Distance Between Point and Parallelogram

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

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Created: October 15, 2024

Contents

1 Motivation: Distance Between a Point and an Oriented Box **2**

2 Distance to a Parallelogram **4**

2.1 Voronoi-Based Algorithm 4

2.2 Convex Quadratic Programming Problem on Square Domain 5

3 Implementation **8**

This document describes an efficient algorithm for a distance and closest point query given an input point and an input parallelogram. All figures were drawn using Mathematica [1]. The implementation section mentions that Mathematica was also used to provide query results to be compared to those of the unit tests.

1 Motivation: Distance Between a Point and an Oriented Box

An oriented bounding box in n-dimensions has an $n \times 1$ center point $\boldsymbol{C}$. The box has n axis directions $\boldsymbol{U}_i$, each axis having 0 in all all components except for the i-th component ($0 \le i < n$) which has 1. The vectors form a right-handed orthonormal basis: $\boldsymbol{U}_i^{\mathsf{T}}\boldsymbol{U}_i = 1$ for all i and $\boldsymbol{U}_i^{\mathsf{T}}\boldsymbol{U}_j = 0$ for all $i \neq j$. The box has n extents $e_i > 0$ for $0 \le i < n$ that are the half-lengths of the box edges. The box center and axes form an affine coordinate system. Points inside the box are represented by

$$\boldsymbol{X} = \boldsymbol{C} + \sum_{i=0}^{n} y_i \boldsymbol{U}_i = \boldsymbol{C} + R\boldsymbol{Y} \tag{1}$$

where R is an $n \times n$ rotation matrix whose columns are the axis directions and where $\boldsymbol{Y}$ is an $n \times 1$ vector whose rows are the y_i. The coefficients are constrained by $|\boldsymbol{Y}| \le \boldsymbol{E}$, where $\boldsymbol{E}$ is an $n \times 1$ vector whose rows are the e_i and where the inequalities are applied componentwise; that is $-e_i \le y_i \le e_i$ for $0 \le i < n$.

The box features are the convex polygons forming the boundary of the box. The Voronoi regions for the box features make it easy to compute the distance from an $n \times 1$ point $\boldsymbol{P}$ to the box. The representation of the point is

$$\boldsymbol{P} = \boldsymbol{C} + \sum_{i=0}^{n} z_i \boldsymbol{U}_i = \boldsymbol{C} + R\boldsymbol{Z} \tag{2}$$

where $\boldsymbol{Z}$ is an $n \times 1$ vector whose rows are the z_i. The squared distance between box points $\boldsymbol{X}$ and $\boldsymbol{P}$ is $f(\boldsymbol{X}) = |\boldsymbol{X} - \boldsymbol{P}|^2 = |R(\boldsymbol{Y} - \boldsymbol{Z})|^2 = |\boldsymbol{Y} - \boldsymbol{Z}|^2$. The last equality omits the rotation matrix R because a rotation does not change the length of vectors. Effectively, the distance query is now that for an axis-aligned box centered at the origin. In the Geometric Tools implementation, such an aligned box is referred to as a *canonical box*. The closest feature of the canonical box is based on identifying the Voronoi regions of which there are 3^n. Listing 1 shows pseudocode for computing the canonical box point $\boldsymbol{K}$ that is closest to $\boldsymbol{Z}$.

