orient

Release 7.0.1-alpha

Azat Ibrakov

Jun 24, 2023

CONTENTS

Note: If object is not listed in documentation it should be considered as implementation detail that can change and should not be relied upon.

orient.planar.**point_in_segment**(*point: Point*, *segment: Segment*, ***, *context: Optional[Context] = None*) → Location

Finds location of point in segment.

Time complexity:
O(1)

Memory complexity:
O(1)

Parameters

- **point** – point to check for.
- **segment** – segment to check in.
- **context** – geometric context.

Returns
location of point in segment.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> point_in_segment(Point(0, 0), segment) is Location.BOUNDARY
True
>>> point_in_segment(Point(1, 0), segment) is Location.BOUNDARY
True
>>> point_in_segment(Point(2, 0), segment) is Location.BOUNDARY
True
>>> point_in_segment(Point(3, 0), segment) is Location.EXTERIOR
True
>>> point_in_segment(Point(0, 1), segment) is Location.EXTERIOR
True
```

orient.planar.**segment_in_segment**(*left: Segment*, *right: Segment*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between segments.

Time complexity:
O(1)

Memory complexity:
O(1)

Parameters

- **left** – segment to check for.
- **right** – segment to check in.
- **context** – geometric context.

Returns
relation between segments.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> segment = Segment(Point(0, 0), Point(2, 0))
>>> (segment_in_segment(Segment(Point(0, 0), Point(0, 2)), segment)
...  is Relation.TOUCH)
True
>>> (segment_in_segment(Segment(Point(0, 0), Point(1, 0)), segment)
...  is Relation.COMPONENT)
True
>>> (segment_in_segment(Segment(Point(0, 0), Point(2, 0)), segment)
...  is Relation.EQUAL)
True
>>> (segment_in_segment(Segment(Point(0, 0), Point(3, 0)), segment)
...  is Relation.COMPOSITE)
True
>>> (segment_in_segment(Segment(Point(1, 0), Point(3, 0)), segment)
...  is Relation.OVERLAP)
True
>>> (segment_in_segment(Segment(Point(2, 0), Point(3, 0)), segment)
...  is Relation.TOUCH)
True
>>> (segment_in_segment(Segment(Point(3, 0), Point(4, 0)), segment)
...  is Relation.DISJOINT)
True
```

orient.planar.**point_in_multisegment**(*point: Point*, *multisegment: Multisegment*, ***, *context: Optional[Context] = None*) → Location

Finds location of point in multisegment.

Time complexity:
`O(len(multisegment.segments))`

Memory complexity:
`O(1)`

Parameters

- **point** – point to check for.
- **multisegment** – multisegment to check in.
- **context** – geometric context.

Returns
location of point in multisegment.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
```

(continues on next page)

(continued from previous page)

```
>>> Segment = context.segment_cls
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 0)),
...                              Segment(Point(3, 0), Point(5, 0))])
>>> point_in_multisegment(Point(0, 0), multisegment) is Location.BOUNDARY
True
>>> point_in_multisegment(Point(0, 1), multisegment) is Location.EXTERIOR
True
>>> point_in_multisegment(Point(1, 0), multisegment) is Location.BOUNDARY
True
>>> point_in_multisegment(Point(2, 0), multisegment) is Location.EXTERIOR
True
>>> point_in_multisegment(Point(3, 0), multisegment) is Location.BOUNDARY
True
>>> point_in_multisegment(Point(4, 0), multisegment) is Location.BOUNDARY
True
```

orient.planar.**segment_in_multisegment**(*segment: Segment*, *multisegment: Multisegment*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between segment and multisegment.

Time complexity:
`O(segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = len(multisegment.segments)`.

Parameters

- **segment** – segment to check for.
- **multisegment** – multisegment to check in.
- **context** – geometric context.

Returns
relation between segment and multisegment.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> multisegment = Multisegment([Segment(Point(0, 0), Point(1, 1)),
...                              Segment(Point(1, 1), Point(3, 3))])
>>> segment_in_multisegment(Segment(Point(0, 0), Point(1, 0)),
...                         multisegment) is Relation.TOUCH
True
>>> segment_in_multisegment(Segment(Point(0, 0), Point(0, 1)),
...                         multisegment) is Relation.TOUCH
True
>>> segment_in_multisegment(Segment(Point(0, 1), Point(1, 0)),
...                         multisegment) is Relation.CROSS
True
>>> segment_in_multisegment(Segment(Point(0, 0), Point(1, 1)),
```

(continues on next page)

(continued from previous page)

```
...                           multisegment) is Relation.COMPONENT
True
>>> segment_in_multisegment(Segment(Point(0, 0), Point(3, 3)),
...                           multisegment) is Relation.EQUAL
True
>>> segment_in_multisegment(Segment(Point(2, 2), Point(4, 4)),
...                           multisegment) is Relation.OVERLAP
True
>>> segment_in_multisegment(Segment(Point(4, 4), Point(5, 5)),
...                           multisegment) is Relation.DISJOINT
True
```

orient.planar.**multisegment_in_multisegment**(*left: Multisegment*, *right: Multisegment*, *, *context: Optional[Context] = None*) → Relation

Finds relation between multisegments.

