How to use the GJK_EPA class

Andrea Casalino

December 16, 2019

1 Collision check for convex sets

The G.J.K. (Gilbert–Johnson–Keerthi) algorithm was designed to check whether two convex sets are or not in collision. An extension of this algorithm is also able, when the two sets don't collide, to compute the closest points. The euclidean distance between these two points can be assumed as the distance between the shapes.
EPA (Expanding Polytope Algorithm) was developed with the aim of computing the penetration depth of two convex sets. Clearly, the EPA algorithm can be applied only to the those pairs that overlaps. i.e. those pairs presenting a collision.
The GJK and the EPA algorithm are integrated into a single class: GJK_EPA. They are internally invoked at the proper time when a certain proximity query is called.

1.0.1 convex shapes

The generic convex shape $\mathcal{S}$, is a set of points for which it holds that:

$$\mathcal{S}\ is\ convex \quad \Rightarrow \quad \forall S_{A,B} \in \mathcal{S} \wedge \forall r \in [0,1] \tag{1}$$

$$\Rightarrow \quad S_A + r\cdot(S_B - S_A) = S(r) \in \mathcal{S} \tag{2}$$

i.e. the segment connecting the generic pair of points $S_A, S_B \in \mathcal{S}$, is entirely contained in $\mathcal{S}$. Examples of convex and non convex shapes are reported in Figure 1. The shapes addressed by GJK_EPA are the convex hulls (METTERE ref) of a set of 3d coordinates $\{S_1, \cdots, S_n\}$. Any internal point of such shapes can be obtained as a convex combination of the vertices $S_{1,\cdots,n}$:

$$S \in \mathcal{S} \Rightarrow \exists c_{1,\cdots,n} \geq 0 \ \ s.t.\ \ S = \sum_{i=1}^{n} S_i \cdot c_i \ \ \wedge \ \ \sum_{i=1}^{n} c_i = 1 \tag{3}$$

Generally speaking, the coordinates of a point in the 3D space assume a different values according to the frame adopted for describing the position that point, see Appendix A. Usually, the coordinates of points $S_{1,\cdots,n}$ are given in a certain local frame (L), whose motion during time is coherent with the motion of the object they represent. The positions of $S_{1,\cdots,n}$ w.r.t. the local frame don't change during time, while the absolute positions can be computed considering an homogeneous matrix $M_{0L} = \{R_{0L}, T_{0L}\}$ [1]:

$$M_{0L} = \begin{bmatrix} R_{0L} & T_{0L} \\ [0 \ \ 0 \ \ 0] & 1 \end{bmatrix} \tag{4}$$

M_{0L} describes the relative position and the orientation of the local frame w.r.t. a fixed absolute one (0). The absolute position of a point $S_{i\ (0)}$ can be computed as follows:

$$S_{i\ (0)} = R_{0L} S_{i\ (L)} + T_{0L} \tag{5}$$

where $S_{i\ (L)}$ denotes the coordinate values w.r.t. the local frame (L).
The $Support$ operator is much relevant for the GJK mechanism. Formally it is defined as follows [2]

$$S^*_D = Support\Big(\mathcal{S}, D\Big) \quad = \quad argmax_{S\in\mathcal{S}}\Big\{\langle S, D\rangle\Big\} \tag{6}$$

$$= \quad argmax_{S_{1,\cdots,n}}\Big\{\langle S_i, D\rangle\Big\} \tag{7}$$

i.e. the support of $\mathcal{S}$ in a certain direction D, is the furthest point $S^*_D \in \mathcal{S}$ in the direction D, see also Figure 2. The simplification done in equation (7) expresses the fact that the farthest point in any direction D must be one the vertices $S_{1,\cdots,n}$. When dealing with moving objects, the $Support$ computation can be done considering matrix M_{0L}, without explicitly compute the absolute coordinates of $S_{0,\cdots,n\ (0)}$. Indeed, the direction $D_{(0)}$, expressed in the absolute frame, can be firstly converted in the local one:

$$D_{(L)} = R^T_{0L} \cdot D_{(0)} \tag{8}$$

Then, the farthest point $S^*_{D\ (L)}$ is searched in the local frame (L) considering the coordinates $S_{1,\cdots,n\ (L)}$, applying equation (7). Finally, the farthest point coordinates in the absolute frame are computed:

$$S^*_{D\ (0)} = R_{0L} \cdot S^*_{D\ (L)} + T_{0L} \tag{9}$$

Figure 3 summarizes the above considerations.

[1] R_{0L} is a rotation matrix (see also Appendix B) describing the relative orientation while T_{0L} is the relative position of the origin.
[2] $\langle , \rangle$ stands for the dot product: $\langle a, b\rangle = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z$

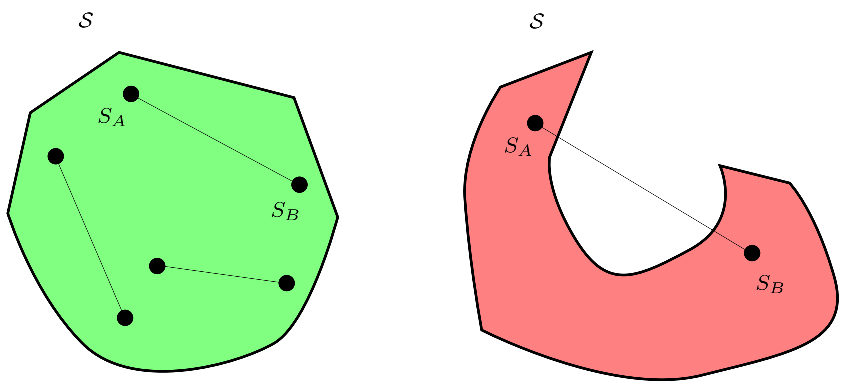


Figure 1: Examples of a convex shape (left) and a non convex one (right). The segment connecting any pair of points within the shape, it's entirely contained in the shape itself. This is not true for the non convex example on the right.

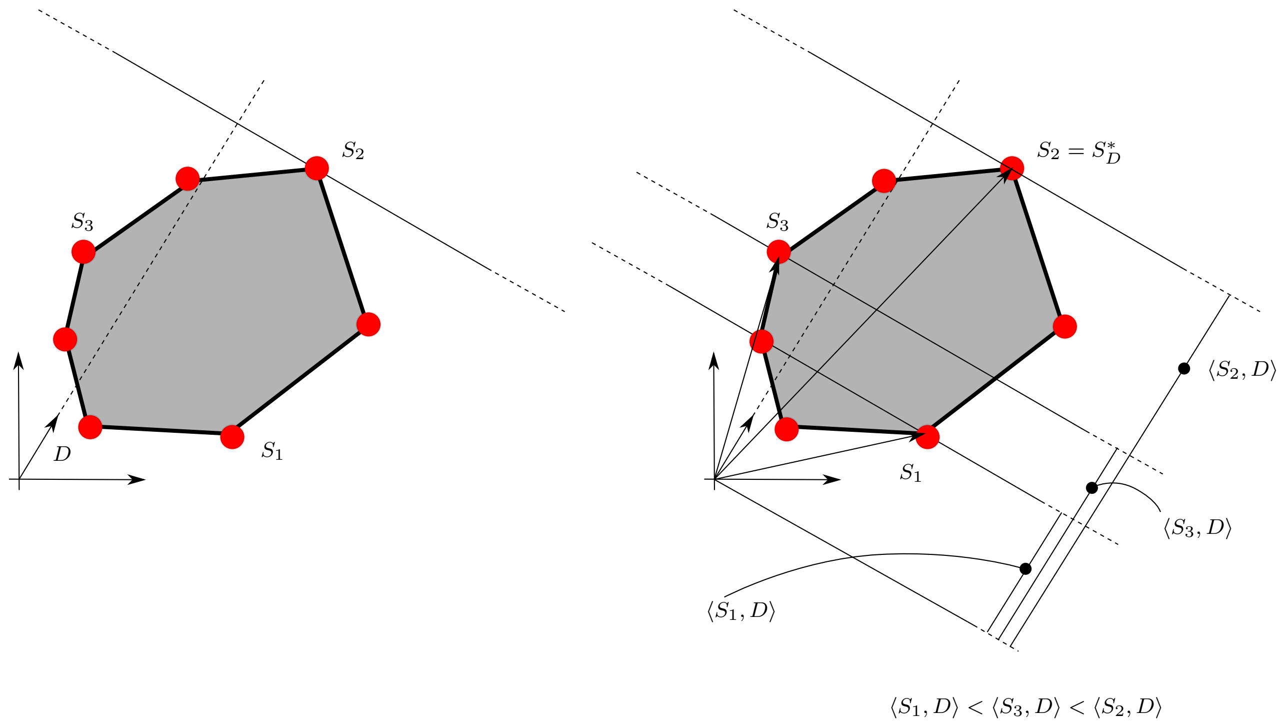


Figure 2: Example of support computation on a convex shape.

