

Finding a Dividing Plane: The Algorithm

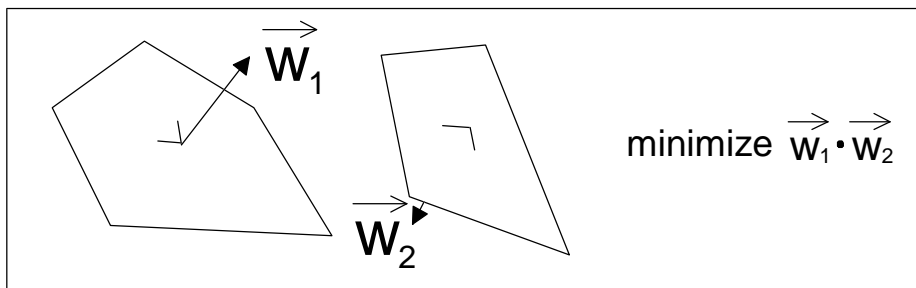
Introduction

The big leap to designing the code is to write the algorithm such that we can understand how all the pieces are put together. Since my last two documents regarding the algorithm, I've made some improvements and now have a clear idea on how to structure the algorithm. The algorithm requires many dot products and compares, with the advantage that it requires no division when compared to possible linear programming algorithms.

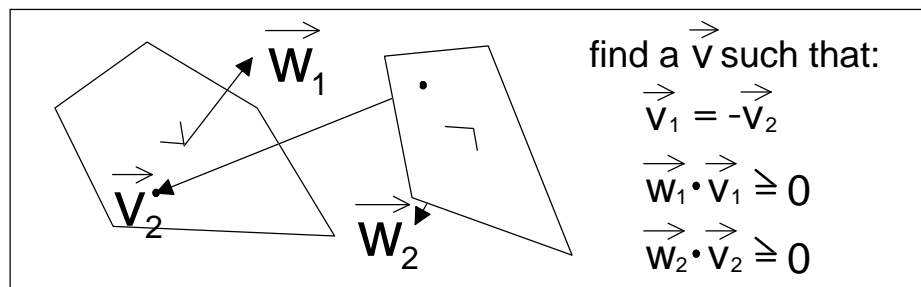
What's the goal?

The goal of the algorithm is to find out where the following conditions apply:

1. Dot product between normalized facet normal vectors is minimized.



2. There exists a vector between both facets which doesn't lie behind them. This can be found via sign of the dot product of a facet normal and the desired vector.



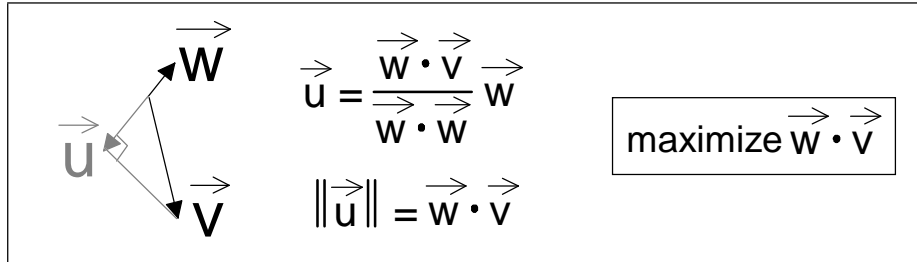
This defines a facet on both polyhedra which most closely point toward each other without one dividing the opposing polyhedron. This doesn't mean that one of the facets' normal vector is the dividing plane, but that one of the vertices of the facets may contain the dividing plane. This is a final case discussed later.

What are the steps?

First we need to think of a way to approach the above goal. My approach is to first satisfy one condition and then work on satisfying the other while at the same time keeping the other satisfied. I first satisfy condition two in the list above. Then I work on condition one.

Satisfying Condition Two (SCT)

A search to satisfy condition two is the same as maximizing the dot product between the vectors v and w . It should be noted that this is not a search to decrease the angle between the two vectors, but a search to decrease the length of the vector which is v projected on w . By decreasing this projection length, we know that the point is approaching one side of the plane.



