Convex Quadratic Programming

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

Created: December 17, 2017

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

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

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 the query for computing the distance between a line and a triangle in 3D. The triangle has vertices $\mathbf{P}_i$ for $0 \leq i \leq 2$ and is parameterized by $\mathbf{P}_0 + x_0\mathbf{E}_0 + x_1\mathbf{E}_1$, where $\mathbf{E}_0 = \mathbf{P}_1 - \mathbf{P}_0$, $\mathbf{E}_1 = \mathbf{P}_2 - \mathbf{P}_0$, $x_0 \geq 0$, $x_1 \geq 0$ and $x_0 + x_1 \leq 1$. The line has origin $\mathbf{Q}$ and nonzero direction vector $\mathbf{D}$ and is parameterized by $\mathbf{Q} + x_2\mathbf{D}$ for $x_2 \in \mathbb{R}$. The variable x_2 is unconstrained.

For a triangle point whose parameters are (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 $x_0 \geq 0$,

$x_1 \geq 0$ and $x_0 + x_1 \leq 1$. Using

$$\mathbf{x} = x_0 \begin{bmatrix} 1 \\ 0 \\ -a_{20}/a_{22} \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ 1 \\ -a_{21}/a_{22} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -b_2/a_{22} \end{bmatrix} = x_0\mathbf{u}_0 + x_1\mathbf{u}_1 + \mathbf{u}_2 \tag{3}$$

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

$$\tilde{A} = \begin{bmatrix} \mathbf{u}_0^\mathsf{T}A\mathbf{u}_0 & \mathbf{u}_0^\mathsf{T}A\mathbf{u}_1 \\ \mathbf{u}_1^\mathsf{T}A\mathbf{u}_0 & \mathbf{u}_q^\mathsf{T}A\mathbf{u}_q \end{bmatrix}, \quad \tilde{\mathbf{b}} = \begin{bmatrix} \mathbf{u}_0^\mathsf{T}A\mathbf{u}_2 \\ \mathbf{u}_1^\mathsf{T}A\mathbf{u}_2 \end{bmatrix}, \quad \tilde{c} = \frac{1}{2}\mathbf{u}_2^\mathsf{T}A\mathbf{u}_2 + \mathbf{b}^\mathsf{T}\mathbf{u}_2 + c \tag{4}$$

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

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix}, \quad A = \begin{bmatrix} A_{cc} & A_{cu} \\ A_{uc} & A_{uu} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_c \\ \mathbf{b}_u \end{bmatrix} \tag{5}$$

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

$$\begin{aligned} f(\mathbf{x}_c, \mathbf{x}_u) &= \tfrac{1}{2}\begin{bmatrix} \mathbf{x}_c^\mathsf{T} & \mathbf{x}_u^\mathsf{T} \end{bmatrix} \begin{bmatrix} A_{cc} & A_{cu} \\ A_{uc} & A_{uu} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix} + \begin{bmatrix} \mathbf{b}_c^\mathsf{T} \\ \mathbf{b}_u^\mathsf{T} \end{bmatrix} \begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_u \end{bmatrix} + c \\ &= \tfrac{1}{2}\mathbf{x}_c^\mathsf{T}A_{cc}\mathbf{x}_c + \left(\mathbf{x}_u^\mathsf{T}A_{uc} + \mathbf{b}_c^\mathsf{T}\right)\mathbf{x}_c + \left(\tfrac{1}{2}\mathbf{x}_u^\mathsf{T}A_{uu}\mathbf{x}_u + \mathbf{b}_u^\mathsf{T}\mathbf{x}_u + c\right) \end{aligned} \tag{6}$$

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

$$\mathbf{0} = \frac{\partial f}{\partial \mathbf{x}_u} = A_{uc}\mathbf{x}_c + A_{uu}\mathbf{x}_u + \mathbf{b}_u \tag{7}$$

The solution is

$$\mathbf{x}_u = -A_{uu}^{-1}\left(A_{uc}\mathbf{x}_c + \mathbf{b}_u\right) \tag{8}$$

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

$$g(\mathbf{x}_c) = \frac{1}{2}\mathbf{x}_c^\mathsf{T}\tilde{A}\mathbf{x}_c + \tilde{\mathbf{b}}^\mathsf{T}\mathbf{x}_c + \tilde{c} \tag{9}$$

where

$$\tilde{A} = A_{cc} - A_{cu}A_{uu}^{-1}A_{uc}, \quad \tilde{b} = \mathbf{b}_c - A_{cu}A_{uu}^{-1}\mathbf{b}_u, \quad \tilde{c} = \frac{1}{2}\mathbf{x}_u^\mathsf{T}A_{uu}\mathbf{x}_u + \mathbf{b}_u^\mathsf{T}\mathbf{x}_u + c \tag{10}$$

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 (8) to obtain $\mathbf{x}_u$.

