Clip a Convex Polygon by a Hyperplane

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 http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Created: February 13, 2019

Contents

1 Introduction

This is a brief document about clipping a convex polygon by a line (2-dimensional) or a plane (3-dimensional) or generally by a hyperplane (n-dimensional). The theoretical framework is described in a straightforward manner, but when computing with floating-point arithmetic, care must be taken to provide a robust solution.

2 The Clipping Algorithm

The hyperplane is defined by $\mathbf{W} \cdot \mathbf{X} = c$, where $\mathbf{W}$ is a hyperplane normal, c is the corresponding hyperplane constant and $\mathbf{X}$ is any point on the hyperplane. The convex polygon has ordered vertices $\{\mathbf{Y}_i\}_{i=0}^{m}$. The clipping algorithm is based on computing the signed distances $h_i = \mathbf{W} \cdot \mathbf{Y}_i - c$ from the $\mathbf{Y}_i$ to the hyperplane when normal vector is unit length. However, $\mathbf{W}$ is not required to be unit length, in which case the h_i are scaled signed distances.

Define the set $L = \{0, 1, \ldots, |L| - 1\}$ to be the set of indices of the convex polygon, where $|L|$ is the number of elements of L. The set L can be treated as a cyclic list by using modular arithmetic. Given an index i, the next index is $i \oplus 1 = (i + 1) \bmod |L|$ and the previous index is $i \ominus 1 = (i + |L| - 1) \bmod |L|$, where both $i \oplus 1$ and $i \ominus 1$ are in L. Any cyclic permutation of the elements of L is allowed for representing the set of indices.

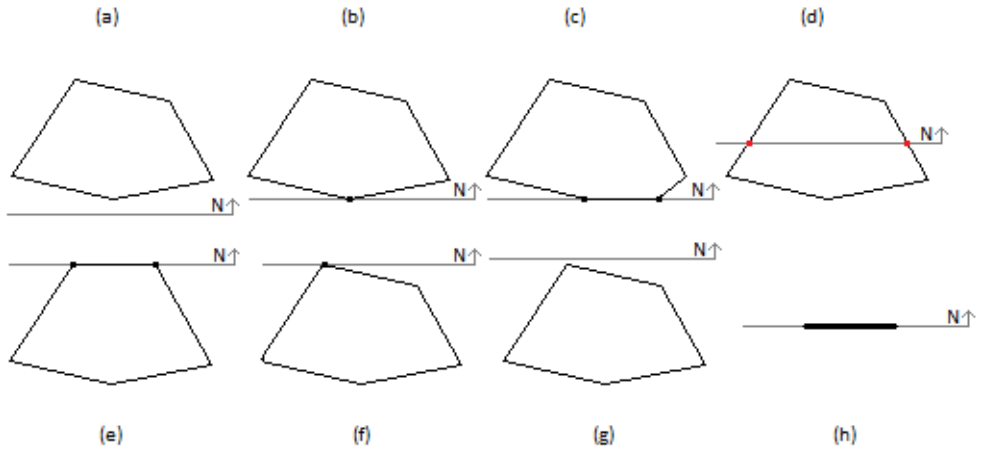
The h-values partition the list as $L = \{Z_0, P, Z_1, N\}$ or $L = \{Z\}$, where P is a subset of L of contiguous indices (possibly empty) for which the corresponding h-values are positive, N is a subset of L of contiguous indices (possibly empty) for which the corresponding h-values are negative and Z_0 and Z_1 are singleton subsets of L (possibly empty) where the h-values are zero. For dimensions $n \geq 3$, if the convex polygon is contained by the hyperplane, $L = \{Z\}$ where the use of Z stresses that all the h-values are zero. Table 1 provides a visual description of how the convex polygon interacts geometrically with the hyperplane.

