2. Computation of intersection of polytopes pairs.

After the first part, suppose that so far we have got the Minkowski sums, $\{P+Q|\forall P,Q\in\mathcal{P}\}$, of a group of polytopes $\mathcal{P}$. Since each Minkowski sum can also be viewed as a polytope, in turn, the problem can be seen as computation of intersection of polytopes pairs.

Analysis

It is quite fair and straightforward to compute all intersections that is made by a polytope from Minkowski sums set $\{P+Q|\forall P,Q\in\mathcal{P}\}$, and a polytope from polytopes set $\mathcal{P}$, but it is not generally observed. An general observation is that most polytope pairs are non-intersected, which means it is unnecessary to compute their intersections since their intersections are empty set. (In fact, we can give a rough statistical estimation here. But since it is not the key point and a little sophisticated, I will leave a space to be filled in the future.) It would be better if there is a way to filter most non-intersected polytope pairs as preprocessing to avoid operations of computing of their intersections.

In fact, this filter can be achieved by using a family of linear projecting functions and a data structure called [AABB tree (https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/)](https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection/), Axis-Aligned Bounding Box Trees. Note that a linear projecting function refers to the functions have the following form:

$f(x)=v\cdot x$, where v is a fixed vector parameter of f and x is a vector variable.

AABB tree is designed for geometrical intersection detection of a large number of boxes, where boxes are Descartes products of k intervals, $[x_1,y_1]\times[x_2,y_2]...\times[x_k,y_k]$. It organizes all k dimensional boxes in a tree, where every leaf node is the real box that is waiting to be detect, every non-leaf node is the minimal box containing all boxes of its children. Besides, each leaf node also has a space to store its own information. Supposing that we have n boxes that have been organized by a AABB tree, and given a request for searching for a box, we can retrieve all the boxes that intersect with a target box in $k*log(n)$ time complexity. In fact, the efficiency of AABB tree comes from clustering of intersected boxes and preserving locality from k dimensional space to the tree data structure.

Despite of efficiency of AABB tree, it is hard to use it explicitly without adjusting for our problem since our polytopes are not rectangular. Even though a circumscribed box of a polytope could be a candidate to represent it in AABB tree, however, as the dimension k increases, the volumn proportion of the polytope in the circumscribed box is getting smaller and smaller. For example, if the polytope is a simplex, then the proportion is exactly $\frac{1}{k!}$, which is too weak to use it.

The following observation gives us a bright way to encode a polytope into a box.

Given two polytopes, P and Q, and a linear projecting function $f(x)$ mapping from $\mathbf{R}^k$ to $\mathbf{R}$:

If two polytopes P and Q have an intersection, then their projections, intervals $f(P)$ and $f(Q)$, must also have an intersection.

Based on the above observation, we can use the same d linear projecting functions, $\{f_i(x)|1\leq i\leq d\}$, to project each polytope P to produce a corresponding box B, which is exactly the Descartes products of d intervals, $f_1(P)\times f_i(P)...\times f_d(P)$. It is obvious that those boxes also maintain the intersection characteristic when their original polytopes have an intersection. Even though this method does not preserve the non-intersection characteristic, with a large enough d, it is believable that the non-intersection characteristic can also be proserved. (In fact, those $\{f_i(x)|1\leq i\leq d\}$ can be generated in self-adjustable or data-drived more than just random, but since it is not the key point here, we can fill it later). "Low-dimensional objects can be easily seperated by a hyperplane when they are projected into higher dimensions", an opinion from statistical machine learning may also be benefitial to understand this method.

We give a simple example to display the basic idea of those projections in the following.