Time complexity:
`O(segments_count * log segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = len(left.segments) + len(right.segments)`.

Parameters

- **left** – multisegment to check for.
- **right** – multisegment to check in.
- **context** – geometric context.

Returns
relation between multisegments.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                 Segment(Point(0, 0), Point(0, 4)),
...                 Segment(Point(4, 0), Point(4, 4)),
...                 Segment(Point(0, 4), Point(4, 4))]
>>> inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                       Segment(Point(1, 3), Point(1, 1)),
...                       Segment(Point(3, 1), Point(3, 3)),
...                       Segment(Point(1, 3), Point(3, 3))]
>>> square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
...                     Segment(Point(2, 2), Point(4, 0)),
...                     Segment(Point(2, 2), Point(4, 4)),
...                     Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_multisegment(Multisegment(inner_square_edges),
...                               Multisegment(square_edges))
...  is Relation.DISJOINT)
True
```

(continues on next page)

(continued from previous page)

```
>>> (multisegment_in_multisegment(Multisegment(square_diagonals),
...                               Multisegment(square_edges))
...  is Relation.TOUCH)
True
>>> (multisegment_in_multisegment(Multisegment(square_diagonals),
...                               Multisegment(inner_square_edges))
...  is Relation.CROSS)
True
>>> (multisegment_in_multisegment(Multisegment(inner_square_edges
...                                            + [square_edges[0]]),
...                               Multisegment(square_edges))
...  is Relation.OVERLAP)
True
>>> (multisegment_in_multisegment(Multisegment(square_edges
...                                            + inner_square_edges),
...                               Multisegment(square_edges))
...  is Relation.COMPOSITE)
True
>>> (multisegment_in_multisegment(Multisegment(square_edges),
...                               Multisegment(square_edges))
...  is Relation.EQUAL)
True
>>> (multisegment_in_multisegment(Multisegment(square_edges),
...                               Multisegment(square_edges
...                                            + inner_square_edges))
...  is Relation.COMPONENT)
True
```

orient.planar.**point_in_contour**(*point: Point*, *contour: Contour*, **, context: Optional[Context] = None*) → Location

Finds location of point in contour.

Time complexity:
`O(len(contour.vertices))`

Memory complexity:
`O(1)`

Parameters

- **point** – point to check for.
- **contour** – contour to check in.
- **context** – geometric context.

Returns
location of point in contour.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> square = Contour([Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2)])
```

(continues on next page)

(continued from previous page)

```
>>> point_in_contour(Point(0, 0), square) is Location.BOUNDARY
True
>>> point_in_contour(Point(1, 1), square) is Location.EXTERIOR
True
>>> point_in_contour(Point(2, 2), square) is Location.BOUNDARY
True
>>> point_in_contour(Point(3, 3), square) is Location.EXTERIOR
True
```

orient.planar.**segment_in_contour**(*segment: Segment*, *contour: Contour*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between segment and contour.

Time complexity:
O(len(contour.vertices))

Memory complexity:
O(1)

Parameters

- **segment** – segment to check for.
- **contour** – contour to check in.
- **context** – geometric context.

Returns
relation between segment and contour.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(3, 0), Point(3, 3), Point(0, 3)])
>>> (segment_in_contour(Segment(Point(0, 0), Point(1, 0)), square)
...  is Relation.COMPONENT)
True
>>> (segment_in_contour(Segment(Point(0, 0), Point(3, 0)), square)
...  is Relation.COMPONENT)
True
>>> (segment_in_contour(Segment(Point(2, 0), Point(4, 0)), square)
...  is Relation.OVERLAP)
True
>>> (segment_in_contour(Segment(Point(4, 0), Point(5, 0)), square)
...  is Relation.DISJOINT)
True
>>> (segment_in_contour(Segment(Point(1, 0), Point(1, 2)), square)
...  is Relation.TOUCH)
True
>>> (segment_in_contour(Segment(Point(0, 0), Point(1, 1)), square)
...  is Relation.TOUCH)
True
```

(continues on next page)

(continued from previous page)

```
>>> (segment_in_contour(Segment(Point(1, 1), Point(2, 2)), square)
...  is Relation.DISJOINT)
True
>>> (segment_in_contour(Segment(Point(2, 2), Point(4, 4)), square)
...  is Relation.CROSS)
True
```

orient.planar.**multisegment_in_contour**(*multisegment: Multisegment*, *contour: Contour*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multisegment and contour.

Time complexity:
`O(segments_count * log segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = len(left.vertices) + len(right.vertices)`.

Parameters

- **multisegment** – multisegment to check for.
- **contour** – contour to check in.
- **context** – geometric context.

Returns
relation between multisegment and contour.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                 Segment(Point(0, 0), Point(0, 4)),
...                 Segment(Point(4, 0), Point(4, 4)),
...                 Segment(Point(0, 4), Point(4, 4))]
>>> inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                       Segment(Point(1, 3), Point(1, 1)),
...                       Segment(Point(3, 1), Point(3, 3)),
...                       Segment(Point(1, 3), Point(3, 3))]
>>> square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
...                     Segment(Point(2, 2), Point(4, 0)),
...                     Segment(Point(2, 2), Point(4, 4)),
...                     Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_contour(Multisegment(inner_square_edges), square)
...  is Relation.DISJOINT)
True
>>> (multisegment_in_contour(Multisegment(square_diagonals), square)
...  is Relation.TOUCH)
```

