Collision detection algorithm for MIRADAS

David Roma[a,b,c] **Jose Bosch**[a,b,d] **Josep Sabater**[b] **and Jose M. Gomez**[a,b,d,*]

[a]Universitat de Barcelona, Department of Electronic and Biomedical Engineering, Barcelona, Spain
[b]Institute of Space Studies of Catalonia, Barcelona, Spain
[c]Institut de Ciències de l'Espai, Campus UAB, Bellaterra, Spain
[d]University of Barcelona, Institute of Cosmos Sciences, Barcelona, Spain

Abstract. Mid-resolution Infrared Astronomical Spectrograph (MIRADAS) is a near-infrared multi-object echelle spectrograph for Gran Telescopio de Canarias. It selects targets from a 5-arc min field of view using up to 12 deployable probe arms with pick-off mirror optics. The focal plane where the probe arms move has a diameter around 250 mm. The specific geometry of the probe arms requires an optimized collision detection algorithm for the determination of the target assignment and the trajectories determination. We present the general polygonal chain intersection algorithm, which is used to detect the possible collisions and avoid them. It is a generalization of the Polygonal Chain Intersection algortihm, allowing to work with vertical segments, providing a solution for the intersection of any class of polygons. Its use has reduced the time required to detect the collisions between 3 and 4 times compared with a naive solution when used in MIRADAS. © *2021 Society of Photo-Optical Instrumentation Engineers (SPIE)* [DOI: 10.1117/1.JATIS.7.1.015003]

Keywords: collision detection; polygon; intersection; sweep line; multiple object; spectrograph.

Paper 20149 received Oct. 5, 2020; accepted for publication Feb. 3, 2021; published online Mar. 2, 2021.

1 Introduction

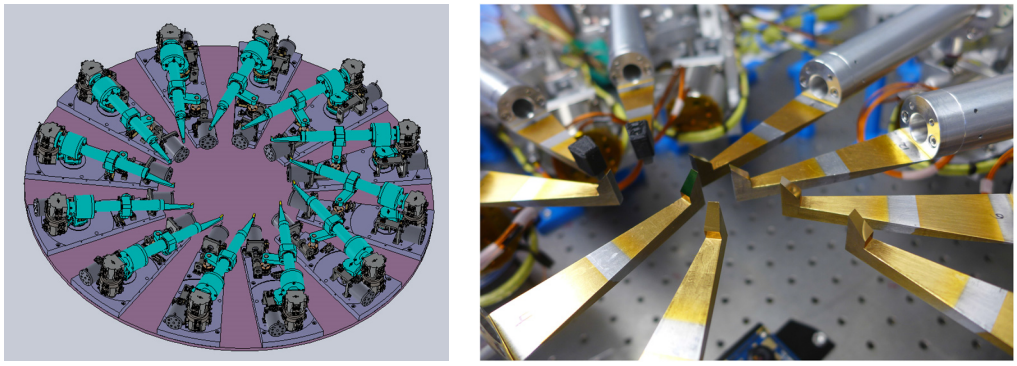
Multi-object spectroscopy (MOS) is a key technique in modern astronomical instrumentation. In case of large telescope, the need to improve their efficiency makes necessary to include a multiplexer (MXS) that selects specific targets in the field of view (FoV). Different solutions have been proposed and are being used, in the most cases requiring mechanical devices that need to be coordinated to avoid losing functionality.

Mid-resolution Infrared Astronomical Spectrograph (MIRADAS) is a near-infrared multi-object echelle spectrograph operating at spectral resolution $R = 20,000$ over the 1- to 2.5-μm bandpass for the Gran Telescopio de Canarias (GTC). MIRADAS selects targets from a 5-arc min FoV using up to 12 deployable probe arms with pick-off mirror optics, each feeding a 3.7×1.2 arc sec^2 FoV to the spectrograph. These arms are distributed around the focal plane, which has ~250 mm diameter (see Fig. 1). There they have to be choreographed.

We define:

- *Assignment*. It identifies which probe arm will observe a target.
- *Configuration*. The set of positions of the different probe arms when an observation is done. Not all the probe arms have to be observing, hence not all the probe arms have an assignment. The ones not assigned have a position where they do not collide with others. This position can be park, or one nearby to park that is calculated *ad hoc* based on the nearby assigned probes.
- *Movement*. The set of positions that a probe arm shall follow to go from one configuration to another. These movements shall comply with some restrictions in order to optimize

*Address all corrspondence to Jose M. Gomez, jm.gomez@ub.edu



(a) (b)

Fig. 1 (a) Drawing of the MXS bench with the 12 probe arms and (b) detail of the pick-off mirrors.

the instrument usage (e.g., the maximum time allowed to reach from one configuration to another).

A planning tool will be used by the astronomers to determine the different assignments/configurations. The requirements of the different targets assignments and the movements to reach them can lead to collisions between the probe arms.[1] The tool shall be able to detect these collisions and change the planed configurations and movements to avoid them. This will guarantee a safe usage of the instrument and optimize its observation time. The number of collision checks per movement can be between tenths to hundreds, depending on the distance of the motion.

For all these reasons, a collision detection algorithm (CDA) is required. In case of MIRADAS, it is based on a polygon intersection algorithm. Each probe arm is described as an irregular convex hexagon, as shown in Fig. 2.

The hexagons presented include a guard buffer (Fig. 3) whose size will depend on the mechanical tolerances and numerical errors, minimizing the risk of collision associated. If there is any intersection between the polygons, it means a collision, indicating that this arrangement is not valid. The algorithm shall be precise, to allow close targets to be picked by different probe arms, increasing the instrument efficiency.

The CDA is constantly used to check the feasibility of a configuration or the movements that are used to go from one configuration to another.[1] As a result, it is necessary to optimize it, as this will reduce significantly the processing time. This will improve also the responsiveness of the interactive targets–probes planning tools.

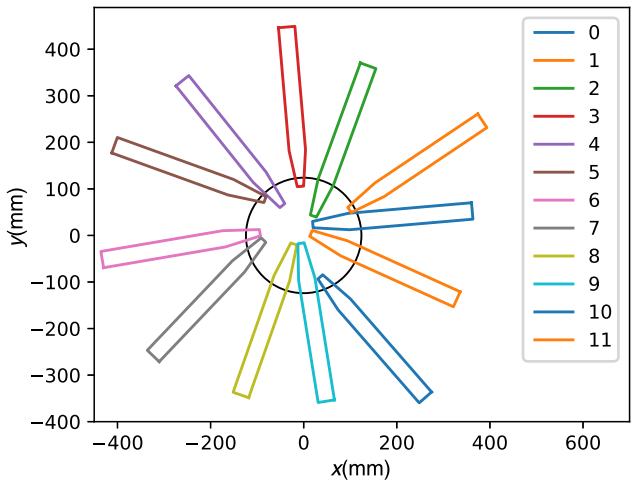


Fig. 2 Example of a probe arm configuration including the guard buffers (see Fig. 3).

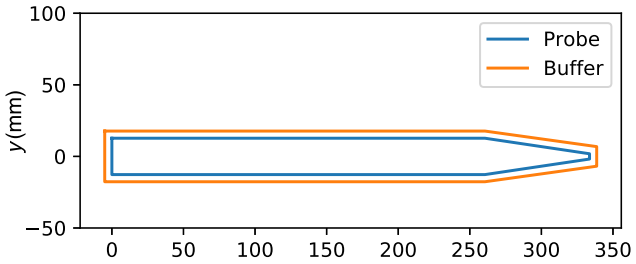


Fig. 3 Example of a probe arm and its guard buffer.

Polygon intersection algorithms are based on line segment intersection algorithms. In the present application, it can be assumed that all the polygons have the same number of vertices (n). A naive algorithm, which pairwise checks the intersection of the full set of segments in the two polygons, will require $\mathcal{O}(n^2)$ operations to find them. As there are p polygons, and we are interested only in intersections between polygons (arms), the problem can be reduced to $\mathcal{O}\left(\binom{p}{2} \cdot n^2\right)$.