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.

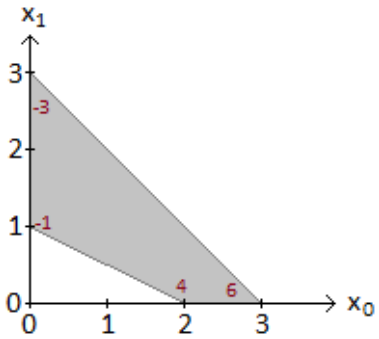
1. The variable complementary to the leaving variable cannot enter the dictionary. In this case, the LCP does not have a solution.
2. It is possible to encounter a cycle in the dictionaries, which prevents the algorithm from converging to a solution. When this happens, one of the components of $\mathbf{q}$ 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

$$\mathbf{q} = \begin{bmatrix} 2 \\ -1 \\ \hline 3 \\ -2 \end{bmatrix}, \quad M = \left[\begin{array}{cc|cc} 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & -2 \\ \hline -1 & -1 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{array}\right], \quad \mathbf{z} = \begin{bmatrix} x_0 \\ x_1 \\ \hline y_0 \\ y_1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} u_0 \\ u_1 \\ \hline v_0 \\ v_1 \end{bmatrix} \tag{11}$$

The initial dictionary with auxiliary variable z_4 is

$$\begin{aligned} w_0 &= 2 + z_2 - z_3 + z_4 \\ w_1 &= -1 + z_2 - 2z_3 + z_4 \\ w_2 &= 3 - z_0 - z_1 + z_4 \\ w_3 &= -2 + z_0 + 2z_1 + z_4 \end{aligned} \tag{12}$$

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

$$\begin{aligned} w_0 &= 4 - z_0 - 2z_1 + z_2 - z_3 + w_3 \\ w_1 &= 1 - z_0 - 2z_1 + z_2 - 2z_3 + w_3 \\ w_2 &= 5 - 2z_0 - 3z_1 + w_3 \\ z_4 &= 2 - z_0 - 2z_1 + w_3 \end{aligned} \tag{13}$$

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,

$$\begin{aligned} w_0 &= (7/2) - (1/2)z_0 - z_1 + (1/2)z_2 + (1/2)w_1 + (1/2)w_3 \\ z_3 &= (1/2) - (1/2)z_0 - z_1 + (1/2)z_2 - (1/2)w_1 + (1/2)w_3 \\ w_2 &= 5 - 2z_0 - 3z_1 + w_3 \\ z_4 &= 2 - z_0 - 2z_1 + w_3 \end{aligned} \tag{14}$$

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,

$$\begin{aligned} w_0 &= 3 + z_3 + w_1 \\ z_1 &= (1/2) - (1/2)z_0 - z_3 + (1/2)z_2 - (1/2)w_1 + (1/2)w_3 \\ w_2 &= (7/2) - (1/2)z_0 + 3z_3 - (3/2)z_2 + (3/2)w_1 - (1/2)w_3 \\ z_4 &= 1 + 2z_3 - z_2 + w_1 \end{aligned} \tag{15}$$

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,

$$\begin{aligned} w_0 &= 3 + z_3 + w_1 \\ z_1 &= 4 - z_0 + 2z_3 - z_2 + w_1 - w_2 \\ w_3 &= 7 - z_0 + 6z_3 - 3z_2 + 3w_1 - 2w_2 \\ z_4 &= 1 + 2z_3 - z_2 + w_1 \end{aligned} \tag{16}$$

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,

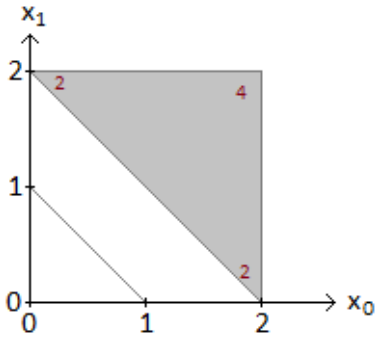
$$\begin{aligned} w_0 &= 3 + z_3 + w_1 \\ z_1 &= 3 - z_0 + z_4 - w_2 \\ w_3 &= 4 - z_0 + 3z_4 - 2w_2 \\ z_2 &= 1 + 2z_3 - z_4 + w_1 \end{aligned} \tag{17}$$