(continues on next page)

(continued from previous page)

```
True
>>> (multisegment_in_contour(Multisegment(square_diagonals), inner_square)
...  is Relation.CROSS)
True
>>> (multisegment_in_contour(
...      Multisegment(square_diagonals + [square_edges[0]]), square)
...  is Relation.OVERLAP)
True
>>> (multisegment_in_contour(Multisegment(square_diagonals + square_edges),
...                          square)
...  is Relation.COMPOSITE)
True
>>> (multisegment_in_contour(Multisegment(square_edges), square)
...  is Relation.EQUAL)
True
>>> (multisegment_in_contour(Multisegment(square_edges[1:]), square)
...  is Relation.COMPONENT)
True
```

orient.planar.**contour_in_contour**(*left: Contour*, *right: Contour*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between contours.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(left.vertices) + len(right.vertices)`.

Parameters

- **left** – contour to check for.
- **right** – contour to check in.
- **context** – geometric context.

Returns
relation between contours.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> triangle = Contour([Point(0, 0), Point(1, 0), Point(0, 1)])
>>> square = Contour([Point(0, 0), Point(1, 0), Point(1, 1), Point(0, 1)])
>>> contour_in_contour(triangle, triangle) is Relation.EQUAL
True
>>> contour_in_contour(triangle, square) is Relation.OVERLAP
True
>>> contour_in_contour(square, triangle) is Relation.OVERLAP
True
>>> contour_in_contour(square, square) is Relation.EQUAL
True
```

orient.planar.**point_in_region**(*point: Point*, *region: Contour*, *, *context: Optional[Context] = None*) → Location

Finds location of point in region.

Based on ray casting algorithm.

Time complexity:
O(len(region.vertices))

Memory complexity:
O(1)

Reference:
https://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm

Parameters

- **point** – point to check for.
- **region** – region to check in.
- **context** – geometric context.

Returns
location of point in region.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> square = Contour([Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2)])
>>> point_in_region(Point(0, 0), square) is Location.BOUNDARY
True
>>> point_in_region(Point(1, 1), square) is Location.INTERIOR
True
>>> point_in_region(Point(2, 2), square) is Location.BOUNDARY
True
>>> point_in_region(Point(3, 3), square) is Location.EXTERIOR
True
```

orient.planar.**segment_in_region**(*segment: Segment*, *region: Contour*, *, *context: Optional[Context] = None*) → Relation

Finds relation between segment and region.

Time complexity:
O(len(region.vertices))

Memory complexity:
O(1)

Parameters

- **segment** – segment to check for.
- **region** – region to check in.
- **context** – geometric context.

Returns
relation between segment and region.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(3, 0), Point(3, 3), Point(0, 3)])
>>> (segment_in_region(Segment(Point(0, 0), Point(1, 0)), square)
...  is Relation.COMPONENT)
True
>>> (segment_in_region(Segment(Point(0, 0), Point(3, 0)), square)
...  is Relation.COMPONENT)
True
>>> (segment_in_region(Segment(Point(2, 0), Point(4, 0)), square)
...  is Relation.TOUCH)
True
>>> (segment_in_region(Segment(Point(4, 0), Point(5, 0)), square)
...  is Relation.DISJOINT)
True
>>> (segment_in_region(Segment(Point(1, 0), Point(1, 2)), square)
...  is Relation.ENCLOSED)
True
>>> (segment_in_region(Segment(Point(0, 0), Point(1, 1)), square)
...  is Relation.ENCLOSED)
True
>>> (segment_in_region(Segment(Point(1, 1), Point(2, 2)), square)
...  is Relation.WITHIN)
True
>>> (segment_in_region(Segment(Point(2, 2), Point(4, 4)), square)
...  is Relation.CROSS)
True
```

orient.planar.**multisegment_in_region**(*multisegment: Multisegment*, *region: Contour*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multisegment and region.

Time complexity:
`O(segments_count * log segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = len(multisegment.segments) + len(region.vertices)`.

Parameters

- **multisegment** – multisegment to check for.
- **region** – region to check in.
- **context** – geometric context.

Returns
relation between multisegment and region.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
```

(continues on next page)