If we focus on convex polygons, it can be reduced to linear complexity[2,3] $\mathcal{O}(2n)$, resulting in $\mathcal{O}\left(\binom{p}{2} \cdot n^2\right)$ for the intersections between polygons. A logarithmic complexity algorithm[4] can be also used, reaching $\mathcal{O}(\log(n^2))$ in the general case, or $\mathcal{O}\left(2 \cdot \binom{p}{2} \cdot \log(n)\right)$ in case of polygons.

Considering a polygon as a closed polygonal chain, the polygonal chain intersection (PCI) algorithm[5] can be used. It has a complexity of $\mathcal{O}((n \cdot p + k)\log(m))$, where m is the number of Monotone Polygonal Chains (MCs), and k is the number of intersections. The MC is defined[6] as "A chain C is said to be monotone with respect to a straight line L if a line orthogonal to L intersects C in exactly one point." Without loosing generality, the line L can be the x axis [Fig. 4(a)].

In this case, all the vertices in the MC will have an increasing value of x ($p_n.x < p_{n+1}.x$). A monotone polygon can be divided in two MCs, and any convex polygon is monotone.[7] Hence, in the present scenario, two MC per probe arm polygon can be expected, so the complexity will become $\mathcal{O}((n \cdot p + k)\log(2p))$.

In many cases, one or more segments of different probes can be parallel to the y axis (vertical), e.g., one of the segments near the pick-off mirror of probe 9, shown in Fig. 2 configuration. This implies the need to improve the PCI algorithm to take into account vertical segments, so it can be used with any possible MIRADAS configuration.

One approach could be to detect *a priori* the vertical segments[8] and manage them specifically. In our case, we will follow a different approach, generalizing the MCs allowing to include perpendicular segments to the line L, an example is represented in [Fig. 4 (b)]. We will call them General Monotone Polygonal Chains (GMCs). In this case, a point can have the same x as the precedent or following point ($p_n.x \leq p_{n+1}.x$). In this case, the monotonicity is also requested on the y axis, henceforth all the perpendicular segments shall follow the same direction. In the case shown in the figure, we can see that both perpendicular segments go in the upward direction ($p_n.y < p_{n+1}.y$). Notice that the oblique have only horizontal monotonicity.

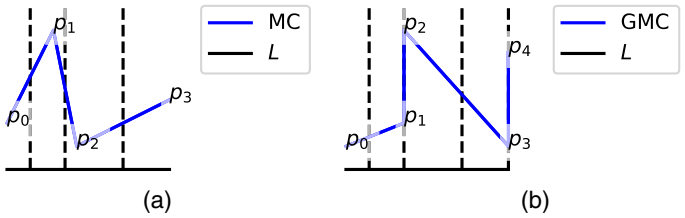


Fig. 4 (a) Monotone polygonal chain with respect to line L and (b) GMC with respect to line L.

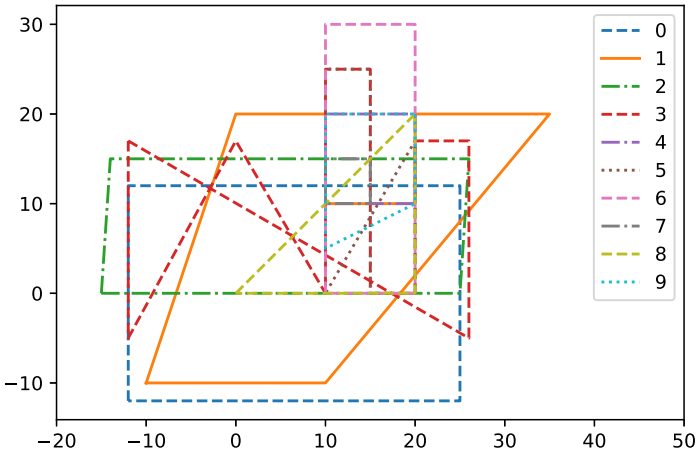


Fig. 5 An example with ten polygons including different complex geometries whose intersections can be detected by the GPCI algorithm.

Therefore, an MC is a GMC that has no perpendicular segments to L or, in this case, vertical segments. Its usage generalizes the algorithm allowing the detection of intersections with any class of polygons (Fig. 5). For this reason, we will call the algorithm General Polygonal Chain Intersection (GPCI). It is described in the next sections.

2 Base Concepts

The GPCI needs some new concepts to generalize the PCI algorithm. They help to improve the numerical stability of the algorithm and allow to work with vertical segments. They are described in the following sections.

2.1 *Difference*

First, the difference between two numbers is defined. Although the calculations are done using floating-point values with double-precision,[9] there are numerical errors. Hence, it is needed to define how separated two numbers can be and still be considered to have the same value. For this purpose, the difference between the number a and b with a bits factor is defined as

$$d_{\text{bits}}(a,b) = \lfloor (a-b)\cdot 2^{\text{bits}} \rceil, \tag{1}$$

where $\lfloor x \rceil$ is the rounding to the nearest integer function. Notice that we use bits instead of decimal digits, as it is optimal when using the functions associated with IEEE floating point numbers exponent. An approximation can be used to convert from decimal error to number of bits:

$$\text{bits} = \lceil -\log_2(\epsilon_{10}) \rceil, \tag{2}$$

where ϵ_{10} is the decimal error and $\lceil x \rceil$ is the ceil value. In case of MIRADAS, the required resolution is better than 1 μm. As the focal plane units are millimiters, the maximum allowed error is 10^{-4}, the bits value will be 14.

2.2 *Sorting*

The algorithm needs also to sort the elements. For this purpose, a less than operator ($<_d$ [bits]) is defined. It makes use of this difference, making it more numerically stable:

$$a <_{d[\text{bits}]} b \equiv \lfloor (a-b)\cdot 2^{\text{bits}} \rceil < 0. \tag{3}$$

To generate the GMCs, the vertices in the polygon must be also sorted. As indicated in Sec. 1, the x and y dimensions shall be taken into account, requiring a 2D less than operator ($<_{2D[\text{bits}]}$). We define it as

$$\boldsymbol{p}_a <_{2D[\text{bits}]} \boldsymbol{p}_b \equiv \begin{cases} x_a <_{d[\text{bits}]} x_b & \text{for } x_a <_{d[\text{bits}]} x_b \vee x_b <_{d[\text{bits}]} x_a \\ y_a <_{d[\text{bits}]} y_b & \text{for } x_a \not<_{d[\text{bits}]} x_b \wedge x_b \not<_{d[\text{bits}]} x_a \end{cases}, \tag{4}$$

where the points $\boldsymbol{p}_a$ and $\boldsymbol{p}_b$ have the coordinates (x_a, y_a) and (x_b, y_b), respectively. The x coordinate is the first used for the sorting. Only if the x_a and x_b can be considered equal (vertical segment), the y coordinate is checked.

This would be the case for the chain in Fig. 4(a). The line L follows the x axis. The point p_n has a lower x value than p_{n+1} ($p_n.x \leq p_{n+1}.x$). In the case of the chain in Fig. 4(b), the segment between the points p_1 and p_2 and the one between p_3 and p_4 are perpendicular to L, and parallel to the y axis. In case of vertical segments, we impose without any loss of generality that they have to go upward. For this reason, the operator will be true if the value of the y coordinate of the starting point is smaller than the one of the ending point ($p_n.y < p_{n+1}.y$).

2.3 *Intersection Detection*

The intersection is detected using a half-open segment,[10] where the first point of the segment is included and the last not. The possible intersections and overlaps are represented in Fig. 6.

It can be seen that three types of intersections and overlaps are defined (X, T, and V) depending on the point and place of the intersection or overlap. Notice that only in the X intersection the point (p_i) is unknown. In all the other cases, it is defined by one or both segments starting points.