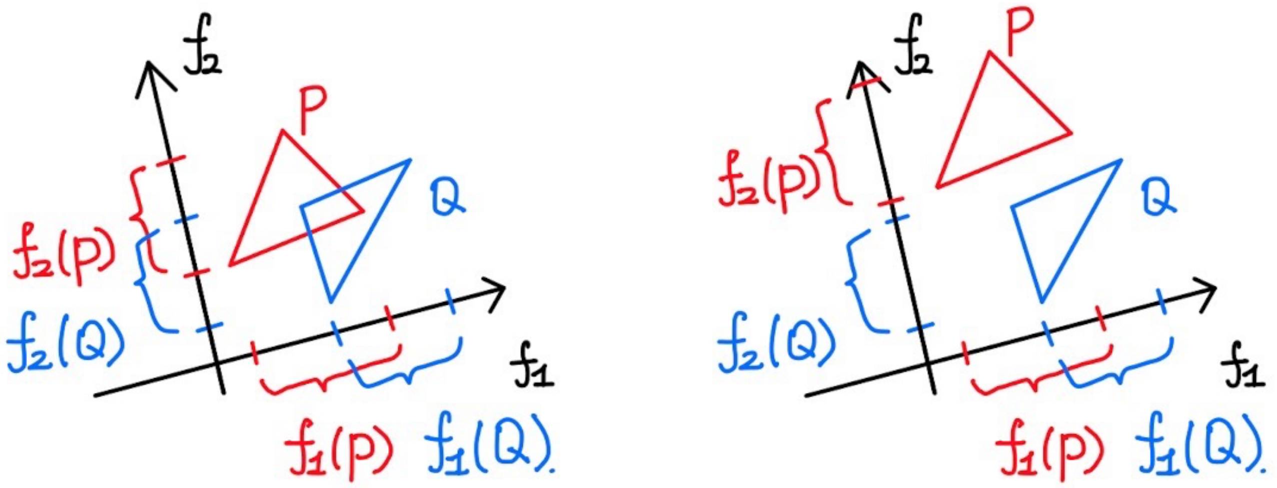

Fig 2. The left graph shows that if two polytopes P and Q have an intersection, then their projections must also have an intersection. The right graph shows that if two polytopes P and Q do not have an intersection, then their projections may not have an intersection.

In a summary, an AABB tree can be constructed by the boxes coming from a family of linear projecting functions inflicting on each polytope. And every time when there is a request of a target polytope for retrieving all polytopes that have an intersection with it, we can firstly encode the target polytope as a box by the same family of linear projecting functions, then we perform the search operations of the AABB tree and it return a set containing all boxes that have an intersection with this target box, which is a superset in terms of intersection of polytopes. And as the number of projecting function increases, the superset finally could converge to an exact set.

Procedures (Algorithm)

After the above analysis, we can divide the second part into 5 subparts,

1. Generate a family of d linear projecting functions $\{f_i(x) | 1 \leq i \leq d\}$,
2. Encode each polytope as a d dimensional box,
3. Construct of an AABB tree for all boxes of Minkowski sums,
4. Retrieve all Minkowski sums that possibly have an intersection for every target polytope in the AABB tree
5. Verify whether those Minkowski sums have an real intersection and then label the corresponding three tuple table to indicate whether they have an intersection or not.

Implementation

We define the following structures and functions to support.

1. List `self.project_function[i]` is to support a family of projecting functions $\{f_i(x) | 1 \leq i \leq d\}$, each element of it is a vector parameter of the corresponding projecting function.
2. Tree `self.minkowski_sums_AABB_tree()` is imported by AABBTree module.

3. Three tuple table `self.intersection[i][j][k]` is to store $(P_i + P_j) \cap P_k$, the value is `False` if it is an empty set, otherwise it will be the corresponding intersection wrapped by `polyhedron()`.
4. Function `boxing(p)` is used to generate a box of polyhedron `p` by using `self.project_function[i]`.

3. Additional materials: tranformation from detect intersection of cones to detect intersection of polytopes

After introducing how to use AABB tree to detect intersection of polytopes, let us go back and review the problem of intersection detection of cones. It is hard to explicitly use the techinques from intersection detection of polytopes to deal with intersection detection of cones, because those normal cones always share the same original point and do not have a finite volumn for projection. However, we can find a way to transform problem of intersection detection of cones to the problem of intersection detection of polytopes so that the time complexity can be reduced from $I * (kn)^2$ to $I * \Sigma_{i=1}^{kn} log(i) \leq I * kn * log(kn)$.

The problem can be settled down once we can find a kind of polytopes to be indicators for each polytope such that any polytope pair has an intersection if and only if their corresponding cones also have a intersection. In fact, given a cone, the polytope, whose vertexes come from the unit vectors of extreme rays of each cone, could be a good indicator for this cone. And due to lack of precision of square root when those extreme rays are unitized, it is better to use l_∞ norm, or l_1 norm, instead of l_2 norm.