(continued from previous page)

```
>>> Contour = context.contour_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                 Segment(Point(0, 0), Point(0, 4)),
...                 Segment(Point(4, 0), Point(4, 4)),
...                 Segment(Point(0, 4), Point(4, 4))]
>>> inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                       Segment(Point(1, 3), Point(1, 1)),
...                       Segment(Point(3, 1), Point(3, 3)),
...                       Segment(Point(1, 3), Point(3, 3))]
>>> square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
...                     Segment(Point(2, 2), Point(4, 0)),
...                     Segment(Point(2, 2), Point(4, 4)),
...                     Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_region(Multisegment(square_edges), inner_square)
...  is Relation.DISJOINT)
True
>>> (multisegment_in_region(
...      Multisegment(square_edges + inner_square_edges), inner_square)
...  is Relation.TOUCH)
True
>>> (multisegment_in_region(Multisegment(square_diagonals), inner_square)
...  is Relation.CROSS)
True
>>> (multisegment_in_region(Multisegment(square_edges), square)
...  is Relation.COMPONENT)
True
>>> (multisegment_in_region(
...      Multisegment(square_edges + inner_square_edges), square)
...  is Relation.ENCLOSED)
True
>>> (multisegment_in_region(Multisegment(inner_square_edges), square)
...  is Relation.WITHIN)
True
```

orient.planar.**contour_in_region**(*contour: Contour*, *region: Contour*, *, *context: Optional[Context] = None*) → Relation

Finds relation between contour and region.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(contour.vertices) + len(region.vertices)`.

Parameters

- **contour** – contour to check for.

- **region** – region to check in.
- **context** – geometric context.

Returns
relation between contour and region.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> contour_in_region(square, inner_square) is Relation.DISJOINT
True
>>> contour_in_region(square, triangle) is Relation.TOUCH
True
>>> contour_in_region(inner_square, triangle) is Relation.CROSS
True
>>> contour_in_region(square, square) is Relation.COMPONENT
True
>>> contour_in_region(triangle, square) is Relation.ENCLOSED
True
>>> contour_in_region(inner_square, square) is Relation.WITHIN
True
```

orient.planar.**region_in_region**(*left: Contour*, *right: Contour*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between regions.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(left.vertices) + len(right.vertices)`.

Parameters

- **left** – region to check for.
- **right** – region to check in.
- **context** – geometric context.

Returns
relation between regions.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> neighbour_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                             Point(4, 4)])
```

(continues on next page)

(continued from previous page)

```
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (region_in_region(inner_square, neighbour_square)
...  is Relation.DISJOINT)
True
>>> region_in_region(square, neighbour_square) is Relation.TOUCH
True
>>> region_in_region(inner_square, triangle) is Relation.OVERLAP
True
>>> region_in_region(square, inner_square) is Relation.COVER
True
>>> region_in_region(square, triangle) is Relation.ENCLOSES
True
>>> region_in_region(square, square) is Relation.EQUAL
True
>>> region_in_region(triangle, square) is Relation.ENCLOSED
True
>>> region_in_region(inner_square, square) is Relation.WITHIN
True
```

orient.planar.**point_in_multiregion**(*point: Point*, *multiregion: Sequence[Contour]*, *, *context: Optional[Context] = None*) → Location

Finds location of point in multiregion.

Time complexity:

`O(sum(len(region.vertices) for region in multiregion))`

Memory complexity:

`O(1)`

Parameters

- **point** – point to check for.
- **multiregion** – multiregion to check in.
- **context** – geometric context.

Returns

location of point in multiregion.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> triangle = Contour([Point(0, 0), Point(2, 0), Point(0, 2)])
>>> square = Contour([Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2)])
>>> point_in_multiregion(Point(0, 0), [triangle]) is Location.BOUNDARY
True
>>> point_in_multiregion(Point(0, 0), [square]) is Location.BOUNDARY
True
>>> point_in_multiregion(Point(1, 1), [triangle]) is Location.BOUNDARY
True
```

(continues on next page)

(continued from previous page)

```
>>> point_in_multiregion(Point(1, 1), [square]) is Location.INTERIOR
True
>>> point_in_multiregion(Point(2, 2), [triangle]) is Location.EXTERIOR
True
>>> point_in_multiregion(Point(2, 2), [square]) is Location.BOUNDARY
True
```

orient.planar.**segment_in_multiregion**(*segment: Segment*, *multiregion: Sequence[Contour]*, *, *context: Optional[Context] = None*) → Relation

Finds relation between segment and multiregion.

Time complexity:
`O(segments_count * log segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = sum(len(region.vertices) for region in multiregion)`.

Parameters

- **segment** – segment to check for.
- **multiregion** – multiregion to check in.
- **context** – geometric context.

Returns
relation between segment and multiregion.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(3, 0), Point(3, 3), Point(0, 3)])
>>> (segment_in_multiregion(Segment(Point(0, 0), Point(1, 0)), [])
...  is Relation.DISJOINT)
True
>>> (segment_in_multiregion(Segment(Point(0, 0), Point(1, 0)), [square])
...  is Relation.COMPONENT)
True
>>> (segment_in_multiregion(Segment(Point(0, 0), Point(3, 0)), [square])
...  is Relation.COMPONENT)
True
>>> (segment_in_multiregion(Segment(Point(2, 0), Point(4, 0)), [square])
...  is Relation.TOUCH)
True
>>> (segment_in_multiregion(Segment(Point(4, 0), Point(5, 0)), [square])
...  is Relation.DISJOINT)
True
>>> (segment_in_multiregion(Segment(Point(1, 0), Point(1, 2)), [square])
...  is Relation.ENCLOSED)
True
>>> (segment_in_multiregion(Segment(Point(0, 0), Point(1, 1)), [square])
...  is Relation.ENCLOSED)
```