The detection of an intersection is done using the orientation between the points[11] defined by the following equation:

$$\begin{aligned} \text{orientation} \equiv (p_c - p_b) \times (p_b - p_a) &= (p_c.x - p_b.x) \\ &\cdot (p_b.y - p_a.y) - (p_b.x - p_a.x)\cdot(p_c.y - p_b.y), \end{aligned} \tag{5}$$

where p_a, p_b, and p_c are the points in a 2D plane, and the $\times$ operator is the two-dimensional cross product, resulting in a scalar value.

The calculations use the $<_{d[\text{bits}]}$ operator. If the orientation is not less than 0 and 0 is not less than orientation, the three points are considered collinear. If the result is positive, the vector $p_b\vec{p}_c$ is rotated clockwise (CW) relatively to $p_a\vec{p}_b$. If it is negative, the rotation is counter clockwise (CCW). The detection of the intersection is highly dependent on the resolution used for the

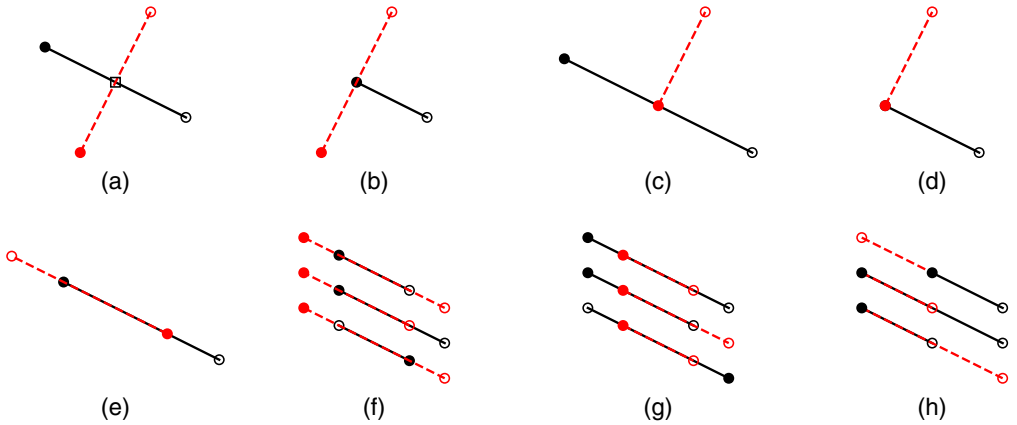


Fig. 6 Possible (a)–(d) intersections and (e)–(h) overlaps between two half-open segments: (a) X intersection; (b), (c) T intersections, (d) V intersection; (e) X overlap; (f), (g) T overlaps; and (h) V overlap.

Algorithm 1 Orientation algorithm

```
1: procedure ORIENTATION(p_a, p_b, p_c, bits)
2:     orientation ← (p_b.y − p_a.y) × (p_c.x − p_b.x) − (p_b.x − p_a.x) × (p_c.y − p_b.y)   ▷ Using Eq. (5)
3:     obits ← (bits + 1) × 2 + 1                                                            ▷ Using Eq. (6)
4:     if orientation <_d[obits] 0 then
5:         return Counter Clockwise
6:     else if 0 <_d[obits] orientation then
7:         return Clockwise
8:     else
9:         return Collinear
```

orientation calculation. Taking into account that there is a subtraction of the product of two subtractions, the number of bits needed to make the calculation without loosing precision is

$$\text{bits}_{\text{orientation}} \equiv (\text{bits} + 1) \times 2 + 1. \tag{6}$$

Example given: in case of MIRADAS, the number of bits is 14 bits (24 bits for the full mantissa). In this case, the $\text{bits}_{\text{orientation}}$ is 39 bits (51 bits full mantissa), just below the limit (53 bits).

If there is an intersection between two segments a and b, the orientation between the first point of a (a_0), the last point (a_1), and the first point of b (b_0) shall be different to the one between a_0, a_1, and b_1, and also the orientation between b_0, b_1, and a_0 shall be different to the one between b_0, b_1, and a_1. An example is presented in Fig. 7.

The calculation of the intersection between segment a and b requires two specific data types.

- *Segment*: represents a line segment that starts at point 0 and ends at point 1.
- *Result*: represents the intersection result. It has the fields.

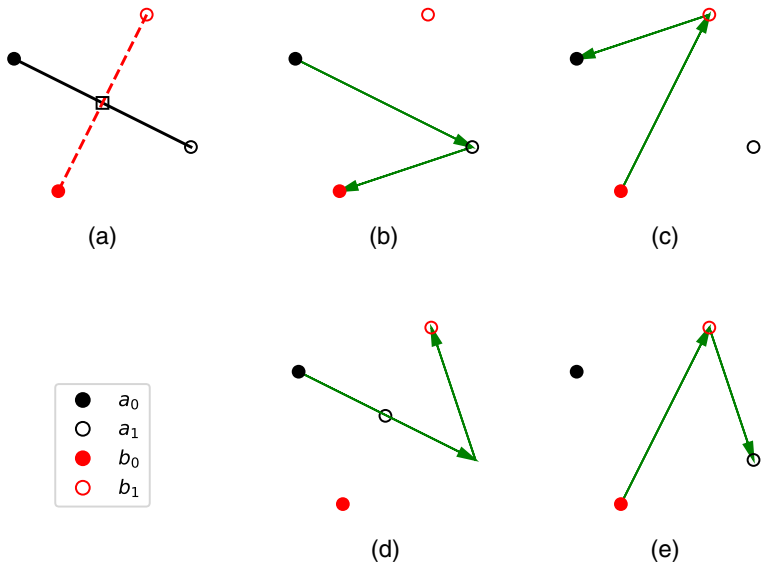


Fig. 7 (a) Intersection detection between segment a and b. The orientation (b) between the point a_0, the point a_1, and the point of b_0 is CW; (d) between a_0, a_1, and b_1 is CCW; (c) between b_0, b_1, and a_0 is CCW; and (e) between b_0, b_1, and a_1 is CW.

Algorithm 2 Intersection algorithm

1: **procedure** INTERSECTION($a, b, bits$)
2: ▷ Initialization of the result
3: $result.point \leftarrow empty$ ▷ Intersection point, initialised as empty
4: $result.type \leftarrow No\ intersection$
5: $ab0 \leftarrow$ ORIENTATION($a_0, a_1, b_0, bits$)
6: $ab1 \leftarrow$ ORIENTATION($a_0, a_1, b_1, bits$)
7: ▷ If the segments overlap
8: **if** $(ab0 = Collinear) \wedge (ab1 = Collinear)$ **then**
9: $result.type \leftarrow$ overlap($a, b, bits$)
10: **return** *result*
11: ▷ If the segments do not overlap
12: $ba0 \leftarrow$ ORIENTATION($b_0, b_1, a_0, bits$)
13: $ba1 \leftarrow$ ORIENTATION($b_0, b_1, a_1, bits$)
14: **if** $(ab0 = Collinear) \wedge (ba0 = Collinear)$ **then**
15: $result.type \leftarrow V\ intersection$ ▷ Fig. 6(d)
16: **return** *result*
17: **if** $(ab0 = Collinear) \wedge (ab1 \neq Collinear) \wedge (ba0 \neq Collinear) \wedge (ba1 \neq Collinear) \wedge (ba0 \neq ba1)$ **then**
18: $result.type \leftarrow T\ intersection\ with\ a$ ▷ Fig. 6(c)
19: **return** *result*
20: **if** $(ab0 \neq Collinear) \wedge (ab1 \neq Collinear) \wedge (ba0 = Collinear) \wedge (ba1 \neq Collinear) \wedge (ab0 \neq ab1)$ **then**
21: $result.type \leftarrow T\ intersection\ with\ b$ ▷ Fig. 6(b)
22: **return** *result*
23: **if** $(ab1 \neq Collinear) \wedge (ba1 \neq Collinear) \wedge (ab0 \neq ab1) \wedge (ba0 \neq ba1)$ **then**
24: $u_a \leftarrow \frac{(b_1 - b_0) \times (a_0 - b_0)}{(a_1 - a_0) \times (b_1 - b_0)}$ ▷ Eq. (8)
25: $result.point \leftarrow a_0 + u_a(a_1 - a_0)$ ▷ Eq. (7)
26: $result.type \leftarrow X\ intersection$ ▷ Fig. 6(a)
27: **return** *result*
28: **return** *result* ▷ No intersection

- *point*: resulting intersection point, in case of X intersection.
- *type*: intersection type, as described in Fig. 6.