Table 1. The visual descriptions of how the convex polygon interacts geometrically with the hyperplane. The contained-by-hyperplane case occurs only for dimensions $n \geq 3$ because I assume the convex polygon is not degenerate to a line segment.

convex polygon relative to hyperplane	partitions of indices
strictly on positive side	$L = \{P\}$
on positive side with one vertex in hyperplane	$L = \{Z_0, P_0\}$
on positive side with one edge in hyperplane	$L = \{Z_0, P, Z_1\}$
split by hyperplane	L one of $\{P, N\}$, $\{Z_0, P, N\}$, $\{P, Z_1, N\}$ or $\{Z_0, P, Z_1, N\}$
on negative side with one edge in hyperplane	$L = \{Z_0, Z_1, N\}$
on negative side with one vertex in hyperplane	$L = \{Z_1, N\}$
strictly on negative side	$L = \{N\}$
contained by hyperplane	$L = \{Z\}$ (dimensions 3 or larger)

Figure 1 shows the configurations visually.

Figure 1. The various configurations for how the convex polygon interacts with the hyperplane. (a) strictly on positive side, (b) on positive side with one vertex in hyperplane, (c) on positive side with one edge in hyperplane, (d) split by hyperplane (edge intersections drawn in red), (e) on negative side with one edge in hyperplane, (f) on negative side with one vertex in hyperplane, (g) strictly on negative side, (h) contained by hyperplane [the plane is perpendicular to the page].



When using floating-point arithmetic, rounding errors can lead to a theoretically invalid configuration. For example, the vertices might be close enough to the plane that some of the signs of the h-values are inaccurate, say, some are positive, some are negative, and 3 or more are zero. It might even be that the ordered list of h-values alternate in sign frequently, leading to multiple subsets of positive, negative and zero values. In 3 dimensions, this is particularly possible when the convex polygon is nearly coincident with the plane. Moreover, consider when the convex polygon vertices are (by definition) on a plane, but numerical computations of the positions have rounding errors that lead to a set of points that—when treated as exact rational-valued points—are no longer coplanar.

Listing 1 provides a 3-dimensional example of a theoretically invalid configuration due to rounding errors. The plane contains the point $\mathbf{K} = (1/10, 2/10, 3/10)$ and is spanned by the unit-length vectors $\mathbf{U} = (1, 1, 2)/\sqrt{6}$ and $\mathbf{V} = (2, 0, -1)/\sqrt{5}$. A unit-length plane normal is $\mathbf{W} = \mathbf{U} \times \mathbf{V} = (-1, 5, -2)/\sqrt{30}$. A 7-sided convex polygon has vertices $\mathbf{Y}_i = \cos(\theta_i)\mathbf{U} + \sin(\theta_i)\mathbf{V}$ for $0 \le i < 7$. The angles are chosen randomly in $[0, 2\pi)$ and are increasing, $\theta_i < \theta_{i+1}$ for all i. Theoretically, the polygon is coincident with the plane.

