* l2i*(l2i.Dp): Dp projected onto l2i */
        rgdTmp[0] -= rgdDp[0];
        rgdTmp[1] -= rgdDp[1];
        dTmp = (rgdl2p[0]*rgdTmp[0] + rgdl2p[1]*rgdTmp[1]) / dl2n;
            /* t = l2p'*(p2cp - Ap*d)/l2n:  'Distance' along l2p for sol'n in plane */
            /* t*l2p is solution in plane, so t*l2 is solution in 3space */
        v3scal(rgdDX+3,rgdl2,dTmp);         /* Compute solution for point 2 */

        /* Point 1 solution may now be found easily. */
        v3sum(rgdTmp,rgdX+3,rgdDX+3);  /* Tmp = new point 2 */
        v3diff(rgdD,rgdTmp,rgdX);       /* D = distance vector, point 1 to new point 2 */

        v3dot(dTmp,rgdalp,rgdD);
        v3scal(rgdTmp,rgdalp,dTmp);         /* Tmp = alp*(alp.D) */
        v3dot(dTmp,pda2,rgdD);
        v3scal(rgdTmp2,pda2,dTmp); /* Tmp2 = a2*(a2.D) */
        v3sum(rgdTmp,rgdTmp,rgdTmp2);  /* Tmp = alp*(alp.D) + a2*(a2.D) */
                                            /* Component of D perp to constraint line 1
*/
        v3diff(rgdDX,rgdD,rgdTmp);      /* Component of D parallel to constraint line 1 */

        return;
    } /* end if(cA2==2) */
    else if(cA2==0 || cA2==3) {
        /* Line - point and line-space cases */
        /* Line-point using cross product: */
        v3cross(rgdl1,pdal,pda2);       /* l1 = alxa2 */
        v3norm(dTmp,rgdl1);
        dTmp=1/dTmp;
        v3scal(rgdl1,rgdl1,dTmp);       /* Normalized l1 */

        v3dot(dTmp,rgdl1,rgdD);             /* D was generated earlier */
        v3scal(rgdDX,rgdl1,dTmp);       /* dx = l1(d.l1) */

        /* If point 2 is unconstrained, move it to this new point */
        if(cA2==0) {
            v3diff(rgdDX+3,rgdDX,rgdD);     /* dx(4:6) = dx(1:3) - d */
        }
        return;

    } /* end if(cA2==2, 0, or 3) */
    else {
        /* Line-plane case */
        v3cross(rgdl1,pdal,pda2);           /* l1 = a1 x a2 */
        v3dot(dTmp,rgdA2,rgdl1);            /* A2'*l1 */
        if(fabs(dTmp) < gdEpsPar12) {
            /* Line and plane are nearly parallel.  Treat as parallel */
            /* Move 1/2 way along L1 to closest point to P2, then find */
            /* the closest point in the plane to this resulting point */

            /* Line-point using cross product: */
            /*v3cross(rgdl1,pdal,pda2);*/  /* l1 = alxa2 - from above. */
            v3norm(dTmp,rgdl1);
            dTmp=1/dTmp;
            v3scal(rgdl1,rgdl1,dTmp); /* Normalized l1 */

            v3dot(dTmp,rgdl1,rgdD);             /* D was generated earlier */
            dTmp=dTmp*0.5;
            v3scal(rgdDX,rgdl1,dTmp);       /* dx = 0.5*l1(d.l1) ('normal' dx) */
```

```
            /* Point-to-plane solution */
            v3diff(rgdD,rgdD,rgdDX);        /* New dist: d=d-dx(1:3) */
            v3dot(dTmp,rgdA2,rgdD);             /* A2(1:3) . d */
            v3scal(rgdTmp,rgdA2,dTmp);      /* A2*(A2.d) - component of d perp to plane
*/
            v3diff(rgdDX+3,rgdTmp,rgdD);    /* component in plane */
            return;
        } /* end if (fabs(dTmp) < gdEpsPar12) */
        /* The line and plane are not parallel, so we proceed normally */
        /* Move along L1 until the intersection with the plane */
        /* v3dot(dTmp,rgdA2,rgdl1); - already done, above, and non-zero. */
        v3dot(dTmp2,rgdA2,rgdD);
        dTmp=dTmp2/dTmp;
        v3scal(rgdDX,rgdl1,dTmp); /* dx(1:3) = l1 * ((A2'*d)/(A2'*l1)) */
        /* Move point P2 to this new point */
        v3diff(rgdDX+3,rgdDX,rgdD);
        return;
    } /* end if(cA2==...) */