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 $0 \leq x_0 \leq 2$, $0 \leq x_1 \leq 2$, $x_0 + x_1 \geq 1$ and $x_0 + x_1 \geq 2$. The figure shows the domain of f that is defined by the inequality constraints. The constraint $x_0 + x_1 \geq 1$ 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

$$\mathbf{q} = \left[\begin{array}{r} 1 \\ 1 \\ \hline -1 \\ -2 \\ 2 \\ 2 \end{array}\right], \quad M = \left[\begin{array}{rr|rrrr} 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \end{array}\right], \quad \mathbf{z} = \left[\begin{array}{c} x_0 \\ x_1 \\ \hline y_0 \\ y_1 \\ y_2 \\ y_3 \end{array}\right], \quad \mathbf{w} = \left[\begin{array}{c} u_0 \\ u_1 \\ \hline v_0 \\ v_1 \\ v_2 \\ v_3 \end{array}\right] \tag{18}$$

The initial dictionary with auxiliary variable z_6 is

$$\begin{aligned} w_0 &= 1 - z_2 - z_3 + z_4 + z_6 \\ w_1 &= 1 - z_2 - z_3 + z_5 + z_6 \\ w_2 &= -1 + z_0 + z_1 + z_6 \\ w_3 &= -2 + z_0 + z_1 + z_6 \\ w_4 &= 2 - z_0 + z_6 \\ w_5 &= 2 - z_1 + z_6 \end{aligned} \tag{19}$$

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

$$\begin{aligned} w_0 &= 3 - z_0 - z_2 - z_3 + z_4 + w_3 \\ w_1 &= 3 - z_0 - z_2 - z_3 + z_5 + w_3 \\ w_2 &= 1 - w_3 \\ z_6 &= 2 - z_0 - z_1 + w_3 \\ w_4 &= 4 - 2z_0 - z_1 + w_3 \\ w_5 &= 4 - z_0 - 2z_1 + w_3 \end{aligned} \tag{20}$$

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,

$$\begin{aligned} z_3 &= 3 - z_0 - z_1 - z_2 - w_0 + z_4 + w_3 \\ w_1 &= 0 + w_0 - z_4 + z_5 \\ w_2 &= 1 - w_3 \\ z_6 &= 2 - z_0 - z_1 + w_3 \\ w_4 &= 4 - 2z_0 - z_1 + w_3 \\ w_5 &= 4 - z_0 - 2z_1 + w_3 \end{aligned} \tag{21}$$

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,

$$\begin{aligned} z_3 &= 1 - z_2 - w_0 + z_4 + z_6 \\ w_1 &= 0 + w_0 - z_4 + z_5 \\ w_2 &= 1 - w_3 \\ z_0 &= 2 - z_6 - z_1 + w_3 \\ w_4 &= 0 + z_1 + 2z_6 - w_3 \\ w_5 &= 2 - z_1 + z_6 \end{aligned} \tag{22}$$

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

$$\mathbf{q} = \begin{bmatrix} 2 \\ -1 \\ 1 \end{bmatrix}, \quad M = \left[\begin{array}{cc|c} 0 & 0 & -1 \\ 0 & 0 & -1 \\ \hline 1 & 1 & 0 \end{array}\right], \quad \mathbf{z} = \left[\begin{array}{c} x_0 \\ x_1 \\ \hline y_0 \end{array}\right], \quad \mathbf{w} = \left[\begin{array}{c} u_0 \\ u_1 \\ \hline v_0 \end{array}\right], \tag{23}$$

The initial dictionary with auxiliary variable z_3 is

$$\begin{aligned} w_0 &= 2 - z_2 + z_3 \\ w_1 &= -1 - z_2 + z_3 \\ w_2 &= 1 + z_0 + z_1 + z_3 \end{aligned} \tag{24}$$

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

$$\begin{aligned} w_0 &= 3 \\ z_3 &= 1 + z_2 + w_1 \\ w_2 &= 2 + z_0 + z_1 + z_3 + w_1 \end{aligned} \tag{25}$$

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 GTEngine 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 $\mathbf{q}$ 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,

$$\begin{aligned} w_0 &= -1 + z_0 - 2z_2 + z_3 \\ w_1 &= -1 + 2z_1 - z_2 + z_3 \\ w_2 &= -1 + 2z_0 + z_1 + z_3 \end{aligned} \tag{26}$$