Algorithm 2 describes the detection of the different intersection cases indicated in Fig. 6 using the orientation calculation. If two segments are collinear, then Algorithm 3 is called to check the overlap cases.

When there is an intersection, the associated point (p_i) can be calculated using any of the segments with the following equation:

Algorithm 3 Overlap algorithm

1: **procedure** OVERLAP($a, b, bits$) ▷ Returns the type of overlap
2: $u_a \leftarrow \frac{(b_0-a_0)\cdot(a_1-a_0)}{(a_1-a_0)\cdot(a_1-a_0)}$ ▷ Eq. (10)
3: $u_b \leftarrow \frac{(a_0-b_0)\cdot(b_1-b_0)}{(b_1-b_0)\cdot(b_1-b_0)}$ ▷ Eq. (11)
4: $u_{a0} \leftarrow d_{\text{bits}}(u_a, 0) = 0$ ▷ u_a can be considered 0
5: $u_{b0} \leftarrow d_{\text{bits}}(u_b, 0) = 0$ ▷ u_b can be considered 0
6: $u_{a1} \leftarrow d_{\text{bits}}(u_a, 1) = 0$ ▷ u_a can be considered 1
7: $u_{b1} \leftarrow d_{\text{bits}}(u_b, 1) = 0$ ▷ u_b can be considered 1
8: $u_{\text{aWithin}} \leftarrow (\neg u_{a0}) \wedge (0 < u_a \wedge u_a < 1) \wedge (\neg u_{a1})$ ▷ $u_a \in (0,1)$
9: $u_{\text{bWithin}} \leftarrow (\neg u_{b0}) \wedge (0 < u_b \wedge u_b < 1) \wedge (\neg u_{b1})$ ▷ $u_a \in (0,1)$
10: **if** $u_{\text{aWithin}} \wedge u_{\text{bWithin}}$ **then return** *X overlap* ▷ Fig. 6(e)
11: **if** $u_{a0} \wedge u_{b0}$ **then return** *V overlap* ▷ Fig. 6(h)
12: **if** u_{bWithin} **then return** *T overlap with b* ▷ Fig. 6(f)
13: **if** u_{aWithin} **then return** *T overlap with a* ▷ Fig. 6(g)
14: **return** *No intersection* ▷ Non-overlapping parallel segments

$$p_i = a_0 + u_a(a_1 - a_0). \tag{7}$$

In this case, the segment a has been used, and u_a is the parameter. If an X intersection is detected using the orientations (Fig. 7), the values of the parameters u_a and u_b can be calculated using the following equations:

$$u_a = \frac{(b_1 - b_0) \times (a_0 - b_0)}{(a_1 - a_0) \times (b_1 - b_0)}, \tag{8}$$

$$u_b = \frac{(a_1 - a_0) \times (a_0 - b_0)}{(a_1 - a_0) \times (b_1 - b_0)}. \tag{9}$$

In all the cases, we will consider there is an intersection or overlap only if u_a and u_b are in the interval $[0,1)$. As indicated previously, only when there is an X intersection p_i is unknown.

If the segments are collinear, it will be necessary to determine if the segments overlap. This is the purpose of the overlap procedure (Algorithm 3).

The parameters u_a and u_b can be calculated using the following equations:

$$u_a = \frac{(b_0 - a_0) \cdot (a_1 - a_0)}{(a_1 - a_0) \cdot (a_1 - a_0)}, \tag{10}$$

$$u_b = \frac{(a_0 - b_0) \cdot (b_1 - b_0)}{(b_1 - b_0) \cdot (b_1 - b_0)}, \tag{11}$$

where the $\cdot$ operator is the inner product. As indicated previously, u_a and u_b shall be in the interval $[0,1)$ to consider it is an overlap. The d_{bits} function ensures the numerical stability of the algorithms.

3 General Polygonal Chain Intersection Algorithm

As said previously, the GPCI algorithm follows a sweep line (SL) strategy.[12] It introduces the use of the GMC that generalizes the algorithms defined by the Monotone Chain Intersection [13] and the PCI,[5] allowing to use it with any class of polygons, like the ones presented in Fig. 5.

An SL can be represented as a vertical line that moves horizontally on a plane. The SL moves from the left side of the plane to the right side, in a quantified process. It will only visit the x positions where there is an GMC vertex or where an intersection is detected. In the present algorithm, every GMC has a token. At the beginning, the token is associated with the first vertex of the GMC. When the SL reaches a vertex in one GMC, the token is passed to the following vertex in the same GMC.

The vertices can have one of the following types.

- *Leftmost*: the first and minimum vertex in a GMC.
- *Rightmost*: the last and maximum vertex in a GMC.
- *Internal*: any other vertex in the GMC.
- *Intersection*: a vertex that is in the intersection between two or more GMCs.
- *Rightmost intersection*: a rightmost vertex in one or more GMCs it belongs that is also an intersection. This case is needed as the segments are considered half-open.

The type of a vertex can evolve as the algorithm progresses, starting with one of the first three types.

All the GMCs are stored in the active chain list (ACL). The ACL indicate the next point in the GMC that will be visited by the SL. To optimize the process, a second list is used, the sweeping chain list (SCL). This list stores the GMCs that intersect with the SL at the present x position.

The algorithm uses the following data types.

- *Point*: represents a 2D point with the x and y coordinates.
- *Polygon*: a circular list of points representing a polygon.
- *Vertex* (v): contains the polygon point, its type, and a list with the GMCs with which it belongs.
- GMC: It is a list of vertex elements. The GMC includes a field indicating the polygon with which it belongs and the token.
- *ACL*: It is a list of GMCs.
- *SCL*: It is also a list of GMCs.

The algorithm has two main processes: the determination of the GMCs and the intersections determination. This are described in the following sections.

3.1 *General Monotone Polygonal Chains Determination Algorithm*

The first step of the algorithm is determined the GMCs of every polygon and storeed them in the ACL. This will be used as the starting point of the GMC, as presented in Algorithm 4. The algorithm can be used for any type of polygon (convex or complex). At least, two GMCs will be generated per polygon.

First, notice that this algorithm ensures that all the chains have a monotone increase, as defined by the $<_{2D[\text{bits}]}$ operator. Second, the first and last vertices of the different chains are shared with other chains of the same polygon. This has to be taken into account to avoid detecting these points as intersections.

To simplify things, the default type of a vertex is internal. When an GMC is added to the ACL, the type of the first and last vertices (v_0 and v_{n-1}) of the GMC are updated to leftmost and rightmost types, respectively, and the token is initialized with the v_0 handle. Finally, the GMC gets the present polygon handle, to manage the GMCs sharing of vertices from the same polygon. The algorithm will consider two leftmost or two rightmost vertices that are close an intersection only if they belong to different polygons.