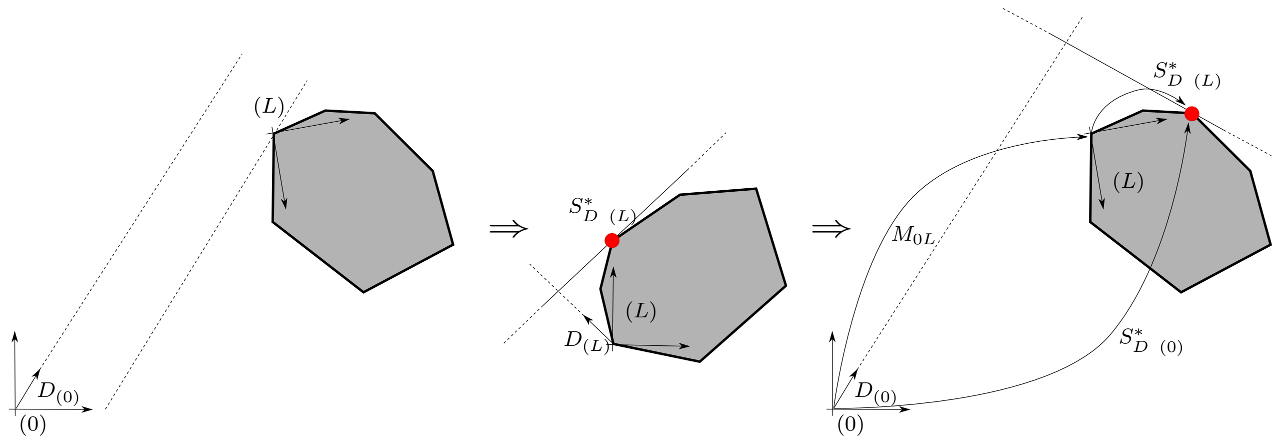


Figure 3: Steps involved in the computation of the furthest point of a shape whose position and orientation is described by an homogeneous matrix M_{0L}.

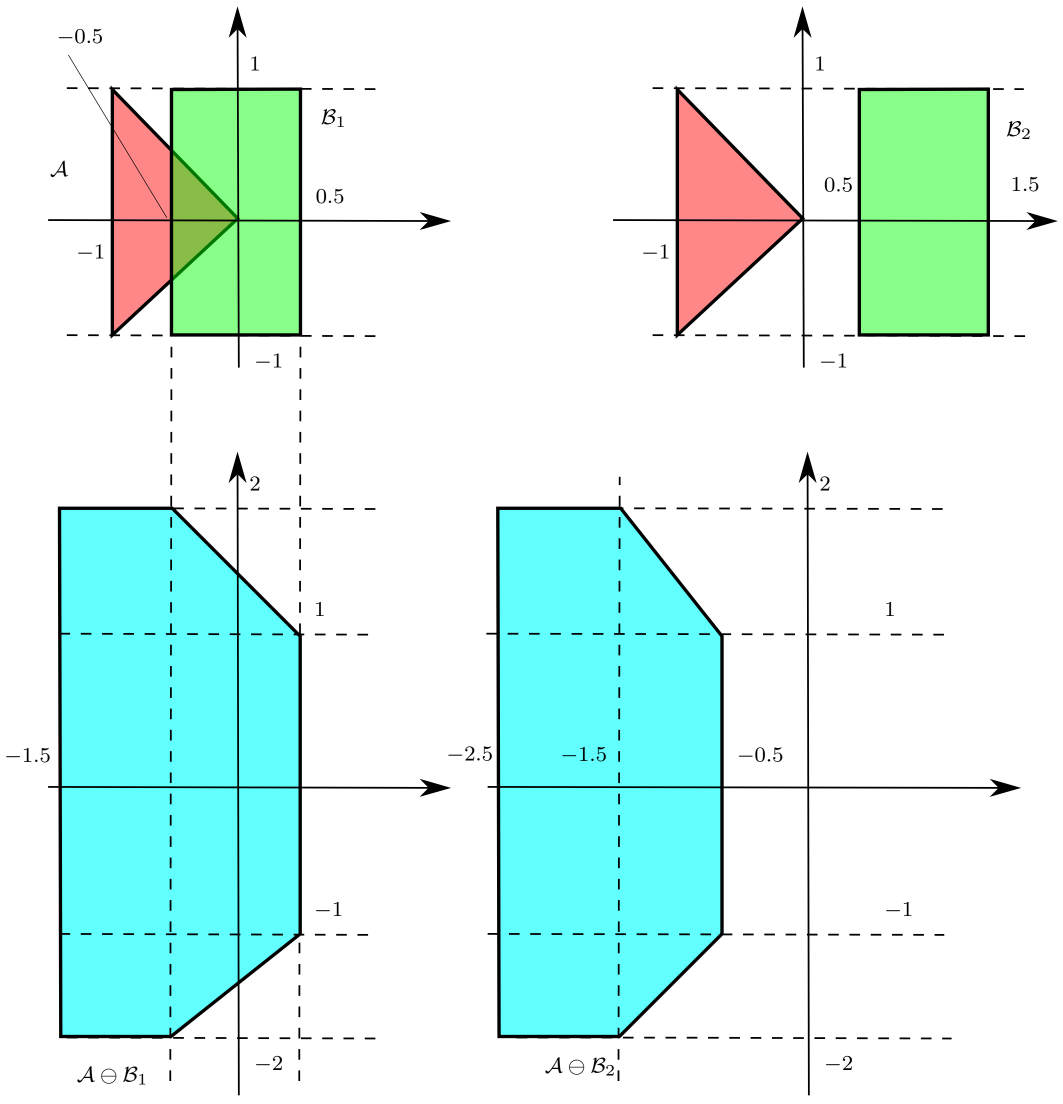


Figure 4: Example of Minkowski differences on the bottom, on the top the pairs for which the differences were computed. In case of overlapping pairs (case on the left), the Minkowski difference contains the origin.

1.0.2 Minkowski difference

Both the GJK and the EPA algorithm, exploit the properties of the Minkowski difference. The Minkowski difference of a pair of shapes $\mathcal{A}, \mathcal{B}$ (not necessary convex) is described as follows:

$$\mathcal{A} \ominus \mathcal{B} = \left\{ c \middle| \forall a \in \mathcal{A}, \forall b \in \mathcal{B} \Rightarrow c = a - b \right\} \tag{10}$$

In case $\mathcal{A}$, $\mathcal{B}$ are convex, the set represented by the Minkowski difference is also convex. Moreover, in case the two shapes are in collision, the Minkowski difference contains the origin, since there must be two points $a \in \mathcal{A}, b \in \mathcal{B}$ such that $a = b$. On the opposite, when $\mathcal{A}$ and $\mathcal{B}$ are disjointed, the distance of the Minkowski difference w.r.t. the origin is the distance of the two shapes. Figure 4 reports an example of Minkowski difference computations.
The $Support$ of the Minkowski difference can be computed as follows:

$$Support\Big(\mathcal{A} \ominus \mathcal{B}, D\Big) = Support\Big(\mathcal{A}, D\Big) - Support\Big(\mathcal{B}, -D\Big) \tag{11}$$