Listing 1. An example of a theoretical invalid configuration due to rounding errors. The numbers in the comments are decimal representations that generate the actual float numbers during program execution.

```
int main()
{
    Vector3<float> U = { 1.0f / sqrt(6.0f), 1.0f / sqrt(6.0f), 2.0f / sqrt(6.0f) };
    Vector3<float> V = { 2.0f / sqrt(5.0f), 0.0f, -1.0f / sqrt(5.0f) };
    Vector3<float> W = { -1.0f / sqrt(30.0f), 5.0f / sqrt(30.0f), -2.0f / sqrt(30.0f) };
    // U = { 0.408248276, 0.408248276, 0.816496551 }
    // V = { 0.894427180, 0.000000000, -0.447213590 }
    // W = { -0.182574183, 0.912870884, -0.365148365 }
    // Rounding errors occur in computing the square roots and in the divisions.

    Vector3<float> K = { 0.1f, 0.2f, 0.3f };
```

```
    // K = { 0.100000001, 0.200000003, 0.300000012 }
    // 0.1, 0.2 and 0.3 do not have finite single-precision floating-point representations

    float c = Dot(W, K);
    // c = 0.0547722429

    size_t const numVertices = 7;
    std::vector<float> angle(numVertices);
    angle[0] = 0.789902925f;
    angle[1] = 0.842714906f;
    angle[2] = 5.06786919f;
    angle[3] = 5.19404840f;
    angle[4] = 5.63434601f;
    angle[5] = 5.68152094f;
    angle[6] = 6.02670002f;

    std::vector<Vector3<float>> polygon(numVertices);
    for (size_t i = 0; i < numVertices; ++i)
    {
        polygon[i] = K + cos(angle[i]) * U + sin(angle[i]) * V;
    }
    // polygon[0] = { 1.02267003f, 0.487371802f, 0.557094455f }
    // polygon[1] = { 1.03931153f, 0.471664190f, 0.509504735f }
    // polygon[2] = { -0.596420169f, 0.342086971f, 1.00342751f }
    // polygon[3] = { -0.503544688f, 0.389121175f, 1.07457530f }
    // polygon[4] = { -0.115181953f, 0.525286376f, 1.22080696f }
    // polygon[5] = { -0.0697017908f, 0.536557734f, 1.22624516f }
    // polygon[6] = { 0.267993093f, 0.594893515f, 1.20323718f }

    std::vector<float> h(numVertices);
    for (size_t i = 0; i < numVertices; ++i)
    {
        h[i] = Dot(W, polygon[i]) - c;
    }
    // h[0] = 1.49011612e-08
    // h[1] = 0.0
    // h[2] = -1.49011612e-08
    // h[3] = 1.49011612e-08
    // h[4] = -1.49011612e-08
    // h[5] = 1.49011612e-08
    // h[6] = 1.49011612e-08

    // Theoretically, the h-values should all be zero.  With floating-point
    // representation and rounding errors, the signs of the h-values are
    // {+, 0, -, +, -, +, +}, which can never happen when using (exact)
    // real-valued arithmetic.  The numerically computed h-values have too
    // many sign changes.

    return 0;
}
```

3 Test-Intersection Query

Ignoring the problems with floating-point rounding errors, a simple algorithm for testing whether the convex polygon intersects the hyperplane is based on computing the h-values and counting the signs. Listing 2 contains pseudocode.

Listing 2. Pseudocode for determining whether a convex polygon intersects a hyperplane without actually computing the set of intersection. The number of vertices is assumed to be 3 or larger.

```
// The configurations are listed in the order shown in Table 1.
enum Configuration
```

```
{
    POSITIVE_SIDE_STRICT,
    POSITIVE_SIDE_VERTEX,
    POSITIVE_SIDE_EDGE,
    SPLIT,
    NEGATIVE_SIDE_EDGE,
    NEGATIVE_SIDE_VERTEX,
    NEGATIVE_SIDE_STRICT,
    CONTAINS
};

// The return value of the query indicates whether or not there is an intersection
// and provides the geometric configuration of the convex polygon and hyperplane.
struct Result
{
    bool intersect;
    Configuration configuration;
};

// N is the dimension of the problem, typically 2 (the hyperplane is a line)
// or 3 (the hyperplane is a plane).
Result TestIntersection(int numVertices, Vector<N,Real> polygon[], Hyperplane<N,Real> hyperplane)
{
    Result result;

    // Determine on which side of the plane the vertices live.
    int numPositive = 0, numNegative = 0, numZero = 0;
    for (int i = 0; i < numVertices; ++i)
    {
        Real h = Dot(hyperplane.normal, polygon[i]) - hyperplane.constant;
        if (h > 0)
        {
            ++numPositive;
        }
        else if (h < 0)
        {
            ++numNegative;
        }
        else
        {
            ++numZero;
        }
    }

    if (numPositive > 0)
    {
        if (numNegative > 0)
        {
            // The polygon and hyperplane intersect transversely.
            result.intersect = true;
            result.configuration = SPLIT;
        }
        else if (numZero == 0)
        {
            // The polygon is strictly on the positive side of the hyperplane.
            result.intersect = false;
            result.configuration = POSITIVE_SIDE_STRICT;
        }
        else if (numZero == 1)
        {
            // The polygon touches the hyperplane in a vertex.
            result.intersect = true;
            result.configuration = POSITIVE_SIDE_VERTEX;
        }
        else // numZero > 1
        {
            // The polygon touches the hyperplane in an edge.
            result.intersect = true;
            result.configuration = POSITIVE_SIDE_EDGE;
        }
    }
    else if (numNegative > 0)
```

```
    {
        if (numZero == 0)
        {
            // The polygon is strictly on the negative side of the hyperplane.
            result.intersect = false;
            result.configuration = NEGATIVE_SIDE_STRICT;
        }
        else if (numZero == 1)
        {
            // The polygon touches the hyperplane in a vertex.
            result.intersect = true;
            result.configuration = NEGATIVE_SIDE_VERTEX;
        }
        else  // numZero > 1
        {
            // The polygon touches the hyperplane in an edge.
            result.intersect = true;
            result.configuration = NEGATIVE_SIDE_EDGE;
        }
    }
    else  // numZero == numVertices
    {
        // The polygon is contained in the hyperplane.  This can happen for
        // a nondegenerate polygon only in dimensions n >= 3.
        result.intersect = true;
        result.configuration = CONTAINED;
    }

    return result;
}
```