} /* end if (cA1==2) */
else if (cA1==1) {
    if (cA2==2) {
        /* Plane-line case */
        pda2=rgdA2;                 /* a2 = A2(1:3,1) - first constraint, P2 */
        pda3=rgdA2+3;           /* a3 = A2(1:3,2) - second constraint, P2 */
        v3cross(rgdl2,pda2,pda3);       /* l2 = a2 x a3 */
        v3dot(dTmp,rgdA1,rgdl2);            /* A1'*l2 */
/* Start from here... copied from 2-1 */
        if(fabs(dTmp) < gdEpsPar12) {
            /* Line and plane are nearly parallel.  Treat as parallel */
            /* Move 1/2 way along L2 to closest point to P1, then find */
            /* the closest point in the plane to this resulting point */

            /* Line-point using cross product: */
            /*v3cross(rgdl2,pda2,pda3);*/  /* l2 = a2xa3  - from above. */
            v3norm(dTmp,rgdl2);
            dTmp=1/dTmp;
            v3scal(rgdl2,rgdl2,dTmp);       /* Normalized l2 */

            v3dot(dTmp,rgdl2,rgdD);             /* D was generated earlier */
            dTmp=dTmp*0.5;
            v3scal(rgdDX+3,rgdl2,-dTmp);    /* dx2 = -0.5*l2(d.l2) ('normal' dx) */
                                                /* (handles reverse of sense of D) */

            /* Point-to-plane solution */
            v3sum(rgdD,rgdD,rgdDX+3);       /* New dist: d=d+dx(4:6) */
            v3dot(dTmp,rgdA1,rgdD);             /* A1(1:3) . d */
            v3scal(rgdTmp,rgdA1,dTmp);      /* A1*(A1.d) - component of d perp to plane
*/
            v3diff(rgdDX,rgdD,rgdTmp);      /* component in plane */
            return;
        } /* end if (fabs(dTmp) < gdEpsPar12) */
        /* The line and plane are not parallel, so we proceed normally */
        /* Move along L2 until the intersection with the plane */
        /* v3dot(dTmp,rgdA1,rgdl2); - already done, above, and non-zero. */
        v3dot(dTmp2,rgdA1,rgdD);
        dTmp=dTmp2/dTmp;
        v3scal(rgdDX+3,rgdl2,-dTmp);        /* dx(1:3) = - l2 * ((A1'*d)/(A1'*l2)) */
        /* Move point P1 to this new point */
        v3sum(rgdDX,rgdDX+3,rgdD);
        return;
    } /* end if(cA2==2) */
    else if (cA2==0 || cA2==3) {
        /* Treat P2 as a fixed point - if it's free (case 0), move it later */
        /* Project d in P1 plane (defined by A1) */
        v3dot(dTmp,rgdA1,rgdD);
        v3scal(rgdTmp,rgdA1,dTmp);
        v3diff(rgdDX,rgdD,rgdTmp);          /* dx(1:3) = d-A1*(A1.d) */
```

```
        if(cA2==0) {
            /* Move free point P2 to new location */
            v3diff(rgdDX+3,rgdDX,rgdD);          /* dx(4:6) = dx(1:3)-d */
        }
        return;
    } else {
        /* Case 1-1, plane-plane */
        v3dot(dal2,rgdA2,rgdA1);                 /* al2 = A2'*A1 */
        v3scal(rgdTmp,rgdA2,dal2);
        v3diff(rgdalp,rgdA1,rgdTmp);             /* alp = A1 - A2*al2: A1 in A2 plane */

        v3dot(dTmp,rgdalp,rgdalp);
        if(fabs(dTmp) < gdEpsPar11) {
            /* Planes are nearly parallel - treat as parallel */
            /* Split the difference between the points */
            v3dot(dTmp,rgdA1,rgdD);
            v3scal(rgdTmp,rgdA1,dTmp);           /* A1*(A1.d) */
            v3diff(rgdDX,rgdD,rgdTmp);           /* dx(1:3)=d - A1*(A1.d) */
            v3scal(rgdDX,rgdDX,0.5);             /* dx(1:3)= (d-A1*(A1.d))*0.5 */
                                                    /* Half normal dx */
            v3diff(rgdD,rgdD,rgdDX);             /* d=d-dx(1:3) - distance to new point */

            v3dot(dTmp,rgdA2,rgdD);
            v3scal(rgdTmp,rgdA2,dTmp);
            v3diff(rgdDX+3,rgdTmp,rgdD);         /* dx(4:6)=-d+A2*(A2.d) */
            return;
        }
        /* Planes intersect.  Project from P1 along a2p, and P2 along alp */
        v3scal(rgdTmp,rgdA1,dal2);
        v3diff(rgda2p,rgdA2,rgdTmp);             /* a2p= A2 - A1*al2 */

