Convex Quadratic Programming

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: December 10, 2017
Last Modified: September 11, 2020

Contents

1 Introduction

This document briefly describes the quadratic programming (QP) problem, a minimization of a quadratic polynomial on a domain defined by linear inequality constraints. The focus is on the convex quadratic programming (CQP) problem, where the matrix of the quadratic polynomial is positive semidefinite. Many geometric algorithms can be formulated as CQPs. A CQP is converted to a Linear Complementarity Problem (LCP) that can be solved using Lemke's Method [1].

The general framework for QP is presented first, showing how to convert a QP to an LCP. Lemke's Method is presented together with several illustrative examples. An implementation for solving an LCP is discussed with attention given to accuracy of the results when using floating-point arithmetic. The LCP solver uses only addition, subtraction, multiplication and division, so assuming the inputs are finite floating-point numbers, such numbers are rational and the solver can use arbitrary-precision floating-point arithmetic to produce exact results.

Some CQPs involve geometric primitives whose parameterizations use unit-length vectors. If these vectors are computed using fixed-precision floating-point arithmetic, numerical rounding errors lead to vectors that are not unit length when interpreted as exact rational inputs. In this situation, the LCP solver will not produce the correct theoretical result that is based on real-valued arithmetic. However, in many cases the concept of *real quadratic field* in abstract algebra can be used to solve the LCP exactly. If a distance query is required within this framework, the distance itself is computed only at the very end of the algorithm by approximating the exact quadratic field result by a fixed-precision floating-point number.

For the sake of notation, the set of $n \times 1$ column vectors with real-valued entries is denoted $\mathbb{R}^n$. The set of $r \times c$ matrices with r rows, c columns, and real-valued entries is denoted $\mathbb{R}^{r \times c}$.

1.1 The Quadratic Programming Problem

The *quadratic program (QP)* is concisely stated as follows.

> Given constants $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$, $c \in \mathbb{R}$, $D \in \mathbb{R}^{m \times n}$, $\mathbf{e} \in \mathbb{R}^m$, and variable $\mathbf{x} \in \mathbb{R}^n$, minimize $f(\mathbf{x}) = \frac{1}{2}\,\mathbf{x}^\mathsf{T} A\mathbf{x} + \mathbf{b}^\mathsf{T}\mathbf{x} + c$ subject to the linear inequality constraints $\mathbf{x} \geq \mathbf{0}$ and $D\mathbf{x} \geq \mathbf{e}$.

The number of linear inequality constraints is $n + m$.

The linear inequalities define an intersection of half spaces. The intersection can be empty, in which case the QP does not have a solution. For a nonempty intersection that is unbounded and with no additional constraints on A, it is possible the QP has no solution. If the nonempty intersection is a bounded set, that set is necessarily convex. The polynomial f is continuous and defined on a closed bounded set, which guarantees that f attains both a minimum and a maximum on the set.

If $\mathbf{x} \in \mathbb{R}^n$ is a local extremum of the QP, then there exists $\mathbf{y} \in \mathbb{R}^m$ such that $(\mathbf{x}, \mathbf{y})$ satisfies the Karesh–Kuhn–Tucker (KKT) conditions

$$\begin{aligned} &\mathbf{u} = \mathbf{b} + A\mathbf{x} - D^\mathsf{T}\mathbf{y} \geq \mathbf{0}, \quad && \mathbf{x} \geq \mathbf{0}, \quad \mathbf{x}^\mathsf{T}\mathbf{u} = 0, \\ &\mathbf{v} = -\mathbf{e} + D\mathbf{x} \geq \mathbf{0}, && \mathbf{y} \geq \mathbf{0}, \quad \mathbf{y}^\mathsf{T}\mathbf{v} = 0 \end{aligned} \tag{1}$$

The KKT conditions are necessary for the existence of a local extremum. When A is positive semidefinite, the KKT conditions are also sufficient for the existence of a local extremum.

1.2 The Linear Complementarity Problem

The *linear complementarity problem (LCP)* is concisely stated as

> Given constants $\mathbf{q} \in \mathbb{R}^k$ and $M \in \mathbb{R}^{k\times k}$, find $\mathbf{z} \in \mathbb{R}^k$ such that $\mathbf{z} \geq \mathbf{0}$, $\mathbf{q} + M\mathbf{z} \geq \mathbf{0}$, and $\mathbf{z}^{\mathsf{T}}(\mathbf{q} + M\mathbf{z}) = \mathbf{0}$.

Define $\mathbf{w} = \mathbf{q} + M\mathbf{z}$. We want $\mathbf{z} \geq \mathbf{0}$ such that $\mathbf{w} \geq \mathbf{0}$ and $\mathbf{z}^{\mathsf{T}}\mathbf{w} = 0$.

Lemke's Method allows us to compute an LCP solution $\mathbf{z}$ if there exists one or to determine that there is no solution.

1.3 The Convex Quadratic Programming Problem

In the quadratic program, when A is positive semidefinite the problem is a *convex quadratic program (CQP)*. The CQP can be converted to an LCP by defining

$$\mathbf{q} = \begin{bmatrix} \mathbf{b} \\ -\mathbf{e} \end{bmatrix}, \quad M = \begin{bmatrix} A & -D^{\mathsf{T}} \\ D & 0 \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{2}$$

where $k = n + m$. The variable names come from the CQP and the KKT conditions. The matrix M is not symmetric, but it is positive semidefinite because $\mathbf{z}^{\mathsf{T}}M\mathbf{z} \geq 0$ for all $\mathbf{z}$. The inequality is guaranteed because A is positive semidefinite.

Once formulated as an LCP, we may solve the problem using Lemke's Method to extract the location $\mathbf{x}$ and value f of the local minimum. Observe that the *linear programming* (LP) problem is a special case of CQP when A is the zero matrix (which is positive semidefinite).

1.4 Eliminating Unconstrained Variables

The CQP problem has the inequality contraint $\mathbf{x} \geq \mathbf{0}$ that says all independent variables must be nonnegative. Some geometric queries involving variables that are unconstrained; that is, they can be any real number. The corresponding CQP must be modified to eliminate such variables.

For example, consider a CQP in 3D with $\mathbf{x} = (x_0, x_1, x_2)$ and whose inequality constraints depend only on x_0 and x_1,

$$x_0 \geq 0,\ x_1 \geq 0,\ D \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \geq \mathbf{e} \tag{3}$$

where D is $m \times 2$ and $\mathbf{e}$ is $m \times 1$. The variable x_2 is unconstrained.

For a fixed pair (x_0, x_1), the function $f(\mathbf{x}) = \mathbf{x}^{\mathsf{T}}A\mathbf{x}/2 + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c$ is quadratic in x_2. The minimum with respect to x_2 must occur when the derivative with respect to x_2 is zero. Let $A = [a_{ij}]$ and $\mathbf{b} = [b_j]$; then $0 = \partial f/\partial x_2 = a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + b_2$ and has solution $x_2 = -(a_{20}x_0 + a_{21}x_1 + b_2)/a_{22}$. The function to minimize is $g(x_0, x_1) = f(x_0, x_1, -(a_{20}x_0 + a_{21}x_1 + b_2)/a_{22})$ subject to the constraints of equation (3).

Using

$$\mathbf{x} = x_0 \begin{bmatrix} 1 \\ 0 \\ -a_{20}/a_{22} \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ 1 \\ -a_{21}/a_{22} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -b_2/a_{22} \end{bmatrix} = x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + \mathbf{u}_2 \tag{4}$$

some algebra will show that $g(s,t) = \tilde{\mathbf{x}}^{\mathsf{T}}\tilde{A}\tilde{\mathbf{x}}/2 + \tilde{\mathbf{b}}^{\mathsf{T}}\tilde{\mathbf{x}} + \tilde{c}$, where

$$\tilde{A} = \begin{bmatrix} \mathbf{u}_0^{\mathsf{T}}A\mathbf{u}_0 & \mathbf{u}_0^{\mathsf{T}}A\mathbf{u}_1 \\ \mathbf{u}_1^{\mathsf{T}}A\mathbf{u}_0 & \mathbf{u}_q^{\mathsf{T}}A\mathbf{u}_q \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \mathbf{u}_0^{\mathsf{T}}A\mathbf{u}_2 \\ \mathbf{u}_1^{\mathsf{T}}A\mathbf{u}_2 \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}\mathbf{u}_2^{\mathsf{T}}A\mathbf{u}_2 + \mathbf{b}^{\mathsf{T}}\mathbf{u}_2 + c \tag{5}$$

In general, let $\mathbf{x}_c$ be the constrained variables and let $\mathbf{x}_u$ are the unconstrained variables. Partition the various quantities by

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix}, \quad A = \begin{bmatrix} A_{cc} & A_{cu} \\ A_{uc} & A_{uu} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_u \end{bmatrix} \tag{6}$$

where the block elements are of the appropriate sizes. The matrix A is symmetric, so $A_{uc} = A_{cu}^{\mathsf{T}}$. The matrix A is also positive definite, so A_{cc} and A_{uu} are positive definite. The quadratic function is

$$\begin{aligned} f(\mathbf{x}_c, \mathbf{x}_u) &= \tfrac{1}{2}\begin{bmatrix} \mathbf{x}_c^{\mathsf{T}} & \mathbf{x}_u^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} A_{cc} & A_{cu} \\ A_{uc} & A_{uu} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix} + \begin{bmatrix} \mathbf{b}_c^{\mathsf{T}} \\ \mathbf{b}_u^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix} + c \\ &= \tfrac{1}{2}\mathbf{x}_c^{\mathsf{T}}A_{cc}\mathbf{x}_c + \left(\mathbf{x}_u^{\mathsf{T}}A_{uc} + \mathbf{b}_c^{\mathsf{T}}\right)\mathbf{x}_c + \left(\tfrac{1}{2}\mathbf{x}_u^{\mathsf{T}}A_{uu}\mathbf{x}_u + \mathbf{b}_u^{\mathsf{T}}\mathbf{x}_u + c\right) \end{aligned} \tag{7}$$

The derivative with respect to the unconstrained variables must be zero,

$$\mathbf{0} = \frac{\partial f}{\partial \mathbf{x}_u} = A_{uc}\mathbf{x}_c + A_{uu}\mathbf{x}_u + \mathbf{b}_u \tag{8}$$

The solution is

$$\mathbf{x}_u = -A_{uu}^{-1}\left(A_{uc}\mathbf{x}_c + \mathbf{b}_u\right) \tag{9}$$

Substituting this back into the quadratic function, we obtain $g(\mathbf{x}_c) = f(\mathbf{x}_c, \mathbf{x}_u)$ and

$$g(\mathbf{x}_c) = \frac{1}{2}\mathbf{x}_c^{\mathsf{T}}\tilde{A}\mathbf{x}_c + \tilde{\mathbf{b}}^{\mathsf{T}}\mathbf{x}_c + \tilde{c} \tag{10}$$

where

$$\tilde{A} = A_{cc} - A_{cu}A_{uu}^{-1}A_{uc}, \quad \tilde{b} = \mathbf{b}_c - A_{cu}A_{uu}^{-1}\mathbf{b}_u, \quad \tilde{c} = \frac{1}{2}\mathbf{x}_u^{\mathsf{T}}A_{uu}\mathbf{x}_u + \mathbf{b}_u^{\mathsf{T}}\mathbf{x}_u + c \tag{11}$$

We solve the CQP to minimize $g = \mathbf{x}_c^{\mathsf{T}}\tilde{A}\mathbf{x}_c + \tilde{\mathbf{b}}^{\mathsf{T}}\mathbf{x}_c + \tilde{c}$ subject to $\mathbf{x}_c \geq 0$ and the problem-specific constraints $\tilde{D}\mathbf{x}_c \geq \tilde{\mathbf{e}}$. The solution $\mathbf{x}_c$ is then substituted into equation (9) to obtain $\mathbf{x}_u$.

1.5 Reduction of Dimension for Equality Constraints

Sometimes the CQP in an n-dimensional setting involves equality constraints. The dimension of the CQP can be reduced by eliminating such contraints. For example, consider a CQP in 3D with $\mathbf{x} = (x_0, x_1, x_2)$ with constraints

$$x_0 \geq 0,\ x_1 \geq 0,\ D\mathbf{x} \geq \mathbf{e},\ \mathbf{n} \cdot \mathbf{x} + d = 0 \tag{12}$$

where D is $m \times 3$, $\mathbf{e}$ is $m \times 1$, and $\mathbf{n} = (n_0, n_1, n_2)$ with $n_2 \neq 0$.

Solve the equality constraint for $x_2 = -(n_0 x_0 + n_1 x_1 + d)/n_2$ and substitute it into both the function $f(\mathbf{x})$ and the inequality constraints $D\mathbf{x} \geq \mathbf{e}$. The reduction $g(x_0, x_1) = f(x_0, x_1, -(n_0 x_0 + n_1 x_1 + d)/n_2)$ uses the same approach that led to equations (4) and (5), except that $\mathbf{u}_0 = (1, 0, -n_0/n_2)$, $\mathbf{u}_1 = (0, 1, -n_1/n_2)$ and $\mathbf{u}_2 = (0, 0, -d/n_2)$.

The reduction of the inequality constraint $D\mathbf{x} \geq \mathbf{e}$ is as follows. For $D = [D_{ij}]$ and $\mathbf{e} = [e_i]$, each inequality constraint is of the form

$$e_i \leq D_{i0} x_0 + D_{i1} x_1 + D_{i2} x_2 = D_{i0} x_0 + D_{i1} x_1 - D_{i2}(n_0 x_0 + n_1 x_1 + d)/n_2 \tag{13}$$

Grouping similar terms, we have

$$(D_{i0} - D_{i2}\, n_0/n_2) x_0 + (D_{i1} - D_{i2}\, n_1/n_2) x_1 \geq e_i + D_{i2}\, d/n_2 \tag{14}$$

Using linear algebra terminology for solving the equality constraint, x_2 is a basic variable and x_0 and x_1 are free variables.

In general, let the ℓ equality constraints for the CQP be $F\mathbf{x} + \mathbf{v} = \mathbf{0}$ where F is $\ell \times n$ and $\mathbf{v}$ is $\ell \times 1$. For a nontrivial problem, it must be that $\ell < n$. Apply row reductions to the linear system of equality constraints to obtain a coefficient matrix that is in reduced row echelon form. Once in this form it is easy to identify the basic variables and the free variables of the linear system. If $\mathbf{x}_b$ is the tuple of basic variables and $\mathbf{x}_f$ is the tuple of free variables, then the reduced row echelon form can be solved for $\mathbf{x}_b = H\mathbf{x}_f + \mathbf{w}$ for some matrix H and vector $\mathbf{w}$.

For simplicity, reorder the components of $\mathbf{x}$ so that $\mathbf{x} = (\mathbf{x}_f, \mathbf{x}_b)$. The general construction for unconstrained variables starting with equation (6) can be duplicated with renamed quantities $\mathbf{x}_f$ for $\mathbf{x}_c$, $\mathbf{x}_b$ for $\mathbf{x}_u$, A_{ff} for A_{uu}, A_{fb} for A_{cu}, A_{bb} for A_{uu}, $\mathbf{b}_f$ for $\mathbf{b}_c$ and $\mathbf{b}_b$ for $\mathbf{b}_u$. The resulting $\tilde{A}$, $\tilde{\mathbf{b}}$ and $\tilde{c}$ are used for the function to be minimized, $g(\mathbf{x}_f) = \mathbf{x}_f^{\mathsf{T}} \tilde{A} \mathbf{x}_f/2 + \tilde{\mathbf{b}}^{\mathsf{T}} \mathbf{x}_f + \tilde{c}$.

Partition $D = [D_f\ D_b]$ such that the number of columns of D_f is the number of components of $\mathbf{x}_f$ and the number of columns of D_b is the number of components of $\mathbf{x}_b$. The inequality constraints are

$$\mathbf{e} \leq D\mathbf{x} = \begin{bmatrix} D_f & D_b \end{bmatrix} \begin{bmatrix} \mathbf{x}_f \\ \mathbf{x}_b \end{bmatrix} = D_f \mathbf{x}_f + D_b \mathbf{x}_b = D_f \mathbf{x}_f + D_b(H\mathbf{x}_f + \mathbf{w}) \tag{15}$$

Grouping similar terms, we have

$$\tilde{D}\mathbf{x}_f = (D_f + D_b H)\mathbf{x}_f \geq \mathbf{e} - D_b \mathbf{w} = \tilde{\mathbf{e}} \tag{16}$$

where the first equality defines $\tilde{D}$ and the last equality defines $\tilde{\mathbf{e}}$.

The reduction in dimension leads to minimizing $g(\mathbf{x}_f) = \mathbf{x}_f^{\mathsf{T}} \tilde{A} \mathbf{x}_f/2 + \tilde{\mathbf{b}}^{\mathsf{T}} \mathbf{x}_f + \tilde{c}$ subject to $\mathbf{x}_f \geq 0$ and $\tilde{D}\mathbf{x}_f \geq \tilde{\mathbf{e}}$.

2 Lemke's Method

2.1 Terms and Framework

The standard approach for solving LP is the simplex algorithm using the tableau method. This may also be used to solve an LCP, but an approach that uses different terminology is Lemke's Method. The presentation here follows that of [2]. The equation $\mathbf{w} = \mathbf{q} + M\mathbf{z}$ is considered to be a *dictionary* for the *basic* variables $\mathbf{w}$ defined in terms of the *nonbasic* variables $\mathbf{z}$. The analogy to a dictionary is that the basic variables are words in the dictionary defined in terms of the nonbasic variables that are other words in the dictionary. If $\mathbf{q} \geq \mathbf{0}$, the dictionary is said to be *feasible*, in which case the LCP has the trivial solution $\mathbf{z} = \mathbf{0}$ and $\mathbf{w} = \mathbf{q}$.

If the dictionary is not feasible, Lemke's Method is applied. Assuming that $\mathbf{z} = (z_0, \ldots, z_{n-1})$, the first phase of the algorithm adds an auxiliary variable $z_n \geq 0$ by modifying the dictionary to $\mathbf{w} = \mathbf{q} + M\mathbf{z} + z_n\mathbf{1}$, where $\mathbf{1}$ is the n-tuple whose components are all 1. The i-th equation is selected according to some criterion (described later) that exchanges z_n and w_i by solving the equation for z_n, which now becomes a basic variable. The right-hand side of the equation contains a w_i term, so w_i now becomes a nonbasic variable. The equation for the now-basic z_n is substituted into the other equations to eliminate the right-hand side occurrences of z_n. The equation to solve for z_n is selected so that after the substitutions in the other equations, the modified dictionary is feasible.

The second phase of the algorithm is designed to obtain a dictionary such that the following two conditions hold:

1. z_n is nonbasic.
2. For each i, either z_i or w_i is nonbasic.

A dictionary that satisfies conditions 1 and 2 is said to be a *terminal dictionary.* If the dictionary satisfies only condition 2, it is said to be a *balanced dictionary.* The first phase produces a balanced dictionary, but z_n is in the dictionary (it is a basic variable), so the dictionary is not terminal. The procedure to reach a terminal dictionary is iterative. Each iteration is designed so that a nonbasic variable enters the dictionary and a basic variable leaves the dictionary. The invariant after each iteration is that the dictionary remain feasible and balanced. To ensure this happens and hopefully to avoid producing the same dictionary twice, if a variable has just left the dictionary, then its complementary variable must enter the dictionary on the next iterations: A variable cannot leave/enter on one iteration and enter/leave on the next iteration. Once z_n leaves the dictionary, we have a terminal dictionary. The condition that z_i or w_i is nonbasic for each $i < n$ means that either $z_i = 0$ or $w_i = 0$; that is, $\mathbf{w}^\mathsf{T}\mathbf{z} = 0$ and we have solved the LCP.

Two problems can occur during the iterations.

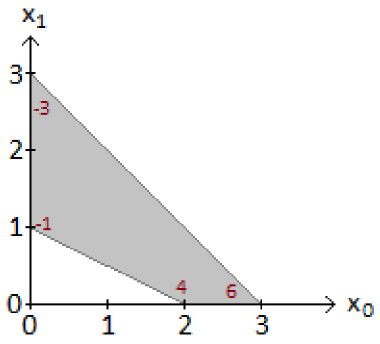
1. The variable complementary to the leaving variable cannot enter the dictionary. In this case, the LCP does not have a solution.
2. It is possible to encounter a cycle in the dictionaries, which prevents the algorithm from converging to a solution. When this happens, one of the components of $\mathbf{q}$ in the dictionary has become zero. This is referred to as a *degeneracy.* The algorithm can be modified by introducting symbolic perturbations of the components of $\mathbf{q}$ to avoid the cycles.

Several examples are presented here to illustrate the algorithm.

2.2 LCP with a Unique Solution

Example 1 shows how one selects the variables to exchange in order to obtain a feasible dictionary.

Example 1. *Linear Programming problem with a unique solution.* Minimize $f(x_0, x_1) = 2x_0 - x_1$ subject to the constraints $x_0 \geq 0$, $x_1 \geq 0$, $x_0 + x_1 \leq 3$ and $x_0 + 2x_1 \geq 0$. The figure shows the domain of f that is defined by the inequality constraints. The function values at the vertices of the domain are shown in red.



Visually, the minimum must occur at $(x_0, x_1) = (0, 3)$. The dimension of the LCP is $n = 4$. The LCP quantities of interest are

$$\mathbf{q} = \left[\begin{array}{c} 2 \\ -1 \\ \hline 3 \\ -2 \end{array}\right], \quad M = \left[\begin{array}{cc|cc} 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & -2 \\ \hline -1 & -1 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{array}\right], \quad \mathbf{z} = \left[\begin{array}{c} x_0 \\ x_1 \\ \hline y_0 \\ y_1 \end{array}\right], \quad \mathbf{w} = \left[\begin{array}{c} u_0 \\ u_1 \\ \hline v_0 \\ v_1 \end{array}\right] \tag{17}$$

The initial dictionary with auxiliary variable z_4 is

$$\begin{aligned} w_0 &= 2 + z_2 - z_3 + z_4 \\ w_1 &= -1 + z_2 - 2z_3 + z_4 \\ w_2 &= 3 - z_0 - z_1 + z_4 \\ w_3 &= -2 + z_0 + 2z_1 + z_4 \end{aligned} \tag{18}$$

We need to exchange z_4 with one of the w_i and then substitute that equation into the others to obtain a feasible dictionary; that is, choose the exchange equation so that the resulting constants for $\mathbf{q}$ are nonnegative. The coefficients of z_4 are positive, so we are limited to examining the two equations with negative constants. We could solve the second equation, $z_4 = 1 - z_2 + 2z_3 + w_1$, but when substituting it in the fourth equation we obtain $w_3 = -1 + z_0 + 2z_1 - z_2 + 2z_3 + w_1$, which has a negative constant. The resulting dictionary is not feasible. Therefore, the exchange equation is the fourth equation in which case $z_4 = 2 - z_0 - 2z_1 + w_3$. The nonbasic variable z_4 becomes basic (enters the dictionary) and the basic variable w_3 becomes nonbasic (leaves the dictionary). Substituting in the other equations, we have

$$\begin{aligned} w_0 &= 4 - z_0 - 2z_1 + z_2 - z_3 + w_3 \\ w_1 &= 1 - z_0 - 2z_1 + z_2 - 2z_3 + w_3 \\ w_2 &= 5 - 2z_0 - 3z_1 + w_3 \\ z_4 &= 2 - z_0 - 2z_1 + w_3 \end{aligned} \tag{19}$$

For the initial dictionary, the exchange equation is the one with the minimum **q**-component.