(continues on next page)

(continued from previous page)

```
True
>>> (segment_in_multiregion(Segment(Point(1, 1), Point(2, 2)), [square])
...  is Relation.WITHIN)
True
>>> (segment_in_multiregion(Segment(Point(2, 2), Point(4, 4)), [square])
...  is Relation.CROSS)
True
```

orient.planar.**multisegment_in_multiregion**(*multisegment: Multisegment*, *multiregion: Sequence[Contour]*, **, context: Optional[Context] = None*) → Relation

Finds relation between multisegment and multiregion.

Time complexity:

`O(segments_count * log segments_count)`

Memory complexity:

`O(segments_count)`

where `segments_count = len(multisegment.segments) + sum(len(region.vertices) for region in multiregion)`.

Parameters

- **multisegment** – multisegment to check for.
- **multiregion** – multiregion to check in.
- **context** – geometric context.

Returns

relation between multisegment and multiregion.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Segment = context.segment_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                          Point(4, 8)])
>>> second_inner_square = Contour([Point(5, 5), Point(7, 5), Point(7, 7),
...                                Point(5, 7)])
>>> first_square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                       Segment(Point(0, 0), Point(0, 4)),
...                       Segment(Point(4, 0), Point(4, 4)),
...                       Segment(Point(0, 4), Point(4, 4))]
>>> first_inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                             Segment(Point(1, 3), Point(1, 1)),
...                             Segment(Point(3, 1), Point(3, 3)),
...                             Segment(Point(1, 3), Point(3, 3))]
>>> first_square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
```

(continues on next page)

(continued from previous page)

```
...                             Segment(Point(2, 2), Point(4, 0)),
...                             Segment(Point(2, 2), Point(4, 4)),
...                             Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_multiregion(Multisegment(first_square_edges),
...                               [first_inner_square, second_inner_square])
...  is Relation.DISJOINT)
True
>>> (multisegment_in_multiregion(Multisegment(first_square_edges
...                                           + first_inner_square_edges),
...                               [first_inner_square, second_inner_square])
...  is Relation.TOUCH)
True
>>> (multisegment_in_multiregion(Multisegment(first_square_diagonals),
...                               [first_inner_square, second_inner_square])
...  is Relation.CROSS)
True
>>> (multisegment_in_multiregion(Multisegment(first_square_edges),
...                               [first_square, second_square])
...  is Relation.COMPONENT)
True
>>> (multisegment_in_multiregion(Multisegment(first_inner_square_edges),
...                               [first_square, second_square])
...  is Relation.WITHIN)
True
```

orient.planar.**contour_in_multiregion**(*contour: Contour*, *multiregion: Sequence[Contour]*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between contour and multiregion.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(contour.vertices) + sum(len(region.vertices) for region in multiregion)`.

Parameters

- **contour** – contour to check for.
- **multiregion** – multiregion to check in.
- **context** – geometric context.

Returns
relation between contour and multiregion.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
```

(continues on next page)

(continued from previous page)

```
...                             Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (contour_in_multiregion(first_square,
...                         [first_inner_square, second_inner_square])
...  is Relation.DISJOINT)
True
>>> (contour_in_multiregion(second_square, [first_square, third_square])
...  is Relation.TOUCH)
True
>>> (contour_in_multiregion(first_inner_square, [triangle, second_square])
...  is Relation.CROSS)
True
>>> (contour_in_multiregion(first_square, [first_square, third_square])
...  is Relation.COMPONENT)
True
>>> (contour_in_multiregion(triangle, [first_square, third_square])
...  is Relation.ENCLOSED)
True
>>> (contour_in_multiregion(first_inner_square,
...                         [first_square, third_square])
...  is Relation.WITHIN)
True
```

orient.planar.**region_in_multiregion**(*region: Contour*, *multiregion: Sequence[Contour]*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between region and multiregion.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(region.vertices) + sum(len(region.vertices) for region in multiregion)`.

Parameters

- **region** – region to check for.
- **multiregion** – multiregion to check in.
- **context** – geometric context.

Returns
relation between region and multiregion.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
```

(continues on next page)

(continued from previous page)