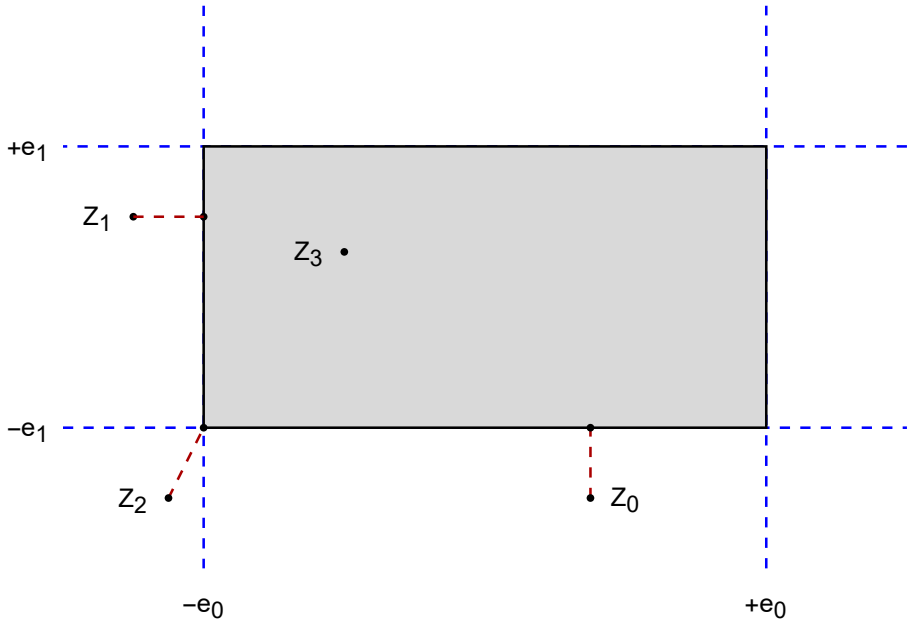
Listing 1. Pseudocode for computing the canonical box point $\boldsymbol{K}$ that is closest to $\boldsymbol{Z}$.

```
void DCPQueryPointCanonicalBox(Vector<Real,n> Z, Vector<Real,n> E, Real& sqrDistance, Vector<Real,n>& K)
{
    sqrDistance = 0;
    K = Z;
    for (int i = 0; i < n; ++i)
    {
        if (K[i] < -E[i])
        {
            Real sum = K[i] + E[i]; sqrDistance += sum * sum; K[i] = -E[i];
        }
        else if (K[i] > E[i])
        {
            Real dif = K[i] - E[i]; sqrDistance += dif * dif; K[i] = E[i];
        }
    }
}
```

Naturally, the transformation from oriented box coordinates to canonical box coordinates must occur first. That is, $\boldsymbol{Z} = R^{\mathsf{T}}(\boldsymbol{P} - \boldsymbol{C})$. After the query, the oriented box closest point is computed to be $\boldsymbol{P} = \boldsymbol{C} + R\boldsymbol{Z}$.

Figure 1 shows several cases for determining closest points, illustrated in 2D.

Figure 1. Several cases for determination of closest points. $\boldsymbol{Z}_0$ is in the Voronoi region for the bottom edge. The projection onto the bottom edge is the box point closest to $\boldsymbol{Z}_0$. $\boldsymbol{Z}_1$ is in the Voronoi region for the left edge. The projection onto the left edge is the box point closest to $\boldsymbol{Z}_1$. $\boldsymbol{Z}_2$ is in the Voronoi region for the lower-left corner. The corner is the box point closest to $\boldsymbol{Z}_2$. $\boldsymbol{Z}_3$ is inside the box, so it already is the point closest to the box and has distance 0 to the box.



Observe that for points in the other Voronoi regions, rather than process those regions directly it is possible to use reflections about the origin to obtain one of the point configurations shown in Figure 1. For example, if (s, t) is in the Voronoi region for the top edge, we can reflect that point to $(s, -t)$ so that the case of processing $\boldsymbol{Z}_0$ applies. The canonical box point closest to $\boldsymbol{Z}_0$ is $\boldsymbol{K}_0 = (s, -1)$. The inverse reflection is applied to obtain $(s, +1)$ which is the canonical box point closest to (s, t). Similarly, if (s, t) is in the Voronoi region for the upper-right corner, we can reflect that point to $(-s, -t)$ so that the case of processing $\boldsymbol{Z}_2$ applies. The canonical box point closest to $\boldsymbol{Z}_2$ is $\boldsymbol{K}_2 = (-1, -1)$. The inverse reflection is applied to obtain $(+1, +1)$ which is the canonical box point closest to (s, t). For the 2D case, reflections map points to 1 of 3 Voronoi regions. These are $\{(s, t) : s < -1 \wedge |t| \le 1\}$ (closest to the bottom edge), $\{(s, t) : |s| \le 1 \wedge t < 0\}$ (closest to the left edge), and $\{(s, t) : s < -1 \wedge t < -1\}$ (closest to the lower-left corner).

The pseudocode corresponding to the reflection preprocessing is more complicated than that of listing 1. However, the idea will be important when processing parallelograms in 2D or parallelepipeds in 3D.