For the remaining iterations, if v_j is the nonbasic variable that is required to enter the dictionary and become basic (v_j is either z_j or w_j), the exchange equation is the one for which the coefficient of v_j is negative and the nonnegative ratio $-q_i/(m_{ij}v_j)$ is minimum for all i.

The variable w_3 left the dictionary, so z_3 must now enter the dictionary. Choose the equation that minimizes the quantity mentioned in the previous paragraph. The first and second equations have negative coefficients for z_3. The ratio for the first equation is 4/1 and the ratio for the second equation is 1/2, so the second equation is the one to exchange. Solve for z_3 and substitute this into the other equations,

$$\begin{aligned} w_0 &= (7/2) - (1/2)z_0 - z_1 + (1/2)z_2 + (1/2)w_1 + (1/2)w_3 \\ z_3 &= (1/2) - (1/2)z_0 - z_1 + (1/2)z_2 - (1/2)w_1 + (1/2)w_3 \\ w_2 &= 5 - 2z_0 - 3z_1 + w_3 \\ z_4 &= 2 - z_0 - 2z_1 + w_3 \end{aligned} \tag{20}$$

The variable w_1 left the dictionary, so z_1 must now enter the dictionary. All four equations have negative coefficients for z_1 and the ratios are 7/2, 1/2, 5/3 and 1, in order of listing of the equations. The minimum ratio is 1/2, generated by the second equation. Solve for z_1 and subtitute this into the other equations,

$$\begin{aligned} w_0 &= 3 + z_3 + w_1 \\ z_1 &= (1/2) - (1/2)z_0 - z_3 + (1/2)z_2 - (1/2)w_1 + (1/2)w_3 \\ w_2 &= (7/2) - (1/2)z_0 + 3z_3 - (3/2)z_2 + (3/2)w_1 - (1/2)w_3 \\ z_4 &= 1 + 2z_3 - z_2 + w_1 \end{aligned} \tag{21}$$

The variable z_3 left the dictionary, so w_3 must now enter the dictionary. Only the third equation has a negative coefficient for w_3. Solve for w_3 and substitute this into the other equations,

$$\begin{aligned} w_0 &= 3 + z_3 + w_1 \\ z_1 &= 4 - z_0 + 2z_3 - z_2 + w_1 - w_2 \\ w_3 &= 7 - z_0 + 6z_3 - 3z_2 + 3w_1 - 2w_2 \\ z_4 &= 1 + 2z_3 - z_2 + w_1 \end{aligned} \tag{22}$$

The variable w_2 left the dictionary, so z_2 must now enter the dictionary. The last 3 equations have a negative coefficient for z_2, so the ratios are 4, 7/3, and 1. The last equation provides the minimum ratio. Solve for

z_2 and substitute this into the other equations,

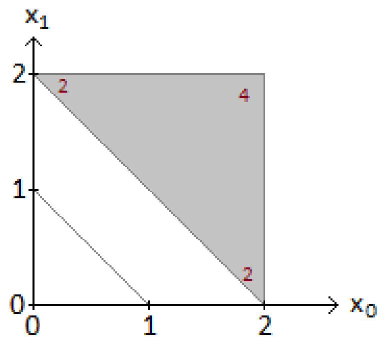
$$\begin{aligned}
w_0 &= 3 + z_3 + w_1 \\
z_1 &= 3 - z_0 + z_4 - w_2 \\
w_3 &= 4 - z_0 + 3z_4 - 2w_2 \\
z_2 &= 1 + 2z_3 - z_4 + w_1
\end{aligned} \tag{23}$$

The auxiliary variable z_4 left the dictionary, returning to its initial role as a nonbasic variable. The iterations terminate here and we have a solution. The variables on the right-hand side of the equation are set to zero: $z_0 = 0$, $z_3 = 0$, $z_4 = 0$, $w_1 = 0$ and $w_2 = 0$. The variables on the left-hand side are then $w_0 = 3$, $z_1 = 3$, $w_3 = 4$ and $z_2 = 0$. The original variables that minimize f are $(x_0, x_1) = (z_0, z_1) = (0, 3)$.

2.3 LCP with Infinitely Many Solutions

Example 2 shows that the algorithm will select one of the locations at which the minimum occurs when there are infinitely many such locations.

Example 2. *Linear Programming problem with infinitely many solutions.* Minimize $f(x_0, x_1) = x_0 + x_1$ subject to the constraints $0 \le x_0 \le 2$, $0 \le x_1 \le 2$, $x_0 + x_1 \ge 1$ and $x_0 + x_1 \ge 2$. The figure shows the domain of f that is defined by the inequality constraints. The constraint $x_0 + x_1 \ge 1$ does not contribute to defining the domain of f; generally, it is not trivial to identify such contraints. The function values at the vertices of the domain are shown in red.



The function is constant along the domain edge $x_0 + x_1 = 2$, so any pair (x_0, x_1) on this edge is a minimizer point.

The dimension of the LCP is $n = 6$. The LCP quantities of interest are

$$\mathbf{q} = \left[\begin{array}{c} 1 \\ 1 \\ \hline -1 \\ -2 \\ 2 \\ 2 \end{array}\right], \quad M = \left[\begin{array}{cc|cccc} 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \end{array}\right], \quad \mathbf{z} = \left[\begin{array}{c} x_0 \\ x_1 \\ \hline y_0 \\ y_1 \\ y_2 \\ y_3 \end{array}\right], \quad \mathbf{w} = \left[\begin{array}{c} u_0 \\ u_1 \\ \hline v_0 \\ v_1 \\ v_2 \\ v_3 \end{array}\right] \tag{24}$$

The initial dictionary with auxiliary variable z_6 is

$$\begin{aligned} w_0 &= 1 - z_2 - z_3 + z_4 + z_6 \\ w_1 &= 1 - z_2 - z_3 + z_5 + z_6 \\ w_2 &= -1 + z_0 + z_1 + z_6 \\ w_3 &= -2 + z_0 + z_1 + z_6 \\ w_4 &= 2 - z_0 + z_6 \\ w_5 &= 2 - z_1 + z_6 \end{aligned} \tag{25}$$

The fourth equation has minimum **q**-component (-2). Solve for z_6 and substitute this into the other equations,

$$\begin{aligned} w_0 &= 3 - z_0 - z_2 - z_3 + z_4 + w_3 \\ w_1 &= 3 - z_0 - z_2 - z_3 + z_5 + w_3 \\ w_2 &= 1 - w_3 \\ z_6 &= 2 - z_0 - z_1 + w_3 \\ w_4 &= 4 - 2z_0 - z_1 + w_3 \\ w_5 &= 4 - z_0 - 2z_1 + w_3 \end{aligned} \tag{26}$$

The variable w_3 left the dictionary, so z_3 must now enter the dictionary. The first two equations have a negative z_3 coefficient and the same ratio, so either equation can be chosen. Let's solve the first equation for z_3 and substitute this into the other equations,

$$\begin{aligned} z_3 &= 3 - z_0 - z_1 - z_2 - w_0 + z_4 + w_3 \\ w_1 &= 0 + w_0 - z_4 + z_5 \\ w_2 &= 1 - w_3 \\ z_6 &= 2 - z_0 - z_1 + w_3 \\ w_4 &= 4 - 2z_0 - z_1 + w_3 \\ w_5 &= 4 - z_0 - 2z_1 + w_3 \end{aligned} \tag{27}$$

The variable w_0 left the dictionary, so z_0 must now enter the dictionary. Of the four equations with a negative z_0 coefficient, two of them attain the minimum ratio—the equation with z_6 and the equation with w_4, both having ratio 2. Solve the z_6-equation for z_0 and substitute this into the other equations,

$$\begin{array}{rcl} z_3 & = & 1 - z_2 - w_0 + z_4 + z_6 \\ w_1 & = & 0 + w_0 - z_4 + z_5 \\ w_2 & = & 1 - w_3 \\ z_0 & = & 2 - z_6 - z_1 + w_3 \\ w_4 & = & 0 + z_1 + 2z_6 - w_3 \\ w_5 & = & 2 - z_1 + z_6 \end{array} \tag{28}$$

The auxiliary variable z_6 has left the dictionary, so we have solved the LCP. The variables on the right-hand side are set to zero: $z_1 = 0$, $z_2 = 0$, $z_4 = 0$, $z_5 = 0$, $z_6 = 0$, $w_0 = 0$ and $w_3 = 0$. The variables on the left-hand side are then $z_3 = 1$, $w_1 = 0$, $w_2 = 1$, $z_0 = 2$, $w_4 = 0$ and $w_5 = 2$. The original variables that minimize f are $(x_0, x_1) = (z_0, z_1) = (2, 0)$. As noted, this is only one of infinitely many minimizers for f.

When w_0 left the dictionary, we had two choices for the equations leading to the minimum ratio. We chose the z_6-equation for the iteration, which led immediately to a solution $(x_0, x_1) = (2, 0)$. Had we chosen the w_4-equation, two additional iterations are required for z_6 to leave the dictionary. The solution in this case is still $(2, 0)$.

2.4 LCP with No Solution

Example 3 has no solution because a complementary variable cannot enter the dictionary. It mentions a general condition that ensures there is no solution in this case.

Example 3. *Linear Programming problem with no solution.* Minimize $f(x_0, x_1) = 2x_0 - x_1$ for $x_0 \geq 0$, $x_1 \geq 0$ and $x_0 + x_1 \geq 0$. The domain of f is an unbounded convex region in the first quadrant. The dimension of the LCP is $n = 3$. The LCP quantities of interest are

$$\mathbf{q} = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad M = \left[\begin{array}{cc|c} 0 & 0 & -1 \\ 0 & 0 & -1 \\ \hline 1 & 1 & 0 \end{array}\right], \quad \mathbf{z} = \left[\begin{array}{c} x_0 \\ x_1 \\ \hline y_0 \end{array}\right], \quad \mathbf{w} = \left[\begin{array}{c} u_0 \\ u_1 \\ \hline v_0 \end{array}\right], \tag{29}$$

The initial dictionary with auxiliary variable z_3 is

$$\begin{array}{rcl} w_0 & = & 2 - z_2 + z_3 \\ w_1 & = & -1 - z_2 + z_3 \\ w_2 & = & 1 + z_0 + z_1 + z_3 \end{array} \tag{30}$$

The second equation has minimum **q**-component. Solve for z_3 and substitute this into the other equatoins,

$$\begin{aligned} w_0 &= 3 \\ z_3 &= 1 + z_2 + w_1 \\ w_2 &= 2 + z_0 + z_1 + z_3 + w_1 \end{aligned} \tag{31}$$

The variable w_1 left the dictionary, so the complementary variable z_1 must now enter the dictionary. However, its coefficient is not negative, so it cannot enter the dictionary. Therefore, the LCP has no solution. This should be intuitively clear because $f(0, x_1) = -x_1$ which has the limit $-\infty$ as $x_1 \to \infty$; that is, f is not bounded below.

2.5 LCP with a Cycle

I have been unable to construct a cycling example that uses the min-ratio algorithm shown in the previous examples. Searching online for such an example has not been successful. Other pivoting strategies exist for entering and leaving the dictionary. Example 4 uses an alternate strategy that generates a cycle.

Example 4. *Linear Programming problem with a cycle.* An example of a LCP with a cycle in the dictionaries is presented in [3]. The cycle example is for a linear programming problem where the objective function is tracked along with the LCP equations. The variable that enters the dictionary is the one in the objective function that has the largest coefficient. The variable that leaves the dictionary is the basic variable with the smallest index, where the z_i variables are assumed to occur before the w_i variables in the indexing. The smallest-index rule is *Bland's rule.*

2.6 Avoiding Cycles when Constant Terms are Zero

This section shows how to avoid cycles by perturbing the **q**-components with powers of a variable ε. The idea is that when the degeneracy occurs the first time because a component of **q** becomes zero, add ε to it, making that component a linear polynomial of ε. The arithmetic operations of the LCP iterations now involve a symbolic component—manipulating the polynomial itself using addition and scalar multiplication. If another component of **q** becomes zero in a later iteration, then add ε^2 to it, making that component a quadratic polynomial of ε. In worst case, all components of **q** become zero during the iterations and the final component has ε^n added to it for an LCP of dimension n. The polynomials are linearly independent throughout the iterations, so the cycling cannot occur. When the iterations terminate and there is an LCP solution, set ε to zero and report the solution **z** and **w** in the usual manner.

In the GTE implementation of the LCP solver, the code is kept simple by adding the powers of ε to the components of **q** even when those components are not zero. The trade-off is that more computations are required to manipulate the polynomials. Of course, the code can be optimized to reduce computations by inserting the powers of ε only when needed.

Example 5 illustrates the idea for an LCP where at least one of the **q** components becomes zero during the iterations.

Example 5. Minimize $f(x_0, x_1) = (x_0^2 + 2x_1^2)/2 - (x_0 + x_1)$ subject to the constraints $x_0 \geq 0$, $x_1 \geq 0$ and $2x_0 + x_1 \geq 1$. The LCP formulation is the following, where z_3 is the auxiliary variable,

$$\begin{aligned} w_0 &= -1 + z_0 - 2z_2 + z_3 \\ w_1 &= -1 + 2z_1 - z_2 + z_3 \\ w_2 &= -1 + 2z_0 + z_1 + z_3 \end{aligned} \tag{32}$$

The variable z_3 must enter the dictionary via the equation that has the minimum $\mathbf{q}$-component. All components attain the minimum, so choose the first equation to solve for z_3. The variable w_0 exits the dictionary,

$$\begin{aligned} z_3 &= 1 - z_0 + 2z_2 + w_0 \\ w_1 &= 0 - z_0 + 2z_1 + z_2 + w_0 \\ w_2 &= 0 + z_0 + z_1 + 2z_2 + w_0 \end{aligned} \tag{33}$$

The variable z_0 must enter the dictionary. The minimum-ratio term is generated by the second equation, so w_1 must leave the dictionary,

$$\begin{aligned} z_3 &= 1 + w_1 - 2z_1 + z_2 \\ z_0 &= 0 - w_1 + 2z_1 + z_2 + w_0 \\ w_2 &= 0 - w_1 + 3z_1 + 3z_2 + 2w_0 \end{aligned} \tag{34}$$

The variable z_1 must enter the dictionary. The minimum-ratio term is generated by the first equation, so z_3 must leave the dictionary,

$$\begin{aligned} z_1 &= (1/2) + (1/2)w_1 - (1/2)z_3 + (1/2)z_2 \\ z_0 &= 1 - z_3 + 2z_2 + w_0 \\ w_2 &= (3/2) + (1/2)w_1 - (3/2)z_3 + (9/2)z_2 + 2w_0 \end{aligned} \tag{35}$$

The auxiliary variable z_3 has exited the dictionary and the $\mathbf{q}$ coefficients are nonnegative, so we have a unique solution to the LCP: $\mathbf{w} = (0, 0, 3/2)$ and $\mathbf{z} = (1, 1/2, 0)$. The CQP solution is $(x_0, x_1) = (1, 1/2)$. Observe that $\nabla f(x_0, x_1) = (x_0 - 1, 2x_1 - 1)$ and the global minimum occurs when $(x_0 - 1, 2x_1 - 1) = (0, 0)$, so $x_0 = 1$ and $x_1 = 1/2$. This is the solution we found via the LCP. The minimizer point is in the domain defined by the inequality constraints.

Although we did not encounter a cycle, we can still perturb the $\mathbf{q}$ components by powers of ε. The LCP is

$$\begin{aligned} w_0 &= (-1 + \varepsilon) + z_0 - 2z_2 + z_3 \\ w_1 &= (-1 + \varepsilon^2) + 2z_1 - z_2 + z_3 \\ w_2 &= (-1 + \varepsilon^3) + 2z_0 + z_1 + z_3 \end{aligned} \tag{36}$$

Determining the minimum-ratio now depends on comparisons of polynomials. The less-than operation uses lexiographical ordering. If $\mathbf{a}(x) = \sum_{i=0}^{n} a_i x^i$ and $\mathbf{b}(x) = \sum_{i=0}^{n} b_i x^i$, pseudocode for the less-than operation is shown next,

```
bool LessThan(Polynomial a, Polynomial b)
{
    for (int i = 0; i <= n; ++i)
    {
        if (a[i] < b[i])
        {
            return true;
        }

        if (a[i] > b[i])
        {
            return false;
        }
    }

    // At this point, a[i] and b[i] are equal for all i.
    return false;
}
```

Of the 3 equations in the LCP, $(-1+\varepsilon^3) < (-1+\varepsilon)$ and $(-1+\varepsilon^3) < (-1+\varepsilon^2)$, so the last equation has the minimum $\mathbf{q}$ component. The variable z_3 enters the dictionary and the variable w_2 leaves the dictionary,

$$\begin{array}{rcl} w_0 &=& (\varepsilon-\varepsilon^3) - z_0 - z_1 - 2z_2 + w_2 \\ w_1 &=& (\varepsilon^2-\varepsilon^3) - 2z_0 + z_1 - z_2 + w_2 \\ z_3 &=& (1-\varepsilon^3) - 2z_0 - z_1 + w_2 \end{array} \tag{37}$$

The variable z_2 must enter the dictionary. The first two equations are candidates for the pivoting. The ratios are, in order, $(\varepsilon-\varepsilon^3)/2$ and $(\varepsilon^2-\varepsilon^3)$. The second ratio is minimum, so w_1 must leave the dictionary,

$$\begin{array}{rcl} w_0 &=& (\varepsilon-2\varepsilon^2+\varepsilon^3) + 3z_0 - 3z_1 + 2w_1 - w_2 \\ z_2 &=& (\varepsilon^2-\varepsilon^3) - 2z_0 + z_1 - w_1 + w_2 \\ z_3 &=& (1-\varepsilon^3) - 2z_0 - z_1 + w_2 \end{array} \tag{38}$$

The variable z_1 must enter the dictionary. The first and last equations are candidates for the pivoting. The ratios are, in order, $(\varepsilon-2\varepsilon^2+\varepsilon^3)/3$ and $(1-\varepsilon^3)$. The first ratio is minimum, so w_0 must leave the dictionary,

$$\begin{array}{rcl} z_1 &=& ((1/3)\varepsilon - (2/3)\varepsilon^2 + (1/3)\varepsilon^3) + z_0 - (1/3)w_0 + (2/3)w_1 - (1/3)w_2 \\ z_2 &=& ((1/3)\varepsilon + (1/3)\varepsilon^2 - (2/3)\varepsilon^3) - z_0 - (1/3)w_0 - (1/3)w_1 + (2/3)w_2 \\ z_3 &=& (1 - (1/3)\varepsilon + (2/3)\varepsilon^2 - (4/3)\varepsilon^3) - 3z_0 + (1/3)w_0 - (2/3)w_1 + (4/3)w_2 \end{array} \tag{39}$$

The variable z_0 must enter the dictionary. The second and third equations are candidates for the pivoting. The ratios are, in order, $((1/3)\varepsilon - (2/3)\varepsilon^2 + (1/3)\varepsilon^3)$ and $((1/3) - (1/9)\varepsilon + (2/9)\varepsilon^2 - (4/9)\varepsilon^3)$. The first ratio is minimum, so z_2 must leave the dictionary,

$$\begin{array}{rcl} z_1 &=& (0 + (2/3)\varepsilon - (1/3)\varepsilon^2 - (1/3)\varepsilon^3) - z_2 - (2/3)w_0 + (1/3)w_1 + (1/3)w_2 \\ z_0 &=& ((1/3)\varepsilon + (1/3)\varepsilon^2 - (2/3)\varepsilon^3) - z_2 - (1/3)w_0 - (1/3)w_1 + (2/3)w_2 \\ z_3 &=& (1 - (4/3)\varepsilon - (1/3)\varepsilon^2 + (2/3)\varepsilon^3) + 3z_2 + (4/3)w_0 + (1/3)w_1 - (2/3)w_2 \end{array} \tag{40}$$

The variable w_2 must enter the dictionary. The last equation is the only pivoting candidate, so z_3 must leave the dictionary,

$$\begin{aligned} z_1 &= ((1/2) - (1/2)\varepsilon^2) + (1/2)z_2 + (1/2)w_1 - (1/2)z_3 \\ z_0 &= (1 - \varepsilon) + 2z_2 + w_0 - z_3 \\ w_2 &= ((3/2) - 2\varepsilon - (1/2)\varepsilon^2 + \varepsilon^3) + (9/2)z_2 + 2w_0 + (1/2)w_1 - (3/2)z_3 \end{aligned} \tag{41}$$

The auxiliary variable left the dictionary and the $\mathbf{q}$ components with $\varepsilon = 0$ are nonnegative, so we have a unique solution to the LCP: $\mathbf{w} = (0, 0, 3/2)$ and $\mathbf{z} = (1, 1/2, 0)$. This is the same solution we found without the perturbations.

---

3 Formulating a Geometric Query as a CQP

The typical geometric queries that can be formulated as CQPs are distance between objects and test-intersection queries between objects. The latter type of query determines whether or not two objects overlap but does not give information (or gives limited information) about the overlap set.

The first stage for implementing a geometric query is to formulate the corresponding CQP. The second stage is to solve the CQP as an LCP.

3.1 Distance Between Oriented Boxes

Example 6 shows how to set up the convex quadratic programming algorithm for computing the distance between two boxes in any dimension.