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,

$$\begin{aligned} z_3 &= 1 - z_0 + 2z_2 + w_0 \\ w_1 &= 0 - z_0 + 2z_1 + z_2 + w_0 \\ w_2 &= 0 + z_0 + z_1 + 2z_2 + w_0 \end{aligned} \tag{27}$$

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,

$$\begin{aligned} z_3 &= 1 + w_1 - 2z_1 + z_2 \\ z_0 &= 0 - w_1 + 2z_1 + z_2 + w_0 \\ w_2 &= 0 - w_1 + 3z_1 + 3z_2 + 2w_0 \end{aligned} \tag{28}$$

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,

$$\begin{aligned} z_1 &= (1/2) + (1/2)w_1 - (1/2)z_3 + (1/2)z_2 \\ z_0 &= 1 - z_3 + 2z_2 + w_0 \\ w_2 &= (3/2) + (1/2)w_1 - (3/2)z_3 + (9/2)z_2 + 2w_0 \end{aligned} \tag{29}$$

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

$$\begin{aligned} w_0 &= (-1 + \varepsilon) + z_0 - 2z_2 + z_3 \\ w_1 &= (-1 + \varepsilon^2) + 2z_1 - z_2 + z_3 \\ w_2 &= (-1 + \varepsilon^3) + 2z_0 + z_1 + z_3 \end{aligned} \tag{30}$$

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,

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

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,

$$\begin{aligned} w_0 &= (\varepsilon - 2\varepsilon^2 + \varepsilon^3) + 3z_0 - 3z_1 + 2w_1 - w_2 \\ z_2 &= (\varepsilon^2 - \varepsilon^3) - 2z_0 + z_1 - w_1 + w_2 \\ z_3 &= (1 - \varepsilon^3) - 2z_0 - z_1 + w_2 \end{aligned} \tag{32}$$

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,

$$\begin{aligned} z_1 &= ((1/3)\varepsilon - (2/3)\varepsilon^2 + (1/3)\varepsilon^3) + z_0 - (1/3)w_0 + (2/3)w_1 - (1/3)w_2 \\ z_2 &= ((1/3)\varepsilon + (1/3)\varepsilon^2 - (2/3)\varepsilon^3) - z_0 - (1/3)w_0 - (1/3)w_1 + (2/3)w_2 \\ z_3 &= (1 - (1/3)\varepsilon + (2/3)\varepsilon^2 - (4/3)\varepsilon^3) - 3z_0 + (1/3)w_0 - (2/3)w_1 + (4/3)w_2 \end{aligned} \tag{33}$$

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,

$$\begin{aligned} z_1 &= (0 + (2/3)\varepsilon - (1/3)\varepsilon^2 - (1/3)\varepsilon^3) - z_2 - (2/3)w_0 + (1/3)w_1 + (1/3)w_2 \\ z_0 &= ((1/3)\varepsilon + (1/3)\varepsilon^2 - (2/3)\varepsilon^3) - z_2 - (1/3)w_0 - (1/3)w_1 + (2/3)w_2 \\ z_3 &= (1 - (4/3)\varepsilon - (1/3)\varepsilon^2 + (2/3)\varepsilon^3) + 3z_2 + (4/3)w_0 + (1/3)w_1 - (2/3)w_2 \end{aligned} \tag{34}$$

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

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

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

$$\mathbf{P}(\boldsymbol{\xi}) = \mathbf{K} + \sum_{j=0}^{n-1} \xi_j \mathbf{U}_j = \mathbf{K} + R\boldsymbol{\xi}, \;\; \mathbf{0} \le \boldsymbol{\xi} \le \boldsymbol{\ell} \tag{36}$$

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

$$\mathbf{P}_i(\boldsymbol{\xi}_i) = \mathbf{K}_i + R_i\boldsymbol{\xi}_i, \;\; \mathbf{0} \le \boldsymbol{\xi}_i \le \boldsymbol{\ell}_i \tag{37}$$

for $i \in \{0, 1\}$. All components are doubly indexed: $\mathbf{P}_i$ has components p_{ij}, $\mathbf{K}_i$ has components k_{ij}, R_i has columns $\mathbf{U}_{ij}$, $\boldsymbol{\ell}_i$ has components ℓ_{ij} and $\boldsymbol{\xi}_i$ has components ξ_{ij}.