2 Distance to a Parallelogram

A parallelogram is defined by a center point $\boldsymbol{C}$ and axes $\{\boldsymbol{V}_0, \boldsymbol{V}_1\}$. As a basis of vectors, the vectors are not necessarily unit length or orthogonal. To be right handed, $\boldsymbol{V}_0 \cdot \boldsymbol{V}_1^{\perp} > 0$, where the perp operator is defined by $(x, y)^{\perp} = (y, -x)$. The points in the parallelogram are represented as

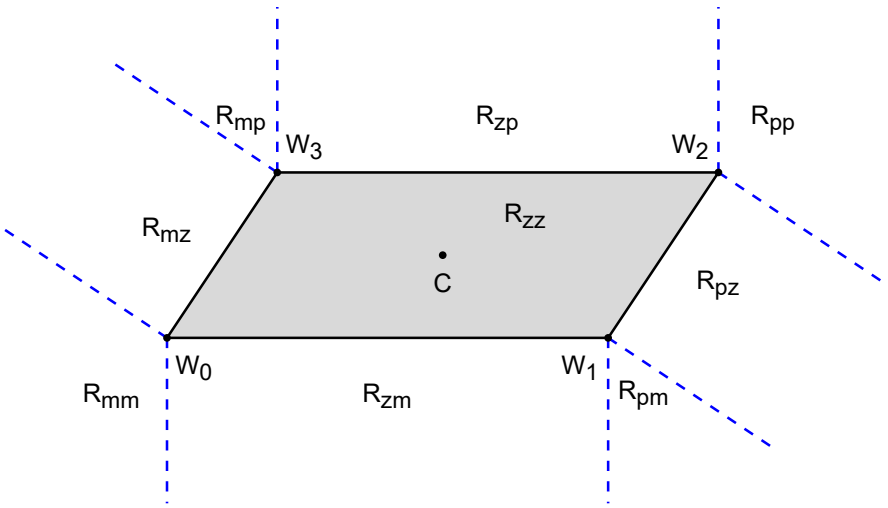
$$\boldsymbol{X} = \boldsymbol{C} + \sum_{i=0}^{1} y_i \boldsymbol{V}_i = \boldsymbol{C} + B\boldsymbol{Y} \tag{3}$$

where B is a 2×2 matrix whose columns are the axis directions which implies $\det(B) > 0$. $\boldsymbol{Y}$ is a 2×1 vector whose rows are the y_i. The coefficients are constrained by $|\boldsymbol{Y}| \le \boldsymbol{1}$, where $\boldsymbol{1}$ is a 2×1 vector whose rows are all scalar 1 and where the inequalities are applied componentwise; that is $-1 \le y_i \le 1$ for $0 \le i \le 1$.

2.1 Voronoi-Based Algorithm

A Voronoi-region-based approach can be used as was done for oriented boxes. The parallelepiped has features: corners, edges, and interior, so there are also 9 regions. However, more computations must be applied to determine the regions. In particular, face normals must be computed to support closest-feature selection. Figure 2 shows the Voronoi regions for a parallelogram in 2D. The axis $\boldsymbol{V}_0$ is half the vector of the bottom (or top) edge. The axis $\boldsymbol{V}_1$ is half the vector of the left (or right) edge.

Figure 2. The Voronoi regions for a parallelogram. The corner points are $\boldsymbol{W}_0 = \boldsymbol{C} - \boldsymbol{V}_0 - \boldsymbol{V}_1$, $\boldsymbol{W}_1 = \boldsymbol{C} + \boldsymbol{V}_0 - \boldsymbol{V}_1$, $\boldsymbol{W}_2 = \boldsymbol{C} + \boldsymbol{V}_0 + \boldsymbol{V}_1$, and $\boldsymbol{W}_3 = \boldsymbol{C} - \boldsymbol{V}_0 + \boldsymbol{V}_1$. The Voronoi regions are labeled R_{ij} where m denotes a region where $y_k < -1$, z denotes a region where $-1 \le y_k \le 1$, and p denotes a region where $1 < y_k$.



Given a query point $\boldsymbol{P}$, it is possible to determine which Voronoi region contains the point using point-in-region tests. For example, if $\boldsymbol{P} \in R_{mm}$, then $\boldsymbol{W}_0$ is the parallelogram point closest to $\boldsymbol{P}$. If $\boldsymbol{P} \in R_{zm}$,

then the projection of $\boldsymbol{P}$ onto the bottom edge is the parallelogram point closest to $\boldsymbol{P}$. However, this is an exhaustive search, which is not efficient.