Example 6. *Convex Quadratic Programming problem: Distance between boxes in n-dimensions.* A box in n-dimensions can be parameterized by choosing an $n \times 1$ point $\mathbf{k}$ as a box corner, a right-handed orthonormal set of axis directions $\{\mathbf{u}_j\}_{j=0}^{n-1}$ and positive edge lengths $\{\ell_j\}_{j=0}^{n-1}$. A point $\mathbf{p}$ in the box is

$$\mathbf{p}(\boldsymbol{\xi}) = \mathbf{k} + \sum_{j=0}^{n-1} \xi_j \mathbf{u}_j = \mathbf{k} + R\boldsymbol{\xi}, \quad \mathbf{0} \le \boldsymbol{\xi} \le \boldsymbol{\ell} \tag{42}$$

where $\boldsymbol{\xi}$ is an $n \times 1$ vector whose components are the ξ_j, R is the $n \times n$ rotation matrix whose columns are the $\mathbf{u}_j$ and $\boldsymbol{\ell}$ is an $n \times 1$ vector whose components are the ℓ_j.

The goal is to formulate the distance between two boxes as a CQP that can then be solved using an LCP. Let the box centers be $\mathbf{k}_i$, the rotation matrices be R_i and the edge lengths be $\boldsymbol{\ell}_i$. The parameterized boxes are

$$\mathbf{p}_i(\boldsymbol{\xi}_i) = \mathbf{k}_i + R_i\boldsymbol{\xi}_i, \quad \mathbf{0} \le \boldsymbol{\xi}_i \le \boldsymbol{\ell}_i \tag{43}$$

for $i \in \{0, 1\}$. All components are doubly indexed: $\mathbf{p}_i$ has components p_{ij}, $\mathbf{k}_i$ has components k_{ij}, $\mathbf{u}_i$ has components u_{ij}, R_i has columns $\mathbf{u}_i$, $\boldsymbol{\ell}_i$ has components ℓ_{ij} and $\boldsymbol{\xi}_i$ has components ξ_{ij}.

Define $\boldsymbol{\Delta} = \mathbf{k}_1 - \mathbf{k}_0$. Half the squared distance between two points, one point from each box, is

$$\begin{aligned}
f(\mathbf{x}) &= \tfrac{1}{2}\left|\mathbf{p}_0(\boldsymbol{\xi}_0) - \mathbf{p}_1(\boldsymbol{\xi}_1)\right|^2 \\
&= \tfrac{1}{2}\left|R_0\boldsymbol{\xi}_0 - R_1\boldsymbol{\xi}_1 - \boldsymbol{\Delta}\right|^2 \\
&= \tfrac{1}{2}\left(\boldsymbol{\xi}_0^{\mathsf{T}} R_0^{\mathsf{T}} R_0 \boldsymbol{\xi}_0 + \boldsymbol{\xi}_1^{\mathsf{T}} R_1^{\mathsf{T}} R_1 \boldsymbol{\xi}_1 + \boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta} - 2\boldsymbol{\xi}_0^{\mathsf{T}} R_0^{\mathsf{T}} R_1 \boldsymbol{\xi}_1 - 2\boldsymbol{\Delta}^{\mathsf{T}} R_0 \boldsymbol{\xi}_0 + 2\boldsymbol{\Delta}^{\mathsf{T}} R_1 \boldsymbol{\xi}_1\right) \\
&= \tfrac{1}{2}\left[\begin{array}{c|c} \boldsymbol{\xi}_0^{\mathsf{T}} & \boldsymbol{\xi}_1^{\mathsf{T}} \end{array}\right] \left[\begin{array}{c|c} I & -R_0^{\mathsf{T}} R_1 \\ \hline -R_1^{\mathsf{T}} R_0 & I \end{array}\right] \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right] + \left[\begin{array}{c|c} -\boldsymbol{\Delta}^{\mathsf{T}} R_0 & \boldsymbol{\Delta}^{\mathsf{T}} R_1 \end{array}\right] \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right] + \tfrac{1}{2}\boldsymbol{\Delta}^{\mathsf{T}}\boldsymbol{\Delta} \\
&= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}} A \mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c
\end{aligned} \tag{44}$$

where

$$\mathbf{x} = \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right], \quad A = \left[\begin{array}{c|c} I & -R_0^{\mathsf{T}} R_1 \\ \hline -R_1^{\mathsf{T}} R_0 & I \end{array}\right], \quad \mathbf{b} = \left[\begin{array}{c} -R_0^{\mathsf{T}}\boldsymbol{\Delta} \\ \hline R_1^{\mathsf{T}}\boldsymbol{\Delta} \end{array}\right], \quad c = \frac{1}{2}|\boldsymbol{\Delta}|^2, \quad \boldsymbol{\ell} = \left[\begin{array}{c} \boldsymbol{\ell}_0 \\ \hline \boldsymbol{\ell}_1 \end{array}\right] \tag{45}$$

and I is the $n \times n$ identity matrix. Note that $R_0^{\mathsf{T}} R_0 = R_1^{\mathsf{T}} R_1 = I$ because R_0 and R_1 are rotation matrices.

The inequatity constraints are $\mathbf{0} \le \mathbf{x} \le \boldsymbol{\ell}$. The formal statement of the inequality constraints for the quadratic program is $D\mathbf{x} \ge \mathbf{e}$. For the current example,

$$D = \left[\begin{array}{c} -I \\ \hline -I \end{array}\right], \quad \mathbf{e} = \left[\begin{array}{c} -\boldsymbol{\ell}_0 \\ \hline -\boldsymbol{\ell}_1 \end{array}\right] \tag{46}$$

3.2 Intersection of Triangle and Cylinder

Example 7 show how to set up the convex quadratic programming algorithm for testing for intersection between a triangle and a cylinder in any dimension. The motivation is the 3D problem, but notice that the specialization of a cylinder to 2D is a rectangle, so the intersection query is for a triangle and rectangle.

Example 7. *Convex Quadratic Programming problem: Intersection of a triangle and a finite cylinder.* A nondegenerate (solid) triangle in n-dimensions has vertices $\mathbf{v}_i$ for $i \in \{0,1,2\}$ and linearly independent edge directions $\mathbf{d}_j = \mathbf{v}_{j+1} - \mathbf{v}_0$ for $j \in \{0,1\}$. Define the parameter pair $\mathbf{x} = (x_0, x_1)$. The triangle is parameterized by

$$\mathbf{p}(\mathbf{x}) = \mathbf{v}_0 + x_0\mathbf{d}_0 + x_1\mathbf{d}_1 = \mathbf{V}_0 + E\mathbf{x}, \;\; x_0 \ge 0, \; x_1 \ge 0, \; x_0 + x_1 \le 1 \tag{47}$$

where E is an $n \times 2$ matrix whose columns are the edge directions. A (solid) infinite cylinder is the set of points that are within r units of distance from an axis with origin $\mathbf{k}$ and unit-length direction $\mathbf{u}_0$; r is the radius of the cylinder. A (solid) finite cylinder is the infinite cylinder truncated by two hyperplanes $\mathbf{u}_0 \cdot (\mathbf{p} - (\mathbf{k} \pm (h/2)\mathbf{u}_0)) = 0$, keeping only those infinite cylinder points between the two hyperplanes; h is the height of the finite cylinder. Let $\{\mathbf{u}_j\}_{j=0}^{n-1}$ be a right-handed orthonormal basis for $\mathbb{R}^n$ for which the first vector in the set is the finite cylinder axis direction. The finite cylinder is parameterized by

$$\mathbf{k} + t_0\mathbf{u}_0 + \sum_{j=1}^{n-1} t_j\mathbf{u}_j = \mathbf{k} + R\mathbf{t}, \;\; |t_0| \le h/2, \; \sum_{j=1}^{n-1} t_j^2 \le r^2 \tag{48}$$

where $\mathbf{t}$ is an $n \times 1$ vector whose components are the t_j and R is the $n \times n$ rotation matrix whose columns are the $\mathbf{u}_j$.

The triangle and cylinder intersect when there is at least one triangle point within r units of the cylinder axis and between the two truncating planes. We can formulate this using a CQP that minimizes a squared distance. Define $\mathbf{\Delta} = \mathbf{v}_0 - \mathbf{k}$. The matrix that projects vectors onto the plane with origin $\mathbf{0}$ and normal $\mathbf{u}_0$ is $\mathcal{P} = I - \mathbf{u}_0\mathbf{u}_0^{\mathsf{T}}$. The right-hand side of the third equality in the next displayed equation uses two properties of a projection matrix: $\mathcal{P}^{\mathsf{T}} = \mathcal{P}$ and $\mathcal{P}^2 = \mathcal{P}$. Half the squared distance between a triangle point and a cylinder axis point is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2}\left|\mathcal{P}\left(\mathbf{P}(\mathbf{x}) - \mathbf{k}\right)\right|^2 \\ &= \tfrac{1}{2}\left|\mathcal{P}\left(E\mathbf{x} + \mathbf{\Delta}\right)\right|^2 \\ &= \tfrac{1}{2}\left(E\mathbf{x} + \mathbf{\Delta}\right)^{\mathsf{T}}\mathcal{P}\left(E\mathbf{x} + \mathbf{\Delta}\right) \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}E^{\mathsf{T}}\mathcal{P}E\mathbf{x} + \mathbf{\Delta}^{\mathsf{T}}\mathcal{P}E\mathbf{x} + \tfrac{1}{2}\mathbf{\Delta}^{\mathsf{T}}\mathcal{P}\mathbf{x} \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \end{aligned} \tag{49}$$

where $A = E^{\mathsf{T}}\mathcal{P}E$, $\mathbf{b} = E^{\mathsf{T}}\mathcal{P}^{\mathsf{T}}\mathbf{\Delta} = E^{\mathsf{T}}\mathcal{P}\mathbf{\Delta}$ and $c = |\mathbf{\Delta}|^2/2$.

The components of x are nonnegative. The other inequality constraints are

$$x_0 + x_1 \le 1, \quad h/2 \ge |t_0| = |\mathbf{u}_0 \cdot (\mathbf{p}(\mathbf{x}) - \mathbf{k})| = |\mathbf{u}_0 \cdot (E\mathbf{x} + \mathbf{\Delta})| = |\mathbf{u}_0^{\mathsf{T}}E\mathbf{x} + \mathbf{U}_0^{\mathsf{T}}\mathbf{\Delta}| \tag{50}$$

In terms of the formal inequality constraints $D\mathbf{x} \ge \mathbf{e}$, we have

$$D = \begin{bmatrix} -\mathbf{1}^{\mathsf{T}} \\ \mathbf{u}_0^{\mathsf{T}}E \\ -\mathbf{u}_0^{\mathsf{T}}E \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \\ -h/2 - \mathbf{u}_0^{\mathsf{T}}\mathbf{\Delta} \\ -h/2 + \mathbf{u}_0^{\mathsf{T}}\mathbf{\Delta} \end{bmatrix} \tag{51}$$

where D is a 3×2 matrix where $\mathbf{1}$ is a 2×1 vector whose components are both 1. The vector $\mathbf{e}$ is 3×1.

An LCP solver is used to compute the minimizer $\hat{\mathbf{x}}$ and the corresponding minimum value $\hat{f} = f(\hat{\mathbf{x}})$. The minimum squared distance is $2\hat{f}$. The triangle and cylinder intersect whenever $2\hat{f} \le r^2$.

4 Implementation Details

The GTE source code that uses an LCP solver is designed to allow you to use fixed-precision floating-point arithmetic (float or double) or arbitrary-precision floating-point arithmetic (via BSRational). See the document GTE: Arbitrary Precision Arithmetic for details. The latter type allows the LCP solver to produce the exact result under the assumption that the inputs are error free; that is, the inputs are assumed to be finite floating-point numbers that, of course, are rational numbers. Any knowledge about numerical rounding errors in producing the inputs is unknown to the LCP solver, so it cannot take advantage of it.

Various geometric primitives have representations that include unit-length vectors. This is problematic when using arbitrary-precision floating-point arithmetic because typically those vectors are obtained by dividing a floating-point vector by its length. The length involves a square root operation, which generally (as a real number) is irrational and requires a numerical approximation to represent it. A section is included on how to deal with the normalization symbolically, a concept related to the abstract algebraic topic of *real quadratic fields*.

4.1 The LCP Solver

The LCP solver in GTE is a straightforward implementation of the algorithm used in Examples 1, 2 and 3. The solver also uses the symbolic perturbation described previously to avoid degeneracy and cycles in the iterations.

Listing 1 shows the public interfaces for the classes used to solve LCPs. The actual source code is found online at LCPSolver.h.

Listing 1. The LCPSolverShared base class encapsulates the support for setting the maximum number of iterations used by the LCP solver and for querying the actual number of iteration used. The Result enumeration is used by derived classes to report the outcome of the solver. The two derived classes include one that uses std::array when the dimension of the LCP is known at compile time and one that uses std::vector when the dimension of the LCP is known only at run time.

```
template <typename Real>
class LCPSolverShared
{
protected:
    // Abstract base class construction.  A virtual destructor is not provided
    // because there are no required side effects when destroying objects from
    // the derived classes.  The member mMaxIterations is set by this call to
    // the default value of n*n.
    LCPSolverShared(int n);

public:
    // Theoretically, when there is a solution the algorithm must converge
    // in a finite number of iterations.  The number of iterations depends
    // on the problem at hand, but we need to guard against an infinite loop
    // by limiting the number.  The implementation uses a maximum number of
    // n*n (chosen arbitrarily).  You can set the number yourself, perhaps
    // when a call to Solve fails--increase the number of iterations and call
    // Solve again.
    inline void SetMaxIterations(int maxIterations);
    inline int GetMaxIterations() const;

    // Access the actual number of iterations used in a call to Solve.
    inline int GetNumIterations() const;

    enum Result
    {
        HAS_TRIVIAL_SOLUTION,
        HAS_NONTRIVIAL_SOLUTION,
        NO_SOLUTION,
        FAILED_TO_CONVERGE,
        INVALID_INPUT
    };
};

template <typename Real, int n>
class LCPSolver<Real, n> : public LCPSolverShared<Real>
{
public:
    // Construction.  The member mMaxIterations is set by this call to the
    // default value of n*n.
    LCPSolver();

    // If you want to know specifically why 'true' or 'false' was returned,
    // pass the address of a Result variable as the last parameter.
    bool Solve(std::array<Real, n> const& q, std::array<std::array<Real, n>, n> const& M,
        std::array<Real, n>& w, std::array<Real, n>& z,
        typename LCPSolverShared<Real>::Result* result = nullptr);
};
```

```
template <typename Real>
class LCPSolver<Real> : public LCPSolverShared<Real>
{
public:
    // Construction.  The member mMaxIterations is set by this call to the
    // default value of n*n.
    LCPSolver(int n);

    // The input q must have n elements and the input M must be an n-by-n
    // matrix stored in row-major order.  The outputs w and z have n elements.
    // If you want to know specifically why 'true' or 'false' was returned,
    // pass the address of a Result variable as the last parameter.
    bool Solve(std::vector<Real> const& q, std::vector<Real> const& M,
        std::vector<Real>& w, std::vector<Real>& z,
        typename LCPSolverShared<Real>::Result* result = nullptr);
};
```

4.2 Distance Between Oriented Boxes in 3D

Example 6 shows the construction of the CQP for computing the distance between two oriented boxes in n dimensions. Listing 2 shows pseudocode for computing the distance between two oriented boxes in 3 dimensions. The representations of an oriented box in GTE and in Wild Magic use a center point and extents (half-lengths), so there is a small adjustment to compute the corners and lengths of the boxes.

Listing 2. The listing contains pseudocode for computing the distance between two oriented boxes in 3 dimensions. A box is parameterized by $\mathbf{p} = \mathbf{c} + \sum_{i=0}^{2} x_i \mathbf{u}_i$ with $|x_i| \leq e_i$. The point $\mathbf{c}$ is the box center and e_i are extents, which are half the edge lengths of the box edges.

```
template <typename Real>
struct Box3
{
    Point3<Real> center;
    Vector3<Real> axis[3];
    Real extent[3];
};

template <typename Real>
struct Box3Box3QueryResult
{
    // Specify the maximum number of LCP iterations.  The default in GTE
    // is N^2 for an LCP with Nx1 vector q and NxN matrix M.  The convergence
    // is not guaranteed to occur within N^2 iterations, so a conservative
    // approach in an application is to examine 'status' after the query.  If
    // the value is FAILED_TO_CONVERGE, repeat with a larger maxLCPIterations
    // if so desired.
    int maxLCPIterations;

    // The number of iterations used by LCPSolver regardless of whether
    // or not the query is successful.
    int numLCPIterations;

    // The information returned by the LCP solver about what it discovered.
    LCPSolver<Real, 12>::Result status;

    // These members are valid only when queryIsSuccessful is true;
    // otherwise, they are all set to zero.
    Real distance, sqrDistance;
    std::array<Real, 3> box0Parameter;  // the x_i for box0
    std::array<Real, 3> box1Parameter;  // the x_i for box1
    Vector3<Real> closestPoint[2];  // (P_0,P_1) wher P_0 is in box0 and P_1 is in box1
};
```

```
// Set result.maxLCPIterations to the desired value before calling this function.
template <typename Real>
void ComputeDistanceAndClosestPoints(Box3<Real> box0, Box3<Real> box1,
    Box3Box3QueryResult<Real>& result)
{
    // Compute the box corners and difference of corners.
    Point3<Real> K0 = box0.center, K1 = box1.center;
    for (int r = 0; r < 3; ++r)
    {
      K0 -= box0.extent[r] * box0.axis[r];
      K1 -= box1.extent[r] * box1.axis[r];
    }
    Vector3<Real> Delta = K1 - K0;

    // Compute R0^T * Delta and R1^T * Delta.
    Vector3<Real> R0TDelta, R1TDelta;
    for (int r = 0; r < 3; ++r)
    {
        R0TDelta[r] = Dot(box0.axis[r], Delta);
        R1TDelta[r] = Dot(box1.axis[r], Delta);
    }

    // Compute R0^T * R1.
    std::array<std::array<Real, 3>, 3> R0TR1;
    for (int r = 0; r < 3; ++r)
    {
        for (int c = 0; c < 3; ++c)
        {
            R0TR1[r][c] = Dot(box0.axis[r], box1.axis[c]);
        }
    }

    // Compute the lengths from the extents (half-lengths).
    std::array<Real, 3> length0, length1;
    for (int r = 0; r < 3; ++r)
    {
        length0[r] = 2 * box0.extent[r];
        length1[r] = 2 * box1.extent[r];
    }

    // The LCP has 6 variables and 6 nontrivial inequality constraints.
    std::array<Real, 12> q =
    {
       -R0TDelta[0], -R0TDelta[1], -R0TDelta[2], R1TDelta[0], R1TDelta[1], R1TDelta[2],  // b
        length0[0], length0[1], length0[2], length1[0], length1[1], length1[2]  // -e
    };

    std::array<std::array<Real, 12>, 12> M;  // {{ A, -D^T }, { D, 0 }}
    {
       M[ 0] = { 1, 0, 0, -R0TR1[0][0], -R0TR1[0][1], -R0TR1[0][2],   1, 0, 0, 0, 0, 0 };
       M[ 1] = { 0, 1, 0, -R0TR1[1][0], -R0TR1[1][1], -R0TR1[1][2],   0, 1, 0, 0, 0, 0 };
       M[ 2] = { 0, 0, 1, -R0TR1[2][0], -R0TR1[2][1], -R0TR1[2][2],   0, 0, 1, 0, 0, 0 };
       M[ 3] = { -R0TR1[0][0], -R0TR1[1][0], -R0TR1[2][0], 1, 0, 0,   0, 0, 0, 1, 0, 0 };
       M[ 4] = { -R0TR1[0][1], -R0TR1[1][1], -R0TR1[2][1], 0, 1, 0,   0, 0, 0, 0, 1, 0 };
       M[ 5] = { -R0TR1[0][2], -R0TR1[1][2], -R0TR1[2][2], 0, 0, 1,   0, 0, 0, 0, 0, 1 };

       M[ 6] = { -1,  0,  0,  0,  0,  0,                               0, 0, 0, 0, 0, 0 };
       M[ 7] = {  0, -1,  0,  0,  0,  0,                               0, 0, 0, 0, 0, 0 };
       M[ 8] = {  0,  0, -1,  0,  0,  0,                               0, 0, 0, 0, 0, 0 };
       M[ 9] = {  0,  0,  0, -1,  0,  0,                               0, 0, 0, 0, 0, 0 };
       M[10] = {  0,  0,  0,  0, -1,  0,                               0, 0, 0, 0, 0, 0 };
       M[11] = {  0,  0,  0,  0,  0, -1,                               0, 0, 0, 0, 0, 0 };
    };

    LCPSolver<Real, 12> lcp;
    lcp.SetMaxLCPIterations(result.maxLCPIterations);
    std::array<Real, 12> w, z;
    if (lcp.Solve(q, M, w, z, &result.status))
    {
        result.closestPoint[0] = box0.center;
        for (int i = 0; i < 3; ++i)
        {
            result.box0Parameter[i] = z[i] - box0.extent[i];
            result.closestPoint[0] += result.box0Parameter[i] * box0.axis[i];
        }

        result.closestPoint[1] = box1.center;
        for (int i = 0, j = 3; i < 3; ++i, ++j)
        {
            result.box1Parameter[i] = z[j] - box1.extent[i];
            result.closestPoint[1] += result.box1Parameter[i] * box1.axis[i];
        }

        Vector3<Real> diff = result.closestPoint[1] - result.closestPoint[0];
        result.sqrDistance = Dot(diff, diff);
        result.distance = sqrt(result.sqrDistance);
    }
    else
    {
        // If you reach this case, the value of 'result' is one of
        // NO_SOLUTION or FAILED_TO_CONVERGE.  The value INVALID_INPUT
        // occurs only when the LCPSolver is passed std::vector inputs
        // whose dimensions are not correct.
        for (int i = 0; i < 3; ++i)
        {
            result.box0Parameter[i] = 0;
            result.box1Parameter[i] = 0;
            result.closestPoint[0][i] = 0;
            result.closestPoint[1][i] = 0;
        }
        result.distance = 0;
        result.sqrDistance = 0;
    }

    result.numLCPIterations = lcp.GetNumIterations();
}
```

4.3 Intersection of Triangle and Cylinder in 3D

Example 7 shows the construction of the CQP for testing for intersection of a triangle and a finite cylinder in n dimensions. Listing 3 shows pseudocode for this query in 3 dimensions.