```
>>> Point = context.point_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> outer_square = Contour([Point(0, 0), Point(8, 0), Point(8, 8),
...                         Point(0, 8)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (region_in_multiregion(third_square,
...                        [first_inner_square, second_inner_square])
...  is Relation.DISJOINT)
True
>>> (region_in_multiregion(second_square, [first_square, third_square])
...  is Relation.TOUCH)
True
>>> (region_in_multiregion(first_square,
...                        [first_inner_square, second_inner_square])
...  is Relation.OVERLAP)
True
>>> (region_in_multiregion(outer_square,
...                        [first_inner_square, second_inner_square])
...  is Relation.COVER)
True
>>> (region_in_multiregion(outer_square, [first_square, third_square])
...  is Relation.ENCLOSES)
True
>>> (region_in_multiregion(triangle, [first_square, third_square])
...  is Relation.ENCLOSED)
True
>>> (region_in_multiregion(first_inner_square,
...                        [first_square, third_square])
...  is Relation.WITHIN)
True
```

orient.planar.**multiregion_in_multiregion**(*left: Sequence[Contour]*, *right: Sequence[Contour]*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multiregions.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = sum(len(region.vertices) for region in left) + sum(len(region.vertices) for region in right)`.

Parameters

- **left** – multiregion to check for.
- **right** – multiregion to check in.
- **context** – geometric context.

Returns

relation between multiregions.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> fourth_square = Contour([Point(0, 4), Point(4, 4), Point(4, 8),
...                          Point(0, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> third_inner_square = Contour([Point(5, 5), Point(7, 5), Point(7, 7),
...                               Point(5, 7)])
>>> (multiregion_in_multiregion([first_inner_square, third_inner_square],
...                             [second_square, fourth_square])
...  is Relation.DISJOINT)
True
>>> (multiregion_in_multiregion([first_square, third_square],
...                             [second_square, fourth_square])
...  is Relation.TOUCH)
True
>>> (multiregion_in_multiregion([first_square, third_inner_square],
...                             [first_inner_square, third_square])
...  is Relation.OVERLAP)
True
>>> (multiregion_in_multiregion([first_square, third_square],
...                             [first_inner_square, third_inner_square])
...  is Relation.COVER)
True
>>> (multiregion_in_multiregion([first_square, third_square],
...                             [first_square, third_inner_square])
...  is Relation.ENCLOSES)
True
>>> (multiregion_in_multiregion(
...      [first_inner_square, second_inner_square, third_inner_square],
...      [first_inner_square, second_inner_square])
...  is Relation.COMPOSITE)
True
>>> (multiregion_in_multiregion([first_square, third_square],
...                             [first_square, third_square])
...  is Relation.EQUAL)
```

(continues on next page)

(continued from previous page)

```
True
>>> (multiregion_in_multiregion(
...      [first_inner_square, second_inner_square],
...      [first_inner_square, second_inner_square, third_inner_square])
...  is Relation.COMPONENT)
True
>>> (multiregion_in_multiregion([first_square, third_inner_square],
...                             [first_square, third_square])
...  is Relation.ENCLOSED)
True
>>> (multiregion_in_multiregion([first_inner_square, third_inner_square],
...                             [first_square, third_square])
...  is Relation.WITHIN)
True
```

orient.planar.**point_in_polygon**(*point: Point*, *polygon: Polygon*, ***, *context: Optional[Context] = None*) → Location

Finds location of point in polygon.

Time complexity:
`O(vertices_count)`

Memory complexity:
`O(1)`

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **point** – point to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns
location of point in polygon.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> outer_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> (point_in_polygon(Point(0, 0), Polygon(inner_square, []))
...  is Location.EXTERIOR)
True
>>> (point_in_polygon(Point(0, 0), Polygon(outer_square, []))
...  is Location.BOUNDARY)
True
>>> (point_in_polygon(Point(1, 1), Polygon(inner_square, []))
...  is Location.BOUNDARY)
```

(continues on next page)

(continued from previous page)

```
True
>>> (point_in_polygon(Point(1, 1), Polygon(outer_square, []))
...  is Location.INTERIOR)
True
>>> (point_in_polygon(Point(2, 2), Polygon(outer_square, []))
...  is Location.INTERIOR)
True
>>> (point_in_polygon(Point(2, 2), Polygon(outer_square, [inner_square]))
...  is Location.EXTERIOR)
True
```

orient.planar.**segment_in_polygon**(*segment: Segment*, *polygon: Polygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between segment and polygon.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **segment** – segment to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns
relation between segment and polygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> outer_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> segment_in_polygon(Segment(Point(0, 0), Point(1, 0)),
...                    Polygon(outer_square, [])) is Relation.COMPONENT
True
>>> (segment_in_polygon(Segment(Point(0, 0), Point(1, 0)),
...                     Polygon(outer_square, [inner_square]))
...  is Relation.COMPONENT)
True
>>> segment_in_polygon(Segment(Point(0, 0), Point(2, 2)),
...                    Polygon(outer_square, [])) is Relation.ENCLOSED
True
>>> (segment_in_polygon(Segment(Point(0, 0), Point(2, 2)),
```

(continues on next page)

(continued from previous page)

```
...                      Polygon(outer_square, [inner_square]))
...  is Relation.CROSS)
True
>>> segment_in_polygon(Segment(Point(1, 1), Point(3, 3)),
...                    Polygon(outer_square, [])) is Relation.WITHIN
True
>>> (segment_in_polygon(Segment(Point(1, 1), Point(3, 3)),
...                      Polygon(outer_square, [inner_square]))
...  is Relation.TOUCH)
True
>>> segment_in_polygon(Segment(Point(0, 0), Point(4, 4)),
...                    Polygon(outer_square, [])) is Relation.ENCLOSED
True
>>> (segment_in_polygon(Segment(Point(0, 0), Point(4, 4)),
...                      Polygon(outer_square, [inner_square]))
...  is Relation.CROSS)
True
```

orient.planar.**multisegment_in_polygon**(*multisegment: Multisegment*, *polygon: Polygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multisegment and polygon.