Algorithm 4 GMC determination algorithm

```
1:  procedure GMCDETERMINATION(polygon, ACL, bits)   ▷ Determines the GMCs of the polygon and stores them in the ACL
2:      p_min ← min_2D(polygon)                      ▷ The minimum point is found using the <_2D[bits] operator
3:      p_-1 ← p_min
4:      idx ← 0
5:      gmc ← new GMC                                ▷ create a new GMC
6:      add p_min to back gmc                        ▷ Append to the end of the GMC
7:      direction ← increase
8:      for p_i ← polygon.after(p_min) do            ▷ Loop over the polygon points,
9:                                                   ▷ starting with point just after p_min
10:         if p_-1 <_2D[bits] p_i then
11:             if direction = increase then
12:                 add p_i to back gmc
13:             else
14:                 add gmc to ACL
15:                 gmc ← new GMC
16:                 add p_-1 to back gmc
17:                 add p_i to back gmc
18:                 direction ← decrease
19:         else
20:             if direction = decrease then
21:                 add p_i to front gmc             ▷ Insert at the beginning of the GMC
22:             else
23:                 add gmc to ACL
24:                 gmc ← new GMC
25:                 add p_-1 to front gmc
26:                 add p_i to front gmc
27:                 direction ← increase
28:             p_-1 ← p_i
29:     if direction = increase then
30:         add gmc to ACL
31:         gmc ← new GMC
32:         add p_-1 to front gmc
33:     add p_min to front gmc
34:     add gmc to ACL
```

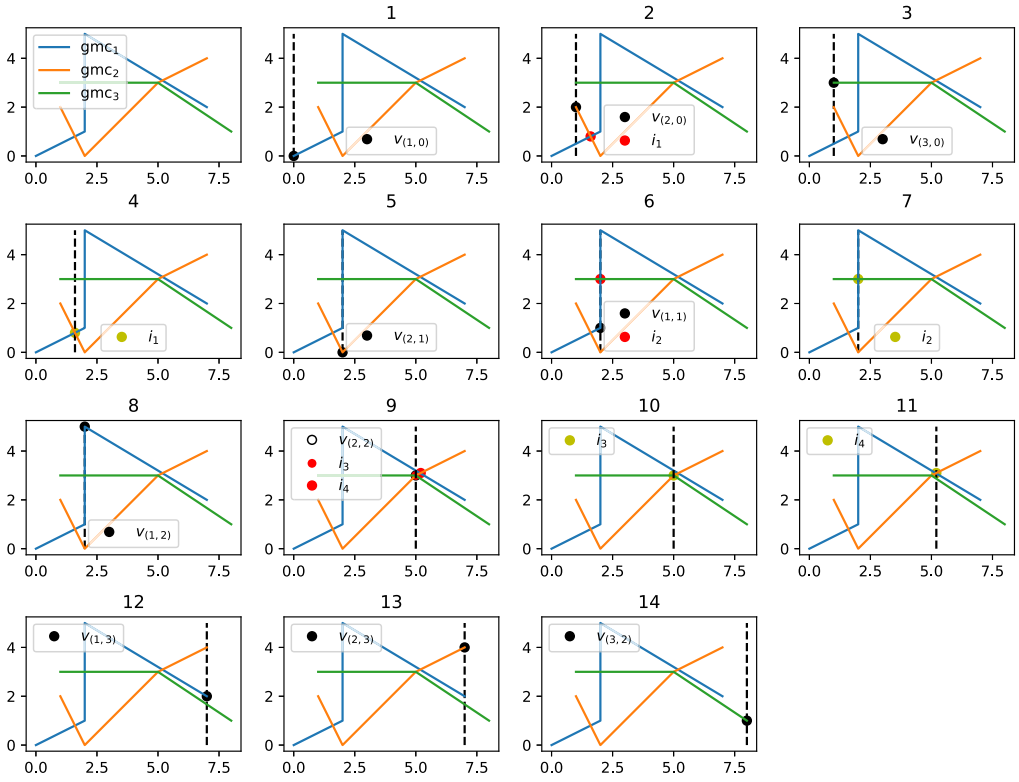


Fig. 8 An example with three GMCs and the GPCI algorithm evolution. The black dots are GMC vertices, yellow ones are intersections that the SL is scanning, whereas the red are the intersections found.

3.2 *General Polygonal Chain Intersection Description*

In this section, we will describe the full GPCI process, step by step, following Fig. 8 sequence. The contents of the state variables for the firsts steps and the last are explicitly indicated in Table 1.

In the PCI algorithm, the slope is used to calculate the precedence of the MCs. In GPCI, the sine of the angle (α) is used, to avoid the infinite value of the slope when there is a vertical segment. For simplicity, we will refer to it as α.

Table 1 GPCI state variables for steps 1 to 5 and the last (15), the $\wedge$ operator indicates the intersection calculation in this case.

	1	2	3	4	5	15
X	0	1	1	1.6	2	8
Vertex	$v_{(1,0)}$	$v_{(2,0)}$	$v_{(3,0)}$	i_1	$v_{(2,1)}$	$v_{(3,2)}$
Token	$v_{(1,1)}$	i_1	$v_{(3,1)}$	$v_{(1,1)}$ $v_{(2,1)}$	$v_{(2,1)}$	$\varnothing$
ACL[0]	gmc_1	gmc_2	gmc_3	gmc_1	gmc_1	gmc_3
SCL	gmc_1	gmc_2 gmc_1	gmc_3 gmc_2 gmc_1	gmc_3 gmc_1 gmc_2	gmc_3 gmc_1 gmc_2	gmc_3
Checks		$gmc_2 \wedge gmc_1$	$gmc_2 \wedge gmc_3$	$gmc_1 \wedge gmc_3$	$gmc_2 \wedge gmc_1$	
Intersection			i_1		i_2	
Chains			gmc_1 gmc_2		gmc_1 gmc_3	

1. The first step can be seen in the second plot of Fig. 8. The SL starts with x value 0. This is where the leftmost vertex of the first GMC in the ACL (gmc_1) is placed. The vertex $v_{(1,0)}$ (where the first subindex is the GMC and the second the associated vertex index) becomes active, and token to the next vertex of the current GMC, which is vertex $v_{(1,1)}$. Afterward, the gmc_1 is stored in the SCL. As there are no other GMC in the SCL, no intersection is considered.
2. The ACL is sorted taking into account the tokens positions [$v_{(1,1)}$, $v_{(2,0)}$, $v_{(3,0)}$]. The SL moves to gmc_2. The token is passed to $v_{(2,1)}$. The gmc_2 is stored in the SCL. As there are two GMCs, a possible intersection with the upper GMC (gmc_1) is checked. In the second plot of Fig. 8, it can be seen that the segments associated with the present SL position intersect in i_1. This point will be added as a new vertex to both gmc_1 and gmc_2, and the token from $v_{(1,1)}$, $v_{(2,1)}$ will be passed back to i_1.
3. The SL moves to the next GMC in the ACL, in this case, the gmc_3. The token is passed to $v_{(3,1)}$. The gmc_3 is stored in the SCL. There are three GMCs, but only gmc_2 is a nearest neighbor. For this reason, only the possible intersection between gmc_3 and gmc_2 is checked with a negative result, as the segments associated with the present SL position do not intersect.
4. The SL moves to gmc_1, as it is the GMC that has the highest α at the intersection i_1. Also, all the GMCs that intersect pass the token to the following vertex [in this case $v_{(1,1)}$ and $v_{(2,1)}$]. The segments that start in the point i_1 are analyzed (in this case only the ones associated with gmc_1 and gmc_2). The segment with the highest α is checked with the upper GMC, and the one with the lowest α is checked with the lower GMC. In this case, only the gmc_1 segment and the gmc_3 one are checked, which do not intersect.
5. The SL continues to gmc_2, on the vertex $v_{(2,1)}$. The token is passed to $v_{(2,2)}$. An intersection between gmc_1 and gmc_2 is checked (upper GMCs), but none is found.
6. The SL continues to gmc_1, on the vertex $v_{(1,1)}$. The token is passed to $v_{(1,2)}$. The intersections with gmc_3 and gmc_2 are checked (upper and lower GMCs). The intersection i_2 is found. It is added to gmc_1 and gmc_3. The tokens are brought to i_2 for both GMCs.
7. The SL continues on gmc_1, on the intersection i_2, as it has the segment with the highest α. The token is passed to the following vertices in the GMCs [$v_{(1,2)}$ and $v_{(3,1)}$]. As there is no upper GMC over i_2, only the intersection between gmc_3 and gmc_2 is checked. Although the segments intersect in the final points, it is not taken into account, as the segments are considered half-open.
8. The SL continues on gmc_1, on the vertex $v_{(1,2)}$. The token is passed to $v_{(1,3)}$. As there is no upper GMC, only the intersection with gmc_3 is checked, giving a negative result.
9. The SL continues to gmc_2, on the vertex $v_{(2,2)}$. The token is passed to $v_{(2,3)}$. An intersection is found with gmc_3 (i_3) and a second with gmc_1 (i_4). The tokens are passed from $v_{(2,2)}$ and $v_{(3,1)}$ back to i_3 and to i_4 from $v_{(1,3)}$.
10. The SL continues on gmc_2, on the intersection i_3. The token is passed to $v_{(3,2)}$ and i_4. No intersection with gmc_1 is found, as the gmc_2 segment ends on i_4.
11. The SL continues on gmc_2, on the intersection i_4. The token is passed to $v_{(2,3)}$ and $v_{(1,3)}$. No intersection with is found.
12. And following. The SL continues scanning all the rightmost vertices, finding no intersection. Once done, the present GMC is removed from the SCL and the ACL.