2.2 Convex Quadratic Programming Problem on Square Domain

Write the query point in the affine coordinate system imposed by the parallelogram,

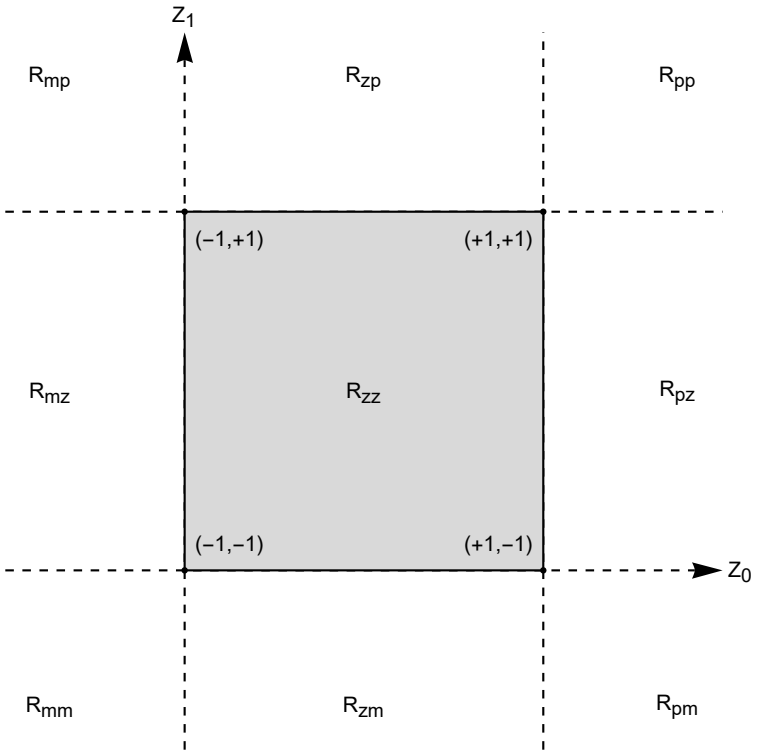
$$\boldsymbol{P} = \boldsymbol{C} + \sum_{i=0}^{1} z_i \boldsymbol{V}_i = C + B\boldsymbol{Z} \tag{4}$$

where $\boldsymbol{Z}$ is a 2×1 vector whose rows are the z_i. The squared distance between parallelogram points $\boldsymbol{X}$ and $\boldsymbol{P}$ is

$$f(\boldsymbol{X}) = |\boldsymbol{X} - \boldsymbol{P}|^2 = |B(\boldsymbol{Y} - \boldsymbol{Z})| = (\boldsymbol{Y} - \boldsymbol{Z})^{\mathsf{T}} B^{\mathsf{T}} B(\boldsymbol{Y} - \boldsymbol{Z}) = (\boldsymbol{Y} - \boldsymbol{Z})^{\mathsf{T}} A(\boldsymbol{Y} - \boldsymbol{Z}) = g(\boldsymbol{Y}) \tag{5}$$

The matrix $A = B^{\mathsf{T}}B$ is 2×2 with entries $a_{ij} = \boldsymbol{V}_i^{\mathsf{T}} \boldsymbol{V}_j$. The convex quadratic programming problem is to minimize $q(\boldsymbol{Y})$ subject to the constraints $|\boldsymbol{Y}| \leq \boldsymbol{1}$ where $\boldsymbol{1}$ is the 2×1 vector with all entries 1. The convex domain defined by the constraints is a square. Figure 3 shows the domain decomposed based on values relative to ± 1. Let $\boldsymbol{Z} = (z_0, z_1)$, written as a 2-tuple.