2 GJK algorithm

The aim of the GJK algorithm is to find the closest point of $\mathcal{M}$ (a politope representing the Minkowski difference of a pair of shapes) w.r.t. the origin, without explicitly compute all the vertices pertaining to $\mathcal{M}$. This can be done in an

iterative fashion, updating at each iteration a $Plex$ and a searching direction. The term $Plex$ refers to a compact set of points, which is essentially the convex hull of a small collection of vertices. We will sometimes refer to the $Plex$, considering its list of points $\{P_1, P_2, \cdots\}$. The list of vertices in the $Plex$ are temporally ordered: P_1 is the last point added to the plex, while the last point in the list is the oldest one. Basically, we avoid to entirely characterize $\mathcal{M}$, by reasoning at every step on a restricted sub set of $\mathcal{M}$, i.e. the $Plex$.
In order to explain the GJK mechanism, we initially suppose to reason on a Minkoski difference $\mathcal{M}$ not containing the origin (i.e. originated by two not colliding shapes). We can compute an initial point in $\mathcal{M}$ by taking the $Support$ in an initial direction D equal to $\hat{x}$, i.e. the versor of the x axis. Therefore, the initial $Plex$ is defined as follows:

$$Plex = \{P_1\} = \{Support(\mathcal{M}, \hat{x})\} \tag{12}$$

Since the aim is to get as much as possible closest to the origin, we consider a searching direction D equal to $-P_1$. After initializing $Plex$ and D, the searching loop can start. At every iteration, the $Plex$ is enriched with the $Support$ of $\mathcal{M}$ in the actual direction D, obtaining $Plex^*$. The closest point to the origin in $Plex^*$ is computed and then, according to which is the closest region in $Plex^*$, the $Plex$ and the searching direction D to consider for the subsequent iteration are updated. The updating process is repeated till it becomes evident that the closest region to the origin was reached. The pseudocode reported in Algorithm 1 summarizes the evolving steps. The updating task, taking into account $Plex^*$, considers 3 possible cases:

- case a: $Plex^* = \{P_1, P_2\}$, i.e. $Plex^*$ is a segment. In this case we can notice that every point P in the segment can be computed as the following combination:

$$\begin{aligned} P(s) &= P_1 + s \cdot (P_2 - P_1) \quad 0 \leq s \leq 1 \\ &= (1-s) \cdot P_1 + s \cdot P_2 \end{aligned} \tag{13}$$

  Finding the closest point can be cast in a constrained optimization problem. Let Δ be the squared distance to the origin of the generic point in the segment. Δ can be computed as follows:

$$\begin{aligned} \Delta(s) &= P(s)_x^2 + P(s)_y^2 + P(s)_z^2 \\ &= \langle P(s), P(s) \rangle \\ &= \langle P_1 + s \cdot (P_2 - P_1), P_1 + s \cdot (P_2 - P_1) \rangle \\ &= \langle P_1, P_1 \rangle + 2s \cdot \langle P_1, P_2 - P_1 \rangle + s^2 \cdot \langle P_2 - P_1, P_2 - P_1 \rangle \end{aligned} \tag{14}$$

  If we get rid of the constraint about s, we can compute the minimum of the above expression by taking the point for which the derivative of the above expression is equal to zero:

$$\begin{aligned} \frac{\partial \Delta}{\partial s} &= 0 \\ 2\big(\langle P_1, P_2 - P_1 \rangle + s \cdot \langle P_2 - P_1, P_2 - P_1 \rangle\big) &= 0 \\ s_{optimum} &= \frac{-\langle P_1, P_2 - P_1 \rangle}{\langle P_2 - P_1, P_2 - P_1 \rangle} \end{aligned} \tag{15}$$

  In the case $0 \leq s_{optimum} \leq 1$, $P(s_{optimum})$ is the closest point to the origin. Such point is contained in the segment representing the actual plex. Therefore, the closest region to the origin is the segment itself: the new $Plex$ to consider is equal to $Plex^*$ and the new searching direction is set equal to [3]:

$$D = \big(P_1 \wedge P_2\big) \wedge \big(P_2 - P_1\big) \tag{16}$$

  This is done in order to have a direction orthogonal to $P_2 - P_1$ and at the same time pointing towards the origin. On the opposite, when $s_{optimum} \leq 0$, the closest point is P_1 and the new plex to consider is simply $\{P_1\}$. The new searching direction is $-P_1$. Figure 5 summarizes the two presented cases. The case for which $s_{optimum} \geq 1$ is not possible, since as stated at the beginning, the points are temporally ordered. Therefore, P_2 was the closest point at the previous iteration and P_1 was found imposing a searching direction equal to $-P_2$. Therefore the closest point cannot be P_2.

- case b: $Plex^* = \{P_1, P_2, P_3\}$, i.e. $Plex^*$ is a triangular facet. Also for this case we can solve an optimization problem for computing the closest point. Every point P within the facet can be computed considering the following combination:

$$\begin{aligned} P(\alpha, \beta) &= P_1 + \alpha \cdot (P_2 - P_1) + \beta \cdot (P_3 - P_1) \\ s.t. & \quad 0 \leq \alpha \wedge 0 \leq \beta \wedge \alpha + \beta \leq 1 \\ &= P_1 + \alpha \cdot \delta_1 + \beta \cdot \delta_2 \end{aligned} \tag{17}$$

[3] The operator $\wedge$ indicates the cross product of the two vectors: $c = a \wedge b = \begin{bmatrix} c_x = a_y \cdot b_z - a_z \cdot b_y & c_y = a_z \cdot b_x - a_x \cdot b_z & c_z = a_x \cdot b_y - a_y \cdot b_x \end{bmatrix}$

As similarly done for case a, the closest point can be computed by taking the minimum of Δ:

$$\begin{aligned}\Delta(\alpha,\beta) &= \langle P(\alpha,\beta), P(\alpha,\beta)\rangle \\ &= \langle P_1+\alpha\cdot\delta_1+\beta\cdot\delta_2, P_1+\alpha\cdot\delta_1+\beta\cdot\delta_2\rangle \\ &= \langle P_1,P_1\rangle + 2\cdot\big(\alpha\cdot\langle\delta_1,P_1\rangle+\beta\cdot\langle\delta_2,P_1\rangle+\alpha\beta\cdot\langle\delta_1,\delta_2\rangle\big)+\alpha^2\langle\delta_1,\delta_1\rangle+\beta^2\langle\delta_2,\delta_2\rangle \qquad (18)\end{aligned}$$

The optimal values are taken imposing the gradient of Δ equal to 0:

$$\begin{aligned}\begin{bmatrix}\frac{\partial\Delta}{\partial\alpha}\\ \frac{\partial\Delta}{\partial\beta}\end{bmatrix} &= 2\begin{bmatrix}\langle\delta_1,P_1\rangle+\beta\cdot\langle\delta_1,\delta_2\rangle+\alpha\cdot\langle\delta_1,\delta_1\rangle\\ \langle\delta_2,P_1\rangle+\alpha\cdot\langle\delta_1,\delta_2\rangle+\beta\cdot\langle\delta_2,\delta_2\rangle\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix} \\ \begin{bmatrix}\langle\delta_1,\delta_1\rangle & \langle\delta_1,\delta_2\rangle\\ \langle\delta_1,\delta_2\rangle & \langle\delta_2,\delta_2\rangle\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} &= \begin{bmatrix}-\langle\delta_1,P_1\rangle\\ -\langle\delta_2,P_1\rangle\end{bmatrix} \\ M\begin{bmatrix}\alpha\\ \beta\end{bmatrix} &= V \\ \begin{bmatrix}\alpha_{optimum}\\ \beta_{optimum}\end{bmatrix} &= M^{-1}V \qquad (19)\end{aligned}$$

The above computations lead to obtain the closest point to the origin lying in the plane defined by the 3 points $P_{1,2,3}$. Only when is verified that $0\leq\alpha_{optimum}$, $0\leq\beta_{optimum}$ and $\alpha_{optimum}+\beta_{optimum}\leq 1$, the closest point belongs to $Plex^*$. In this circumstance, the plex to consider for the subsequent iteration is $Plex^*$ and the new searching direction D is assumed as the normal of the plane $P_{1,2,3}$, pointing towards the origin:

$$\hat{D} = \big(P_2-P_1\big)\wedge\big(P_3-P_1\big) \qquad (20)$$

$$if\ \ \langle\hat{D},P_1\rangle>0\Rightarrow D = -\hat{D} \qquad (21)$$

$$else\ \ D = \hat{D} \qquad (22)$$

When the closest point is not contained in the facet, the closest region can be: vertex P_1 (the last added to the plex), the segment $P_{1,2}$ or the segment $P_{1,3}$. The steps reported for case a are applied independently for $P_{1,2}$ and $P_{1,3}$ and the region having the $P(s_{optimum})$ closer to the origin is assumed as the closest region. Consequently the plex and D are updated, following the directions introduced in the previous case. For example if the closer region is $P_{1,3}$, point P_1 and P_3 are the only ones remaining in the plex and D is assumed equal to $\big(P_1\wedge P_3\big)\wedge\big(P_3-P_1\big)$. Figure 6 summarizes the updating procedure.