Listing 3. The listing contains pseudocode for testing for the intersection of a triangle and a finite cylinder in 3 dimensions.

```
template <typename Real>
struct Triangle3
{
    Point3<Real> vertex[3];
};

template <typename Real>
struct Cylinder3
{
    Point3<Real> center;
    Vector3<Real> direction;
    Real radius;
    Real height;
};

template <typename Real>
struct Triangle3Cylinder3QueryResult
{
    // Specify the maximum number of LCP iterations.  The default in GTE
    // is N^2 for an LCP with Nx1 q and NxN M.  The convergence is not
    // guaranteed to occur within N^2 iterations, so a conservative approach
    // in an application is to examine 'status' after the query.  If the value
    // is FAILED_TO_CONVERGE, repeat with a larger maxLCPIterations if so desired.
    int maxLCPIterations;

    // The number of iterations used by LCPSolver regardless of whether
    // or not the query is successful.
    int numLCPIterations;

    // The information returned by the LCP solver about what it discovered.
    LCPSolver<Real, 5>::Result status;

    // The query is test-intersection that returns only a Boolean result.
    bool intersects;
};

// Set result.maxLCPIterations to the desired value before calling this function.
template <typename Real>
void TestIntersection(Triangle3<Real> triangle, Cylinder3<Real> cylinder,
    Triangle3Cylinder3QueryResult<Real>& result)
{
    Vector3<Real> delta = triangle.vertex[0] - cylinder.center;
    Vector3<Real> edge0 = triangle.vertex[1] - triangle.vertex[0];
    Vector3<Real> edge1 = triangle.vertex[2] - triangle.vertex[0];
    Matrix<Real, 3, 2> E;
    E[0][0] = edge0[0];  E[0][1] = edge1[0];
    E[1][0] = edge0[1];  E[1][1] = edge1[1];
    E[2][0] = edge0[2];  E[2][1] = edge1[2];
    Matrix<Real, 3, 3> P = Matrix<Real, 3, 3>::Identity()
        - OuterProduct(cylinder.direction, cylinder.direction);

    Matrix<Real, 2, 3> ETP = Transpose(E) * P;
    Matrix<Real, 2, 2> A = ETP * E;
    Vector2<Real> b = ETP * delta;
    Vector2<Real> U0TE = cylinder.direction * E;
    Real U0Tdelta = Dot(cylinder.direction, delta);
    Matrix<Real, 3, 2> D;
    D[0][0] = -1;         D[0][1] = -1;
    D[1][0] = U0TE[0];    D[1][1] = U0TE[1];
    D[2][0] = -U0TE[0];   D[2][1] = -U0TE[1];
    Vector3<Real> e;
    e[0] = -1.0;
    e[1] = -0.5 * cylinder.height - U0Tdelta;
    e[1] = -0.5 * cylinder.height + U0Tdelta;

    std::array<Real, 5> q = { b[0], b[1], -e[0], -e[1], -e[2] };
    std::array<std::array<Real, 5>, 5> M;
    {
      M[0] = { A[0][0], A[0][1],   -D[0][0], -D[1][0], -D[2][0] },
      M[1] = { A[1][0], A[1][1],   -D[0][1], -D[1][1], -D[2][1] },

      M[2] = { D[0][0], D[0][1],    0, 0, 0 },
      M[3] = { D[1][0], D[1][1],    0, 0, 0 },
      M[4] = { D[2][0], D[2][1],    0, 0, 0 }
    };


    LCPSolver<Real, 5> lcp;
    lcp.SetMaxLCPIterations(result.maxLCPIterations);
    std::array<Real, 5> w, z;
    LCPSolver<Real, 5> lcp;
    if (lcp.Solve(q, M, w, z, &result.status))
    {
        result.intersects = true;
    }
    else
    {
        // If you reach this case, the value of 'result' is one of
        // NO_SOLUTION or FAILED_TO_CONVERGE.  The value INVALID_INPUT
        // occurs only when the LCPSolver is passed std::vector inputs
        // whose dimensions are not correct.
        result.intersects = false;
    }
    result.numLCPIterations = lcp.GetNumIterations;
}
```

4.4 Accuracy Problems when using Fixed-Precision Floating-Point Arithmetic

Although the LCP solver allows for fixed-precision or arbitrary-precision floating-point arithmetic, certain geometric configurations can produce inaccurate results when using fixed-precision. The problem is that rounding errors can cause the choices of basic and nonbasic variables in the pivoting of the LCP tableau to be different from those when using arbitrary-precision arithmetic.

In particular, the function LCPSolverShared<Real>::Solve in LCPSolver.h has a block of code

```
    if (Augmented(r, driving) < (Real)0)
    {
        // execute when the coefficient of the nonbasic variable is negative
    }
```

Rounding errors can lead to a misclassification. The arbitrary-precision code will enter the conditional block when the coefficient is negative—no matter how small the magnitude—but the fixed-precision code will not when rounding errors cause the computed coefficient to be a small positive number. The opposite can also happen, where the arbitrary-precision code skips the conditional block but the fixed-precision code enters it.

An example for inaccurate results due to rounding error is shown next when computing the distance between a triangle and an oriented box in 3D. The LCP solver code is DistTriangle3AlignedBox3.h. Listing 4 shows a test program that computes the distance using fixed precision and using arbitrary precision.

Listing 4. The listing contains an example for an inaccurate distance calculation because of rounding errors when using fixed-precision floating-point arithmetic.

```
int main()
{
    Triangle3<double> triangle;
    triangle.v[0] = { 0.5, 0.5, 1.5 };
    triangle.v[1] = { 0.50000000000000178, 25.5, 1.5 };
    triangle.v[2] = { -0.50000000000000355, 0.5, 1.5 };

    AlignedBox3<double> box;
    box.min = { -28.666800635711962, 12.285771701019407, -48.666800635711965 };
    box.max = { -20.476286168365689, 20.476286168365682, -40.476286168365689 };

    DCPQuery<double, Triangle3<double>, AlignedBox3<double>> query;
    auto result = query(triangle, box);
    // result.queryIsSuccessful = true
    // result.distance = 47.6918933732887069
    // result.sqrDistance = 2274.5166935291390473
    // result.triangleParameter = (0.0199525116590519, 0.4351332588306535, 0.5449142295102947)
    // result.boxParameter = (-22.6653617332430883, 12.2857717010194065, -40.4762861683656610)
    // result.closestPoint[0] = (-0.0449142295102958, 11.3783314707663372, 1.5000000000000000)
    // result.closestPoint[1] = (-22.6653617332430883, 12.2857717010194065, -40.4762861683656610)
    // result.numLCPIterations = 11

    typedef BSRational<UIntegerAP32> Rational;
    Triangle3<Rational> rtriangle;
    rtriangle.v[0] = { 0.5, 0.5, 1.5 };
    rtriangle.v[1] = { 0.50000000000000178, 25.5, 1.5 };
    rtriangle.v[2] = { -0.50000000000000355, 0.5, 1.5 };
    AlignedBox3<Rational> rbox;
    rbox.min = { -28.666800635711962, 12.285771701019407, -48.666800635711965 };
    rbox.max = { -20.476286168365689, 20.476286168365682, -40.476286168365689 };

    DCPQuery<Rational, Triangle3<Rational>, AlignedBox3<Rational>> rquery;
    auto rresult = rquery(rtriangle, rbox);
    // rresult.queryIsSuccessful = true
    // rresult.distance = 46.6845780373756085
    // rresult.sqrDistance = 2179.4498265278130020
    // rresult.triangleParameter = (0.0000000000000000, 0.4387667833180821, 0.5612332166819179)
    // rresult.boxParameter = (-20.4762861683656894, 12.2857717010194065, -40.4762861683656894)
    // rresult.closestPoint[0] = (-0.0612332166819192, 11.4691695829520519, 1.5000000000000000)
    // rresult.closestPoint[1] = (-20.4762861683656894, 12.2857717010194065, -40.4762861683656894)
    // rresult.numLCPIterations = 7
    return 0;
}
```

The relative error in the distance is approximately 0.0216. The pairs of closest points are approximately the same in the y- and z-components, but they differ by a significant amount in the x-component.

The geometric issue is that the plane of the triangle is parallel to a face of the box. A very small rotation of the plane of the triangle, say, about the center of the triangle, can cause a large change in the closest points. The closest points can vary greatly with small changes in the triangle vertices.

If you must use fixed-precision floating-point arithmetic, the problems with parallel configurations in the geometric primitives should be handled differently. In the next major release of the source code (the Geometric Tools Library), LCP-based algorithms are provided for the queries, but specialized algorithms will also be provided that try to resolve the accuracy problems with parallel configurations.

4.5 Dealing with Vector Normalization

To motivate the discussion, consider Example 7 analyzed previously for the test-intersection query between a triangle and a finite cylinder in 3 dimensions. The construction of the matrices and vectors in the CQP assumes real-valued arithmetic (error-free computations). In particular, the cylinder axis direction is a unit-length vector $\mathbf{u}_0$.

The problem in an implementation is that if the axis direction is computed by normalizing a vector, and then that direction is passed to the query and treated as a 3-tuple of rational numbers, the length is not guaranteed to be 1 (due to rounding errors). For example, suppose the cylinder axis is in the direction of $(1, 2, 3)$. The normalized vector is $(1, 2, 3)/\sqrt{14}$. The normalization code is

```
Vector3<double> u0 = { 1.0, 2.0, 3.0 };
double length = sqrt(u0[0] * u0[0] + u0[1] * u0[1] + u0[2] * u0[2]);  // = sqrt(14.0)
u0 /= length;
// u0 = (0.26726124191242440, 0.53452248382484879, 0.80178372573727319)

typedef BSRational<UIntegerAP32> Rational;
Vector3<Rational> ru0 = { u0[0], u0[1], u0[2] };
Rational rSqrLength = Dot(ru0, ru0);
// rSqrLength.biasedExponent = -105
// rSqrLength.bits = 0x00000200 0x00000000 0x000cc8b2 0xff10b80f
// Moving the binary point from the right-most bit 105 units to the left,
//   rSqrLength = 1.0^{53}110011001000101100101011111111100010000101110000000001111
// where 0^{53} denotes the occurrence of 53 0-valued bits.  Therefore,
//   rSqrLength = 1.t where t > 0
```

Suppose that $\mathbf{u}_0$ was normalized from a vector $\mathbf{v}$; that is, $\mathbf{u}_0 = \mathbf{v}/|\mathbf{v}|$. The vector $\mathbf{v}$ has rational components but its length $|\mathbf{v}|$ is usually irrational. Replace this expression in the CQP for the triangle-cylinder test-intersection query. The projection matrix is $\mathcal{P} = I - \mathbf{v}\mathbf{v}^{\mathsf{T}}/|\mathbf{v}|^2$ and can be computed exactly using rational arithmetic because of the occurrence of the squared distance. The quadratic matrix is $A = E^{\mathsf{T}}\mathcal{P}E$ which is also rational because E involves quantities generated by the differences of rational points. The quadratic vector $\mathbf{b} = E^{\mathsf{T}}\mathcal{P}\boldsymbol{\Delta}$, which is also rational. The quadratic scalar $c = |\boldsymbol{\Delta}|^2/2$ is rational.

Two of the inequality constraints in $D\mathbf{x} \geq \mathbf{e}$ involve the length $|\mathbf{v}|$,

$$(\mathbf{v}/|\mathbf{v}|)^{\mathsf{T}}E\mathbf{x} \geq -h/2 - (\mathbf{v}/|\mathbf{v}|)^{\mathsf{T}}\boldsymbol{\Delta}, \;\; -(\mathbf{v}/|\mathbf{v}|)^{\mathsf{T}}E\mathbf{x} \geq -h/2 + (\mathbf{v}/|\mathbf{v}|)^{\mathsf{T}}\boldsymbol{\Delta} \tag{52}$$

Multiplying the inequalites by the length eliminates the division, but the length term itself cannot be eliminated,

$$\mathbf{v}^{\mathsf{T}}E\mathbf{x} \geq -h|\mathbf{v}|/2 - \mathbf{v}^{\mathsf{T}}\boldsymbol{\Delta}, \;\; -\mathbf{v}^{\mathsf{T}}E\mathbf{x} \geq -h|\mathbf{v}|/2 + \mathbf{v}^{\mathsf{T}}\boldsymbol{\Delta} \tag{53}$$

If $|\mathbf{v}|$ is irrational, we can approximate it by a rational number and then execute the LCP solver using arbitrary-precision floating-point arithmetic. However, the resulting minimizer point $\mathbf{x}$ and corresponding minimum function value $f(\mathbf{x})$ are considered to be approximations.

It is possible to avoid the approximation of the length of a vector that is an input to the LCP solver by using real quadratic fields. The idea is to introduce a symbolic component to the computations that involves the vector length as the square root of a rational number. Details for such an approach can be found in GTE: Arbitrary Precision Arithmetic.

To illustrate the use of real quadratic fields, consider the LCP formulation of the convex quadratic program for determining whether a triangle and cylinder intersect. The implementations shown next are for double-precision floating-point arithmetic, for rational arithmetic and for a real quadratic field where d is the rational squared length of the cylinder axis direction.

Listing 5 shows the source code for the query when the numeric type is `double` (64-bit floating-point arithmetic).

Listing 5. The listing uses double-precision arithmetic for executing the LCP solver for triangle-cylinder intersection. The computations necessarily have rounding errors.

```
std::array<double, 2> ExecuteDouble(Triangle3<double> const& triangle, Cylinder3<double> const& cylinder)
{
    Vector3<double> delta = triangle.v[0] - cylinder.axis.origin;
    Vector3<double> edge1 = triangle.v[1] - triangle.v[0];
    Vector3<double> edge2 = triangle.v[2] - triangle.v[0];
    Matrix<3, 2, double> E;
    E.SetCol(0, edge1);
    E.SetCol(1, edge2);
    Matrix<3, 3, double> P = Matrix<3, 3, double>::Identity() -
        OuterProduct(cylinder.axis.direction, cylinder.axis.direction);

    Matrix<2, 3, double> ETP = MultiplyATB(E, P);
    Matrix<2, 2, double> A = ETP * E;
    Vector2<double> b = ETP * delta;
    Vector2<double> u0TE = cylinder.axis.direction * E;
    double u0Tdelta = Dot(cylinder.axis.direction, delta);
    Matrix<3, 2, double> D;
    D(0, 0) = -1.0;
    D(0, 1) = -1.0;
    D(1, 0) = u0TE[0];
    D(1, 1) = u0TE[1];
    D(2, 0) = -u0TE[0];
    D(2, 1) = -u0TE[1];
    Vector3<double> e;
    e[0] = -1.0;
    e[1] = -0.5 * cylinder.height - u0Tdelta;
    e[2] = -0.5 * cylinder.height + u0Tdelta;

    std::array<double, 5> q = { b[0], b[1], -e[0], -e[1], -e[2] };
    std::array<std::array<double, 5>, 5> M;
    {
        M[0] = { A(0, 0), A(0, 1), -D(0, 0), -D(1, 0), -D(2, 0) };
        M[1] = { A(1, 0), A(1, 1), -D(0, 1), -D(1, 1), -D(2, 1) };
        M[2] = { D(0, 0), D(0, 1), 0.0, 0.0, 0.0 };
        M[3] = { D(1, 0), D(1, 1), 0.0, 0.0, 0.0 };
        M[4] = { D(2, 0), D(2, 1), 0.0, 0.0, 0.0 };
    }

    std::array<double, 5> w, z;
    LCPSolver<double, 5> lcp;
    lcp.Solve(q, M, w, z);

    std::array<double, 2> result = { z[0], z[1] };
    return result;
}
```

The returned numbers are the triangle parameters for determining the triangle point closest to the cylinder axis and that is between the two planes of the cylinder caps.

Listing 6 shows the source code for the query when the numeric type is BSRational<UIntegerAP32> (arbitrary-precision arithmetic).

Listing 6. The listing uses exact rational arithmetic for executing the LCP solver for triangle-cylinder intersection. The computations can be inaccurate when the cylinder axis direction is not unit length when computed as the square root of the sum of squares of rational components.

```
typedef BSRational<UIntegerAP32> Rational;

std::array<Rational, 2> ExecuteRational(Triangle3<double> const& inTri, Cylinder3<double> const& inCyl)
{
    Triangle3<Rational> triangle;
    triangle.v[0] = { inTri.v[0][0], inTri.v[0][1], inTri.v[0][2] };
    triangle.v[1] = { inTri.v[1][0], inTri.v[1][1], inTri.v[1][2] };
    triangle.v[2] = { inTri.v[2][0], inTri.v[2][1], inTri.v[2][2] };

    Cylinder3<Rational> cylinder;
    cylinder.axis.origin =
        { inCyl.axis.origin[0], inCyl.axis.origin[1], inCyl.axis.origin[2] };
    cylinder.axis.direction =
        { inCyl.axis.direction[0], inCyl.axis.direction[1], inCyl.axis.direction[2] };
    cylinder.radius = inCyl.radius;
    cylinder.height = inCyl.height;

    Vector3<Rational> delta = triangle.v[0] - cylinder.axis.origin;
    Vector3<Rational> edge1 = triangle.v[1] - triangle.v[0];
    Vector3<Rational> edge2 = triangle.v[2] - triangle.v[0];
    Matrix<3, 2, Rational> E;
    E.SetCol(0, edge1);
    E.SetCol(1, edge2);
    Matrix<3, 3, Rational> P = Matrix<3, 3, Rational>::Identity() -
        OuterProduct(cylinder.axis.direction, cylinder.axis.direction);

    Matrix<2, 3, Rational> ETP = MultiplyATB(E, P);
    Matrix<2, 2, Rational> A = ETP * E;
    Vector2<Rational> b = ETP * delta;
    Vector2<Rational> u0TE = cylinder.axis.direction * E;
    Rational u0Tdelta = Dot(cylinder.axis.direction, delta);
    Matrix<3, 2, Rational> D;
    Rational rNegOne(-1), rNegHalf(-0.5), rZero(0);
    D(0, 0) = rNegOne;
    D(0, 1) = rNegOne;
    D(1, 0) = u0TE[0];
    D(1, 1) = u0TE[1];
    D(2, 0) = -u0TE[0];
    D(2, 1) = -u0TE[1];
    Vector3<Rational> e;
    e[0] = rNegOne;
    e[1] = rNegHalf * cylinder.height - u0Tdelta;
    e[2] = rNegHalf * cylinder.height + u0Tdelta;

    std::array<Rational, 5> q = { b[0], b[1], -e[0], -e[1], -e[2] };
    std::array<std::array<Rational, 5>, 5> M;
    {
        M[0] = { A(0, 0), A(0, 1), -D(0, 0), -D(1, 0), -D(2, 0) };
        M[1] = { A(1, 0), A(1, 1), -D(0, 1), -D(1, 1), -D(2, 1) };
        M[2] = { D(0, 0), D(0, 1), rZero, rZero, rZero };
        M[3] = { D(1, 0), D(1, 1), rZero, rZero, rZero };
        M[4] = { D(2, 0), D(2, 1), rZero, rZero, rZero };
    }

    std::array<Rational, 5> w, z;
    LCPSolver<Rational, 5> lcp;
    lcp.Solve(q, M, w, z);

    std::array<Rational, 2> result = { z[0], z[1] };
    return result;
}
```

The returned numbers are the triangle parameters for determining the triangle point closest to the cylinder axis and that is between the two planes of the cylinder caps.

Listing 7 shows the source code for the query when the numeric type is QFElement for a real quadratic field.

Listing 7. The listing uses arithmetic for a real quadratic field when executing the LCP solver for triangle-cylinder intersection. The computations are exact in the sense of returning parameters of the form $x + y\sqrt{d}$ where x and y are rational numbers and $\sqrt{d}$ is represented symbolically.

```
typedef BSRational<UIntegerAP32> Rational;
typedef QFElement<Rational, 0> QFType;
Rational QFType::DSqr;

std::array<QFType, 2> ExecuteQFType(Triangle3<double> const& inTri, Cylinder3<double> const& inCyl)
{
    Triangle3<Rational> triangle;
    triangle.v[0] = { inTri.v[0][0], inTri.v[0][1], inTri.v[0][2] };
    triangle.v[1] = { inTri.v[1][0], inTri.v[1][1], inTri.v[1][2] };
    triangle.v[2] = { inTri.v[2][0], inTri.v[2][1], inTri.v[2][2] };

    Cylinder3<Rational> cylinder;
    cylinder.axis.origin =
        { inCyl.axis.origin[0], inCyl.axis.origin[1], inCyl.axis.origin[2] };
    cylinder.axis.direction =
        { inCyl.axis.direction[0], inCyl.axis.direction[1], inCyl.axis.direction[2] };
    cylinder.radius = inCyl.radius;
    cylinder.height = inCyl.height;

    QFType::DSqr = Dot(cylinder.axis.direction, cylinder.axis.direction);

    Vector3<Rational> delta = triangle.v[0] - cylinder.axis.origin;
    Vector3<Rational> edge1 = triangle.v[1] - triangle.v[0];
    Vector3<Rational> edge2 = triangle.v[2] - triangle.v[0];
    Matrix<3, 2, Rational> E;
    E.SetCol(0, edge1);
    E.SetCol(1, edge2);
    Matrix<3, 3, Rational> P = Matrix<3, 3, Rational>::Identity() -
        OuterProduct(cylinder.axis.direction, cylinder.axis.direction) / QFType::DSqr;

    Matrix<2, 3, Rational> ETP = MultiplyATB(E, P);
    Matrix<2, 2, Rational> A = ETP * E;
    Vector2<Rational> b = ETP * delta;
    Vector2<Rational> u0TE = cylinder.axis.direction * E;
    Rational u0Tdelta = Dot(cylinder.axis.direction, delta);
    Matrix<3, 2, Rational> D;
    Rational rNegOne(-1), rNegHalf(-0.5), rZero(0);
    D(0, 0) = rNegOne;
    D(0, 1) = rNegOne;
    D(1, 0) = u0TE[0];
    D(1, 1) = u0TE[1];
    D(2, 0) = -u0TE[0];
    D(2, 1) = -u0TE[1];
    Vector3<QFType> e;
    e[0] = (Rational)-1.0;
    e[1][0] = -u0Tdelta;
    e[1][1] = rNegHalf * cylinder.height;
    e[2][0] = u0Tdelta;
    e[2][1] = rNegHalf * cylinder.height;

    std::array<QFType, 5> q = { b[0], b[1], -e[0], -e[1], -e[2] };
    std::array<std::array<QFType, 5>, 5> M;
    {
        M[0] = { A(0, 0), A(0, 1), -D(0, 0), -D(1, 0), -D(2, 0) };
        M[1] = { A(1, 0), A(1, 1), -D(0, 1), -D(1, 1), -D(2, 1) };
        M[2] = { D(0, 0), D(0, 1), rZero, rZero, rZero };
        M[3] = { D(1, 0), D(1, 1), rZero, rZero, rZero };
        M[4] = { D(2, 0), D(2, 1), rZero, rZero, rZero };
    }

    std::array<QFType, 5> w, z;
    LCPSolver<QFType, 5> lcp;
    lcp.Solve(q, M, w, z);

    std::array<QFType, 2> result = { z[0], z[1] };
    return result;
}
```

The returned numbers are the triangle parameters for determining the triangle point closest to the cylinder axis and that is between the two planes of the cylinder caps.