If you need a solution more robust to floating-point rounding errors, use instead the find-intersection query as discussed in the next section because a rewrite of the test-intersection query would require effectively the same work as the find-intersection query.

4 Find-Intersection Query

The find-intersection query is designed to be robust to floating-point rounding errors. The algorithm is designed to partition the list of indices into $L = \{P', N'\}$, where $h_i \geq 0$ for $i \in P'$ and $h_j \leq 0$ for $j \in N'$. Each of P' and N', when not empty, contains a list of contiguous indices in the cyclic sense; for example, $\{2, 3, 0\}$ is a subset of $\{0, 1, 2, 3\}$ and has contiguous indices.

If all $h_i \geq 0$, then $L = P'$ and $N' = \emptyset$; if all $h_i \neq 0$, then $L = N'$ and $P' = \emptyset$; otherwise, there is at least one sign change in the h_i, which can occur only when the polygon is split by the hyperplane into two nondegenerate convex polygons and both P' and N' are not the empty set. The remainder of this section is constrained to the split-by-hyperplane case.

Identify those indices $p_{\max}$ and $n_{\max}$ for which $p_{\max} = \max\{i : h_i > 0\}$ and $n_{\max} = \max\{i : h_i < 0\}$. We are guaranteed that the aforementioned indices exist with $h_{p_{\max}} > 0 > h_{n_{\max}}$.

If $|h_{p_{\max}}| \geq |h_{n_{\max}}|$, choose P' to be the largest subset of contiguous indices of L that contains $p_{\max}$ and for which $h_i \geq 0$. Choose $N' = L \setminus P'$ to be the remaining subset of contiguous indices of L; necessarily $n_{\max} \in N'$. If $|h_{n_{\max}}| > |h_{p_{\max}}|$, choose N' to be the largest subset of contiguous indices of L that contains $n_{\max}$ and for which $h_i \leq 0$. Choose $P' = L \setminus N'$ to be the remaining subset of contiguous indices of L; necessarily $p_{\max} \in P'$.

To avoid having source code blocks with similar logic, if we find that $|h_{n_{\max}}| > |h_{p_{\max}}|$, we can replace the hyperplane normal $\mathbf{W}$ by $-\mathbf{W}$ and the hyperplane constant c by $-c$. The equations $\mathbf{W} \cdot \mathbf{X} = c$ and $(-\mathbf{W}) \cdot \mathbf{X} = (-c)$ define the same hyperplane, but by using the second equation we have converted the problem to one where $|h_{p_{\max}}| > |h_{n_{\max}}|$, because the h-values are all negated from their previous values when negating the hyperplane normal and constant.

The logic modification of the previous paragraph ensures that $|h_{p_{\max}}| \geq |h_{n_{\max}}|$, and we can proceed with subdividing the convex polygon into two convex polygons. One polygon is on the positive side of the hyperplane with possibly a vertex or edge contained by the hyperplane. The other polygon is on the negative side of the hyperplane with possibly a vertex or edge contained by the hyperplane. If both polygons have an edge contained by the hyperplane, then that edge is the same for both polygons.