In this stage of the algorithm, both polyhedra we will be moving around their facets trying to find the location where $\vec{w} \cdot \vec{v} \geq 0$ for the two polyhedra *Poly1* and *Poly2*.

We select a facet on *Poly1* to start on. One of the facet's vertices is chosen. If condition two is not met with this facet's vertex and a point on *Poly2*, then we search the neighboring facets to try and maximize $\vec{w}_n \cdot \vec{v}_1$. It should be noted that the same v vector is used for each new facet. This is because we are looking for a facet that faces the other point. If we changed v , then we'd be perturbing more than we really want to.

These steps are done iteratively, moving once among *Poly1*'s facets, and then moving once among *Poly2*'s facets, switching back and forth until condition two is met for both polygons. It needs to be done this way because we only take into account condition two from a single polyhedron's perspective. Moving to satisfy one perspective may change the other's condition.

Simultaneously Satisfying Conditions (SSC)

To find the final position, the idea is to improve condition one while keeping condition two satisfied. This is done by improving the condition by moving on the facets of one polyhedron until it can no longer be improved, then by switching over to the other polyhedron to do the same. The switching between polyhedra continues until both can no longer be improved.

The conditions to move to a new facet are as follows assuming we are moving along facets on *Poly1* and have one fixed facet on *Poly2*:

1. $\vec{w}_n \cdot \vec{w}_2 < \vec{w}_1 \cdot \vec{w}_2$ where 1 is the current facet on *Poly1* and n are the facets being considered.
2. $\exists! \vec{v}_{n,m} \mid \vec{w}_2 \cdot \vec{v}_{n,m} \leq 0$ where 2 is the current facet on *Poly2*, n are the facets being considered, and m are vertices around the given facet.
3. $\max(\vec{w}_n \cdot \vec{v}_1) \geq 0$ where 1 is the current facet on *Poly1* and n are the facets being considered.

The first condition ensures that we always make some type of progress towards our goal. The second condition ensures that the facet we choose is still partially in front of the facet on *Poly2*. The third condition tells us we want the chosen facet to contain the facet on *Poly2* as much as possible, as well as contain it.

Since all of these conditions must hold, it is wise to select them in order of least difficult to most difficult. For example, the second condition may need to visit each vertex of *Poly2*'s facet.

This can be very wasteful. In the third and first condition however, we can eliminate the need of checking for the second condition through simple dot products.