- case c: $Plex^*=\{P_1,P_2,P_3,P_4\}$, i.e. $Plex^*$ is a tetrahedron. Here it is important to remark that the facet $P_{2,3,4}$ cannot be the one closer to the origin, since the corresponding points are the oldest ones in the vertices: the searching direction that leads to obtain P_1 was computed considering that points. Therefore, the closest region must be one of the facets having P_1 as a vertex, i.e. the facets $P_{1,2,3}$, $P_{1,2,4}$ and $P_{1,3,4}$. The points $P_{1,2,3}(\alpha_{optimum},\beta_{optimum})$, $P_{1,2,4}(\alpha_{optimum},\beta_{optimum})$ and $P_{1,3,4}(\alpha_{optimum},\beta_{optimum})$, with the obvious meaning of notation, are computed as indicated for case b and the closest one to the origin is found. The corresponding closest region in the plex is identified and the plex is updated as similarly done for the previous cases. Visual examples are provided in Figure 7.

Figure 8 reports an example. Notice that it is not necessary to know the entire set of vertices pertaining to the Minkowski difference $\mathcal{M}$, since only the $Support$ computation is needed. Although the example is planar, the considerations must be generalized to the 3D case, considering the updating rules provided above.

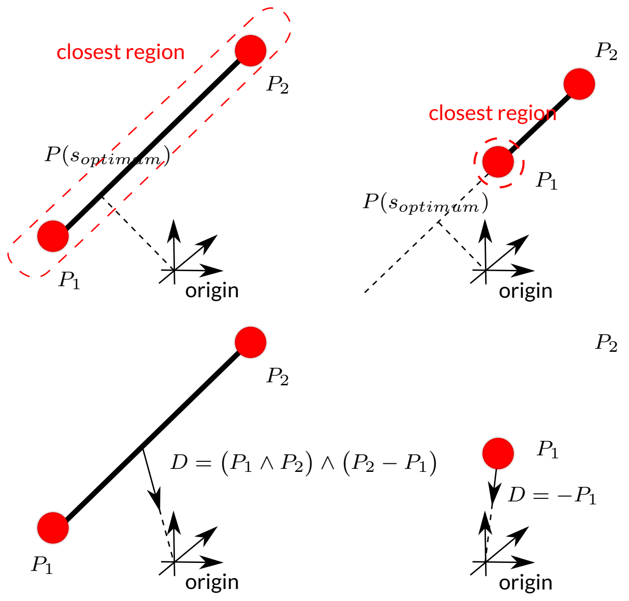


Figure 5: Examples of plex update. On the top, $Plex^*$, while on the bottom the updated $Plex$ and the new searching direction.

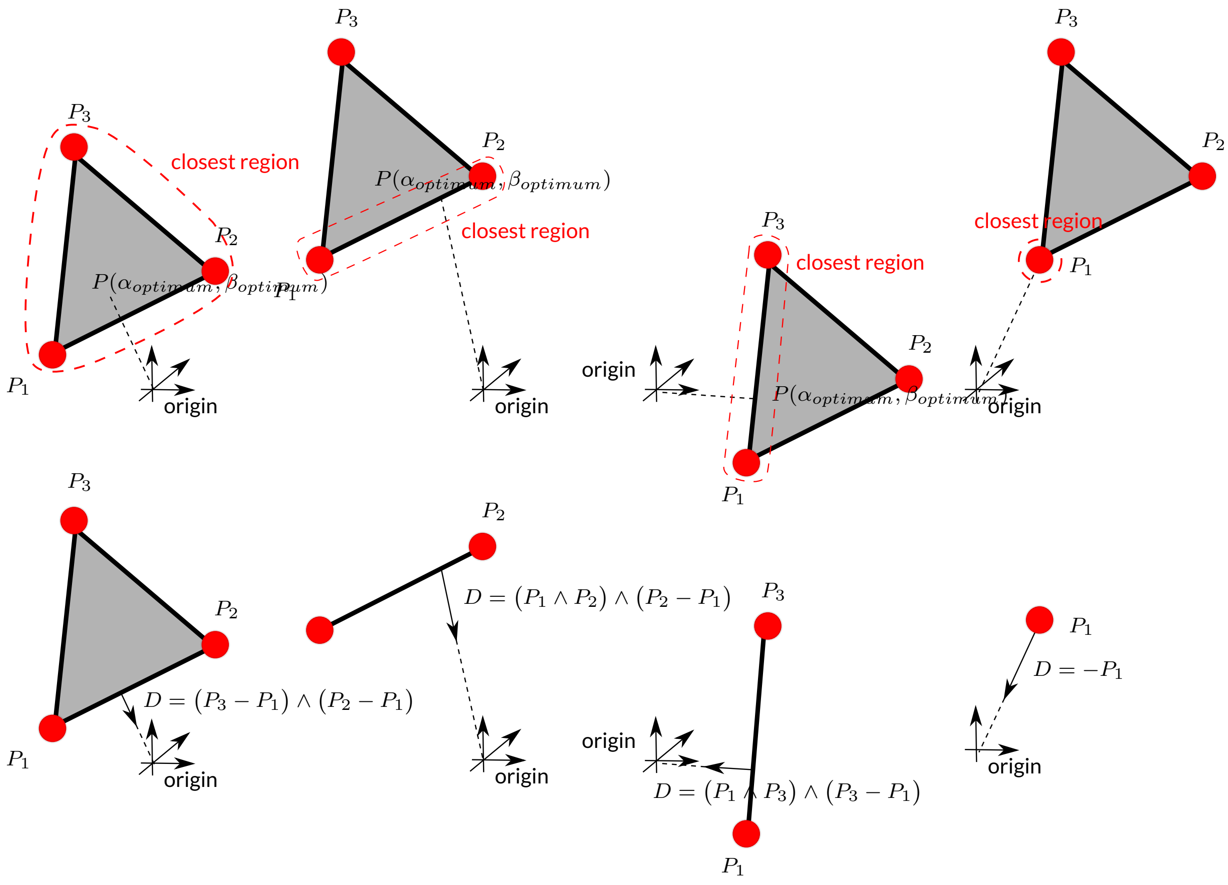

P_3
closest region
P_2
$P(\alpha_{optimum}, \beta_{optimum})$
P_1
origin
P_3
P_2
$P(\alpha_{optimum}, \beta_{optimum})$
P_1
closest region
origin
P_3
closest region
P_2
$P(\alpha_{optimum}, \beta_{optimum})$
P_1
origin
P_3
P_2
closest region
P_1
origin
P_3
P_2
P_1
$D = (P_3 - P_1) \wedge (P_2 - P_1)$
origin
P_2
$D = (P_1 \wedge P_2) \wedge (P_2 - P_1)$
origin
P_3
origin
$D = (P_1 \wedge P_3) \wedge (P_3 - P_1)$
P_1
P_1
$D = -P_1$
origin

Figure 6: Examples of plex update. On the top, $Plex^*$, while on the bottom the updated $Plex$ and the new searching direction.