Listing 3 contains pseudocode.

Listing 3. Pseudocode for determining the whether a convex polygon intersects a hyperplane. When the hyperplane splits the polygon into two nondegenerate polygons, the pseudocode also constructs those polygons. The number of vertices is assumed to be 3 or larger.

```
// The configurations are listed in the order shown in Table 1.
enum Configuration
{
    POSITIVE_SIDE_STRICT,
    POSITIVE_SIDE_VERTEX,
    POSITIVE_SIDE_EDGE,
    SPLIT,
    NEGATIVE_SIDE_EDGE,
    NEGATIVE_SIDE_VERTEX,
    NEGATIVE_SIDE_STRICT,
    CONTAINS
};

// The return value provides the geometric configuration of the convex polygon and hyperplane.
// The intersection, if any, is reported and is either a vertex (one point), an edge (two
// points) or the input polygon when it is already contained in the hyperplane.
struct Result
{
    Configuration configuration;
    int numIntersections;
    Vector<N, Real> intersection[];

    // If 'configuration' is POSITIVE_* or SPLIT, this polygon is the
    // portion of the query input 'polygon' on the positive side of
    // the hyperplane with possibly a vertex or edge on the hyperplane.
    int numPositives;
    Vector<N, Real> positivePolygon[];

    // If 'configuration' is NEGATIVE_* or SPLIT, this polygon is the
    // portion of the query input 'polygon' on the negative side of
    // the hyperplane with possibly a vertex or edge on the hyperplane.
    int numNegatives;
    Vector<N, Real> negativePolygon[];
};

Result FindIntersection(int numVertices, Vector<N,Real> polygon[], Hyperplane<N,Real> hyperplane)
{
    Result result;

    size_t const numVertices = polygon.size();

    // Determine on which side of the hyperplane the vertices live.  The index maxPosIndex stores
    // the index of the vertex on the positive side of the hyperplane that is farthest from the
    // hyperplane.  The index maxNegIndex stores the index of the vertex on the negative side of
    // the hyperplane that is farthest from the hyperplane.
    Real h[numVertices];
```

```
int zeroHIndices[numVertices];
int numZeroHIndices = 0;
int numPositive = 0, numNegative = 0;
Real maxPosH = -infinity, maxNegH = +infinity;
int maxPosIndex = -1, maxNegIndex = -1;
for (int i = 0; i < numVertices; ++i)
{
    h[i] = Dot(hyperplane.normal, polygon[i]) - hyperplane.constant;
    if (h[i] > 0)
    {
        ++numPositive;
        if (h[i] > maxPosH)
        {
            maxPosH = h[i];
            maxPosIndex = i;
        }
    }
    else if (h[i] < 0)
    {
        ++numNegative;
        if (h[i] < maxNegH)
        {
            maxNegH = h[i];
            maxNegIndex = i;
        }
    }
    else
    {
        zeroHIndices[numZeroHIndices++] = i;
    }
}

if (numPositive > 0)
{
    if (numNegative > 0)
    {
        result.configuration = SPLIT;
        bool doSwap = (maxPosHeight < -maxNegHeight);
        if (doSwap)
        {
            for (int i = 0; i < numVertices; ++i)
            {
                h[i] = -h[i];
            }
            swap(maxPosIndex, maxNegIndex);
        }
        SplitPolygon(numVertices, polygon, h, maxPosIndex, result);
        if (doSwap)
        {
            swap(result.positivePolygon, result.negativePolygon);
        }
    }
    else
    {
        if (numZeroHIndices == 0)
        {
            result.configuration = POSITIVE_SIDE_STRICT;
            result.numIntersections = 0;
        }
        else if (numZeroHIndices == 1)
        {
            result.configuration = POSITIVE_SIDE_VERTEX;
            result.numIntersections = 1;
            result.intersection[0] = polygon[zeroHIndices[0]];
        }
        else // numZeroHIndices > 1
        {
            result.configuration = POSITIVE_SIDE_EDGE;
            result.numIntersections = 2;
            result.intersection[0] = polygon[zeroHeightIndices[0]];
            result.intersection[1] = polygon[zeroHeightIndices[1]];