Define $\mathbf{\Delta} = \mathbf{K}_1 - \mathbf{K}_0$. Half the squared distance between box points is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2}\left|\mathbf{P}_0(\boldsymbol{\xi}_0) - \mathbf{P}_1(\boldsymbol{\xi}_1)\right|^2 \\ &= \tfrac{1}{2}\left|R_0\boldsymbol{\xi}_0 - R_1\boldsymbol{\xi}_1 - \mathbf{\Delta}\right|^2 \\ &= \tfrac{1}{2}\left(\boldsymbol{\xi}_0^{\mathsf{T}}R_0^{\mathsf{T}}R_0\boldsymbol{\xi}_0 + \boldsymbol{\xi}_1^{\mathsf{T}}R_1^{\mathsf{T}}R_1\boldsymbol{\xi}_1 + \mathbf{\Delta}^{\mathsf{T}}\mathbf{\Delta} - 2\boldsymbol{\xi}_0^{\mathsf{T}}R_0^{\mathsf{T}}R_1\boldsymbol{\xi}_1 - 2\mathbf{\Delta}^{\mathsf{T}}R_0\boldsymbol{\xi}_0 + 2\mathbf{\Delta}^{\mathsf{T}}R_1\boldsymbol{\xi}_1\right) \\ &= \tfrac{1}{2}\left[\begin{array}{c|c} \boldsymbol{\xi}_0^{\mathsf{T}} & \boldsymbol{\xi}_1^{\mathsf{T}} \end{array}\right] \left[\begin{array}{c|c} I & -R_0^{\mathsf{T}}R_1 \\ \hline -R_1^{\mathsf{T}}R_0 & I \end{array}\right] \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right] + \left[\begin{array}{c|c} -\mathbf{\Delta}^{\mathsf{T}}R_0 & \mathbf{\Delta}^{\mathsf{T}}R_1 \end{array}\right] \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right] + \tfrac{1}{2}\mathbf{\Delta}^{\mathsf{T}}\mathbf{\Delta} \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \end{aligned} \tag{38}$$

where

$$\mathbf{x} = \left[\begin{array}{c} \boldsymbol{\xi}_0 \\ \hline \boldsymbol{\xi}_1 \end{array}\right], \quad A = \left[\begin{array}{c|c} I & -R_0^{\mathsf{T}}R_1 \\ \hline -R_1^{\mathsf{T}}R_0 & I \end{array}\right], \quad \mathbf{b} = \left[\begin{array}{c} -R_0^{\mathsf{T}}\mathbf{\Delta} \\ \hline R_1^{\mathsf{T}}\mathbf{\Delta} \end{array}\right], \quad c = \frac{1}{2}\left|\mathbf{\Delta}\right|^2, \quad \mathbf{L} = \left[\begin{array}{c} \boldsymbol{\ell}_0 \\ \hline \boldsymbol{\ell}_1 \end{array}\right] \tag{39}$$

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

$$D = \left[\begin{array}{c} -I \\ \hline -I \end{array}\right], \quad \mathbf{e} = \left[\begin{array}{c} -\boldsymbol{\ell}_0 \\ \hline -\boldsymbol{\ell}_1 \end{array}\right] \tag{40}$$

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 triangle in n-dimensions has vertices $\mathbf{V}_i$ for $i \in \{0, 1, 2\}$ and linearly independent edges $\mathbf{E}_j = \mathbf{V}_{j+1} - \mathbf{V}_0$ for $j \in \{0, 1\}$. Define the parameter pair $\mathbf{x} = (x_0, x_1)$. The solid triangle is parameterized by

$$\mathbf{P}(\mathbf{x}) = \mathbf{V}_0 + x_0\mathbf{E}_0 + x_1\mathbf{E}_1 = \mathbf{V}_0 + E\mathbf{x}, \;\; x_0 \ge 0, \; x_1 \ge 0, \; x_0 + x_1 \le 1 \tag{41}$$

where E is an $n \times 2$ matrix whose columns are the edges. A solid and 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 and finite cylinder truncates the solid and infinite cylinder by two hyperplanes $\mathbf{U}_0 \cdot (\mathbf{P} - (\mathbf{K} \pm (h/2)\mathbf{U}_0)) = 0$, keeping only those solid and infinite cylinder points between the two hyperplanes; h is the height of the 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

$$\mathbf{Q}(\mathbf{t}) = \mathbf{K} + t_0\mathbf{U}_0 + \sum_{j=1}^{n-1} t_j\mathbf{U}_j = \mathbf{K} + R\mathbf{t}, \;\; |t_0| \le h/2, \; \sum_{j=1}^{n-1} t_j^2 \le r^2 \tag{42}$$