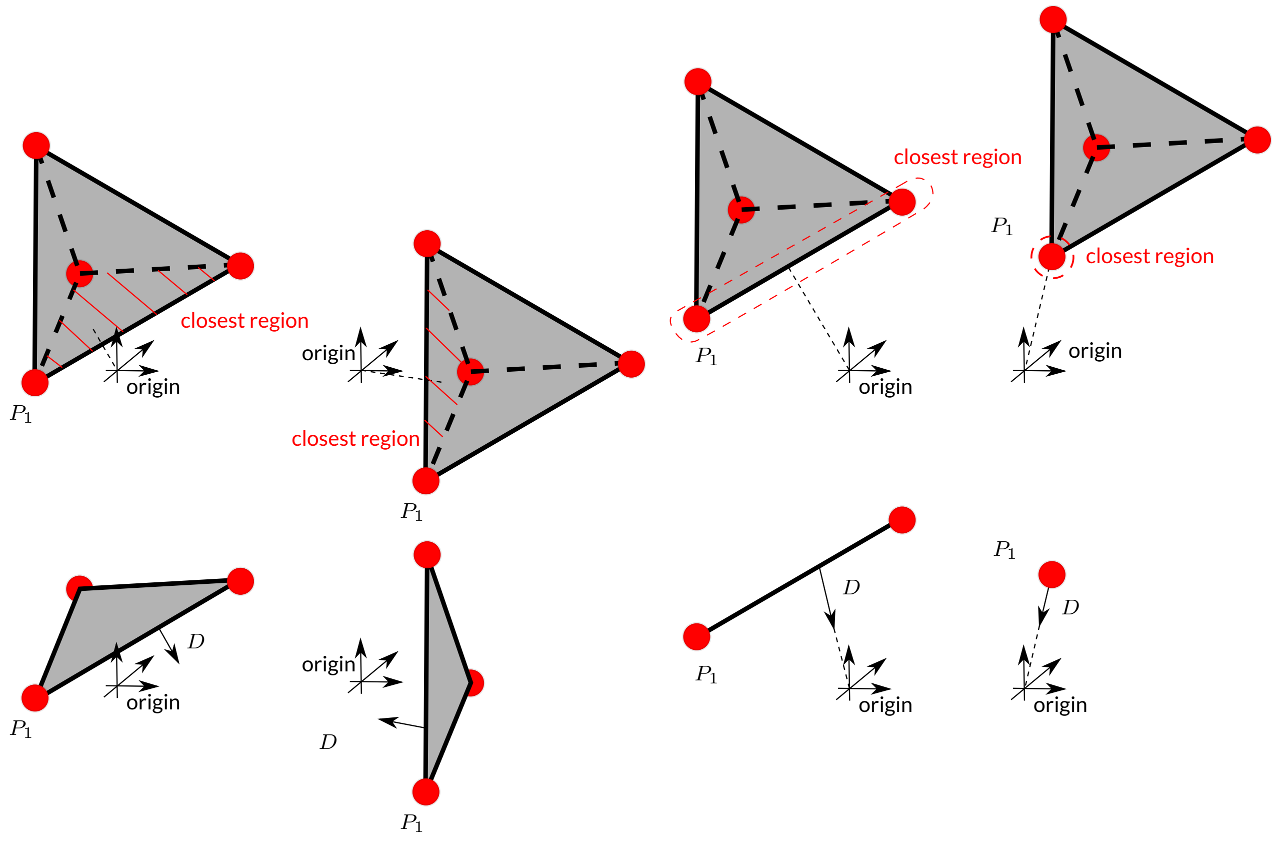


Figure 7: Examples of plex update. On the top, $Plex^*$, while on the bottom the updated $Plex$ and the new searching direction.

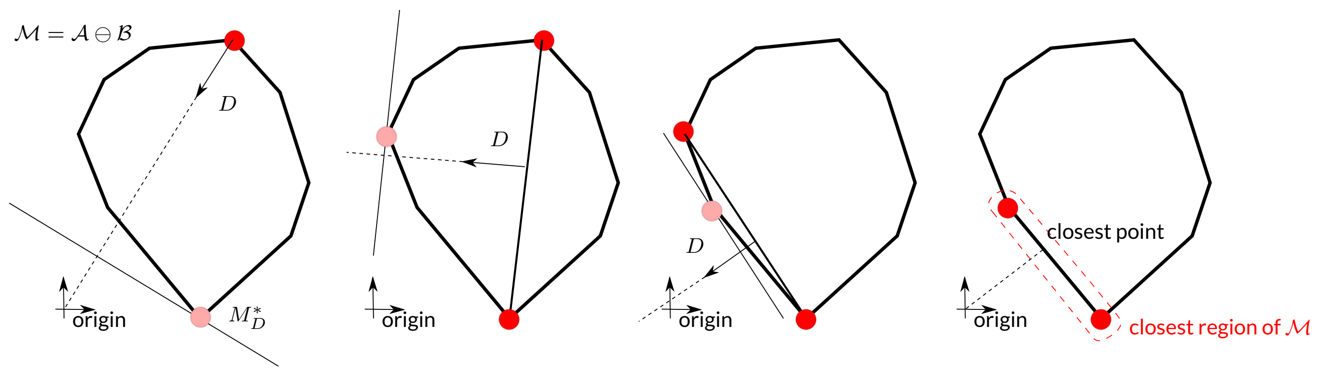

$\mathcal{M} = \mathcal{A} \ominus \mathcal{B}$
D
M_D^*
origin
closest point
closest region of $\mathcal{M}$

Figure 8: Examples of plex and searching direction updates: the search is terminated when the closest region of the Minkowski difference is found.

```
$P_1 = Support\Big(\mathcal{M}, \hat{x}\Big)$;
$Plex = \{P_1\}$;
$D = -P_1$;
$P^*_D = Support\Big(Plex, D\Big)$;
while $P^*_D \notin Plex$ do
    $P^*_D = Support\Big(Plex, D\Big)$;
    $Plex^* = P^*_D \cup Plex$;
    if $|Plex^*| = 2$ then
        update following the steps of case a;
    end
    else if $|Plex^*| = 3$ then
        update following the steps of case b;
    end
    else if $|Plex^*| = 4$ then
        update following the steps of case c;
    end
end
```

Algorithm 1: Computation of the closest region in the Minkowski difference $\mathcal{M}$.

2.1 Simple collision check

Sometimes, we are not interested in finding the closest region of the Minkowski difference to the origin, since we want just to know whether it contains or not the origin, i.e. check whether two objects are or not in collision. This can be done by slightly modify the pseudo code of Algorithm 1. Consider the first picture from the left of Figure 8: it is evident also from the initial update that the Minkowski difference depicted cannot contain the origin. Indeed, for the first $Support$ found, it is verified that $\langle D, M^*_D \rangle < 0$. This implies that $\mathcal{M}$ is a politope that is entirely above the plane having D as normal and passing from M^*_D. For this reason, if we are simply interested in performing a collision check, we can stop the search at the first iteration.
In the previous Section we assumed that the Minkowski difference did not contain the origin. Anyway, the updating process proposed can be still valid in the contrary case, arresting the search as soon we detect that $Plex$ contains the origin. Checking whether $Plex$ contains the origin is trivial for case a and b (since it happens when the closest point found has a null distance to the origin), while is a little bit more complex when dealing with case c. In the last case, we can notice that the tetrahedron represented by $Plex$ contains the origin only when the following is verified:

$$\langle P_1, n_1 \rangle \geq 0 \tag{23}$$
$$\langle P_1, n_2 \rangle \geq 0 \tag{24}$$
$$\langle P_1, n_3 \rangle \geq 0 \tag{25}$$

where $n_{1,2,3}$ are the outside normals of the facets $P_{1,2,3}$, $P_{1,2,4}$ and $P_{1,3,4}$ respectively, see also Figure 9.
Therefore, introducing the above additional checks, we obtain the GJK algorithm adopted for performing simple collision check. The corresponding pseudocode is reported by Algorithm 2.

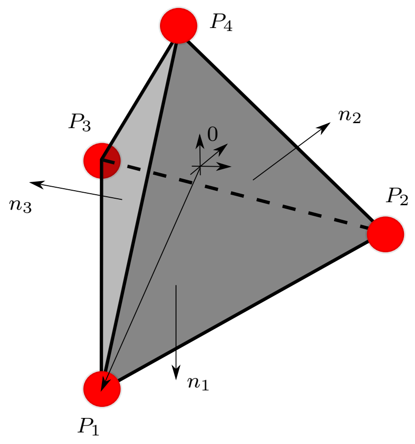


Figure 9: Example of a tetrahedron containing the origin.

```
$P_1 = Support\Big(\mathcal{M}, \hat{x}\Big)$;
$Plex = \{P_1\}$;
$D = -P_1$;
$P_D^* = Support\Big(Plex, D\Big)$;
**while** $P_D^* \notin Plex$ **do**
    $P_D^* = Support\Big(Plex, D\Big)$;
    **if** $\langle D, P_D^* \rangle < 0$ **then**
        **return** collision absent;
    **end**
    $Plex^* = P_D^* \cup Plex$;
    **if** $0 \in Plex^*$ **then**
        **return** collision detected;
    **end**
    **if** $|Plex^*| = 2$ **then**
        update following the steps of case a;
    **end**
    **else if** $|Plex^*| = 3$ **then**
        update following the steps of case b;
    **end**
    **else if** $|Plex^*| = 4$ **then**
        update following the steps of case c;
    **end**
**end**
```

Algorithm 2: GJK steps for performing a simple collision check.