Notice that most of the quantities in the code are rational numbers. The first introduction of real quadratic field numbers is in the assignment to the 3-tuple **e** in the inequality constraints of equation (53); that is, e[1] and e[2] are elements of $\mathbb{Q}(\sqrt{d})$. The call to lcp.Solve will involve arithmetic in the real quadratic field.

Executions of the functions of Listings 5, 6 and 7 are shown in Listing 8. In the comments, the rational numbers are listed as odd integers times powers of two, a format described in GTE: Arbitrary Precision Arithmetic.

Listing 8. The listing contains the main function to compare the results of the triangle-cylinder intersection query for various numeric types.

```
int main()
{
    Triangle3<double> triangle;
    triangle.v[0] = { 0.5, -1.0, 0.0 };
    triangle.v[1] = { 3.0, 1.0, 0.0 };
    triangle.v[2] = { 0.5, 2.0, 0.0 };
    Vector3<double> nonUnitDirection{ 1.0, 2.0, 3.0 };
    Cylinder3<double> cylinder;
    cylinder.axis.origin = { 0.0, 0.0, 0.0 };
    cylinder.axis.direction = nonUnitDirection;
    Normalize(cylinder.axis.direction);
    cylinder.radius = 1.0;
    cylinder.height = 2.0;

    // The point on the triangle closest to the cylinder axis is
    // V0 + (0)*(V1 - V0) + (11/30)*(V2 - V0). In the LCP solver, we expect
    // that z = (0,11/30,*).  Note that 11/30 = 0.3666... where the 6 repeats
    // ad infinitum.

    std::array<double, 2> result;
    result = ExecuteDouble(triangle, cylinder);
    // result = (0.00000000000000000, 0.36666666666666670)
    // The second component is an approximation to 11/30.

    std::array<Rational, 2> rresult;
    rresult = ExecuteRational(triangle, cylinder);
    // rresult[0].numerator   = 0
    // rresult[0].denominator = 1
    // rresult[1].numerator   = [0x0000096DB6DB6DB6DB5D4719DCA15C7F, -108]
    // rresult[1].denominator = [0x0000066DB6DB6DB6DB5D4719DCA15C7F, -106]
    double temp;
    temp = rresult[0];  // 0.00000000000000000
    temp = rresult[1];  // 0.36666666666666670
    // The second component is an approximation to 11/30.

    cylinder.axis.direction = nonUnitDirection;
    std::array<QFType, 2> qfresult;
    qfresult = ExecuteQFType(triangle, cylinder);
    // qfresult[0][0].numerator   = 0
    // qfresult[0][0].denominator = 1
    // qfresult[0][1].numerator   = 0
    // qfresult[0][1].denominator = 1
    // qfresult[0] = 0 + 0 * sqrt(14)
    // qfresult[1][0].numerator   = [0x0007C5AB, -20] = 11 * 46305 * 2^{-20}
    // qfresult[1][0].denominator = [0x000A992F, -19] = 30 * 46305 * 2^{-20}
    // qfresult[1][1].numerator   = 0
    // qfresult[1][1].denominator = 1
    // qfresult[1] = 11/30 + 0 * sqrt(14)
    // The second component is exactly 11/30.
    return 0;
}
```

Another slightly more interesting example is shown in Listing 9. The triangle intersects the cylinder and the plane of one of the cylinder caps.

Listing 9. The triangle point inside the cylinder and closest to the cylinder axis is a point on the plane that bounds the top of the cylinder.

```
int main()
{
    Vector3<double> nonUnitDirection{ 1.0, 2.0, 3.0 };
    Vector3<double> perp{ -3.0, 0.0, 1.0 };

    Triangle3<double> triangle;
    triangle.v[0] = 0.125 * perp + 0.5 * nonUnitDirection;
    triangle.v[1] = 0.25 * perp;
    triangle.v[2] = perp;

    Cylinder3<double> cylinder;
    cylinder.axis.origin = { 0.0, 0.0, 0.0 };
    cylinder.axis.direction = nonUnitDirection;
    Normalize(cylinder.axis.direction);
    cylinder.radius = 1.0;
    cylinder.height = 2.0;

    // The point on the triangle inside the planes of the cylinder
    // caps and closest to the cylinder axis is
    // V0 + (1 - (1/7) * sqrt(14))*(V1 - V0) + (0)*(V2 - V0).

    Normalize(cylinder.axis.direction);
    std::array<double, 2> result;
    result = ExecuteDouble(triangle, cylinder);
    // result = (0.46547751617515137, 0.00000000000000000)
    // The first component is an approximation to 1-(1/7)*sqrt(14).

    std::array<Rational, 2> rresult;
    rresult = ExecuteRational(triangle, cylinder);
    // rresult[0].numerator   = [0x001bddd422d07e93, -53]
    // rresult[0].denominator = [0x003bddd422d07e93, -53]
    // rresult[1].numerator   = 0
    // rresult[1].denominator = 1
    double temp;
    temp = rresult[0];  // 0.46547751617515126
    temp = rresult[1];  // 0.00000000000000000

    cylinder.axis.direction = nonUnitDirection;
    std::array<QFType, 2> qfresult;
    qfresult = ExecuteQFType(triangle, cylinder);
    // qfresult[0][0].numerator   = [+0x00000031, -5]
    // qfresult[0][0].denominator = [+0x00000031, -5]
    // qfresult[0][1].numerator   = [-0x00000007, -5]
    // qfresult[0][1].denominator = [+0x00000031, -5]
    // qfresult[0] = 1 - (1/7) * sqrt(14)
    // qfresult[1][0].numerator   = 0
    // qfresult[1][0].denominator = 1
    // qfresult[1][1].numerator   = 0
    // qfresult[1][1].denominator = 1
    // qfresult[1] = 0 + 0 * sqrt(14)
    // 1 - (1/7)*sqrt(14) is approximately 0.46547751617515123063089303824049
    return 0;
}
```

5 Geometric Primitives

The remainder of the document is about formulating distance queries as CQP problems in 2D and in 3D. Each of the primitives involved is parameterized in a manner that is suited for the inequality constraints of the CQP. The 1-dimensional primitives include lines, rays and segments. The 2-dimensional primitives include planes and objects that live in a plane such as triangles, rectangles and convex polygons. The 3-dimensional primitives are convex polyhedra including tetrahedra and boxes.

5.1 Linear Objects

The linear objects are lines, rays and segments. Each object has a single parameter in its representation.

5.1.1 Lines

A line has an origin point $\mathbf{p}$ and a direction vector $\mathbf{u}$ that is not the zero vector. Usually, $\mathbf{u}$ is specified as a unit-length vector. The parameterization is

$$\mathbf{p} + t\mathbf{u}, \quad t \in \mathbb{R} \tag{54}$$

The parameter t is unconstrained, so Section 1.4 is applicable when formulating distance queries between lines and other objects.

5.1.2 Rays

A ray is a subset of a line. It has an origin point $\mathbf{p}$ and a direction vector $\mathbf{u}$ that is not the zero vector. Usually, $\mathbf{u}$ is specified as a unit-length vector. The parameterization is

$$\mathbf{p} + t\mathbf{u}, \quad t \geq 0 \tag{55}$$

5.1.3 Segments

A segment is a subset of a line. The classical parameterization uses endpoints $\mathbf{p}_0 = \mathbf{p}$ and $\mathbf{p}_1 = \mathbf{p} + \mathbf{u}$, where $\mathbf{u}$ is not the zero vector (and generally not unit length). The parameterization is

$$\mathbf{p} + t\mathbf{u} = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1, \quad 0 \leq t \leq 1 \tag{56}$$

Other representations are possible. Using the line representation of equation (54), the segment is specified by an interval $[t_0, t_1]$ for $t_0 < t_1$. Another representation involves choosing a center point $\mathbf{p}$, a unit-length direction $\mathbf{u}$ and a radius $r > 0$, namely, $\mathbf{p} + t\mathbf{u}$ with $|t| \leq r$.

5.2 Planar Objects

The planar objects are planes or convex polygons contained in the plane, including triangles and rectangles.

5.2.1 Planes

Whether living in 2D or 3D, a plane has an origin point $\mathbf{p}$ and two linearly independent direction vectors $\mathbf{u}_0$ and $\mathbf{u}_1$. In 2D, the directions are 2-tuples with $\mathbf{u}_0 \cdot \mathbf{u}_1^\perp \neq 0$. In 3D, the directions are 3-tuples with $\mathbf{u}_0 \times \mathbf{u}_1 \neq \mathbf{0}$. The parameterization is

$$\mathbf{p} + t_0\mathbf{u}_0 + t_1\mathbf{u}_1, \quad t_0 \in \mathbb{R},\ t_1 \in \mathbb{R} \tag{57}$$

Usually the directions $\mathbf{u}_0$ and $\mathbf{u}_1$ are chosen to be unit length and perpendicular. The parameters t_0 and t_1 are unconstrained, so Section 1.4 is applicable when formulating distance queries between planes and other objects in 3D.

5.2.2 Triangles

A triangle is defined by a point $\mathbf{p}$ and two linearly independent vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ for the directions of the edges emanating from $\mathbf{p}$. The triangle vertices are $\mathbf{p}_0 = \mathbf{p}$, $\mathbf{p}_1 = \mathbf{p} + \mathbf{u}_0$ and $\mathbf{p}_2 = \mathbf{p} + \mathbf{u}_1$. The parameterization is

$$\mathbf{p} + t_0\mathbf{u}_0 + t_1\mathbf{u}_1,\ \ t_0 \ge 0,\ t_1 \ge 0,\ t_0 + t_1 \le 1 \tag{58}$$

Generally, the edge directions are not unit length.

5.2.3 Rectangles

A rectangle is defined by a point $\mathbf{p}$ and two perpendicular vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ for the directions of the edges emanating from $\mathbf{p}$. The rectangle vertices are $\mathbf{p}_{00} = \mathbf{p}$, $\mathbf{p}_{10} = \mathbf{p} + \mathbf{u}_0$, $\mathbf{p}_{01} = \mathbf{p} + \mathbf{u}_1$ and $\mathbf{p}_{11} = \mathbf{p} + \mathbf{u}_0 + \mathbf{u}_1$. The parameterization is

$$\mathbf{p} + t_0\mathbf{u}_0 + t_1\mathbf{u}_1,\ \ 0 \le t_0 \le 1,\ 0 \le t_1 \le 1 \tag{59}$$

Sometimes the edge directions are specified by unit-length vectors $\mathbf{u}_0$ and $\mathbf{u}_1$. The corresponding edge lengths are ℓ_0 and ℓ_1. The parameterization is the same as equation (59) but with constraints $0 \le t_i \le \ell_i$. A common representation of a rectangle uses a center point $\mathbf{p}$, two unit-length and perpendicular directions $\mathbf{u}_0$ and $\mathbf{u}_1$, and radii $r_0 > 0$ and $r_1 > 0$. The parameterization is the same as equation (59) but with constraints $|t_i| \le r_i$.

5.2.4 Convex Polygons

A simple parameterization is not possible, although the polygon can be triangulated and then each triangle processed separately using the parameterization of equation (58). However, when formulating CQP problems, it is sufficient to define a convex polygon as the intersection of half-spaces. Let the polygon have n ordered vertices named $\mathbf{p}_i$ for $0 \le i < n$.

In 2D, let the vertices be counterclockwise ordered. The edge directions are $\mathbf{d}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$ with the understanding that the indices are computed modulo n; that is, $\mathbf{p}_n = \mathbf{p}_0$ and $\mathbf{p}_{-1} = \mathbf{p}_{n-1}$. A normal to the edge $\mathbf{d}_i$ that points to the polygon interior is $\mathbf{n}_i = -\mathbf{d}_i^{\perp}$, where $(u, v)^{\perp} = (v, -u)$. The polygon $\mathcal{P}$ is defined by

$$\mathcal{P} = \{\mathbf{y} \in \mathbb{R}^2 : \mathbf{n}_i \cdot (\mathbf{y} - \mathbf{p}_i) \ge 0,\ 0 \le i < n\} \tag{60}$$

In 3D, the vertices are coplanar where the plane has origin $\mathbf{p}_0$ and normal direction $\mathbf{m}$. Let the vertices be counterclockwise ordered to an observer positioned on the side of the plane to which $\mathbf{m}$ points and who is looking in the direction $-\mathbf{m}$ at the polygon in the plane. As in 2D, the edge directions are $\mathbf{d}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$. A normal to the edge that lives in the plane and points to the polygon interior is $\mathbf{n}_i = \mathbf{m} \times \mathbf{d}_i$. The polygon $\mathcal{P}$ is defined by

$$\mathcal{P} = \{\mathbf{y} \in \mathbb{R}^3 : \mathbf{m} \cdot (\mathbf{y} - \mathbf{p}_0) = 0,\ \mathbf{n}_i \cdot (\mathbf{y} - \mathbf{p}_i) \ge 0,\ 0 \le i < n\} \tag{61}$$

Whether 2D or 3D, observe that one or more components of $\mathbf{y} \in \mathcal{P}$ can be negative. This prevents us from choosing $\mathbf{x} = \mathbf{y}$ as the independent variables for the quadratic function of the CQP. We can remedy this by translating the polygon to the first quadrant in 2D or to the first octant in 3D. Let $\boldsymbol{\mu}$ be the vector with the largest components for $\mathbf{y} \ge \boldsymbol{\mu}$. We can choose the quadratic function variables as $\mathbf{x} = \mathbf{y} - \boldsymbol{\mu}$, in which case $\mathbf{x} \ge \mathbf{0}$ and the nonnegativity constraints are satisfied. When computing the distance between the polygon and another object, we must also translate that object by subtracting $\boldsymbol{\mu}$ from its points.

5.3 Volumetric Objects

The volumetric objects are convex polyhedra in space, including tetrahedra and boxes.

5.3.1 Tetrahedra

A tetrahedron is defined by a point $\mathbf{p}$ and three linearly independent vectors $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$ for the directions of the edges emanating from $\mathbf{p}$. The tetrahedron vertices are $\mathbf{p}_0 = \mathbf{p}$, $\mathbf{p}_1 = \mathbf{p} + \mathbf{u}_0$, $\mathbf{p}_2 = \mathbf{p} + \mathbf{u}_1$ and $\mathbf{p} + \mathbf{u}_2$. The convention is that the points are ordered so that $\mathbf{u}_0 \cdot \cdot \mathbf{u}_1 \times \mathbf{u}_2 > 0$. The canonical tetrahedron has vertices $\mathbf{p}_0 = (0,0,0)$, $\mathbf{p}_1 = (1,0,0)$, $\mathbf{p}_2 = (0,1,0)$ and $\mathbf{p}_3 = (0,0,1)$. The parameterization is

$$\mathbf{p} + t_0\mathbf{u}_0 + t_1\mathbf{u}_1 + t_2\mathbf{u}_2, \;\; t_0 \geq 0, \; t_1 \geq 0, \; t_2 \geq 0, \; t_0 + t_1 + t_2 \leq 1 \tag{62}$$

Generally, the edge directions are not unit length.

5.3.2 Boxes

A box is defined by a point $\mathbf{p}$ and three mutually perpendicular vectors $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$ for the directions of the edges emanating from $\mathbf{p}$. The box vertices are $\mathbf{p}_{i_0 i_1 i_2} = \mathbf{p} + i_0\mathbf{u}_0 + i_1\mathbf{u}_1 + i_2\mathbf{u}_2$ for $i_j \in \{0,1\}$ for $j = 0,1,2$. The convention is that $\mathbf{u}_0 \cdot \mathbf{u}_1 \times \mathbf{u}_2 > 0$. The canonical box has vertices $\mathbf{p}_{i_0 i_1 i_2} = (i_0, i_1, i_2)$ for $i_j \in \{0,1\}$ for $j = 0,1,2$. The parameterization is

$$\mathbf{p} + t_0\mathbf{u}_0 + t_1\mathbf{u}_1 + t_2\mathbf{u}_2, \;\; 0 \leq t_0 \leq 1, \; 0 \leq t_1 \leq 1, \; 0 \leq t_2 \leq 1 \tag{63}$$

Sometimes the edge directions are specified by unit-length vectors $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$. The corresponding edge lengths are ℓ_0, ℓ_1 and ℓ_2. The parameterization is the same as equation (63) but with constraints $0 \leq t_i \leq \ell_i$. A common representation of a box uses a center point $\mathbf{p}$, three unit-length and perpendicular directions $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$, and radii $r_0 > 0$, $r_1 > 0$ and $r_2 > 0$. The parameterization is the same as equation (63) but with constraints $|t_i| \leq r_i$.

5.3.3 Convex Polyhedra

A simple parameterization is not possible, although the polyhedron can be tetrahedralized and each tetrahedron processed separately using the parameterization of equation (62). However, when formulating CQP problems, it is sufficient to define a convex polyhedron as the intersection of half-spaces.

Let the polyhedron have m vertices named $\mathbf{p}_i$ for $0 \leq i < m$. Assume that the polyhedron has n faces each with normal vector $\mathbf{n}_i$ for $0 \leq i < n$ that points towards the polyhedron interior. The normals are not necessarily unit length. In the common case that the polyhedron faces are triangles, consider a face $\langle \mathbf{p}_{i_0}, \mathbf{p}_{i_1}, \mathbf{p}_{i_2} \rangle$ whose vertices are counterclockwise ordered when viewed by an observer outside the polyhedron. An inner-pointing normal vector is $\mathbf{n}_{i_0} = (\mathbf{p}_{i_2} - \mathbf{p}_{i_0}) \times (\mathbf{p}_{i_1} - \mathbf{p}_{i_0})$. The polyhedron $\mathcal{P}$ is defined by

$$\mathcal{P} = \{\mathbf{y} \in \mathbb{R}^3 : \mathbf{n}_i \cdot (\mathbf{y} - \mathbf{p}_{j_i}) \geq 0, \; 0 \leq i < n\} \tag{64}$$

where $\mathbf{p}_{j_i}$ is a point on the ith face. Generally, for a face that is a convex polygon, choose any three noncollinear points of the face and compute the normal vector as shown for a triangle.

Observe that one or more components of $\mathbf{y} \in \mathcal{P}$ can be negative. This prevents us from choosing $\mathbf{x} = \mathbf{y}$ as the independent variables for the quadratic function of the CQP. We can remedy this by translating the polyhedron to the first octant in 3D. Let $\boldsymbol{\mu}$ be the vector with the largest components for $\mathbf{y} \geq \boldsymbol{\mu}$. We can choose the quadratic function variables as $\mathbf{x} = \mathbf{y} - \boldsymbol{\mu}$, in which case $\mathbf{x} \geq \mathbf{0}$ and the nonnegativity constraints are satisfied. When computing the distance between the polyhedron and another object, we must also translate that object by subtracting $\boldsymbol{\mu}$ from its points.

5.4 Data Structures for the Primitives

The linear primitives all have an origin $\mathbf{p}$ and a direction $\mathbf{u}$. The planar primitives, not including convex polygons, have an origin $\mathbf{p}$ and two directions $\mathbf{u}_0$ and $\mathbf{u}$. The volumetric primitives, not including convex polyhedra, have an origin $\mathbf{p}$ and three directions $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_3$. The data structures used in the pseudocode for distance queries are shown in Listing 10 and use C++ template notation. In actual GTE code, the Real type can be float or double, although floating-point rounding errors have the potential to cause the LCP solver to generate inaccurate information. The Real type can also represent an arbitrary-precision number system for exact computation as mentioned previously in this document.

Listing 10. The data structures used to represent geometric primitives are listed here.

```
// linear primitives
template <typename Real, int N> struct Line { Point<Real, N> p; Vector<Real, N> u; }
template <typename Real, int N> struct Ray { Point<Real, N> p; Vector<Real, N> u; }
template <typename Real, int N> struct Segment { Point<Real, N> p; Vector<Real, N> u; }

// planar primitives not including convex polygons
template <typename Real, int N> struct Plane { Point<Real, N> p; Vector<Real, N> u0, u1; }
template <typename Real, int N> struct Triangle { Point<Real, N> p; Vector<Real, N> u0, u1; }
template <typename Real, int N> struct Rectangle { Point<Real, N> p; Vector<Real, N> u0, u1; }

// volumetric primitives not including convex polyhedra
template <typename Real, int N> struct Tetrahedra { Point<Real, N> p; Vector<Real, N> u0, u1, u2; }
template <typename Real, int N> struct Box { Point<Real, N> p; Vector<Real, N> u0, u1, u2; }
```

Although each class of primitives (such as linear primitives) has the same form for the structure, they vary based on constraints for the coefficients of the **u**-vectors.

Listing 11 contains data structures for convex polygons with sufficient information to support the distance queries.

Listing 11. The data structures for convex polygons living in $\mathbb{R}^2$ or $\mathbb{R}^3$ are listed here.

```
template <typename Real> struct ConvexPolygon2
{
    std::vector<Point<Real, 2>> points;
    std::vector<Vector<Real, 2>> normals;
    Point<Real, 2> minimum;
}

template <typename Real> struct ConvexPolygon3
{
    std::vector<Point<Real, 3>> points;
    std::vector<Vector<Real, 3>> normals;
    Point<Real, 3> minimum;
    Vector<Real, 3> planeNormal;
    std::array<int, 3> permute, invPermute;
}
```

The points array stores the vertices of the polygon. The vertices are ordered, and it does not matter whether that ordering is clockwise or counterclockwise. The distance query depends only on having normals that are directed to the polygon interior. The normals array has the same number of elements as the points array. The vector normal[i] is perpendicular to the edge with points[i+1] - points[i] and must be directed to the interior of the polygon. The minimum point has components that store the minimum values for the vertices. This member supports translation of the convex polygon to the first quadrant for convex polygons in 2D or to the first octant for convex polygons in 3D.

For convex polygons living in 3-dimensional space, we need to know the plane that contains the polygon. The normal for that plane is planeNormal. The vertex point[0] is chosen to be the plane origin. The vector normal[i] is perpendicular to both the edge direction points[i+1] - points[i] and the plane normal planeNormal; it must be directed to the interior of the polygon. The 3-tuple permute stores a permutation of $\{0, 1, 2\}$, call it $\{i_0, i_1, i_2\}$, so that the plane normal has its maximum absolute component at index i_2; thus, if the plane normal is $\mathbf{m} = (m_0, m_1, m_2)$, then $|m_{i_2}| = \max\{|m_0|, |m_1|, |m_2|\}$. The 3-tuple invPermute is the inverse of the permutation. Table 1 shows the permutations and their inverses.