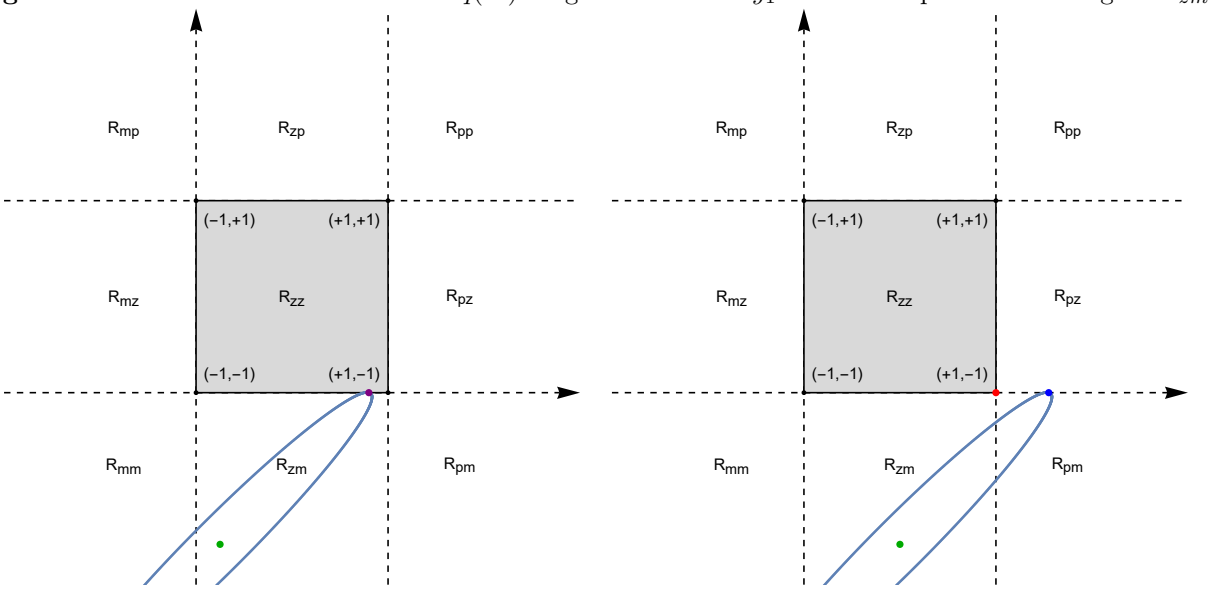
Figure 3. The convex domain for the convex quadratic programming problem is the unit square. The regions R_{*m} occur when $z_1 < -1$, R_{mm} occurs when $z_0 < -1$, R_{zm} occurs when $|z_0| \leq 1$, and R_{pm} occurs when $z_0 > +1$. The other regions are labeled similarly. The region R_{zz} is convex domain itself.



If $\boldsymbol{Z}$ is in R_{zz}, then the distance is 0 and $\boldsymbol{P}$ is already the closest parallelogram point. For $\boldsymbol{P}$ strictly outside the square domain, the ensuing argument applies.

Suppose that $|z_0| \le 1$ and $z_1 < 0$, in which case $\boldsymbol{Z}$ is in R_{zm}. The level curves of $q(\boldsymbol{Y})$ are ellipses $(\boldsymbol{Y}-\boldsymbol{Z})^{\mathsf{T}} A(\boldsymbol{Y}-\boldsymbol{Z}) = \lambda^2$. Each level curve represents points that are equidistant from $\boldsymbol{Z}$, with the squared distance λ^2. Imagine starting with a degenerate ellipse which is the point $\boldsymbol{Y} = \boldsymbol{Z}$, in which case $\lambda = 0$. Increase λ until the elliptical level curve is tangent to the line $y_1 = -1$. Figure 4 illustrates this.

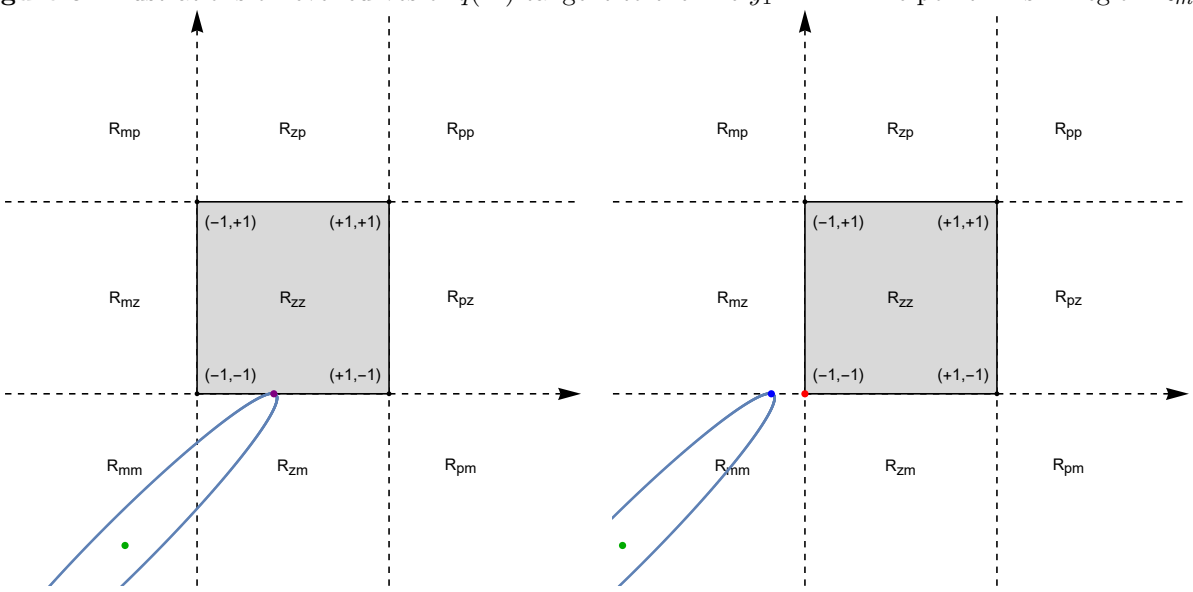
Figure 4. Illustrations of level curves of $q(\boldsymbol{Y})$ tangent to the line $y_1 = -1$. The point $\boldsymbol{Z}$ is in region R_{zm}.