2.2 Absence of a collision: get the closest points

In case a collision between two shapes $\mathcal{A}$ and $\mathcal{B}$ is not present, Algorithm 1 can be applied to obtain the closest $Plex$ to the origin. Then, the distance among the shapes can be assumed equal to the distance of the $Plex$ to the origin. Moreover, the closest points in the shapes generating the point in the Minkowski difference closer to the origin can be computed. Indeed, the closest point in the $Plex$ to the origin can be obtained as a combination in the form (equation METTERE and METTERE):

$$P_{closest} = \sum_{i=1}^{|Plex|} c_i P_i \tag{26}$$

Every vertex $P_i \in Plex$ was obtained as a difference $A_i - B_i$, i.e. the $Supports$ in the two shapes to check, equation (11). Therefore, the coefficients in the above equation can be used for computing the points in $\mathcal{A}$ and $\mathcal{B}$ whose

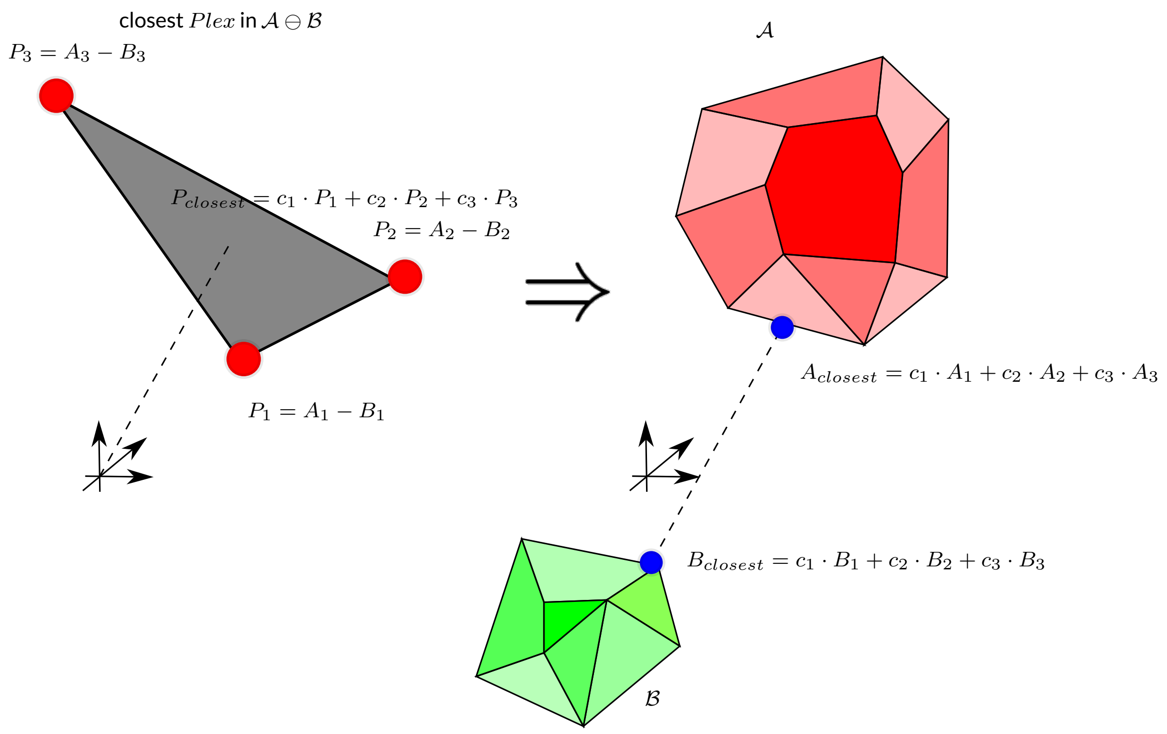

closest $Plex$ in $\mathcal{A} \ominus \mathcal{B}$
$P_3 = A_3 - B_3$
$P_{closest} = c_1 \cdot P_1 + c_2 \cdot P_2 + c_3 \cdot P_3$
$P_2 = A_2 - B_2$
$P_1 = A_1 - B_1$
$\mathcal{A}$
$A_{closest} = c_1 \cdot A_1 + c_2 \cdot A_2 + c_3 \cdot A_3$
$B_{closest} = c_1 \cdot B_1 + c_2 \cdot B_2 + c_3 \cdot B_3$
$\mathcal{B}$

Figure 10: Example of closest points computation, for non colliding shapes.

difference leads to obtain the closest point to the origin of the Minkowski difference:

$$A_{closest} = \sum_{i=1}^{|Plex|} c_i A_i \tag{27}$$

$$B_{closest} = \sum_{i=1}^{|Plex|} c_i B_i \tag{28}$$

$A_{closest}$ and $B_{closest}$ are the closest pair of points in the two shapes. Clearly $\|A_{closest} - B_{closest}\|$ is equal to the distance of between $\mathcal{A}$ and $\mathcal{B}$. Figure 10 reports an example.

3 Collision detected: get the penetration depth

In case of an overlapping pair $\mathcal{A}$,$\mathcal{B}$, we could be interested in determining the penetration vector ρ. Formally ρ is defined in this way:

$$\rho = argmin_{\rho} \left\{ \|\rho\|_2^2 \middle| \mathcal{B}^* = \{b^* = b + \rho \quad \forall b \in \mathcal{B}\} \Rightarrow 0 \notin \mathcal{A} \ominus \mathcal{B} \right\} \tag{29}$$

i.e. ρ is the minimum length translation to apply for $\mathcal{B}$ to realize that $\mathcal{A}$ and $\mathcal{B}$ are not anymore in collision. ρ can be computed as the following difference:

$$\rho = \rho_A - \rho_B \tag{30}$$

where $\rho_A \in \mathcal{A}$ and $\rho_B \in \mathcal{B}$ can be obtained by using the EPA algorithm. Refer also to Figure 11.
Indeed, ρ is a vector pointing to the closest point to the origin in the surface of $\mathcal{M} = \mathcal{A} \ominus \mathcal{B}$. In order to get such a point, an algorithm specular to the GJK can be adopted: the EPA algorithm. Also in this case, the entire shape of the Minkowski difference will not be explicitly computed. The algorithm starts by building an initial tetrahedron contained in $\mathcal{M}$[4]. This initial politope is constantly inflated in order to find ρ. More precisely, at each iteration the closest facet to the origin is identified and the $Support$ in the direction of the outgoing normal of that facet is searched. In

[4]The last $Plex$ obtained by the initial GJK iterations is considered as a starting point. In case it is not a tetrahedron, points are added, finding the $Support$ in proper directions.

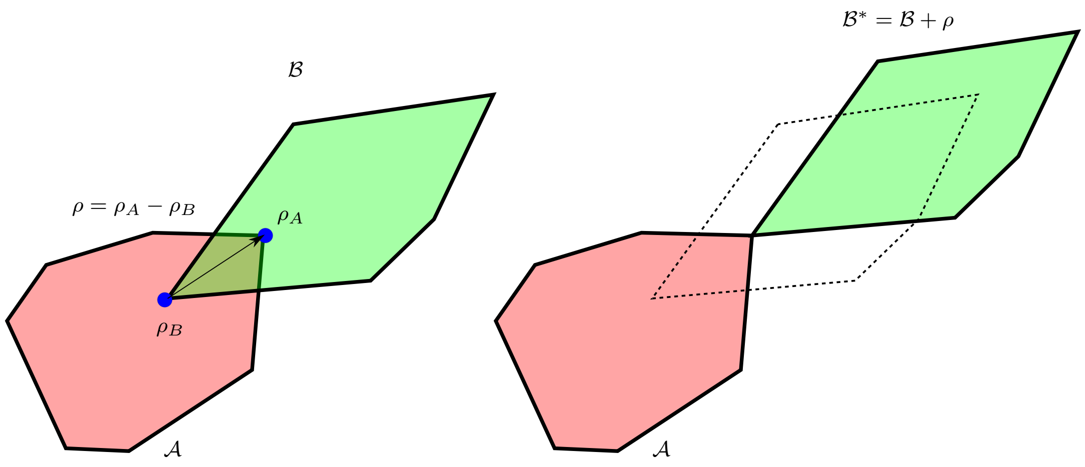


Figure 11: Penetration vector.