The different steps of the algorithm are detailed in the following section.

3.3 *General Polygonal Chain Intersection Algorithm*

The full GPCI is resumed in Algorithm 5. It is based on the previous description.

Notice that the proposed implementation of the GPCI (Algorithm 3.3) forces the token to be in the following vertex of the GMC. The different steps are detailed in the following sections.

Algorithm 5 GPCI algorithm

```
1:  procedure GPCI(ACL)                          ▷ Determines the intersection between the GMCs
2:  while ACL is not empty do
3:    sort the GMCs in ACL
4:    gmc_a ← ACL[0]
5:    v_front ← gmc_a[token]
6:    advance the token in gmc_a
7:    switch v_front.type do
8:      case Leftmost
9:        add gmc_a in SCL
10:         sort SCL at v_front point
11:         find intersection between gmc_a and upper GMC
12:         find intersection between gmc_a and lower GMC
13:     case Internal
14:         sort SCL at v_front point
15:         find intersection between gmc_a and upper GMC
16:         find intersection between gmc_a and lower GMC
17:     case Rightmost
18:         sort SCL at v_front point
19:         find intersection between gmc_a and upper GMC
20:         find intersection between gmc_a and lower GMC
21:         find intersection between gmc_a's upper and lower GMC
22:         remove gmc_a from SCL
23:         remove gmc_a from ACL
24:     case Rightmost intersection
25:         mcs ← v_front chains
26:         sort the mcs chains by the segment gradient
27:         gmc_max ← GMC from mcs with maximum segment gradient
28:         gmc_min ← GMC from mcs with minimum segment gradient
29:         find intersection between gmc_max and upper GMC
30:         find intersection between gmc_min and lower GMC
31:         if gmc_max = gmc_a ∨ gmc_min = gmc_a then
32:           find intersection between gmc_a's upper and lower GMC
33:         remove gmc_a from SCL
34:         remove gmc_a from ACL
35:     case Intersection
36:         mcs ← v_front chains
37:         sort the mcs chains by the segment gradient
38:         gmc_max ← GMC from mcs with maximum segment gradient
39:         gmc_min ← GMC from mcs with minimum segment gradient
40:         find intersection between gmc_max and upper GMC
41:         find intersection between gmc_min and lower GMC
42:   return intersections
```

Algorithm 6 $<_{ACL}$ algorithm

```
1:  procedure <_ACL (v_rhs, v_lhs)        ▷ RHS and LHS vertex objects
2:      if v_lhs.x < v_rhs.x then return true
3:      if v_rhs.x < v_lhs.x then return false
4:      if v_lhs.y < v_rhs.y then return true
5:      if v_rhs.y < v_lhs.y then return false
6:      if v_lhs.type = Leftmost ∧ v_rhs.type ≠ Leftmost then return true
7:      if v_rhs.type = Leftmost ∧ v_lhs.type ≠ Leftmost then return false
8:      if v_lhs.type = Rightmost intersection ∧ v_rhs.type ≠ Rightmost intersection then return true
9:      if v_rhs.type = Rightmost intersection ∧ v_lhs.type ≠ Rightmost intersection then return false
10:     if v_lhs.type = Rightmost ∧ v_rhs.type ≠ Rightmost then return true
11:     if v_rhs.type = Rightmost ∧ v_lhs.type ≠ Rightmost then return false
12:     if v_lhs.type = Intersection ∧ v_rhs.type ≠ Intersection then return true
13:     if v_rhs.type = Intersection ∧ v_lhs.type ≠ Intersection then return false
14:     if v_lhs.type = Internal ∧ v_rhs.type ≠ Internal then return true
15:     if v_rhs.type = Internal ∧ v_lhs.type ≠ Internal then return false
16:     return false
```

3.4 *ACL and SCL Sorting*

The SL displaces from the leftmost vertex to the rightmost one. This process is done in a quantified manner, moving from one vertex, or intersection to another of the different GMCs. In the PCI algorithm, once all the GMCs of the different polygons have been added to the ACL, it is sorted by the x value of the first vertex on every GMC. The first element would be the one that has the leftmost x value. Also the type of vertex is used, making the left most go before. In the example with three GMCs shown in Fig. 8, it can be seen that a vertical segment has two vertices at the same x, requiring a modification to the algorithm.

To solve this issue, the GPCI uses the y for the sorting, having preference the ones with lower value. Also the rightmost intersection type is taken into account, as it shall have precedence over the rightmost type. These aspects have to be taken into account while the ACL is sorted, as it indicates the following vertex/intersection where the SL shall stop. For this purpose, a new less than ($<_{\mathrm{ACL}}$) operator has been defined. It is described in Algorithm 6. As said, it uses not only the position of the right-hand side (RHS) and left-hand side (LHS) vertices, but also their type.

The SL makes also a scan in the y direction, starting from the upper GMC in the present SL x position. The SCL is used for this purpose, as it stores the GMCs that are active on the present SL position. As in the PCI algorithm, the SCL is sorted by the y position in the present x, and the angle α of the segment associated with the present position. In case of vertical segments, the y value is not determined. For this reason, two modifications have been made, fixing the y on vertical segments to its lowest y (remember that segments are half-open), and adding to the sorting of the SCL the ending vertex of the present segment. The intersections are checked between the GMC of the present vertex or the GMCs of the present intersection and the upper and lower one. The SCL sorting uses a different less than operator ($<_{\mathrm{SCL}}$). It is described in Algorithm 7.

The operator assumes that the GMC object is able to determine the y and sine of α at the present x position ($\mathrm{gmc}[x].y$ and $\mathrm{mc}[x].\mathrm{sine}$, respectively). Also in case two segments are vertical

Algorithm 7 $<_{\text{SCL}}$ algorithm

1: **procedure** $<_{\text{SCL}}$ $(gmc_{rhs}, gmc_{lhs}, x)$ ▷ RHS and LHS GMCs objects, the position x
2: **if** $gmc_{rhs}[x].y < gmc_{lhs}[x].y$ **then return** true
3: **if** $gmc_{lhs}[x].y < gmc_{rhs}[x].y$ **then return** false
4: **if** $gmc_{rhs}[x].sine < gmc_{lhs}[x].sine$ **then return** true
5: **if** $gmc_{lhs}[x].sine < gmc_{rhs}[x].sine$ **then return** false
6: **if** $gmc_{lhs}[token].x < gmc_{rhs}[token].x$ **then return** true
7: **if** $gmc_{rhs}[token].x < gmc_{lhs}[token].x$ **then return** false
8: **if** $gmc_{lhs}[token].y < gmc_{rhs}[token].y$ **then return** true
9: **if** $gmc_{rhs}[token].y < gmc_{lhs}[token].y$ **then return** false
10: **return** false

and start at the same vertex, they are ordered starting from the one that goes further down, ensuring that no intersection is lost.