        /* If rgdalp is too small, we need to rescale - expensive, but worthwhile */
        if (fabs(dTmp) < 0.0001)  {
            dTmp=1/sqrt(dTmp);
            v3scal(rgdalp,rgdalp,dTmp);             /* Normalize alp */
            v3dot(dTmp,rgdalp,rgdA2);
            v3scal(rgdTmp,rgdA2,dTmp);           /* A2*(A2.alp) */
            v3diff(rgdalp,rgdalp,rgdTmp);        /* alp - ( " )   Part of alp in A2 plane */

            v3dot(dTmp,rgda2p,rgda2p);
            dTmp=1/sqrt(dTmp);
            v3scal(rgda2p,rgda2p,dTmp);             /* Normalize a2p */
            v3dot(dTmp,rgda2p,rgdA1);
            v3scal(rgdTmp,rgdA1,dTmp);           /* A1*(A1.a2p) */
            v3diff(rgda2p,rgda2p,rgdTmp);        /* a2p - ( " )   Part of a2p in A1 plane */
        }

        /* Compute the new point */
        /* p = ((x(4:6) - alp*(d'*A1)/(alp'*A1)) + (x(1:3) + a2p*(d'*A2)/(a2p'*A2)) ) *
0.5; */
        v3dot(dTmp,rgdD,rgdA1);
        v3dot(dTmp2,rgdalp,rgdA1);
        dTmp=dTmp/dTmp2;                         /* (d.A1)/(alp.A1) */
        v3scal(rgdTmp,rgdalp,dTmp);
        v3diff(rgdTmp,rgdX+3,rgdTmp);            /* x(4:6) - alp*(d.A1)/(alp.A1) */

        v3dot(dTmp,rgdD,rgdA2);
        v3dot(dTmp2,rgda2p,rgdA2);
        dTmp=dTmp/dTmp2;                         /* (d.A2)/(a2p.A2) */
        v3scal(rgdTmp2,rgda2p,dTmp);
        v3sum(rgdTmp2,rgdX,rgdTmp2);             /* x(1:3) + a2p*(d.A2)/(a2p.A2) */

        v3sum(rgdTmp,rgdTmp,rgdTmp2);
        v3scal(rgdTmp,rgdTmp,0.5);               /* (term1 + term2) * 0.5 */

        /* Calculate the resulting dx, given this final point */
        v3diff(rgdDX,rgdTmp,rgdX);               /* dx(1:3) = p - x(1:3) */
        v3diff(rgdDX+3,rgdTmp,rgdX+3);           /* dx(4:6) = p - x(4:6) */
        return;
```

```
    } /* end if (cA2==0..3) */
} /* end if(cA1==2)... else if (cA1==1) */
else {
/* cA1==0 or 3, handle case by case in cA2 */
    if(cA2==2) {
        /* 3,2 is common, so handle this case first */
        /* point-line or volume-line case */
        /* Generate closest point on the line... */
        pda2=rgdA2;                 /* a2 = A2(1:3,1) - first constraint, P2 */
        pda3=rgdA2+3;           /* a3 = A2(1:3,2) - second constraint, P2 */

        /* Line-point using cross product: */
        v3cross(rgdl2,pda2,pda3);       /* l2 = a2xa3 */
        v3norm(dTmp,rgdl2);
        dTmp=1/dTmp;
        v3scal(rgdl2,rgdl2,dTmp);       /* Normalized l2 */

        v3dot(dTmp,rgdl2,rgdD);             /* D was generated earlier */
        v3scal(rgdDX+3,rgdl2,-dTmp);    /* dx2 = -l2(d.l2)  */
                                            /* (Handles reverse of sense of D) */

        if(cA1==0) {
            /* Volume-line case: move P1 to newly generated point */
            v3sum(rgdDX,rgdDX+3,rgdD);
        }
        return;
    } /* end if (cA2==2) */
    else if (cA2==3) {
        /* point-point or volume-point */
        if (cA1==0) {
            /* Volume-point: move P1 to P2 */
            v3copy(rgdDX,rgdD);                         /* dx(1:3)=d */
        }
        /* point-point case returns 0, obviously. */
        return;
    }
    else if (cA2==0) {
        /* Point-volume or volume-volume */
        if(cA1==3) {
            /* Point-volume: move P2 to P1 */
            v3scal(rgdDX+3,rgdD,-1);            /* dx(4:6)=-d */
            return;
        }
        /* Volume-volume: take average */
        v3scal(rgdDX,rgdD,0.5);                     /* dx(1:3) = d*0.5 */
        v3scal(rgdDX+3,rgdD,-0.5);              /* dx(4:6) = -d*0.5 */
        return;
    }
    else /* cA2==1 */ {
        /* Point-plane or volume-plane */
        /* Generate closest point */
        v3dot(dTmp,rgdA2,rgdD);
        v3scal(rgdTmp,rgdA2,dTmp);
        v3diff(rgdDX+3,rgdTmp,rgdD);            /* dx(4:6) = A2*(A2.d) - d */
        if(cA1==0) {
            /* Volume-plane - move P1 to new point */
            v3sum(rgdDX,rgdDX+3,rgdD);          /* dx(1:3) = dx(4:6) + d */
        }
    return;
    } /* end if (cA2==2,3,0)..else */
} /* end if(cA1==2 or 1)... else */
} /* end fnDSolvQPE */
```