Table 1. The permutations and their inverse.

permute	$(0,1,2)$	$(0,2,1)$	$(2,0,1)$	$(1,0,2)$	$(1,2,0)$	$(2,1,0)$
invPermute	$(0,1,2)$	$(0,2,1)$	$(1,2,0)$	$(1,0,2)$	$(2,0,1)$	$(2,1,0)$

Convex polyhedra are assumed to have triangle faces. Listing 12 contains a data structure with sufficient information to support the distance queries.

Listing 12. The data structure for convex polyhedra living in $\mathbb{R}^3$ is listed here.

```
template <typename Real> struct ConvexPolyhedron
{
    std::vector<Point<Real, 3>> points;
    std::vector<std::array<int, 3>> triangles;
    std::vector<Vector<Real, 3>> normals;
    Point<Real, 3> minimum;
}
```

The points array stores the vertices of the polygon. The triangles stores triples of indices that are relative to the points array. For example, face i of the polygon has triple triangles[i] and the vertices that form the face are points[triangles[i][0]], points[triangles[i][1]] and points[triangles[i][2]]. The normals array has the same number of elements as the triangles array. The vector normal[i] is perpendicular to the triangle face determined by the triangle[i] triple. The minimum point has components that store the minimum values for the vertices. This member supports translation of the convex polyhedron to the first octant in order to satisfy the nonnegativity constraints.

6 Distance Queries

Each distance query is formulated as a CQP. Alternatively, it is possible to formulate a query in a feature-based manner by decomposing the objects into vertices and edges, computing the distance queries for those features, and then selecting the feature pair that leads to the object-object distance, but this style of query is not discussed in the document.

The CQP formulations use the object definitions presented in Section 5. The input variable of the CQP is $\mathbf{x} = (x_0, \ldots, x_{n-1})$ of the appropriate dimension n. All the queries are formulated as the minimization of the quadratic function f subject to inequality constraints, namely,

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathsf{T}} A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c, \;\; \mathbf{x} \geq \mathbf{0}, \; D\mathbf{x} \geq \mathbf{e} \tag{65}$$

Each subsection has a construction for A, $\mathbf{b}$, c, D and $\mathbf{e}$ with the appropriate selection of parameters $\mathbf{x}$ depending on the types of primitives of the query.

The pseudocode for a distance query is shown in Listing 13. The squared distance is computed so that exact arithmetic can be supported. If you need the distance, compute the square root of the squared distance using whatever support is required for sqrt of your numeric type.

Listing 13. The listing contains pseudocode for a distance query using an LCP solver. The number of components of $\mathbf{x}$ and the number of inequality constraints depends on the pair of objects participating in the query.

```
template <typename Real, int N>
struct QueryResult
{
    QueryResult(int numXComponents) : x(numXComponents) {}

    // The squared distance between object0 and object1.
    Real sqrDistance;

    // The parameters x that minimize f(x), the half-squared-distance
    // function.
    Vector<Real> x;  // x has numXComponents elements

    // A pair of closest points that generate the squared distance
    // between object0 and object1.  The closest point of object0 is
    // closestPoint[0] and the closest point of object1 is closestPoint[1].
    Point<Real, N> closestPoint[2];
};

template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Object0Type<Real> object0, Object1Type<Real> object1)
{
    // Preprocess object data to support nonnegativity constraints.  This
    // step always applies to convex polygons and convex polyhedra.

    // Compute the coefficients of the quadratic function f(x).
    int numXComponents;  // depends on types of the input objects
    Matrix<Real> A(numXComponents, numXComponents);  // ... assign A elements
    Vector<Real> b(numXComponents);  // ... assign b elements
    Real c;  // ... assign c value

    // Compute the inequality constraint coefficients.
    Matrix<Real> D(numXComponents, numConstraints);  // ... assign D elements
    Vector<Real> e(numConstraints);  // ... assign e elements

    // Compute the LCP inputs.
    int lcpSize = numXComponents + numConstraints;
    Vector<Real> q(lcpSize);  // = { b, -e }
    Matrix<Real> M(lcpSize, lcpSize);  // = {{A, -Transpose(D)}, {D, 0}}

    // Solve the LCP and extract the x-portion from z = { x, y }.
    Vector<Real> w(lcpSize), z(lcpSize);
    LCPSolver<Real> LCP(q, M, w, z);
    Vector<Real> x(numXComponents);
    for (int i = 0; i < numXComponents; ++i)
    {
        x[i] = z[i];
    }

    // Report the query results to the caller.  NOTE: In theory, the
    // squared distance is nonnegative.  In practice when using floating-point
    // arithmetic for objects that are very close together, rounding errors
    // can cause sqrDistance to be a small negative number.  It is better
    // in practice to compute sqrDistance as the squared length of the
    // difference of the closest points.
    QueryResult<Real, N> result(numXComponents);
    result.x = x;
    result.sqrDistance = Dot(x, A * x) + 2 * Dot(b, x) + 2 * c;

    // Compute the closest points from x using the parameterizations of
    // the types Object0Type and Object1Type.  Postprocess the closest
    // points to undo, if necessary, any adjustments made during
    // preprocessing.
    result.closestPoint[0];  // ... assign the N-tuple
    result.closestPoint[1];  // ... assign the N-tuple

    return result;
}
```

When using fixed-precision floating-point arithmetic, it is better in practice to compute result.sqrDistance by constructing the closest points first and then computing the squared length of the difference. This avoids the computed value for $\mathbf{x}^{\mathsf{T}}A\mathbf{x} + 2\mathbf{b}^{\mathsf{T}}\mathbf{x} + 2c$ from being slightly negative caused by rounding errors when the objects are nearly touching or slight overlapping (distance nearly zero). This approach allows you to skip the numerical computation of c.

A common subroutine in the queries for any pair of objects is the conversion of A, $\mathbf{b}$, D and $\mathbf{e}$ to the LCP inputs $\mathbf{q}$ and M. Moreover, the LCP solver returns a vector $\mathbf{z}$ whose first several rows stores the output $\mathbf{x}$.

The DoQuery implementation of Listing 13 may then be refactored and converted to that of Listing 14.

Listing 14. The listing contains refactored pseudocode for a distance query using an LCP solver.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Object0Type<Real> object0, Object1Type<Real> object1)
{
    // Preprocess object data to support nonnegativity constraints.  This
    // step always applies to convex polygons and convex polyhedra.

    // Compute the coefficients of the quadratic function f(x).
    int numXComponents;  // depends on types of the input objects
    Matrix<Real> A(numXComponents, numXComponents);  // ... assign A elements
    Vector<Real> b(numXComponents);  // ... assign b elements

    // Compute the inequality constraint coefficients.
    Matrix<Real> D(numXComponents, numConstraints);  // ... assign D elements
    Vector<Real> e(numConstraints);  // ... assign e elements

    // Set up the LCP and solve it for vector x.  The numXComponents and
    // numConstraints values are is accessible to ComputeMinimizer from the
    // inputs.  In queries without inequality constraints, ComputeMinimizer
    // does not have D or e inputs.
    QueryResult<Real, N> result(numXComponents);
    result.x = ComputeMinimizer(A, b, D, e);

    // Compute the closest points from x using the parameterizations of
    // the types Object0Type and Object1Type.  Postprocess the closest
    // points to undo, if necessary, any adjustments made during
    // preprocessing.
    result.closestPoint[0];  // ... assign the N-tuple
    result.closestPoint[1];  // ... assign the N-tuple
    Vector<Real> diff = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(diff, diff);

    return result;
}

// The minimizer function when the CQP has inequality constraints.
template <typename Real>
Vector<Real> ComputeMinimizer(Matrix<Real> A, Vector<Real> b, Matrix<Real> D, Vector<Real> e)
{
    // Compute the LCP inputs.
    int numXComponents = b.size();
    int numConstraints = e.size();

    Vector<Real> q(lcpSize);  // = { b, -e }
    for (int r = 0; r < numXComponents; ++r)
    {
        q[r] = b[r];
    }
    for (int r = 0; r < numConstraints; ++r)
    {
        q[r + numXComponents] = -e[r];
    }

    Matrix<Real> M(lcpSize, lcpSize);  // = {{A, -Transpose(D)}, {D, 0}}
    for (int r = 0; r < numXComponents; ++r)
    {
        for (int c = 0; c < numXComponents; ++c)
        {
            M[r][c] = A[r][c];
        }

        for (int c = 0; c < numConstraints; ++c)
        {
            M[r][c + numXComponents] = -D[c][r];
        }
    }
    for (int r = 0; r < numConstraints; ++r)
    {
        for (int c = 0; c < numXComponents; ++c)
        {
            M[r + numXComponents][c] = D[r][c];
        }

        for (int c = 0; c < numConstraints; ++c)
        {
            M[r + numXComponents][c + numXComponents] = 0;
        }
    }

    // Solve the LCP and extract the x-portion from z = { x, y }.
    int lcpSize = numXComponents + numConstraints;
    Vector<Real> w(lcpSize), z(lcpSize);
    LCPSolver<Real> LCP(q, M, w, z);
    Vector<Real> x(numXComponents);
    for (int i = 0; i < numXComponents; ++i)
    {
        x[i] = z[i];
    }
    return x;
}

// The minimizer function when the CQP has no inequality constraints.  The LCP
// solver still involves the nonnegativity constraints (internally).
template <typename Real>
Vector<Real> ComputeMinimizer(Matrix<Real> A, Vector<Real> b)
{
    // With no inequality constraints, q = b and M = A.  Solve the LCP;
    // the z vector is the solution x.
    Vector<Real> w(b.size()), z(b.size());
    LCPSolver<Real> LCP(q, M, w, z);
    return z;
}
```

Naturally, the implementation for a specific pair of object types can be optimized to avoid filling in A, $\mathbf{b}$, D and $\mathbf{e}$ only to use these to fill in $\mathbf{q}$ and M. The pseudocode in this document does not use such optimizations to ensure that the readability and structure of the pseudocode is understandable at a high level.

6.1 Point to Line

The point is $\mathbf{s}$ and the line is $\mathbf{p}+x_0\mathbf{u}$. The dimension of the CQP is 1, so $\mathbf{x}=(x_0)$. Define $\mathbf{\Delta}=\mathbf{p}-\mathbf{s}$. Half the squared distance between a line point and the point is

$$f(\mathbf{x}) = \frac{1}{2}\left|x_0\mathbf{u}+\mathbf{p}-\mathbf{s}\right|^2 = \frac{1}{2}\left|x_0\mathbf{u}+\mathbf{\Delta}\right|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x}+\mathbf{b}^{\mathsf{T}}\mathbf{x}+c \tag{66}$$

The quadratic coefficients are

$$A = \left[\begin{array}{c} |\mathbf{u}|^2 \end{array}\right],\quad \mathbf{b} = \left[\begin{array}{c} \mathbf{u}\cdot\mathbf{\Delta} \end{array}\right],\quad c = \frac{1}{2}|\mathbf{\Delta}|^2 \tag{67}$$

There are no constrained variables, so the nonnegativity constraint does not exist and D and $\mathbf{e}$ do not exist.

The variable x_0 is unconstrained, so we can eliminate it according to Section 1.4 by solving $df/dx_0=0$. The solution is the parameter of the point that minimizes the distance,

$$x_0 = -\mathbf{u}\cdot\mathbf{\Delta}/|\mathbf{u}|^2 \tag{68}$$

Listing 15 contains pseudocode for the distance query.

Listing 15. The listing contains pseudocode for the point-line distance query. The number of $\mathbf{x}$-components is 1 and the number of inequality constraints is 0.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Point<Real, N> point, Line<Real, N> line)
{
    Vector<Real, N> delta = line.p - point;
    Real A00 = Dot(ray.u, ray.u);
    Real b0 = Dot(ray.u, delta);

    QueryResult<Real, N> result(1);
    result.x[0] = -b0 / A00;
    result.closestPoint[0] = point;
    result.closestPoint[1] = line.p + result.x[0] * line.u;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.2 Point to Ray

The point is $\mathbf{s}$ and the ray is $\mathbf{p} + x_0\mathbf{u}$ where $x_0 \geq 0$. Half the squared distance between a ray point and the point is given by equation (66) and the quadratic coefficients are given by equation (67).

The nonnegativity constraint $x_0 \geq 0$ is summarized by $\mathbf{x} \geq \mathbf{0}$. There are no inequality constraints of the form $D\mathbf{x} \geq \mathbf{e}$, so D and $\mathbf{e}$ do not exist. The LCP coefficients are therefore $\mathbf{q} = \mathbf{b}$ and $M = A$.

Listing 16 contains pseudocode for the distance query.

Listing 16. The listing contains pseudocode for the point-ray distance query. The number of $\mathbf{x}$-components is 1 and the number of inequality constraints is 0.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Point<Real, N> point, Ray<Real, N> ray)
{
    Vector<Real, N> delta = ray.p - point;

    Matrix<Real> A(1, 1);
    Vector<Real> b(1);
    A[0][0] = Dot(ray.u, ray.u);
    b[0] = Dot(ray.u, delta);

    QueryResult<Real, N> result(1);
    result.x = ComputeMinimizer(A, b);
    result.closestPoint[0] = point;
    result.closestPoint[1] = ray.p + result.x[0] * ray.u;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

Use of the general LCP solver is not necessary. You can implement the algorithm manually and inline it for performance.

6.3 Point to Segment

The point is $\mathbf{s}$ and the segment is $\mathbf{p}_0 + x_0\mathbf{u}$ where $x_0 \in [0,1]$. Half the squared distance between a segment point and the point is given by equation (66) and the quadratic coefficients are given by equation (67).

The nonnegativity constraint $x_0 \geq 0$ is summarized by $\mathbf{x} \geq \mathbf{0}$. The inequality constraint $x_0 \leq 1$ is summarized by $\mathbf{Dx} \geq \mathbf{e}$ where

$$D = \begin{bmatrix} -1 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \end{bmatrix} \tag{69}$$

Listing 17 contains pseudocode for the distance query.

Listing 17. The listing contains pseudocode for the point-segment distance query. The number of $\mathbf{x}$-components is 1 and the number of inequality constraints is 1, so the LCP size is 2.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Point<Real, N> point, Segment<Real, N> segment)
{
    Vector<Real, N> delta = segment.p - point;

    Matrix<Real> A(1, 1), D(1, 1);
    Vector<Real> b(1), e(1);
    A[0][0] = Dot(ray.u, ray.u);
    b[0] = Dot(ray.u, delta);
    D[0][0] = -1;
    e[0] = -1;

    QueryResult<Real, N> result(1);
    result.x = ComputeMinimizer(A, b, D, e);
    result.closestPoint[0] = point;
    result.closestPoint[1] = segment.p + result.x[0] * segment.u;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

Use of the general LCP solver is not necessary. You can implement the algorithm manually and inline it for performance.

6.4 Point to Plane

The 2D query has the trivial solution of distance zero because the point is already in the plane, so consider the query for 3D. The point is $\mathbf{s}$ and the plane is $\mathbf{p} + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$, where $\mathbf{u}_0$ and $\mathbf{u}_1$ are not necessarily unit length or perpendicular. The dimension of the CQP is 2, so $\mathbf{x} = (x_0, x_1)$. Define $\boldsymbol{\Delta} = \mathbf{p} - \mathbf{s}$. Half the squared distance between a plane point and the point is

$$f(\mathbf{x}) = \frac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + \mathbf{p} - \mathbf{s}\right|^2 = \frac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + \boldsymbol{\Delta}\right|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \tag{70}$$

The quadratic coefficients are

$$A = \begin{bmatrix} \mathbf{u}_0 \cdot \mathbf{u}_0 & \mathbf{u}_0 \cdot \mathbf{u}_1 \\ \mathbf{u}_1 \cdot \mathbf{u}_0 & \mathbf{u}_1 \cdot \mathbf{u}_1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{u}_0 \cdot \boldsymbol{\Delta} \\ \mathbf{u}_1 \cdot \boldsymbol{\Delta} \end{bmatrix}, \quad c = \frac{1}{2}\left|\boldsymbol{\Delta}\right|^2 \tag{71}$$

There are no constrained variables, so the nonnegativity constraint does not exist and D and $\mathbf{e}$ do not exist.

The variables x_0 and x_1 are unconstrained, so we can eliminate them according to Section 1.4 by solving $\nabla f(x_0, x_1) = (0, 0)$. The solution is the parameter pair that minimizes the distance,

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \frac{1}{|\mathbf{u}_0|^2|\mathbf{u}_1|^2 - (\mathbf{u}_0 \cdot \mathbf{u}_1)^2} \begin{bmatrix} (\mathbf{u}_0 \cdot \mathbf{u}_1)(\mathbf{u}_1 \cdot \boldsymbol{\Delta}) - (\mathbf{u}_1 \cdot \mathbf{u}_1)(\mathbf{u}_0 \cdot \boldsymbol{\Delta}) \\ (\mathbf{u}_0 \cdot \mathbf{u}_1)(\mathbf{u}_0 \cdot \boldsymbol{\Delta}) - (\mathbf{u}_0 \cdot \mathbf{u}_0)(\mathbf{u}_1 \cdot \boldsymbol{\Delta}) \end{bmatrix} \tag{72}$$

Observe (in 3D) that $|\mathbf{u}_0|^2|\mathbf{u}_1|^2 - (\mathbf{u}_0 \cdot \mathbf{u}_1)^2 = |\mathbf{u}_0 \times \mathbf{u}_1|^2$, which is not zero because $\mathbf{u}_0$ and $\mathbf{u}_1$ are linearly independent. Similar cross product identities lead to

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \frac{1}{|\mathbf{u}_0 \times \mathbf{u}_1|^2} \begin{bmatrix} (\mathbf{u}_0 \times \mathbf{u}_1) \cdot (\mathbf{u}_1 \times \boldsymbol{\Delta}) \\ -(\mathbf{u}_0 \times \mathbf{u}_1) \cdot (\mathbf{u}_0 \times \boldsymbol{\Delta}) \end{bmatrix} \tag{73}$$

Listing 18 contains pseudocode for the distance query.

Listing 18. The listing contains pseudocode for the point-plane distance query in 3D. The dimension `N` is specialized to 3 in the listing.

```
template <typename Real>
QueryResult<Real, 3>
DoQuery(Point<Real, 3> point, Plane<Real, 3> plane)
{
    Vector<Real, 3> delta = plane.p - point;
    Vector<Real, 3> u0xu1 = Cross(plane.u0, plane.u1);
    Vector<Real, 3> u0xDelta = Cross(plane.u0, delta);
    Vector<Real, 3> u1xDelta = Cross(plane.u1, delta);
    Real dot0 = Dot(u0xu1, u0xu1);
    Real dot1 = Dot(u0xu1, u0xDelta);
    Real dot2 = Dot(u0xu1, u1xDelta);

    QueryResult<Real, 3> result(2);
    result.x[0] = dot1 / dot0;
    result.x[1] = -dot2 / dot0;
    result.closestPoint[0] = point;
    result.closestPoint[1] = plane.p + result.x[0] * plane.u0 + result.x[1] * plane.u1;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.5 Point to Triangle

The point is $\mathbf{s}$ and the triangle is $\mathbf{p} + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$ where $x_0 \geq 0$, $x_1 \geq 0$ and $x_0 + x_1 \leq 1$. Half the squared distance between a triangle point and the point is given by equation (70). The quadratic coefficients are given by equation (71).

The nonnegativity constraints $x_0 \geq 0$ and $x_1 \geq 0$ are summarized by $\mathbf{x} \geq \mathbf{0}$. The inequality constraint $x_0 + x_1 \leq 1$ is summarized by $D\mathbf{x} \geq \mathbf{e}$ where

$$D = \begin{bmatrix} -1 & -1 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \end{bmatrix} \tag{74}$$

Listing 19 contains pseudocode for the distance query.

Listing 19. The listing contains pseudocode for the point-triangle distance query. The number of $\mathbf{x}$-components is 2 and the number of inequality constraints is 1, so the LCP size is 3.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Point<Real, N> point, Triangle<Real, N> triangle)
{
    Vector<Real, N> delta = triangle.p - point;

    Matrix<Real> A(2, 2), D(1, 2);
    Vector<Real> b(2), e(1);
    A[0][0] = Dot(triangle.u0, triangle.u0);
    A[0][1] = Dot(triangle.u0, triangle.u1);
    A[1][0] = A[0][1];
    A[1][1] = Dot(triangle.u1, triangle.u1);
    b[0] = Dot(triangle.u0, delta);
    b[1] = Dot(triangle.u1, delta);
    D[0][0] = -1;
    D[0][1] = -1;
    e[0] = -1;

    QueryResult<Real, N> result(2);
    result.x = ComputeMinimizer(A, b, D, e);
    result.closestPoint[0] = point;
    result.closestPoint[1] = triangle.p + result.x[0] * triangle.u0 + result.x[1] * triangle.u1;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.6 Point to Rectangle

The point is $\mathbf{s}$ and the rectangle is $\mathbf{p} + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$ where $\mathbf{u}_0 \cdot \mathbf{u}_1 = 0$, $x_0 \in [0, 1]$ and $x_1 \in [0, 1]$. Half the squared distance between a rectangle point and the point is given by equation (70). The quadratic coefficients are given by equation (71).

The nonnegativity constraints $x_0 \geq 0$ and $x_1 \geq 0$ are summarized by $\mathbf{x} \geq \mathbf{0}$. The inequality constraints $x_0 \leq 1$ and $x_1 \leq 1$ are summarized by $D\mathbf{x} \geq \mathbf{e}$ where

$$D = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{75}$$

Listing 20 contains pseudocode for the distance query.