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 the cylinder axis is

$$\begin{aligned} f(\mathbf{x}) &= \tfrac{1}{2}\left|\mathcal{P}\left(\mathbf{P}(\mathbf{x}) - \mathbf{K}\right)\right|^2 \\ &= \tfrac{1}{2}\left|\mathcal{P}\left(E\mathbf{x} + \mathbf{\Delta}\right)\right|^2 \\ &= \tfrac{1}{2}\left(E\mathbf{x} + \mathbf{\Delta}\right)^{\mathsf{T}}\mathcal{P}\left(E\mathbf{x} + \mathbf{\Delta}\right) \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}E^{\mathsf{T}}\mathcal{P}E\mathbf{x} + \mathbf{\Delta}^{\mathsf{T}}\mathcal{P}E\mathbf{x} + \tfrac{1}{2}\mathbf{\Delta}^{\mathsf{T}}\mathcal{P}\mathbf{x} \\ &= \tfrac{1}{2}\mathbf{x}^{\mathsf{T}}A\mathbf{x} + \mathbf{b}^{\mathsf{T}}\mathbf{x} + c \end{aligned} \tag{43}$$

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

$$x_0 + x_1 \le 1, \quad h/2 \ge |t_0| = |\mathbf{U}_0 \cdot (P(\mathbf{x}) - \mathbf{K})| = |\mathbf{U}_0 \cdot (E\mathbf{x} + \mathbf{\Delta})| = |\mathbf{U}_0^{\mathsf{T}}E\mathbf{x} + \mathbf{U}_0^{\mathsf{T}}\mathbf{\Delta}| \tag{44}$$

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

$$D = \begin{bmatrix} -\mathbf{1}^{\mathsf{T}} \\ \mathbf{U}_0^{\mathsf{T}}E \\ -\mathbf{U}_0^{\mathsf{T}}E \end{bmatrix}, \quad \mathbf{e} = \begin{bmatrix} -1 \\ -h/2 - \mathbf{U}_0^{\mathsf{T}}\mathbf{\Delta} \\ -h/2 + \mathbf{U}_0^{\mathsf{T}}\mathbf{\Delta} \end{bmatrix} \tag{45}$$

where D is a 3×2 matrix with $\mathbf{1}$ a 2×1 vector whose components are both 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 GTEngine 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 GTEngine: 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 GTEngine 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 GteLCPSolver.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 GTEngine 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. 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 $|x_i| \le e_i$. 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 GTEngine
    // 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, 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. Pseudocode for computing the distance between two oriented boxes 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 GTEngine
    // 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 GteLCPSolver.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 GteDistTriangle3AlignedBox3.h. Listing 4 shows a test program that computes the distance using fixed precision and using arbitrary precision.

Listing 4. 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}1100110010001011001011111111000100001011100000001111
// 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 $\mathbf{b} = E^{\mathsf{T}}\mathcal{P}\mathbf{\Delta}$, which is also rational. The quadratic scalar $c = |\mathbf{\Delta}|^2/2$ is rational.

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

$$(\mathbf{V}/|\mathbf{V}|)^{\mathsf{T}}E\mathbf{x} \geq -h/2 - (\mathbf{V}/|\mathbf{V}|)^{\mathsf{T}}\mathbf{\Delta}, \quad -(\mathbf{V}/|\mathbf{V}|)^{\mathsf{T}}E\mathbf{x} \geq -h/2 + (\mathbf{V}/|\mathbf{V}|)^{\mathsf{T}}\mathbf{\Delta} \tag{46}$$

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

$$\mathbf{V}^{\mathsf{T}}E\mathbf{x} \geq -h|\mathbf{V}|/2 - \mathbf{V}^{\mathsf{T}}\mathbf{\Delta}, \quad -\mathbf{V}^{\mathsf{T}}E\mathbf{x} \geq -h|\mathbf{V}|/2 + \mathbf{V}^{\mathsf{T}}\mathbf{\Delta} \tag{47}$$

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 a 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 GTEngine: 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. Use of 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. Use of 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. Use of 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 (47); 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 GTEngine: Arbitrary Precision Arithmetic.

Listing 8. 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 $\mathbf{D} \cdot \mathbf{X} = 1$ of one of the cylinder caps.

Listing 9. The main function to compare the results of the triangle-cylinder intersection query for various numeric types.

```
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;
}
```

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.

[4] Wikipedia.
Methods of Computing Square Roots.
https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Babylonian_method.
accessed November 26, 2017.