Time complexity:

`O(segments_count * log segments_count)`

Memory complexity:

`O(segments_count)`

where `segments_count = len(multisegment.segments) + len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **multisegment** – multisegment to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns

relation between multisegment and polygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                 Segment(Point(0, 0), Point(0, 4)),
...                 Segment(Point(4, 0), Point(4, 4)),
...                 Segment(Point(0, 4), Point(4, 4))]
```

(continues on next page)

(continued from previous page)

```
>>> inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                       Segment(Point(1, 3), Point(1, 1)),
...                       Segment(Point(3, 1), Point(3, 3)),
...                       Segment(Point(1, 3), Point(3, 3))]
>>> square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
...                     Segment(Point(2, 2), Point(4, 0)),
...                     Segment(Point(2, 2), Point(4, 4)),
...                     Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_polygon(Multisegment(square_edges),
...                          Polygon(inner_square, []))
...  is Relation.DISJOINT)
True
>>> (multisegment_in_polygon(Multisegment(square_edges
...                                       + inner_square_edges),
...                          Polygon(inner_square, []))
...  is Relation.TOUCH)
True
>>> (multisegment_in_polygon(Multisegment(square_diagonals),
...                          Polygon(inner_square, []))
...  is Relation.CROSS)
True
>>> (multisegment_in_polygon(Multisegment(square_edges),
...                          Polygon(square, []))
...  is Relation.COMPONENT)
True
>>> (multisegment_in_polygon(Multisegment(square_edges
...                                       + inner_square_edges),
...                          Polygon(square, []))
...  is Relation.ENCLOSED)
True
>>> (multisegment_in_polygon(Multisegment(inner_square_edges),
...                          Polygon(square, []))
...  is Relation.WITHIN)
True
```

orient.planar.**contour_in_polygon**(*contour: Contour*, *polygon: Polygon*, *, *context: Optional[Context] = None*) → Relation

Finds relation between contour and polygon.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = len(contour.vertices) + len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **contour** – contour to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns
relation between contour and polygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (contour_in_polygon(square, Polygon(inner_square, []))
...  is Relation.DISJOINT)
True
>>> contour_in_polygon(square, Polygon(triangle, [])) is Relation.TOUCH
True
>>> (contour_in_polygon(inner_square, Polygon(triangle, []))
...  is Relation.CROSS)
True
>>> contour_in_polygon(square, Polygon(square, [])) is Relation.COMPONENT
True
>>> contour_in_polygon(triangle, Polygon(square, [])) is Relation.ENCLOSED
True
>>> (contour_in_polygon(inner_square, Polygon(square, []))
...  is Relation.WITHIN)
True
```

orient.planar.**region_in_polygon**(*region: Contour*, *polygon: Polygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between region and polygon.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = len(region.vertices) + len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **region** – region to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns
relation between region and polygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
```

(continues on next page)

(continued from previous page)

```
>>> square = Contour([Point(0, 0), Point(4, 0), Point(4, 4), Point(0, 4)])
>>> neighbour_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                             Point(4, 4)])
>>> inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                         Point(1, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (region_in_polygon(inner_square, Polygon(neighbour_square, []))
...  is Relation.DISJOINT)
True
>>> (region_in_polygon(square, Polygon(neighbour_square, []))
...  is Relation.TOUCH)
True
>>> (region_in_polygon(inner_square, Polygon(triangle, []))
...  is Relation.OVERLAP)
True
>>> region_in_polygon(square, Polygon(inner_square, [])) is Relation.COVER
True
>>> region_in_polygon(square, Polygon(triangle, [])) is Relation.ENCLOSES
True
>>> region_in_polygon(square, Polygon(square, [])) is Relation.EQUAL
True
>>> region_in_polygon(triangle, Polygon(square, [])) is Relation.ENCLOSED
True
>>> region_in_polygon(inner_square, Polygon(square, [])) is Relation.WITHIN
True
```

orient.planar.**multiregion_in_polygon**(*multiregion: Sequence[Contour]*, *polygon: Polygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multiregion and polygon.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = sum(len(region.vertices) for region in multiregion) + len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`.

Parameters

- **multiregion** – multiregion to check for.
- **polygon** – polygon to check in.
- **context** – geometric context.

Returns
relation between multiregion and polygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
```

(continues on next page)