Listing 20. The listing contains pseudocode for the point-rectangle distance query. The number of $\mathbf{x}$-components is 2 and the number of inequality constraints is 2, so the LCP size is 4.

```
template <typename Real, int N>
QueryResult<Real, N>
DoQuery(Point<Real, N> point, Rectangle<Real, N> rectangle)
{
    Vector<Real, N> delta = rectangle.p - point;

    Matrix<Real> A(2, 2), D(2, 2);
    Vector<Real> b(2), e(2);
    A[0][0] = Dot(rectangle.u0, rectangle.u0);
    A[0][1] = 0;
    A[1][0] = 0;
    A[1][1] = Dot(rectangle.u1, rectangle.u1);
    b[0] = Dot(rectangle.u0, delta);
    b[1] = Dot(rectangle.u1, delta);
    D[0][0] = -1;
    D[0][1] = 0;
    D[1][0] = 0;
    D[1][1] = -1;
    e[0] = -1;
    e[1] = -1;

    QueryResult<Real, N> result;
    result.x = ComputeMinimizer(A, b, D, e);
    result.closestPoint[0] = point;
    result.closestPoint[1] = rectangle.p + result.x[0] * rectangle.u0 + result.x[1] * rectangle.u1;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.7 Point to Convex Polygon

6.7.1 Convex Polygons in 2D

In 2D the point is $\mathbf{s}$ and the convex polygon contains points $\mathbf{y}$ defined by equation (60). In order to satisfy nonnegativity constraints, the polygon must be translated to the first quadrant by choosing $\mathbf{x} = \mathbf{y} - \boldsymbol{\mu} \geq \mathbf{0}$ according to the description in Section 5.2.4. The point must be translated accordingly. Define $\boldsymbol{\Delta} = \boldsymbol{\mu} - \mathbf{s}$. The dimension of the CQP is 2, so $\mathbf{x} = (x_0, x_1)$. Half the squared distance between a convex polygon point and the point is

$$f(\mathbf{x}) = \frac{1}{2}|\mathbf{y} - \mathbf{s}|^2 = \frac{1}{2}|\mathbf{x} + \boldsymbol{\Delta}|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \tag{76}$$

The quadratic coefficients are

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \boldsymbol{\Delta}, \quad c = \frac{1}{2}|\boldsymbol{\Delta}|^2 \tag{77}$$

The inequality constraints are $D\mathbf{x} \geq \mathbf{e}$ where D is $\ell \times 2$ and $\mathbf{e}$ is $\ell \times 1$ when the polygon has ℓ edges (and vertices). The matrices are

$$D = \begin{bmatrix} \mathbf{n}_0^{\mathsf{T}} \\ \vdots \\ \mathbf{n}_{\ell-1}^{\mathsf{T}} \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} \mathbf{n}_0^{\mathsf{T}}(\mathbf{p}_0 - \boldsymbol{\mu}) \\ \vdots \\ \mathbf{n}_{\ell-1}^{\mathsf{T}}(\mathbf{p}_{\ell-1} - \boldsymbol{\mu}) \end{bmatrix} \tag{78}$$

The vertices of the polygon have been translated by $\boldsymbol{\mu}$. The CQP is solved for $\mathbf{x}$ from which we can extract $\mathbf{y} = \mathbf{x} + \boldsymbol{\mu}$ for the closest point to $\mathbf{s}$. Listing 21 contains pseudocode for the distance query.

Listing 21. The listing contains pseudocode for the point-convex polygon distance query in 2D. The number of $\mathbf{x}$-components is 2 and the number of inequality constraints is ℓ, so the LCP size is $\ell + 2$.

```
template <typename Real>
QueryResult<Real, 2>
DoQuery(Point<Real, 2> point, ConvexPolygon2<Real> polygon)
{
    Vector<Real, 2> delta = polygon.minimum - point;

    int L = polygon.normals.size();
    Matrix<Real> A(2, 2), D(L, 2);
    Vector<Real> b(2), e(L);
    A[0][0] = 1;
    A[0][1] = 0;
    A[1][0] = A[0][1];
    A[1][1] = 1;
    b[0] = delta[0];
    b[1] = delta[1];
    for (int j = 0; j < L; ++j)
    {
        D[j][0] = polygon.normals[j][0];
        D[j][1] = polygon.normals[j][1];
        e[j] = Dot(polygon.normals[j], polygon.points[j] - polygon.minimum);
    }

    QueryResult<Real, N> result(2);
    result.x = ComputeMinimizer(A, b, D, e);
    result.closestPoint[0] = point;
    result.closestPoint[1] = result.x + polygon.minimum;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.7.2 Convex Polygons in 3D

In 3D the point is $\mathbf{s}$ and the convex polygon contains points $\mathbf{y}$ defined by equation (61). In order to satisfy the nonnegativity constraints, the polygon must be translated by choosing $\mathbf{x} = \mathbf{y} - \boldsymbol{\mu} \geq \mathbf{0}$ according to the desription in Section 5.2.4. The point must be translated accordingly. Define $\boldsymbol{\Delta} = \boldsymbol{\mu} - \mathbf{s}$. Half the squared distance between a convex polygon point and the point is provided by equation (76) except that $\mathbf{x}$ is a 3-tuple.

We have an equality constraint, $\mathbf{m} \cdot (\mathbf{x} - \mathbf{p}_0) = 0$, that defines the plane containing the convex polygon. The point $\mathbf{p}_0$ is the first point in the list of polygon vertices and $\mathbf{m}$ is a plane normal. Let $\mathbf{m} = (m_0, m_1, m_2)$ and $\boldsymbol{\Delta} = (\Delta_0, \Delta_1, \Delta_2)$. Choose a permutation $(i_0, i_1, i_2) \in \{(0,1,2), (2,0,1), (1,2,0)\}$ for which $|m_{i_2}| = \max\{|m_0|, |m_1|, |m_2|\}$. The plane equation can be solved for

$$x_{i_2} = \frac{\mathbf{m} \cdot \mathbf{p}_0 - m_{i_0} x_{i_0} - m_{i_1} x_{i_1}}{m_{i_2}} = \alpha_0 x_{i_0} + \alpha_1 x_{i_1} + \alpha_2 \tag{79}$$

where the last equality defines the scalars α_i.

Substitute x_{i_2} into $f(\mathbf{x})$ of equation (76). Define $\tilde{\mathbf{x}} = (x_{i_0}, x_{i_1})$. The quadratic equation is

$$\tilde{f}(\tilde{\mathbf{x}}) = \frac{1}{2}\tilde{\mathbf{x}}^{\mathsf{T}} \tilde{A} \tilde{\mathbf{x}} + \tilde{\mathbf{b}}^{\mathsf{T}} \tilde{\mathbf{x}} + \tilde{c} \tag{80}$$

where

$$\tilde{A} = \begin{bmatrix} 1 + \alpha_0^2 & \alpha_0 \alpha_1 \\ \alpha_0 \alpha_1 & 1 + \alpha_1^2 \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \Delta_{i_0} + \alpha_0(\Delta_{i_2} + \alpha_2) \\ \Delta_{i_1} + \alpha_1(\Delta_{i_2} + \alpha_2) \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}|\boldsymbol{\Delta}|^2 + \alpha_2 \Delta_{i_2} + \frac{1}{2}\alpha_2^2 \tag{81}$$

Substitute x_{i_2} into the inequality constraints $D\mathbf{x} \geq \mathbf{e}$. If $\mathbf{n}_j = (n_0^{(j)}, n_1^{(j)}, n_2^{(j)})$, then the constraints $\mathbf{n}_j^{\mathsf{T}} \mathbf{x} \geq \mathbf{n}_j^{\mathsf{T}}(\mathbf{p}_j - \boldsymbol{\mu})$ become $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \begin{bmatrix} n_{i_0}^{(0)} + \alpha_0 n_{i_2}^{(0)} & n_{i_1}^{(0)} + \alpha_1 n_{i_2}^{(0)} \\ \vdots & \vdots \\ n_{i_0}^{(n-1)} + \alpha_0 n_{i_2}^{(n-1)} & n_{i_1}^{(\ell-1)} + \alpha_1 n_{i_2}^{(\ell-1)} \end{bmatrix}, \quad \tilde{\mathbf{e}} = \begin{bmatrix} \mathbf{n}_0^{\mathsf{T}}(\mathbf{p}_0 - \boldsymbol{\mu}) - \alpha_2 n_{i_2}^{(0)} \\ \vdots \\ \mathbf{n}_{\ell-1}^{\mathsf{T}}(\mathbf{p}_{\ell-1} - \boldsymbol{\mu}) - \alpha_2 n_{i_2}^{(\ell-1)} \end{bmatrix} \tag{82}$$

The CQP is solved for $\tilde{\mathbf{x}}$ from which we can compute $\mathbf{x}$ and then $\mathbf{y} = \mathbf{x} + \boldsymbol{\mu}$. Listing 22 contains pseudocode for the distance query.

Listing 22. The listing contains pseudocode for the point-convex polygon distance query in 3D. The number of $\tilde{\mathbf{x}}$-components is 2 and the number of inequality constraints is ℓ, so the LCP size is $\ell + 2$.

```
template <typename Real>
QueryResult<Real, 3>
DoQuery(Point<Real, 3> point, ConvexPolygon3<Real> polygon)
{
    Vector<Real, 3> delta = polygon.minimum - point;
    int i0 = polygon.permute[0], i1 = polygon.permute[1], i2 = polygon.permute[2];
    Real alpha0 = -polygon.planeNormal[i0] / polygon.planeNormal[i2];
    Real alpha1 = -polygon.planeNormal[i1] / polygon.planeNormal[i2];
    Real alpha2 = Dot(polygon.planeNormal, polygon.points[0]) / polygon.planeNormal[i2];

    int L = polygon.normals.size();
    Matrix<Real> A(2, 2), D(L, 2);
    Vector<Real> b(2), e(L);
    A[0][0] = 1 + alpha0 * alpha0;
    A[0][1] = alpha0 * alpha1;
    A[1][0] = A[0][1];
    A[1][1] = 1 + alpha1 * alpha1;
    b[0] = delta[i0] + alpha0 * (delta[i2] + alpha2);
    b[1] = delta[i1] + alpha1 * (delta[i2] + alpha2);
    for (int j = 0; j < L; ++j)
    {
        D[j][0] = polygon.normals[j][i0] + alpha0 * polygon.normals[j][i2];
        D[j][1] = polygon.normals[j][i1] + alpha1 * polygon.normals[j][i2];
        Real dot = Dot(polygon.normals[j], polygon.points[j] - polygon.minimum);
        e[j] = dot - alpha2 * polygon.normals[j][i2];
    }

    QueryResult<Real, N> result(2);
    result.x = ComputeMinimizer(A, b, D, e);  // (x[i0],x[i1])
    Vector<Real, 3> x;  // (x[0],x[1],x[2])
    x[0] = result.x[polygon.invPermute[i0]];
    x[1] = result.x[polygon.invPermute[i1]];
    x[2] = result.x[polygon.invPermute[i2]];
    result.closestPoint[0] = point;
    result.closestPoint[1] = x + polygon.minimum;
    delta = result.closestPoint[1] - result.closestPoint[0];
    result.sqrDistance = Dot(delta, delta);
    return result;
}
```

6.8 Point to Tetrahedron

The point is $\mathbf{s}$ and the tetrahedron is $\mathbf{p}+x_0\mathbf{u}_0+x_1\mathbf{u}_1+x_2\mathbf{u}_2$ with $x_0 \geq 0$, $x_1 \geq 0$, $x_2 \geq 0$ and $x_0+x_1+x_2 \leq 1$. The dimension of the CQP is 3, so $\mathbf{x} = (x_0, x_1, x_2)$. Define $\mathbf{\Delta} = \mathbf{p} - \mathbf{s}$. Half the squared distance between a tetrahedron point and the point is

$$f(\mathbf{x}) = \frac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2 + \mathbf{p} - \mathbf{s}\right|^2 = \frac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2 + \mathbf{\Delta}\right|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \tag{83}$$

The quadratic coefficients are

$$A = \begin{bmatrix} \mathbf{u}_0\cdot\mathbf{u}_0 & \mathbf{u}_0\cdot\mathbf{u}_1 & \mathbf{u}_0\cdot\mathbf{u}_2 \\ \mathbf{u}_1\cdot\mathbf{u}_0 & \mathbf{u}_1\cdot\mathbf{u}_1 & \mathbf{u}_1\cdot\mathbf{u}_2 \\ \mathbf{u}_2\cdot\mathbf{u}_0 & \mathbf{u}_2\cdot\mathbf{u}_1 & \mathbf{u}_2\cdot\mathbf{u}_2 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{u}_0\cdot\mathbf{\Delta} \\ \mathbf{u}_1\cdot\mathbf{\Delta} \\ \mathbf{u}_2\cdot\mathbf{\Delta} \end{bmatrix}, \quad c = \frac{1}{2}|\mathbf{\Delta}|^2 \tag{84}$$

The nonnegativity constraints $x_0 \geq 0$, $x_1 \geq 0$ and $x_2 \geq 0$ are summarized by $\mathbf{x} \geq \mathbf{0}$. The inequality constraint $x_0 + x_1 + x_2 \leq 0$ is summarized by $D\mathbf{x} \geq \mathbf{e}$ where

$$D = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \end{bmatrix} \tag{85}$$

6.9 Point to Box

The point is $\mathbf{s}$ and the box is $\mathbf{p} + x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2$ with $x_0 \in [0,1]$, $x_1 \in [0,1]$ and $x_2 \in [0,1]$. Half the squared distance between a box point and the point is given by equation (83). The quadratic coefficients are given by equation (84).

The nonnegativity constraints $x_0 \geq 0$, $x_1 \geq 0$ and $x_2 \geq 0$ are summarized by $\mathbf{x} \geq \mathbf{0}$. The inequality constraints $x_0 \leq 1$, $x_1 \leq 1$ and $x_2 \leq 1$ are summzrized by $D\mathbf{x} \geq \mathbf{e}$ where

$$D = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \tag{86}$$

6.10 Point to Convex Polyhedron

The point is $\mathbf{s}$ and the convex polyhedron contains points $\mathbf{x}$ defined by equation 64. The dimension of the CQP is 3, so $\mathbf{x} = (x_0, x_1, x_2)$. Half the squared distance between a convex polyhedron point and the point is

$$f(\mathbf{x}) = \frac{1}{2}|\mathbf{x} - \mathbf{s}|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}} A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \tag{87}$$

The quadratic coefficients are

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = -\mathbf{s}, \quad c = \frac{1}{2}|\mathbf{s}|^2 \tag{88}$$

The inequality constraints are $D\mathbf{x} \geq \mathbf{e}$ where D is $n \times 3$ and $\mathbf{e}$ is $n \times 1$ when the polyhedron has ℓ faces. The matrices are

$$D = \begin{bmatrix} \mathbf{n}_0^{\mathsf{T}} \\ \vdots \\ \mathbf{n}_{\ell-1}^{\mathsf{T}} \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} \mathbf{n}_0^{\mathsf{T}}\mathbf{p}_0 \\ \vdots \\ \mathbf{n}_{\ell-1}^{\mathsf{T}}\mathbf{p}_{n-1} \end{bmatrix} \tag{89}$$

We have a technical problem to resolve. In the definition for a convex polyhedron, it is not necessary that $\mathbf{x} \geq \mathbf{0}$; that is, not all the vertices are necessarily in the first octant. To formulate the CQP, we need to translate the vertices to the first octant. Do so by computing an axis-aligned bounding box for the polyhedron, say, $[\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}]$. Subtract the minimum point from the polygon vertices and from the query point, $\mathbf{y} = \mathbf{x} - \boldsymbol{\mu}_{\min}$ and $\mathbf{r} = \mathbf{s} - \boldsymbol{\mu}_{\min}$; then

$$f(\mathbf{x}) = \frac{1}{2}|\mathbf{x} - \mathbf{s}|^2 = \frac{1}{2}|\mathbf{y} - \mathbf{r}|^2 = g(\mathbf{y}) \tag{90}$$

where the last equality defines the quadratic function g. The nonnegativity constraints $\mathbf{y} \geq \mathbf{0}$ are now feasible because the translated vertices are in the first quadrant. The inequality constraints $D\mathbf{x} \geq \mathbf{e}$ become $D\mathbf{y} \geq \mathbf{e} - D\boldsymbol{\mu}_{\min} = \mathbf{h}$, where the last equality defines the vector $\mathbf{h}$. The quadratic coefficients for $g(\mathbf{y})$ are

$$A = I, \quad \mathbf{b} = -\mathbf{r}, \quad c = \frac{1}{2}|\mathbf{r}|^2 \tag{91}$$

where I is the 3×3 identity matrix. The nonnegativity constraints are $\mathbf{y} \geq \mathbf{0}$ and the inequality constraints are $D\mathbf{y} \geq \mathbf{h}$.

The CQP is solved for $\mathbf{y}$ from which we can extract $\mathbf{x} = \mathbf{y} + \boldsymbol{\mu}_{\min}$ for the closest point to $\mathbf{s}$. The distance can be computed either from $f(\mathbf{x})$ or $g(\mathbf{y})$.

6.11 Line to Line

The lines are $\mathbf{p}_i + x_i\mathbf{u}_i$ for $i = 0, 1$. The dimension of the CQP is 2, so $\mathbf{x} = (x_0, x_1)$. Define $\boldsymbol{\Delta} = \mathbf{p}_0 - \mathbf{p}_1$. Half the squared distance between a point on one line and a point on the other line is

$$f(\mathbf{x}) = \frac{1}{2}\left|(\mathbf{p}_0 + x_0\mathbf{u}_0) - (\mathbf{p}_1 + x_1\mathbf{u}_1)\right|^2 = \frac{1}{2}\left|x_0\mathbf{u}_0 - x_1\mathbf{u}_1 + \boldsymbol{\Delta}\right|^2 = \frac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \tag{92}$$

The quadratic coefficients are

$$A = \begin{bmatrix} \mathbf{u}_0 \cdot \mathbf{u}_0 & -\mathbf{u}_0 \cdot \mathbf{u}_1 \\ -\mathbf{u}_1 \cdot \mathbf{u}_0 & \mathbf{u}_1 \cdot \mathbf{u}_1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{u}_0 \cdot \boldsymbol{\Delta} \\ -\mathbf{u}_1 \cdot \boldsymbol{\Delta} \end{bmatrix}, \quad c = \frac{1}{2}|\boldsymbol{\Delta}|^2 \tag{93}$$

There are no constrained variables, so the nonnegativity constraint does not exist and D and $\mathbf{e}$ do not exist.

Both x_0 and x_1 are unconstrained, so we can eliminate them according to Section 1.4. There are no constrained variables, so $(0, 0) = \nabla f(x_0, x_1) = A\mathbf{x} + \mathbf{b}$ provides the parameters of the points that minimize the distance,

$$\begin{bmatrix} \mathbf{u}_0 \cdot \mathbf{u}_0 & -\mathbf{u}_0 \cdot \mathbf{u}_1 \\ -\mathbf{u}_1 \cdot \mathbf{u}_0 & \mathbf{u}_1 \cdot \mathbf{u}_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} -\mathbf{u}_0 \cdot \boldsymbol{\Delta} \\ \mathbf{u}_1 \cdot \boldsymbol{\Delta} \end{bmatrix} \tag{94}$$

The linear system is invertible if and only if $\mathbf{u}_0$ and $\mathbf{u}_1$ are not parallel. If they are parallel, there are infinitely many pairs of closest points—any such pair may be used to compute the line-line distance. The simplest pair to choose is the line origin $\mathbf{p}_1$ and its projection onto the other line. Let $\mathbf{p}_1 = \mathbf{p}_0 + y_0\mathbf{u}_0 + \mathbf{u}_0^{\perp}$ where $\mathbf{u}_0^{\perp}$ is a vector perpendicular to $\mathbf{u}_0$. The closest point is $\mathbf{p}_1' = \mathbf{u}_0 + y_0\mathbf{u}_0$, where $y_0 = -\boldsymbol{\Delta} \cdot \mathbf{u}_0/|\mathbf{u}_0|^2$. The distance is $|\mathbf{p}_1' - \mathbf{p}_1|$.

6.12 Line to Ray

The ray is $\mathbf{p}_0 + x_0\mathbf{u}_0$ for $x_0 \geq 0$ and the line is $\mathbf{p}_1 + x_1\mathbf{u}_1$. The dimension of the CQP is 2, so $\mathbf{x} = (x_0, x_1)$. Half the squared distance between a ray point and a line point is given by equation (92). The variable x_1 is unconstrained, so we can eliminate it according to Section 1.4. Solve $\partial f/\partial x_1 = 0$ to obtain

$$x_1 = \frac{x_0\mathbf{u}_0 \cdot \mathbf{u}_1 + \mathbf{u}_0 \cdot \boldsymbol{\Delta}}{|\mathbf{u}_1|^2} = \alpha_0 x_0 + \alpha_1 \tag{95}$$

where the last equality defines the scalars α_i. Define $\tilde{\mathbf{x}} = (x_0)$, $g(\tilde{\mathbf{x}}) = f(\mathbf{x})$ and substitute the x_1-equation into equation (92) to obtain

$$\begin{aligned} g(\tilde{\mathbf{x}}) &= \tfrac{1}{2}\left|x_0(\mathbf{u}_0 - \alpha_0\mathbf{u}_1) + (\boldsymbol{\Delta} - \alpha_1\mathbf{u}_1)\right|^2 \\ &= \tfrac{1}{2}\left|x_0\tilde{\mathbf{u}}_0 + \tilde{\boldsymbol{\Delta}}\right|^2 \\ &= \tfrac{1}{2}\tilde{\mathbf{x}}^{\mathsf{T}}\tilde{A}\tilde{\mathbf{x}} + \tilde{\mathbf{b}}^{\mathsf{T}}\tilde{\mathbf{x}} + \tilde{c} \end{aligned} \tag{96}$$

where the second equality defines $\tilde{\mathbf{u}}_0$ and $\tilde{\mathbf{\Delta}}$ and the third equality defines the quadratic coefficients,

$$\tilde{A} = \left[\begin{array}{c} \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_0 \end{array} \right], \quad \tilde{\mathbf{b}} = \left[\begin{array}{c} \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{\Delta}} \end{array} \right], \quad \tilde{c} = \frac{1}{2}|\tilde{\mathbf{\Delta}}|^2 \tag{97}$$

Observe that all 3 quantities are scalars.

The nonnegativity constraint $x_0 \geq 0$ is summarized by $\tilde{\mathbf{x}} \geq \mathbf{0}$. There are no other inequality constraints.