The left image of figure 4 shows that the minimum of $q(\boldsymbol{Y})$ occurs at the purple point where $y_1 = -1$ and $|y_0| < 1$. The right image of figure 4 shows that the minimum occurs at the blue point where $y_1 = -1$ and $y_0 > 1$, but that point is outside the square. The point is clamped to the domain at the red point $(1, -1)$.

Figure 5 shows two possible configurations when $\boldsymbol{Z}$ is in region R_{mm}.

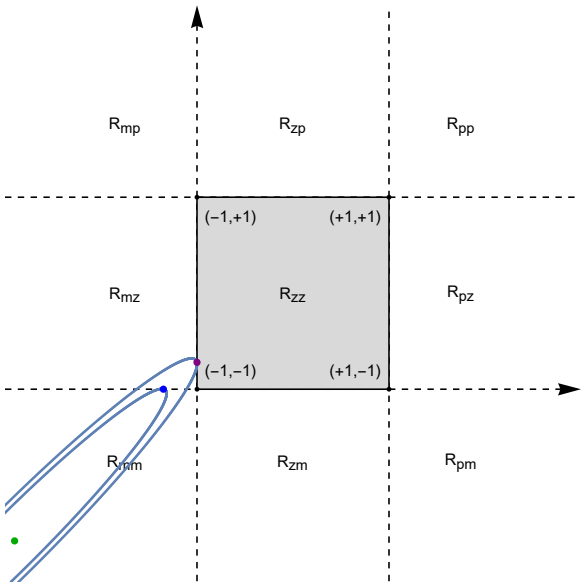
Figure 5. Illustrations of level curves of $q(\boldsymbol{Y})$ tangent to the line $y_1 = -1$. The point $\boldsymbol{Z}$ is in region R_{mm}.



The left image of figure 5 shows that the minimum of $q(\boldsymbol{Y})$ occurs at the purple point where $y_1 = -1$ and $|y_0| < 1$. The right image of figure 5 shows that the minimum occurs at the blue point where $y_1 = -1$ and $y_0 < -1$, but that point is outside the square. The point is clamped to the domain at the red point $(-1, -1)$.

In the right image of figure 5, imagine the ellipse increasing in size but the same center point. When the ellipse grows to just touch the square domain, the first point of contact can be $(-1, -1)$. In this case, the minimum of $q(\boldsymbol{Y})$ occurs at that corner. However, it is possible that the first point of contact occurs on the edge $y_0 = -1$ at a point where $|y_1| < 1$. Figure 6 illustrates this.

Figure 6. Illustrations of a level curve of $q(\boldsymbol{Y})$ tangent to the line $y_0 = -1$ at a point where $|y_1| < 1$. The ellipse that is tangent to the line $y_1 = -1$ has $y_0 < -1$, so the candidate minimum point is the clamped point $(-1, -1)$. As the ellipse is grown, the first point of contact is on the line $y_0 = -1$ at $|y_1| < 1$, so this point is where the minimum occurs.



The conclusion for $\boldsymbol{Z} \in R_{zm}$ is that the minimum of $q(\boldsymbol{Y})$ occurs at a point with $y_1 = -1$ and $|y_0| \le 1$. To compute the point, solve $\partial q/\partial y_0 = 0$ along the line $y_1 = -1$. If $|y_0| < 1$, the point is interior to the bottom edge of the parallelogram. If $|y_0| \ge 1$, the point is clamped to the appropriate corner.