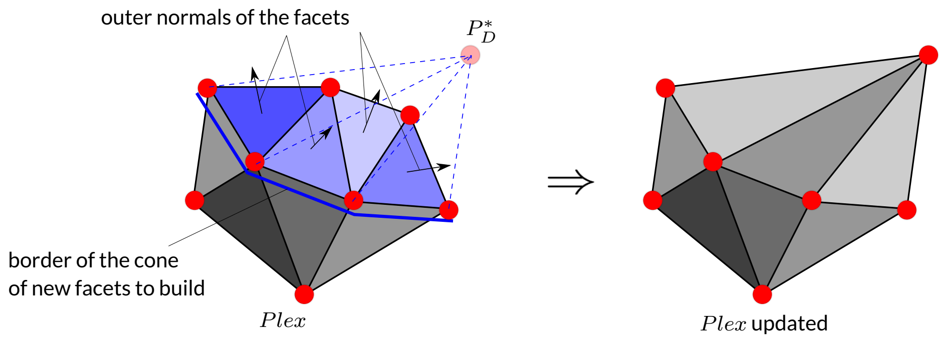


Figure 12: Example of $Plex$ update. All the blue facets are visible from P_D^* and are those that will be replaced by the cone of new facets in the updated version of the $Plex$.

case a new vertex is found, it is used for updating the politope: a cone of new facets is build and an updated politope is obtained, see Figure 12. Otherwise, when the found vertex is already present in the politope, the last facet considered for the expansion is recognized to be the closest to the origin. Therefore, the procedure described in case b of Section 2 is adopted for precisely identify the closest point in the facet and consequently the penetration vector ρ. Algorithm 3 reports the pseudocode of the EPA algorithm. Point ρ_A and ρ_B can be identified by following a procedure similar to the one adopted in Section 2.2 for identify the closest points of the shape. Indeed, considering the points characterizing the vertices of the closest facet to the origin, $P_{1,2,3}$, the penetration vector can be obtained as a combination in the form:

$$\rho = c_1 \cdot P_1 + c_2 \cdot P_2 + c_3 \cdot P_3 \tag{31}$$

The list of vertices in the original shapes that produced $P_{1,2,3}$ are $A_{1,2,3}$ and $B_{1,2,3}$ (review Section 2.2). Therefore:

$$\begin{aligned} \rho_A &= c_1 \cdot A_1 + c_2 \cdot A_2 + c_3 \cdot A_3 \\ \rho_B &= c_1 \cdot B_1 + c_2 \cdot B_2 + c_3 \cdot B_3 \end{aligned} \tag{32}$$

Figure 13 reports a planar example. The tridimensional case is analogous, considering to update a tridimensional convex hull (Figure 12).

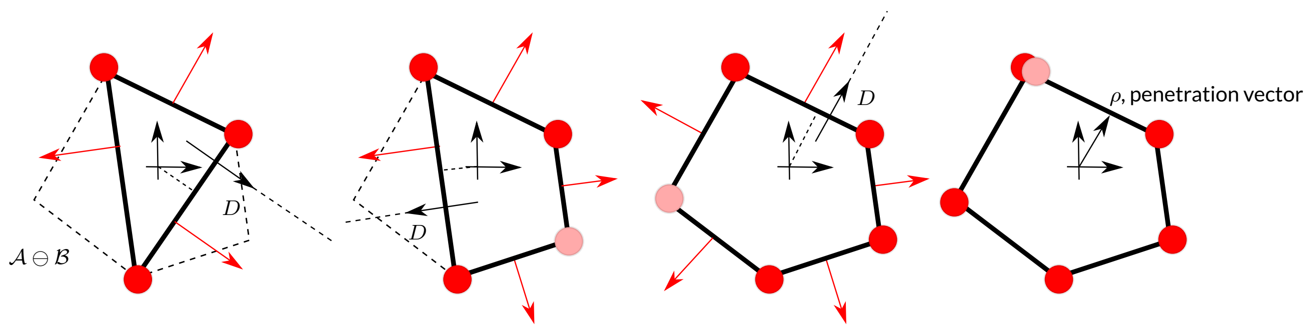


Figure 13: Example of penetration vector computation. Red arrows indicate the outgoing normal of the facets.

```
$Plex = \{P_1, P_2, P_3, P_4\}$;
$F_{clst}$ = find the closest facet to the origin in $Plex$;
$D$ = outgoing normal of $F_{clst}$;
$P^*_D = Support\Big(Plex, D\Big)$;
if $P^*_D \in Plex$ then
    return $\rho$ = closest point to origin in $F_{clst}$;
end
Update $Plex$ building a cone of new facets around $P^*_D$ (Figure 12);
while TRUE do
    $F_{clst}$ = find the closest facet to the origin in $Plex$;
    $D$ = outgoing normal of $F_{clst}$;
    $P^*_D = Support\Big(Plex, D\Big)$;
    if $P^*_D \in Plex$ then
        return $\rho$ = closest point to origin in $F_{clst}$;
    end
    Update $Plex$ building a cone of new facets around $P^*_D$ (Figure 12);
end
```

Algorithm 3: EPA algorithm for penetration vector computation.

4 How to use the GJK_EPA class

The GJK_EPA class implements the GJK and EPA algorithm. The pair of shapes $\mathcal{A}$,$\mathcal{B}$ to consider can be specified using the following methods: Set_shape_A, Set_shape_B, Set_shape_A_transformed and Set_shape_B_transformed. Basically, Set_shape_A accepts as input only the list of vertices pertaining to $\mathcal{A}$. Therefore, when invoking that setter, the local frame (L) (Section 1.0.1) is assumed to be coincident with the absolute one (0). On the opposite, Set_shape_A_transformed must be called passing not only the vertices of $\mathcal{A}$, but also the orientation (the rotation angles across the x,y and z axis are passed, see Section B) and the position of the local frame. Similarly Set_shape_B and Set_shape_B_transformed are used for specifying $\mathcal{B}$.
$\mathcal{A}$,$\mathcal{B}$ are considered undefined until of the above setters are not invoked. After the pair of shapes is completely specified, the proximity queries can be performed. Results are internally stored and possibly recycled between subsequent queries. More specifically, the state machine reported in Figure 14 is considered. The transition between state 0 and 1, or 3, is done when Are_in_collision is called. In this case the GJK version reported in the pseudocode of Algorithm 2 is performed to check whether the shapes are or not in collision. In case a collision is detected, state 1 is reached, otherwise the system goes to state 3.
When Get_penetration is invoked, the system checks whether the actual state is 1. In such a case, the EPA algorithm is invoked and the penetration depth is computed and returned. On the opposite, if the state is 3 or 4, a null value is returned. Finally, if the actual state is 0, the Are_in_collision method is internally called. If after the computations the state reached is 1 the EPA algorithm is invoked and the penetration distance is returned, while in the opposite case a null value is returned. A similar logic is valid for the Get_distance method. In this case the closest region to the origin in the Minkowski difference must be computed and the closest points can be identified adopting Algorithm 1.