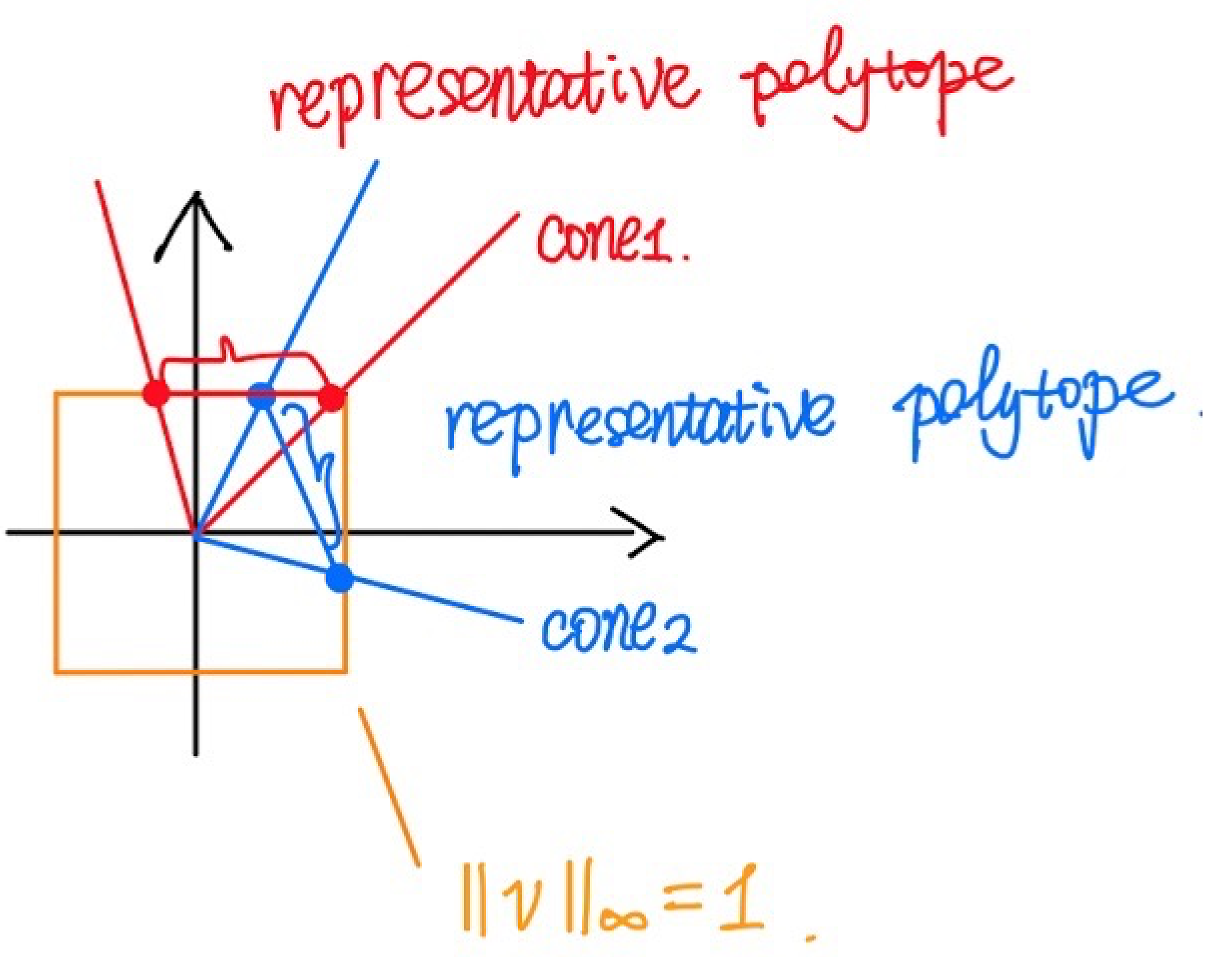

Fig 3. Suppose that 2 normal cones in red and blue seperately are shown in the graph. Unit vectors of extreme rays are shown in red and blue solid circles. Two representative polytopes are the red segment, whose endpoints are red solid circles, and the blue segment, whose endpoints are blue solid circles.

Above all, with the help of unit vectors of extreme rays of each cone, each cone can be transformed into a polytope as a representative so that the intersection of those polytopes will indicate that their corresponding cones also have an intersection. Next, we can deal with those polytopes by using previous mentioned method and therefore, AABB tree can help to reduce the time complexity.

Implementation

We define the following structures to support the above unit vectors.

`self.representatives_for_normal_cones` is to support representatives of normal cones. Inside, each element of it has the form of tuple `(i, v, representative)` , where `i` and `v` are the index and vertex of $N_{p_i}(v)$, `representative` is a polytope whose vertexes are the unit vectors of extreme rays of the normal cone $N_{p_i}(v)$.