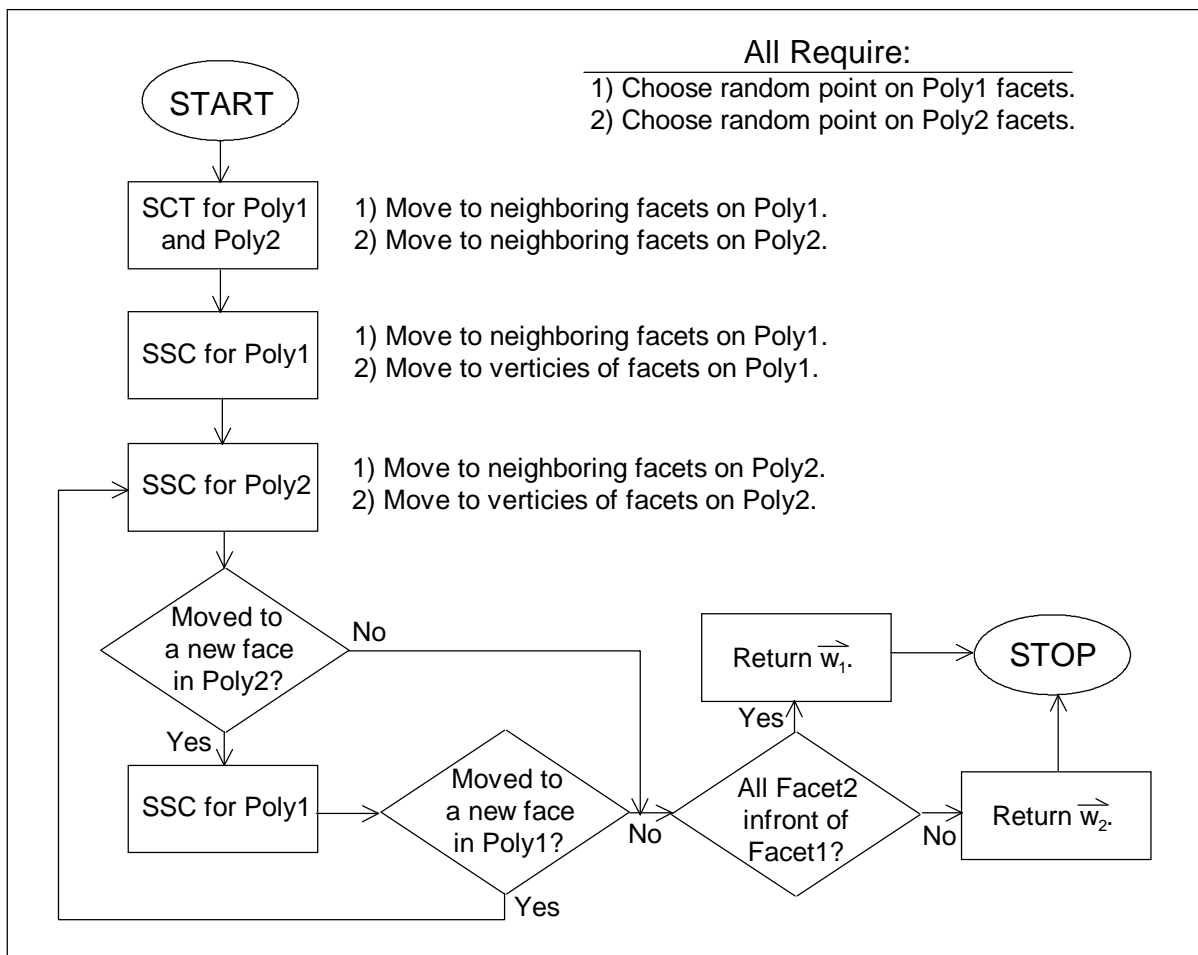
I suggest checking for each neighboring facet the first condition, and on success, checking for the third condition, and placing it in an ordered list. For each facet which makes it into the list, check for the second condition in a way that you can maximize $\vec{w}_n \cdot \vec{v}_1$ and continue on without needing to check further possibilities once you find a successful path.

What if a facet doesn't make up a dividing plane?

Is this possible? I can't visualize any possibility of this sort in my mind unless the polyhedra intersect each other. I would assume that you should just randomly choose a facet's normal vector at this point because the test is probably encountering math errors. Or you could choose the facet with the least amount of error.

The flowchart

To better illustrate how the *SCT* and *SSC* steps are put together, I've developed a flow chart shown below. Notice the comments on the side which indicate what values are used in each search. This will be important when you think about making an optimal datastructure to hold the polyhedra in.



The datastructures

Examination of the flowchart shows that for each polyhedron, there is motion about the facets neighboring facets and points on a facet. This means the connectivity requirements of the datastructure can be built around a facet (edges and vertices do not need to have anything concerning connectivity).

An example of the data needed in a facet would be:

- vertices *of type point*
- neighboring facets *of type facet*
- normalized vector perpendicular to the facet *of type vector*

Some thoughts on how to optimize the way data is stored would be to create conditions on the order of the stored vertices and facets. By storing them clockwise or counterclockwise, if the tests are done in a clockwise or counterclockwise fashion, it can be determined with further tests in a particular direction will lead to failures. This information can be used to reduce the number of extraneous operations.

Possible improvements

Assuming we only consider points on the facets of the polyhedra facing each other, it is possible to produce a search that finds the two closest vertices between both polyhedra. We must only consider those facets facing the opposing polyhedra because if they didn't, it is possible that the distance between neighboring vertices wouldn't be decreasing, but would instead be increasing. It's possible that the two closest vertices could be found assuming that the desired facets would be nearby. The square of the distance between two points is only a single dot product which *may* end up using fewer multiplies.