The conclusion for $\boldsymbol{Z} \in R_{mm}$ is that the minimum of $q(\boldsymbol{Y})$ occurs either on the bottom edge or the left edge of the parallogram. It does not matter which edge is examined first, but the Geometric Tools implementation always tests the bottom edge first and the left edge second. If the minimum point occurs at an interior point of the bottom edge, there is no need to examine the left edge. However, if the solution to $\partial q/\partial y_0 = 0$ along the line $y_1 = -1$ has $y_0 < -1$, the point is clamped to $(-1, -1)$. Now the left edge must be examined in a similar fashion. Solve $\partial q/\partial y_1 = 0$ along the line $y_0 = -1$. If $|y_1| < 1$, the point is interior to the left edge of the parallelogram. If $|y_1| \ge 1$, the point is clamped to the appropriate corner.

The analyses of regions R_{mz}, R_{zp}, and R_{pz} are similar to that of R_{zm}. The analyses of regions R_{mp}, R_{pm}, and R_{pp} are similar to that of R_{mm}.

3 Implementation

The Geometric Tools implementation is listed in this section. The parallelogram is defined by the code shown in listing 2.

Listing 2. The Geometric Tools C++ code for a parallelogram. The axis[] vectors need not be unit length nor do they need to be orthogonal. However, a requirement is that the axes form a right-handed basis.

```
#include <Mathematics/Logger.h>
#include <Mathematics/Vector2.h>
#include <array>

namespace gte
{
    template <typename T>
    class Parallelogram2
    {
    public:
        // The default constructor sets the center to (0,0), axis[0] to (1,0),
        // and axis[1] to (0,1).
        Parallelogram2()
            :
            center(Vector2<T>::Zero()),
            axis{ Vector2<T>::Unit(0), Vector2<T>::Unit(1) }
        {
        }

        // The axes must form a right−handed basis. The axes do not have to be
        // orthogonal. The axis lengths do not have to be unit length.
        Parallelogram2(Vector2<T> const& inCenter, std::array<Vector2<T>, 2> const& inAxis)
            :
            center(inCenter),
            axis(inAxis)
        {
            LogAssert(
                DotPerp(inAxis[0], inAxis[1]) > static_cast<T>(0),
                "The axes must form a right−handed basis.");
        }

        // The vertices are listed in counterclockwise order.
        void GetVertices(std::array<Vector2<T>, 4>& vertices)
        {
            vertices[0] = center − axis[0] − axis[1];
            vertices[1] = center + axis[0] − axis[1];
            vertices[2] = center + axis[0] + axis[1];
            vertices[3] = center − axis[0] + axis[1];
        }

        // Public member access.
        Vector2<T> center;
        std::array<Vector2<T>, 2> axis;

    public:
        // Comparisons to support sorted containers.
        bool operator==(Parallelogram2 const& other) const
        {
            return center == other.center && axis == other.axis;
        }

        bool operator!=(Parallelogram2 const& other) const
        {
            return !operator==(other);
        }

        bool operator< (Parallelogram2 const& other) const
        {
            if (center < other.center)
            {
                return true;
            }

            if (center > other.center)
```

```
            {
                return false;
            }

            if (axis < other.axis)
            {
                return true;
            }

            if (axis > other.axis)
            {
                return false;
            }

            return false;
        }

        bool operator<=(Parallelogram2 const& other) const
        {
            return !other.operator<(*this);
        }

        bool operator> (Parallelogram2 const& other) const
        {
            return other.operator<(*this);
        }

        bool operator>=(Parallelogram2 const& other) const
        {
            return !operator<(other);
        }
    };
}
```

Listing 3 shows the Geometric Tools code for the point-parallelogram distance and closest point query.