        }
```

```
            result.numPositives = numVertices;
            result.positivePolygon = polygon;
        }
    }
    else if (numNegative > 0)
    {
        if (numZeroHIndices == 0)
        {
            result.configuration = NEGATIVE_SIDE_STRICT;
            result.numIntersections = 0;
        }
        else if (numZeroHIndices == 1)
        {
            result.configuration = NEGATIVE_SIDE_VERTEX;
            result.numIntersections = 1;
            result.intersection[0] = polygon[zeroHeightIndices[0]];
        }
        else // numZeroHIndices > 1
        {
            result.configuration = NEGATIVE_SIDE_EDGE;
            result.numIntersections = 2;
            result.intersection[0] = polygon[zeroHeightIndices[0]];
            result.intersection[1] = polygon[zeroHeightIndices[1]];
        }
        result.numNegatives = numVertices;
        result.negativePolygon = polygon;
    }
    else  // numZero == numVertices
    {
        result.configuration = CONTAINED;
        result.numIntersections = numVertices;
        result.intersection = polygon;
        result.numPositives = 0;
        result.numNegatives = 0;
    }

    return result;
}

void SplitPolygon(int numVertices, Vector<N, Real> polygon[], Real h[], int maxPosIndex, Result& result)
{
    // Find the largest contiguous subset of indices for which h[i] >= 0.
    list<Vector<N, Real>> positiveList;
    positiveList.InsertBack(polygon[maxPosIndex]);
    int end0 = maxPosIndex, end0prev = -1;
    for (int i = 0; i < numVertices; ++i)
    {
        end0prev = (end0 + numVertices - 1) MOD numVertices;
        if (h[end0prev] >= 0)
        {
            positiveList.InsertFront(polygon[end0prev]);
            end0 = end0prev;
        }
        else
        {
            break;
        }
    }

    int end1 = maxPosIndex, end1next = -1;
    for (int i = 0; i < numVertices; ++i)
    {
        end1next = (end1 + 1) MOD numVertices;
        if (h[end1next] >= 0)
        {
            positiveList.InsertBack(polygon[end1next]);
            end1 = end1next;
        }
        else
        {
            break;
        }
```

```
    }

    int index = end1next;
    list<Vector<N, Real>> negativeList;
    for (int i = 0; i < numVertices; ++i)
    {
        negativeList.InsertBack(polygon[index]);
        index = (index + 1) MOD numVertices;
        if (index == end0)
        {
            break;
        }
    }

    // Clip the polygon.
    result.numIntersections = 0;
    if (h[end0] > 0)
    {
        Real t = -h[end0prev] / (h[end0] - h[end0prev]);
        Real omt = 1 - t;
        Vector<N, Real> V = omt * polygon[end0prev] + t * polygon[end0];
        positiveList.InsertFront(V);
        negativeList.InsertBack(V);
        result.intersection[result.numIntersections++] = V;
    }
    else
    {
        negativeList.InsertBack(polygon[end0]);
        result.intersection[result.numIntersections++] = polygon[end0];
    }

    if (h[end1] > 0)
    {
        Real t = -h[end1next] / (h[end1] - h[end1next]);
        Real omt = 1 - t;
        Vector<N, Real> V = omt * polygon[end1next] + t * polygon[end1];
        positiveList.InsertBack(V);
        negativeList.InsertFront(V);
        result.intersection[result.numIntersections++] = V;
    }
    else
    {
        negativeList.InsertFront(polygon[end1]);
        result.intersection[result.numIntersections++] = polygon[end1];
    }

    result.numPositives = 0;
    for (each p in the ordered list positiveList) do
    {
        result.positivePolygon[result.numPositives++] = p;
    }

    result.numNegatives = 0;
    for (each p in the ordered list negativeList) do
    {
        result.negativePolygon[result.numNegatives++] = p;
    }
}
```

---

5 Implementation

An implementation of both the test-intersection query and the find-intersection query is provided in the GTEngine code GteIntrConvexPolygonHyperplane.h.