To ensure the consistency of the sine the operator $\not<_{d[\text{bits}]}$ is also used to check the verticality of the segment:

$$\text{sine} = \begin{cases} 1, & \text{if } \hat{u}.x \not<_{d[\text{bits}]} 0 \wedge 0 \not<_{d[\text{bits}]} \hat{u}.x \\ \hat{u}.y, & \text{otherwise} \end{cases}, \tag{12}$$

where both components of the segment unit vector ($\hat{u}$) are used.

3.5 *Finding Intersections*

To find the intersections between GMCs, three different scenarios are considered as follows.

- *General case*. The present point is not a rightmost point in the GMC it belongs, and it is checked against a second GMC (Algorithm 8).
- *Rightmost general case*. The present point is the rightmost vertex of the present GMC and is checked against a segment in a second GMC (Algorithm 9).
- *Rightmost close case*. The rightmost vertex of the present GMC is close enough to the rightmost vertex of the second GMC segment to be considered the same vertex. In this case, the half-open segment rule is broken to avoid loosing the intersection (Algorithm 10).

As described in Algorithm 8, the intersections have always to come from the present GMC. For this reason, the T intersection that is the starting point of the second GMC and the same case for the T overlap are not considered. If any of these intersections is detected, it shall be considered an error.

Notice also the behaviour for an X intersection. There are pathological cases where the intersection vertex is inside the segment s_s, but the operator $<_{2D}$ indicates that it is just after the token vertex v_{ts}. When this is found, the token vertex is substituted by the detected intersection.

In Algorithm 9, only the rightmost vertex of the present GMC is available. For this reason, a trick is used to detect the intersection, using the token vertex in the second GMC as the second vertex of the present segment. With this configuration, only the T overlap with s is feasible. Any other result shall be considered an error.

This three procedures (Algorithms 8 to 10) should be called depending on the vertex type, as indicated previously.

Algorithm 8 General intersection case

1: **procedure** *generalIntersection*(gmc_p, gmc_s, *intersections*) ▷ Present and second GMCs, and the intersections list
2: $v_p \leftarrow gmc_p[token-1]$
3: $v_s \leftarrow gmc_s[token-1]$
4: $v_{tp} \leftarrow gmc_p[token]$
5: $v_{ts} \leftarrow gmc_s[token]$
6: $s_p \leftarrow segment(v_p, v_{tp})$
7: $s_s \leftarrow segment(v_s, v_{ts})$;
8: $intersection \leftarrow intersect(s_p, s_s)$
9: **switch** $intersection.type$ **do**
10: **case** *No intersection*
11: **return**
12: **case** *V intersection* ∨ *V overlap* ▷ Figs. 6(d), 6(h)
13: $v_s.type \leftarrow$ *Intersection*
14: add gmc_p to $v_s.chains$
15: substitute v_p by v_s in gmc_p
16: **if** gmc_p and gmc_s are from different polygons **then**
17: add v_s to *intersections*
18: **case** *X intersection* ▷ Fig. 6(a)
19: create v_i
20: $v_i.point \leftarrow intersection.point$
21: $v_i.type \leftarrow$ *intersection*
22: add gmc_p to $v_i.chains$
23: add gmc_s to $v_i.chains$
24: insert v_i in gmc_p
25: **if** $v_i <_{2D} v_{ts}$ **then**
26: insert v_i in gmc_s
27: **else**
28: substitute v_{ts} by v_i in gmc_s
29: add v_i to *intersections*
30: **case** *T intersection with s* ∨ *T overlap with s* ▷ Figs. 6(b), 6(f)
31: $v_p.type \leftarrow$ *Intersection*
32: add gmc_s to $v_p.chains$
33: insert v_p in gmc_s
34: add v_p to *intersections*
35: **return** ▷ The definition of the GMC makes the cases c, e and g in Fig. 6 not valid

Algorithm 9 Rightmost general case

1: **procedure** *rightmostInter* sec *tion*($gmc_p, gmc_s, inter$ sec *tions*) ▷ Present and second GMCs, and the intersections list

2: $v_p \leftarrow gmc_p[token - 1]$

3: $v_s \leftarrow gmc_s[token - 1]$

4: $v_{ts} \leftarrow gmc_s[token]$ $s_p \leftarrow segment(v_p, v_{ts})$ ▷ There are no more points in the present GMC. For this reason, we use the following vertex from the second GMC (v_{ts}) and we only consider three cases.

5: $s_s \leftarrow segment(v_s, v_{ts})$

6: $inter$ sec $tion \leftarrow inter$ sec $t(s_p, s_s)$;

7: **switch** $inter$ sec $tion.type$

8: **case** *No intersection*

9: **return**

10: **case** *T overlap with s* ▷ Fig. 6 (f)

11: $v_p.type \leftarrow$ *Intersection*

12: add gmc_s to $v_p.chains$

13: insert v_p in gmc_s

14: add v_p to $inter$ sec $tions$

15: **return** ▷ The cases a, b, c, d, e, g and h in Fig. 6 are not valid

Algorithm 10 Rightmost close case

1: **procedure** *rightmostsClose*($gmc_p, gmc_s, inter$ sec *tions*) ▷ Present and second GMCs, and the intersections list

2: $v_s \leftarrow gmc_s[token]$

3: $v_s.type \leftarrow$ *Rightmost Intersection*

4: add gmc_p to $v_s.chains$

5: substitute v_p by v_s in gmc_p

6: add v_s to $inter$ sec $tions$

7: **return**

4 Results

The GPCI algorithm has been compared with the naive one (pairwise checking the segments) on two platforms: MacOS and Linux. Both algorithms have been programmed in C++ using the same intersection algorithm (2) and have been run in the same program sequentially. The number of intersections detected is the same for the GPCI and naive algorithms. The points are coincident within the precision margin. The algorithm implementations make extensive use of the Standard Template Library (STL). In both cases, the native compiler has been used: clang in MacOS and gcc in linux. The optimization flag "–O3" has been used also in both platforms. The hardware is different, so we are going to focus on the ratios between both algorithms running times.

Table 2 GPCI results compared with naive algorithm ones.

OS	Algorithm	Polygons (Fig. 9) (ms)	No intersection (Fig. 2) (ms)	Collision (Fig. 10) (ms)
MacOS	Naive	0.7	0.93	0.99
	GPCI	1.0	0.16	0.28
Linux	Naive	0.6	0.99	1.02
	GPCI	0.9	0.22	0.35

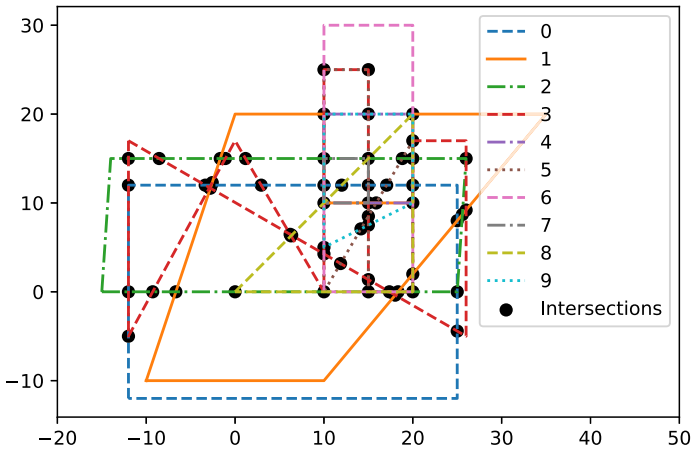


Fig. 9 Intersections calculated for 10 complex polygons.

The results are similar, as seen in Table 2. For this reason, the following values shown will be the MacOS ones.

In the test polygons, the algorithm finds the intersections for a complex scenario like the one shown in Fig. 9. In this case, the number of polygon segments is 52, whereas the number of chains is 26, and there are 60 intersections. It is a worse case scenario for the GPCI, and the naive algorithm has slightly better results: 1.0 and 0.7 ms, respectively. A profile analysis shows that this difference mainly comes from the time required by the sorting processes that are done on every step on the ACL and the SCL.