Listing 3. The Geometric Tools C++ code for the point-parallelogram distance and closest point query. The unit tests provided 100% code coverage. The query outputs were compared to those of Mathematica [1] using NMinimize for the squared-distance function between a point and parallelogram points. The 4 linear inequality constraints are the algebraic conditions for a point to be inside the parallelogram.

```
#include <Mathematics/DCPQuery.h>
#include <Mathematics/Parallelogram2.h>

namespace gte
{
    template <typename T>
    class DCPQuery<T, Vector2<T>, Parallelogram2<T>>
    {
    public:
        struct Result
        {
            Result()
                :
                distance(static_cast<T>(0)),
                sqrDistance(static_cast<T>(0)),
                closest{ Vector2<T>::Zero(), Vector2<T>::Zero() }
            {
            }

            // The point closest[0] is the query point. The point closest[1]
            // is the parallelogram point closest to the query point. The
            // two points are the same when the query point is contained by
            // the parallelogram.
            T distance, sqrDistance;
```

```
    std::array<Vector2<T>, 2> closest;
};

Result operator()(Vector2<T> const& point, Parallelogram2<T> const& pgm)
{
    Result result{};

    // For a parallelogram point X, let Y = {Dot(V0,X-C),Dot(V1,X-C)}.
    // Compute the quadratic function q(Y) = (Y-Z)^T * A * (Y-Z) / 2
    // where A = B^T * B is a symmetric matrix.
    T b00 = pgm.axis[0][0];
    T b10 = pgm.axis[0][1];
    T b01 = pgm.axis[1][0];
    T b11 = pgm.axis[1][1];
    T detB = b00 * b11 - b01 * b10;
    T a00 = b00 * b00 + b10 * b10;
    T a01 = b00 * b01 + b10 * b11;
    T a11 = b01 * b01 + b11 * b11;

    // Transform the query point to parallelogram coordinates,
    // Z = Inverse(B) * (P - C).
    Vector2<T> diff = point - pgm.center;
    Vector2<T> Z
    {
        (b11 * diff[0] - b01 * diff[1]) / detB,
        (b00 * diff[1] - b10 * diff[0]) / detB
    };

    // Determine the region containing Z. The point K is the closest
    // point to Z relative to the metric tensor A.
    T const negOne = static_cast<T>(-1);
    T const posOne = static_cast<T>(+1);
    T root{};
    Vector2<T> K{};

    if (Z[1] < negOne)
    {
        // Examine the bottom edge.
        root = Z[0] - a01 * (negOne - Z[1]) / a00;
        K[0] = Clamp(root, negOne, posOne);
        K[1] = negOne;

        if (Z[0] < negOne)
        {
            if (K[0] == negOne)
            {
                // Examine the left edge.
                root = Z[1] - a01 * (negOne - Z[0]) / a11;
                K[1] = Clamp(root, negOne, posOne);
                K[0] = negOne;
            }
        }
        else if (posOne < Z[0])
        {
            if (K[0] == posOne)
            {
                // Examine the right edge.
                root = Z[1] - a01 * (posOne - Z[0]) / a11;
                K[1] = Clamp(root, negOne, posOne);
                K[0] = posOne;
            }
        }
    }
    else if (Z[1] <= posOne)
    {
        if (Z[0] < negOne)
        {
            // Examine the left edge.
            root = Z[1] - a01 * (negOne - Z[0]) / a11;
            K[1] = Clamp(root, negOne, posOne);
            K[0] = negOne;
        }
```

```
            else if (Z[0] <= posOne)
            {
                // The query point is inside the parallelogram.
                K = Z;
            }
            else
            {
                // Examine the right edge.
                root = Z[1] - a01 * (posOne - Z[0]) / a11;
                K[1] = Clamp(root, negOne, posOne);
                K[0] = posOne;
            }
        }
        else
        {
            // Examine the top edge.
            root = Z[0] - a01 * (posOne - Z[1]) / a00;
            K[0] = Clamp(root, negOne, posOne);
            K[1] = posOne;

            if (Z[0] < negOne)
            {
                if (K[0] == negOne)
                {
                    // Examine the left edge.
                    root = Z[1] - a01 * (negOne - Z[0]) / a11;
                    K[1] = Clamp(root, negOne, posOne);
                    K[0] = negOne;
                }
            }
            else if (posOne < Z[0])
            {
                if (K[0] == posOne)
                {
                    // Examine the right edge.
                    root = Z[1] - a01 * (posOne - Z[0]) / a11;
                    K[1] = Clamp(root, negOne, posOne);
                    K[0] = posOne;
                }
            }
        }

        result.closest[0] = point;
        result.closest[1] = pgm.center + K[0] * pgm.axis[0] + K[1] * pgm.axis[1];
        diff = result.closest[0] - result.closest[1];
        result.sqrDistance = diff[0] * diff[0] + diff[1] * diff[1];
        result.distance = std::sqrt(result.sqrDistance);
        return result;
    }

private:
    inline static T Clamp(T const& u, T const& umin, T const& umax)
    {
        return (u <= umin ? umin : (u >= umax ? umax : u));
    }
};
}
```

The Intel® VTune™Profiler 2024.3 showed that the bottleneck is the std::sqrt function call, Executing a query 2^{28} times used 7898 milliseconds on an Intel® Core™i9-10900 CPU at 2.80GHz. The square root call used 827 milliseconds (approximately 10.5% of the time).

References

[1] Wolfram Research, Inc. *Mathematica 14.1.0*. Wolfram Research, Inc., Champaign, Illinois, 2024.