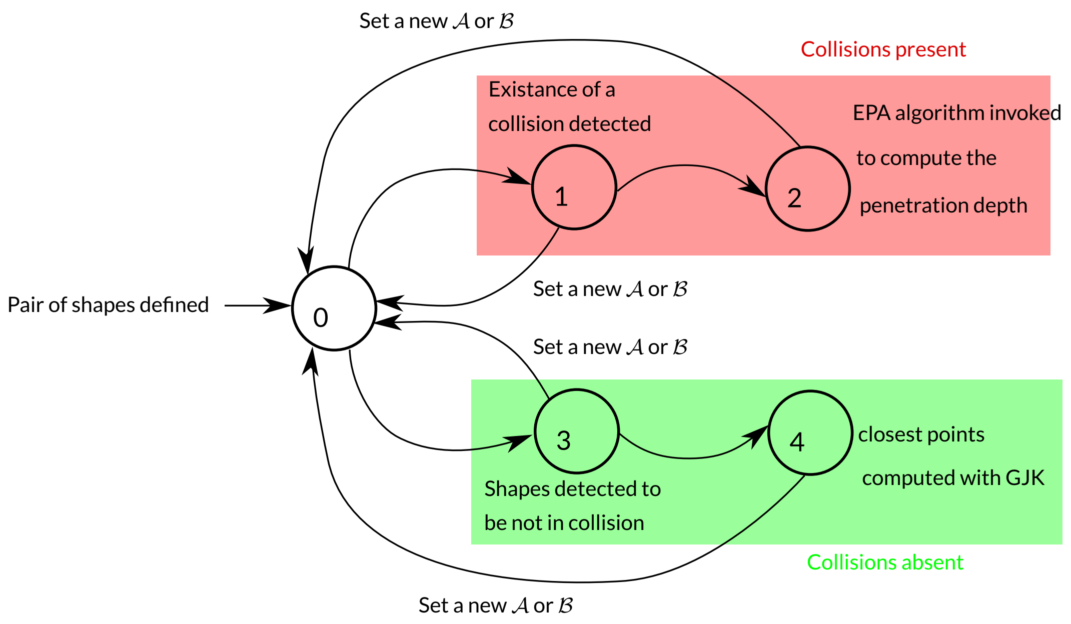

Set a new $\mathcal{A}$ or $\mathcal{B}$
Collisions present
Existance of a collision detected
EPA algorithm invoked to compute the penetration depth
1
2
Pair of shapes defined
0
Set a new $\mathcal{A}$ or $\mathcal{B}$
Set a new $\mathcal{A}$ or $\mathcal{B}$
3
4
closest points computed with GJK
Shapes detected to be not in collision
Collisions absent
Set a new $\mathcal{A}$ or $\mathcal{B}$

Figure 14: The state machine implemented in the GJK_EPA class.

5 Logging for debugging

METTERE struttura JSON che viene prodotto se log è abilitato.

Appendices

A Homogeneous matrix

Homogeneous matrix are adopted for describing the relative position of different frames. Essentially, an homogeneous matrix M_{AB} contains an orientation matrix R_{AB} (see Appendix B) and a translation vector T_{AB} describing, respectively, the relative orientation and position of a frame (B) w.r.t. (A). When a point V is defined in the frame (B), its coordinates $V_{(B)}$ are known, i.e. the coordinates with respect to (B). Generally speaking, the coordinates of V w.r.t. to the frame (A), i.e. $V_{(A)}$, are different (refer to Figure 15). To pass from $V_{(B)}$ to $V_{(A)}$ the homogeneous matrix is exploited, leading to the following expression:

$$V_{(A)} = R_{AB}V_{(B)} + T_{AB} \tag{33}$$

B Orientation matrix

A rotation matrix R_{AB} is a 3×3 matrix describing the orientation of a frame (B) w.r.t to another one (A). R_{AB} is composed as follows:

$$R_{AB} = \begin{bmatrix} X & Y & Z \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} X_x \\ X_y \\ X_z \end{bmatrix} & \begin{bmatrix} Y_x \\ Y_y \\ Y_z \end{bmatrix} & \begin{bmatrix} Z_x \\ Z_y \\ Z_z \end{bmatrix} \end{bmatrix} \tag{34}$$

X, Y and Z defines the direction of the axis of frame (B) seen from frame (A). Computing R_{AB} can be difficult. For this reason an alternative adopted representation are angles $\gamma_x, \gamma_y, \gamma_z$. The idea is to obtain frame (B) starting from frame (A) by applying three subsequent rotations: the first one around the x axis, the second around the y axis of the obtained frame and the last around the resulting z axis, see Figure 16. To pass from the rotation angles $\gamma_x, \gamma_y, \gamma_z$ to R_{AB}, the three matrices reported in Figure 16 are multiplied:

$$R_{AB} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma_x) & -sin(\gamma_x) \\ 0 & sin(\gamma_x) & cos(\gamma_x) \end{bmatrix} \cdot \begin{bmatrix} cos(\gamma_y) & 0 & sin(\gamma_y) \\ 0 & 1 & 0 \\ -sin(\gamma_y) & 0 & cos(\gamma_y) \end{bmatrix} \cdot \begin{bmatrix} cos(\gamma_z) & -sin(\gamma_z) & 0 \\ sin(\gamma_z) & cos(\gamma_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{35}$$

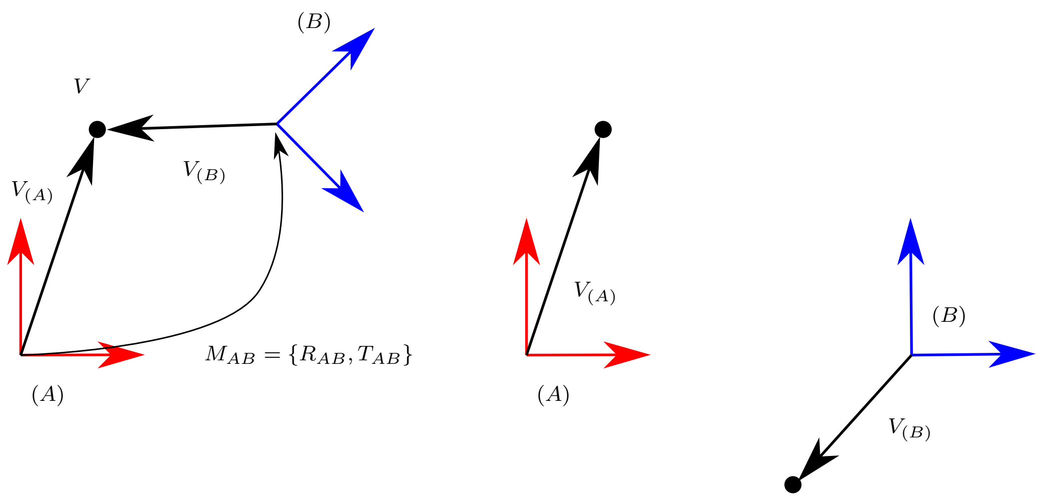
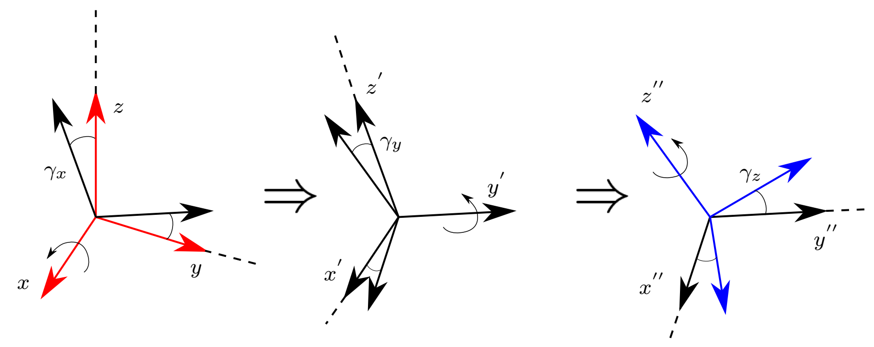


Figure 15: The same point has different coordinates in different frames.



rotation around x:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma_x) & -sin(\gamma_x) \\ 0 & sin(\gamma_x) & cos(\gamma_x) \end{bmatrix}$$

rotation around y:

$$\begin{bmatrix} cos(\gamma_y) & 0 & sin(\gamma_y) \\ 0 & 1 & 0 \\ -sin(\gamma_y) & 0 & cos(\gamma_y) \end{bmatrix}$$

rotation around z:

$$\begin{bmatrix} cos(\gamma_z) & -sin(\gamma_z) & 0 \\ sin(\gamma_z) & cos(\gamma_z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 16: Angles $\gamma_x, \gamma_y, \gamma_z$ are interpreted as subsequent rotations to apply for getting the orientation of frame (B) (in blue), starting from frame (A) (in red)