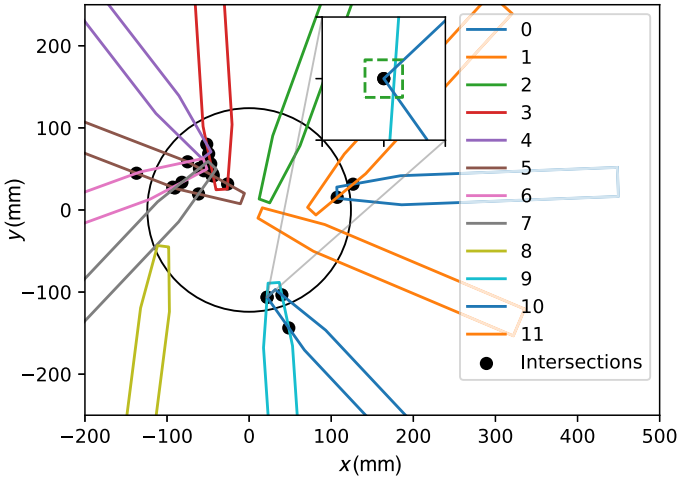


Fig. 10 Intersections calculated in a collision configuration for the MIRADAS probe-arms. In the inset there is a detail of the intersection between probe-arms 9 and 10, and the error limits (dashed square). The results is that two intersections are considered one.

To validate the algorithm, a Fuzz test[14] has been done. Random configurations for the MIRADAS MXS have been generated, and the results compared. In this case (e.g., Fig. 2), the number of segments is 72 and the number of chains 24, which helps the GPCI.

In all the cases tested, the calculation times are better for the GPCI algorithm compared to the naive one. If there is no intersection (Fig. 2), the time needed by the GPCI is 0.16 ms, whereas the naive one needs around 0.93 ms. In the case presented in Fig. 10, the number of intersections is 22, being the collision between five probes (3 to 7) an unlikely situation. In this extreme case, these values are 0.28 ms for the GPCI, whereas the naive is similar, 0.99 ms.

The Fuzz test also showed intersections like the one between the arms 9 and 10 (zoomed in Fig. 10 inset). The determination of the number of intersections depends on the required precision. In this case, two intersection points are considered one, as both are inside the vertex error limits (showed as a dashed square). The use of Eq. (6) ensures that the points determined are coherent with the precision required, ensuring the same results for the GPCI and naive algorithms.

5 Conclusions

The GPCI algorithm has been presented. It allows to calculate intersection between multiple complex polygons. Its usage to calculate the collisions between the probes of an MXS has been presented. The different steps have been described, and improvements added to the PCI to allow its usage in complex polygons provided. We can conclude that GPCI performs best when the number of segments per GMC is over 2. On the contrary, in a scenario where this ratio is lower, the performance of the naive solution is slightly better. In the particular case of the robotic arms of MIRADAS, where the number of segments per chain is equal to or larger than this threshold, the performance improvement using the GPCI has been found to be between 3 and 4 times better, depending on the number of intersections.

Acknowledgments

The authors acknowledge the financial support from the State Agency for Research of the Spanish Ministry of Science and Innovation through the "Unit of Excellence María de Maeztu 2020–2023" award to the Institute of Cosmos Sciences (No. CEX2019-000918-M). The authors have no relevant financial interests in the manuscript and no other potential conflicts of interest to disclose.

References

1. J. Sabater et al., "Roadmap search based motion planning for MIRADAS probe arms," *J. Astron. Telesc. Instrum. Syst.* **4**(3), 034401 (2018).
2. M. I. Shamos, "Geometric complexity," in *Proc. Seventh Annu. ACM Symp. Theory Comput., STOC '75*, Association for Computing Machinery, New York, pp. 224–233 (1975).
3. J. O'Rourke et al., "A new linear algorithm for intersecting convex polygons," *Comput. Graphics Image Process.* **19**(4), 384–391 (1982).
4. D. P. Dobkin and D. L. Souvaine, "Detecting the intersection of convex objects in the plane," *Comput. Aided Geometr. Design* **8**(3), 181–199 (1991).
5. S. C. Park and H. Shin, "Polygonal chain intersection," *Comput. Graphics* **26**(2), 341–350 (2002).
6. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer Science & Business Media (1985).
7. F. P. Preparata and K. J. Supowit, "Testing a simple polygon for monotonicity," *Inf. Process. Lett.* **12**(4), 161–164 (1981).
8. T. T. El-Midany, A. Elkeran, and H. Tawfik, "A sweep-line algorithm and its application to spiral pocketing," *Int. J. CAD/CAM* **2**(1), 23–38 (2002).

9. "IEEE Std 754–2008—IEEE standard for floating-point arithmetic," IEEE (2008).
10. E. L. Foster, K. Hormann, and R. T. Popa, "Clipping simple polygons with degenerate intersections," *Comput. Graphics: X* **2**, 100007 (2019).
11. E. D. Jou, *Determine Whether Two Line Segments Intersect*, pp. 265–274, Springer, Tokyo (1991).
12. M. I. Shamos and D. Hoey, "Geometric intersection problems," in *17th Annu. Symp. Found. Comput. Sci. (SFCS 1976)*, pp. 208–215 (1976).
13. S. C. Park, H. Shin, and B. K. Choi, *A Sweep Line Algorithm for Polygonal Chain Intersection and Its Applications*, pp. 309–321. Springer US, Boston, Massachusetts (2001).
14. M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley Professional (2007).

David Roma received his PhD in electronic engineering in 2019 from the Polytechnic University of Catalonia. He is an assistant professor of the Department of Electronic and Biomedical Engineering at the University of Barcelona (UB). He has worked as an electronics engineer at the Polarimetric and Helioseismic Imager (SO/PHI) Instrument for Solar Orbiter (SO) Mission, developing and testing the correlation camera hardware and firmware. He has been also deeply involved in the validation and software tools for the full image stabilization system. At the same time, he has also been working in the UPC-IonSAT Group under the direction of Manuel Hernandez Pajares, with topics related to real-time ionospheric maps generation and validation. He has six published articles in peer-reviewed papers, six conference presentations, and involved in three international projects related to space science and ionosphere. He is now working at the Institute of Space Sciences from the Spanish National Research Council in the LISA mission.

Jose Bosch received his PhD in electronics and applied physics and his MSc degree in materials science from UB. He is a professor of the Department of Electronic and Biomedical Engineering at UB. From 1987 to 1993, the research field was the analysis and characterization of photoluminiscent semiconductors and quantum wells electronic-based devices. Since 1994, he has been working in the design of digital systems, mainly with microcontrollers and embedded systems, and smart instrumentation. He is a member of the team that developed the Image Stabilization System (ISS) for the SO/PHI Instrument for SO Mission.

Josep Sabater received his BS and MS degrees in computer science and computer engineering from the University Ramon Llull, Barcelona, Spain, in 2000 and 2001, respectively, his MS degree in electronics from the University of Barcelona, Spain in 2010, and his PhD in industrial, computer science, and environmental engineering in 2019 from the University of La Laguna. In 2010, he joined the Institute of Space Studies of Catalonia (IEEC). Since then, he has been working on electronics and control software for satellites and ground telescopes. He was previously involved in the design of the ISS for SO/PHI. Furthermore, he is currently developing the control system for the cryogenic probe arms of MIRADAS instrument for Gran Telescopio Canarias (GTC). His current research interests include motion planning, robotics, control systems, and artificial intelligence.

Jose M. Gomez is an associate professor of the Department of Electronic and Biomedical Engineering at UB. He is also a member of the Institute of Cosmos Sciences at University of Barcelona (ICCUB) and the IEEC. His research is focused on the development of ground-based and space borne instrumentation for telescopes. He is a principal investigator of the UB contribution to the ISS for SO/PHI, and the IEEC contribution to the Ariel Mission and the instrument MIRADAS for the GTC. He has also worked at IBM.