6.13 Line to Segment

The segment is $\mathbf{p}_0 + x_0\mathbf{u}_0$ for $x_1 \in [0, 1]$ and the line is $\mathbf{p}_1 + x_1\mathbf{u}_1$. The conversion to a CQP with elimination of x_1 is identical to that of the line-ray distance query except that we now have an inequality constraint $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \left[\begin{array}{c} -1 \end{array} \right], \quad \tilde{\mathbf{e}} = \left[\begin{array}{c} -1 \end{array} \right] \tag{98}$$

6.14 Line to Plane

The 2D query has the trivial solution of distance zero because the line is already in the plane, so consider the query for 3D. The plane is $\mathbf{p}_0 + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$, where $\mathbf{u}_0$ and $\mathbf{u}_1$ are linearly independent but not necessarily unit length or perpendiculat. The line is $\mathbf{p}_1 + x_2\mathbf{u}_2$. The dimension of the CQP is 3, so $\mathbf{x} = (x_0, x_1, x_2)$. Define $\mathbf{\Delta} = \mathbf{p}_0 - \mathbf{p}_1$. Half the squared distance between a plane point and a line point is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2}\left|(x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + \mathbf{p}_0) - (x_2\mathbf{u}_2 + \mathbf{p}_1)\right|^2 \\ &= \tfrac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 - x_2\mathbf{u}_2 + \mathbf{\Delta}\right|^2 \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \end{aligned} \tag{99}$$

The quadratic coefficients are

$$A = \left[\begin{array}{ccc} \mathbf{u}_0 \cdot \mathbf{u}_0 & \mathbf{u}_0 \cdot \mathbf{u}_1 & -\mathbf{u}_0 \cdot \mathbf{u}_2 \\ \mathbf{u}_1 \cdot \mathbf{u}_0 & \mathbf{u}_1 \cdot \mathbf{u}_1 & -\mathbf{u}_1 \cdot \mathbf{u}_2 \\ -\mathbf{u}_2 \cdot \mathbf{u}_0 & -\mathbf{u}_2 \cdot \mathbf{u}_1 & \mathbf{u}_2 \cdot \mathbf{u}_2 \end{array} \right], \quad \mathbf{b} = \left[\begin{array}{c} \mathbf{u}_0 \cdot \mathbf{\Delta} \\ \mathbf{u}_1 \cdot \mathbf{\Delta} \\ -\mathbf{u}_2 \cdot \mathbf{\Delta} \end{array} \right], \quad c = \frac{1}{2}|\mathbf{\Delta}|^2 \tag{100}$$

All three variables are unconstrained, so we eliminate all variables using the method in Section 1.4. We need to solve $\mathbf{0} = \nabla f(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$. By assumption, $\mathbf{u}_0$ and $\mathbf{u}_1$ are linearly independent. The matrix A is invertible when $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$ are linearly independent. This happens geometrically when the line is not parallel to the plane. The solution $\mathbf{x} = -A^{-1}\mathbf{b}$ is the point of intersection of the line with the plane, in which case the distance is 0. If $J = [\mathbf{u}_0\ \mathbf{u}_1\ -\mathbf{u}_2]$, then $A = J^{\mathsf{T}}J$, $\mathbf{b} = J^{\mathsf{T}}\mathbf{\Delta}$ and $f(\mathbf{x}) = |J\mathbf{x} + \mathbf{\Delta}|^2/2$. We can instead solve $J\mathbf{x} + \mathbf{\Delta} = \mathbf{0}$ for $\mathbf{x} = -J^{-1}\mathbf{\Delta}$.

If A is not invertible, then $\mathbf{u}_2$ is a linear combination of $\mathbf{u}_0$ and $\mathbf{u}_1$. The line is parallel to—or coincident with—the plane. Infinitely many pairs of points achieve the minimum. The solution space to $A\mathbf{x} = -\mathbf{b}$ is 1-dimensional, so any solution in this space generates a pair that achieves the minimum. The simplest pair to choose is the line origin $\mathbf{p}_1$ and its perpendicular projection onto the plane. Let $\mathbf{p}_1 = \mathbf{p}_0 + y_0\mathbf{u}_0 + y_1\mathbf{u}_1 + y_2\mathbf{u}_0 \times \mathbf{u}_1$. Dotting the equation with $\mathbf{u}_0 \times \mathbf{u}_1$ leads to

$$y_2 = \frac{-\boldsymbol{\Delta} \cdot \mathbf{u}_0 \times \mathbf{u}_1}{|\mathbf{u}_0 \times \mathbf{u}_1|^2} \tag{101}$$

The projection onto the plane is $\mathbf{p}_1' = \mathbf{p}_1 - y_2\mathbf{u}_0 \times \mathbf{u}_1$, which is the closest plane point to $\mathbf{p}_1$. The distance is $|\mathbf{p}_1' - \mathbf{p}_1|$.

6.15 Line to Triangle

The triangle is $\mathbf{p}_0 + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$ for $x_0 \geq 0$, $x_1 \geq 0$ and $x_0 + x_1 \leq 1$ and the line is $\mathbf{p}_1 + x_2\mathbf{u}_2$. The dimension of the CQP is 3, so $\mathbf{x} = (x_0, x_1, x_2)$. Define $\boldsymbol{\Delta} = \mathbf{p}_0 - \mathbf{p}_1$. Half the squared distance between a triangle point and a line point is provided by equation (99). The quadratic coefficients are provided by equation (100). The variable x_2 is unconstrained, so we can eliminate it according to Section 1.4. Solve $\partial f / \partial x_2 = 0$ to obtain

$$x_2 = \frac{x_0\mathbf{u}_0 \cdot \mathbf{u}_2 + x_1\mathbf{u}_1 \cdot \mathbf{u}_2 + \boldsymbol{\Delta} \cdot \mathbf{u}_2}{|\mathbf{u}_2|^2} = x_0\alpha_0 + x_1\alpha_1 + \alpha_2 \tag{102}$$

where the last equality defines the α_i. Define $\tilde{\mathbf{x}} = (x_0, x_1)$, $g(\tilde{\mathbf{x}}) = f(\mathbf{x})$ and substitute the x_2-equation into equation (100) to obtain

$$\begin{aligned} g(\tilde{\mathbf{x}}) &= \tfrac{1}{2}\left|x_0(\mathbf{u}_0 - \alpha_0\mathbf{u}_2) + x_1(\mathbf{u}_1 - \alpha_1\mathbf{u}_2) + (\boldsymbol{\Delta} - \alpha_2\mathbf{u}_2)\right|^2 \\ &= \tfrac{1}{2}\left|x_0\tilde{\mathbf{u}}_0 + x_1\tilde{\mathbf{u}}_1 + \tilde{\boldsymbol{\Delta}}\right|^2 \\ &= \tfrac{1}{2}\tilde{\mathbf{x}}^{\mathsf{T}}\tilde{A}\tilde{\mathbf{x}} + \tilde{\mathbf{b}}^{\mathsf{T}}\tilde{\mathbf{x}} + \tilde{c} \end{aligned} \tag{103}$$

where the second equality defines $\tilde{\mathbf{u}}_0$, $\tilde{\mathbf{u}}_1$ and $\tilde{\boldsymbol{\Delta}}$ and the third equality defines the quadratic coefficients

$$\tilde{A} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_1 \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_1 \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\boldsymbol{\Delta}} \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\boldsymbol{\Delta}} \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}|\tilde{\boldsymbol{\Delta}}|^2 \tag{104}$$

The nonnegativity constraints $x_0 \geq 0$ and $x_1 \geq 0$ are summarized by $\tilde{\mathbf{x}} \geq \mathbf{0}$. The inequality constraint $x_0 + x_1 \leq 1$ is summarized by $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \begin{bmatrix} -1 & -1 \end{bmatrix}, \quad \tilde{\mathbf{e}} = \begin{bmatrix} -1 \end{bmatrix} \tag{105}$$

6.16 Line to Rectangle

The rectangle is $\mathbf{p}_0 + x_0\mathbf{u}_0 + x_1\mathbf{u}_1$ where $\mathbf{u}_0 \cdot \mathbf{u}_1 = 0$, $x_0 \in [0, 1]$ and $x_1 \in [0, 1]$. The line is $\mathbf{p}_1 + x_2\mathbf{u}_2$. The conversion to a CQP with elimination of x_2 is identical to that of the line-triangle distance query except that the inequality constraints are $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \tilde{\mathbf{e}} = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{106}$$

6.17 Line to Convex Polygon

6.17.1 Convex Polygons in 2D

In 2D the convex polygon contains points (x_0, x_1) defined by equation (60). Define the basis vectors $\mathbf{u}_0 = (1, 0)$ and $\mathbf{u}_1 = (0, 1)$. The line is $\mathbf{s} + x_2\mathbf{u}_2$. The dimension of the CQP is 3, so $\mathbf{x} = (x_0, x_1, x_2)$. Define $\boldsymbol{\Delta} = -\mathbf{s}$. Half the squared distance between a convex polygon point and a line point is

$$\begin{array}{rcl} f(\mathbf{x}) & = & \frac{1}{2}\left|(x_0\mathbf{u}_0 + x_1\mathbf{u}_1) - (x_2\mathbf{u}_2 + \mathbf{s})\right|^2 \\ & = & \frac{1}{2}\left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 - x_2\mathbf{u}_2 + \boldsymbol{\Delta}\right|^2 \end{array} \tag{107}$$

The variable x_2 is unconstrained, so we can eliminate it according to Section 1.4. Solve $\partial f/\partial x_2 = 0$ to obtain

$$x_2 = \frac{x_0\mathbf{u}_0 \cdot \mathbf{u}_2 + x_1\mathbf{u}_1 \cdot \mathbf{u}_2 + \boldsymbol{\Delta} \cdot \mathbf{u}_2}{|\mathbf{u}_2|^2} = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 \tag{108}$$

where the last equality defines the scalars α_i. Define $\tilde{\mathbf{x}} = (x_0, x_1)$, $g(\tilde{\mathbf{x}}) = f(\mathbf{x})$ and substitute the x_2-equation into equation (107) to obtain

$$\begin{array}{rcl} g(\tilde{\mathbf{x}}) & = & \frac{1}{2}\left|x_0(\mathbf{u}_0 - \alpha_0\mathbf{u}_2) + x_1(\mathbf{u}_1 - \alpha_1\mathbf{u}_2) + (\boldsymbol{\Delta} - \alpha_2\mathbf{u}_2)\right|^2 \\ & = & \frac{1}{2}\left|x_0\tilde{\mathbf{u}}_0 + x_1\tilde{\mathbf{u}}_1 + \tilde{\boldsymbol{\Delta}}\right|^2 \\ & = & \frac{1}{2}\tilde{\mathbf{x}}^{\mathsf{T}}\tilde{A}\tilde{\mathbf{x}} + \tilde{\mathbf{b}}^{\mathsf{T}}\tilde{\mathbf{x}} + \tilde{c} \end{array} \tag{109}$$

where the second equality defines $\tilde{\mathbf{u}}_0$, $\tilde{\mathbf{u}}_1$ and $\tilde{\boldsymbol{\Delta}}$, and the third equality defines the quadratic coefficients

$$\tilde{A} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_1 \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_1 \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\boldsymbol{\Delta}} \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\boldsymbol{\Delta}} \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}|\tilde{\boldsymbol{\Delta}}|^2 \tag{110}$$

The inequality constraints are those of equation (78) and involve only the polygon components $\tilde{\mathbf{x}} = (x_0, x_1)$, so we will write the constraints as $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$.

As in Section 6.7.1, to obtain the nonnegativity constraints $\tilde{\mathbf{x}} \geq \mathbf{0}$, we must translate the convex polygon into the first quadrant. Compute the axis-aligned bounding rectangle $[\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}]$ for the polygon and subtract the minimum point to force the translated polygon into the first quadrant: $\tilde{\mathbf{y}} = \tilde{\mathbf{x}} - \boldsymbol{\mu}_{\min}$. The quadratic function is $g(\tilde{\mathbf{x}}) = h(\tilde{\mathbf{y}}) = \tilde{\mathbf{y}}^{\mathsf{T}}\hat{A}\tilde{\mathbf{y}}/2 + \hat{\mathbf{b}}^{\mathsf{T}}\tilde{\mathbf{y}} + \hat{c}$ where

$$\hat{A} = \tilde{A}, \;\; \hat{\mathbf{b}} = \tilde{A}\boldsymbol{\mu}_{\min} + \tilde{\mathbf{b}}, \;\; \hat{c} = \frac{1}{2}\boldsymbol{\mu}_{\min}^{\mathsf{T}}\tilde{A}\boldsymbol{\mu}_{\min} + \tilde{\mathbf{b}}^{\mathsf{T}}\boldsymbol{\mu}_{\min} + \tilde{c} \tag{111}$$

The nonnegativity constraints are $\tilde{\mathbf{y}} \geq \mathbf{0}$ and the inequality constraints are $\tilde{D}\tilde{\mathbf{y}} \geq \tilde{\mathbf{e}} - \tilde{D}\boldsymbol{\mu}_{\min}$.

The CQP is solved for $\tilde{\mathbf{y}}$ after which $\tilde{\mathbf{x}} = \tilde{\mathbf{y}} + \boldsymbol{\mu}_{\min}$. The distance can be computed from $f(\mathbf{x})$, $g(\tilde{\mathbf{x}})$ or $h(\tilde{\mathbf{y}})$.

6.17.2 Convex Polygons in 3D

In 3D the convex polygon contains points (x_0, x_1, x_2) defined by equation (61). Define the basis vectors $\mathbf{u}_0 = (1, 0, 0)$, $\mathbf{u}_1 = (0, 1, 0)$ and $\mathbf{u}_2 = (0, 0, 1)$. The line is $\mathbf{s} + x_3\mathbf{u}_3$. The dimension of the CQP is 4, so

$\mathbf{x} = (x_0, x_1, x_2, x_3)$. Define $\boldsymbol{\Delta} = -\mathbf{s}$. Half the squared distance between a convex polygon point and a line point is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2} \left|(x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2) - (x_3\mathbf{u}_3 + \mathbf{s})\right|^2 \\ &= \tfrac{1}{2} \left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2 - x_3\mathbf{u}_3 + \boldsymbol{\Delta}\right|^2 \end{aligned} \tag{112}$$

The variable x_3 is unconstrained, so we can eliminate it according to Section 1.4. Solve $\partial f/\partial x_3 = 0$ to obtain

$$x_3 = \frac{x_0\mathbf{u}_0 \cdot \mathbf{u}_3 + x_1\mathbf{u}_1 \cdot \mathbf{u}_3 + x_2\mathbf{u}_2 \cdot \mathbf{u}_3 + \boldsymbol{\Delta} \cdot \mathbf{u}_3}{|\mathbf{u}_3|^2} = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 \tag{113}$$

where the last equality defines the scalars α_i. Define $\tilde{\mathbf{x}} = (x_0, x_1, x_2)$, $g(\tilde{\mathbf{x}}) = f(\mathbf{x})$ and substitute the x_3-equation into equation (112) to obtain

$$\begin{aligned} g(\tilde{\mathbf{x}}) &= \tfrac{1}{2} \left|x_0(\mathbf{u}_0 - \alpha_0\mathbf{u}_3) + x_1(\mathbf{u}_1 - \alpha_1\mathbf{u}_3) + x_2(\mathbf{u}_2 - \alpha_2\mathbf{u}_3) + (\boldsymbol{\Delta} - \alpha_3\mathbf{u}_3)\right|^2 \\ &= \tfrac{1}{2} \left|x_0\tilde{\mathbf{u}}_0 + x_1\tilde{\mathbf{u}}_1 + x_2\tilde{\mathbf{u}}_2 + \tilde{\boldsymbol{\Delta}}\right|^2 \\ &= \tfrac{1}{2}\tilde{\mathbf{x}}^\mathsf{T}\tilde{A}\tilde{\mathbf{x}} + \tilde{\mathbf{b}}^\mathsf{T}\tilde{\mathbf{x}} + \tilde{c} \end{aligned} \tag{114}$$

where the second equality defines $\tilde{\mathbf{u}}_0$, $\tilde{\mathbf{u}}_1$, $\tilde{\mathbf{u}}_2$ and $\tilde{\boldsymbol{\Delta}}$, and the third equality defines the quadratic coefficients

$$\tilde{A} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_1 & \tilde{\mathbf{u}}_0 \cdot \tilde{\mathbf{u}}_2 \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_1 & \tilde{\mathbf{u}}_1 \cdot \tilde{\mathbf{u}}_2 \\ \tilde{\mathbf{u}}_2 \cdot \tilde{\mathbf{u}}_0 & \tilde{\mathbf{u}}_2 \cdot \tilde{\mathbf{u}}_1 & \tilde{\mathbf{u}}_2 \cdot \tilde{\mathbf{u}}_2 \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \tilde{\mathbf{u}}_0 \cdot \tilde{\boldsymbol{\Delta}} \\ \tilde{\mathbf{u}}_1 \cdot \tilde{\boldsymbol{\Delta}} \\ \tilde{\mathbf{u}}_2 \cdot \tilde{\boldsymbol{\Delta}} \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}|\tilde{\boldsymbol{\Delta}}|^2 \tag{115}$$

As in Section 6.7.2, to obtain the nonnegativity constraints $\tilde{\mathbf{x}} \ge \mathbf{0}$, we must translate the convex polygon into the first octant. Compute the axis-aligned bounding box $[\boldsymbol{\mu}_{\min}, \boldsymbol{\mu}_{\max}]$ for the polygon and subtract the minimum point to force the translated polygon into the first octant: $\tilde{\mathbf{y}} = \tilde{\mathbf{x}} - \boldsymbol{\mu}_{\min}$. The quadratic function is $g(\tilde{\mathbf{x}}) = h(\tilde{\mathbf{y}}) = \tilde{\mathbf{y}}^\mathsf{T}\hat{A}\tilde{\mathbf{y}}/2 + \hat{\mathbf{b}}^\mathsf{T}\tilde{\mathbf{y}} + \hat{c}$ where

$$\hat{A} = \tilde{A}, \quad \hat{\mathbf{b}} = \tilde{A}\boldsymbol{\mu}_{\min} + \tilde{\mathbf{b}}, \quad \hat{c} = \frac{1}{2}\boldsymbol{\mu}_{\min}^\mathsf{T}\tilde{A}\boldsymbol{\mu}_{\min} + \tilde{\mathbf{b}}^\mathsf{T}\boldsymbol{\mu}_{\min} + \tilde{c} \tag{116}$$

The nonnegativity constraints are $\tilde{\mathbf{y}} \ge \mathbf{0}$ and the inequality constraints are $\tilde{D}\tilde{\mathbf{y}} \ge \tilde{\mathbf{e}} - \tilde{D}\boldsymbol{\mu}_{\min}$.

The CQP is solved for $\tilde{\mathbf{y}}$ after which $\tilde{\mathbf{x}} = \tilde{\mathbf{y}} + \boldsymbol{\mu}_{\min}$. The distance can be computed from $f(\mathbf{x})$, $g(\tilde{\mathbf{x}})$ or $h(\tilde{\mathbf{y}})$.

6.18 Line to Tetrahedron

The tetrahedron is $\mathbf{p}_0 + x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2$ with $x_0 \ge 0$, $x_1 \ge 0$, $x_2 \ge 0$ and $x_0 + x_1 + x_2 \le 1$. The line is $\mathbf{p}_1 + x_3\mathbf{u}_3$. The dimension of the CQP is s, so $\mathbf{x} = (x_0, x_1, x_2, x_3)$. Define $\boldsymbol{\Delta} = \mathbf{p}_0 - \mathbf{p}_1$. Half the squared distance between a tetrahedron point and a line point is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2} \left|(x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2 + \mathbf{p}_0) - (x_3\mathbf{u}_3 + \mathbf{p}_1)\right|^2 \\ &= \tfrac{1}{2} \left|x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2 - x_3\mathbf{u}_3 + \boldsymbol{\Delta}\right|^2 \end{aligned} \tag{117}$$

The variable x_3 is unconstrained, so we can eliminate it according to Section 1.4. Solve $\partial f/\partial x_3 = 0$ to obtain

$$x_3 = \frac{x_0\mathbf{u}_0 \cdot \mathbf{u}_3 + x_1\mathbf{u}_1 \cdot \mathbf{u}_3 + x_2\mathbf{u}_2 \cdot \mathbf{u}_3 + \boldsymbol{\Delta} \cdot \mathbf{u}_3}{|\mathbf{u}_3|^2} = x_0\alpha_0 + x_1\alpha_1 + x_2\alpha_2 + x_3 \tag{118}$$

where the last equality defines the scalars α_i. Define $\tilde{\mathbf{x}} = (x_0, x_1, x_2)$.

We can then write

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \alpha_0 & \alpha_1 & \alpha_2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \alpha_3 \end{bmatrix} = J\tilde{\mathbf{x}} + \mathbf{t} \tag{119}$$

where the last equality defines the 4×3 matrix J, the 3×1 vector $\tilde{\mathbf{x}}$ and the 4×1 vector $\mathbf{t}$. Substitute this into the quadratic equation to obtain $g(x_0, x_1, x_2) = f(x_0, x_1, x_2, x_3)$. In CQP notation, $g(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}}^\mathsf{T}\tilde{A}\tilde{\mathbf{x}} + \tilde{\mathbf{b}}^\mathsf{T}\tilde{\mathbf{x}} + \tilde{c}$ where

$$\tilde{A} = J^\mathsf{T} A J, \quad \tilde{\mathbf{b}} = J^\mathsf{T}(A\mathbf{t} + \mathbf{b}), \quad \tilde{c} = \frac{1}{2}\mathbf{t}^\mathsf{T} A\mathbf{t} + \mathbf{b}^\mathsf{T}\mathbf{t} + c \tag{120}$$

The nonnegativity constraints $x_0 \geq 0$, $x_1 \geq 0$ and $x_2 \geq 0$ are summarized by $\tilde{\mathbf{x}} \geq \mathbf{0}$. The inequality constraint $x_0 + x_1 + x_2 \leq 1$ is summarized by $D\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix}, \quad \tilde{\mathbf{e}} = \begin{bmatrix} -1 \end{bmatrix} \tag{121}$$

6.19 Line to Box

The box is $\mathbf{p}_0 + x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + x_2\mathbf{u}_2$ where $\mathbf{u}_0$, $\mathbf{u}_1$ and $\mathbf{u}_2$ are mutually perpendicular and where $x_0 \in [0,1]$, $x_1 \in [0,1]$ and $x_2 \in [0,1]$. The line is $\mathbf{p}_1 + x_3\mathbf{u}_3$. The conversion to a CQP with elimination of x_3 is identical to that of the line-tetrahedron distance query except that the inequality constraints are $\tilde{D}\tilde{\mathbf{x}} \geq \tilde{\mathbf{e}}$ where

$$\tilde{D} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad \tilde{\mathbf{e}} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix} \tag{122}$$

6.20 Line to Convex Polyhedron

References

[1] Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone.
The Linear Complementarity Problem.
Academic Press, San Diego, CA, 1992.

[2] Joel Friedman.
Linear complementarity and mathematical (non-linear) programming.
http://www.math.ubc.ca/~jf/courses/340/pap.pdf,
April 1998.

[3] Robert J. Vanderbei.
Linear Programming: Chapter 3 - Degeneracy.
http://www.princeton.edu/~rvdb/522/Fall13/lectures/lec3.pdf,
September 2013.