(continued from previous page)

```
...                           Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                           Point(4, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> outer_square = Contour([Point(0, 0), Point(8, 0), Point(8, 8),
...                           Point(0, 8)])
>>> (multiregion_in_polygon([first_square, third_square],
...                           Polygon(second_inner_square, []))
...  is Relation.DISJOINT)
True
>>> (multiregion_in_polygon([first_inner_square, second_inner_square],
...                           Polygon(first_square, [first_inner_square]))
...  is Relation.TOUCH)
True
>>> (multiregion_in_polygon([first_inner_square, second_inner_square],
...                           Polygon(first_square, []))
...  is Relation.OVERLAP)
True
>>> (multiregion_in_polygon([first_square, second_inner_square],
...                           Polygon(first_inner_square, []))
...  is Relation.COVER)
True
>>> (multiregion_in_polygon([first_square, second_inner_square],
...                           Polygon(first_square, [first_inner_square]))
...  is Relation.ENCLOSES)
True
>>> (multiregion_in_polygon([first_square, second_inner_square],
...                           Polygon(first_square, []))
...  is Relation.COMPOSITE)
True
>>> (multiregion_in_polygon([first_square, second_inner_square],
...                           Polygon(outer_square, []))
...  is Relation.ENCLOSED)
True
>>> (multiregion_in_polygon([first_inner_square, second_inner_square],
...                           Polygon(outer_square, []))
...  is Relation.WITHIN)
True
```

orient.planar.**polygon_in_polygon**(*left: Polygon*, *right: Polygon*, *, *context: Optional[Context] = None*) → Relation

Finds relation between polygons.

Time complexity:

```
O(vertices_count * log vertices_count)
```

Memory complexity:

```
O(vertices_count)
```

where `vertices_count = len(left.border.vertices) + sum(len(hole.vertices) for hole in left.holes) + len(right.border.vertices) + sum(len(hole.vertices) for hole in right.holes)`.

Parameters

- **left** – polygon to check for.
- **right** – polygon to check in.
- **context** – geometric context.

Returns

relation between polygons.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> outer_square = Contour([Point(0, 0), Point(7, 0), Point(7, 7),
...                         Point(0, 7)])
>>> inner_square = Contour([Point(1, 1), Point(6, 1), Point(6, 6),
...                         Point(1, 6)])
>>> innermore_square = Contour([Point(2, 2), Point(5, 2), Point(5, 5),
...                             Point(2, 5)])
>>> innermost_square = Contour([Point(3, 3), Point(4, 3), Point(4, 4),
...                             Point(3, 4)])
>>> (polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                     Polygon(innermore_square, []))
...  is polygon_in_polygon(Polygon(innermore_square, []),
...                        Polygon(outer_square, [inner_square]))
...  is polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                        Polygon(innermore_square, [innermost_square]))
...  is polygon_in_polygon(Polygon(innermore_square, [innermost_square]),
...                        Polygon(outer_square, [inner_square]))
...  is Relation.DISJOINT)
True
>>> (polygon_in_polygon(Polygon(inner_square, []),
...                     Polygon(outer_square, [inner_square]))
...  is polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                        Polygon(inner_square, []))
...  is polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                        Polygon(inner_square, [innermore_square]))
...  is polygon_in_polygon(Polygon(inner_square, [innermore_square]),
...                        Polygon(outer_square, [inner_square]))
...  is Relation.TOUCH)
True
>>> (polygon_in_polygon(Polygon(inner_square, []),
...                     Polygon(outer_square, [innermore_square]))
...  is polygon_in_polygon(Polygon(outer_square, [innermore_square]),
...                        Polygon(inner_square, []))
...  is polygon_in_polygon(Polygon(outer_square, [innermore_square]),
...                        Polygon(inner_square, [innermost_square]))
...  is polygon_in_polygon(Polygon(inner_square, [innermost_square]),
...                        Polygon(outer_square, [innermore_square]))
```

(continues on next page)

(continued from previous page)

```
...  is Relation.OVERLAP)
True
>>> (polygon_in_polygon(Polygon(outer_square, []),
...                     Polygon(inner_square, []))
...  is polygon_in_polygon(Polygon(outer_square, [innermost_square]),
...                        Polygon(inner_square, [innermore_square]))
...  is Relation.COVER)
True
>>> (polygon_in_polygon(Polygon(outer_square, []),
...                     Polygon(outer_square, [inner_square]))
...  is polygon_in_polygon(Polygon(outer_square, [innermore_square]),
...                        Polygon(outer_square, [inner_square]))
...  is polygon_in_polygon(Polygon(outer_square, [innermore_square]),
...                        Polygon(inner_square, [innermore_square]))
...  is Relation.ENCLOSES)
True
>>> (polygon_in_polygon(Polygon(outer_square, []),
...                     Polygon(outer_square, []))
...  is polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                        Polygon(outer_square, [inner_square]))
...  is Relation.EQUAL)
True
>>> (polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                     Polygon(outer_square, []))
...  is polygon_in_polygon(Polygon(outer_square, [inner_square]),
...                        Polygon(outer_square, [innermore_square]))
...  is polygon_in_polygon(Polygon(inner_square, [innermore_square]),
...                        Polygon(outer_square, [innermore_square]))
...  is Relation.ENCLOSED)
True
>>> (polygon_in_polygon(Polygon(inner_square, []),
...                     Polygon(outer_square, []))
...  is polygon_in_polygon(Polygon(inner_square, [innermore_square]),
...                        Polygon(outer_square, [innermost_square]))
...  is Relation.WITHIN)
True
```

orient.planar.**point_in_multipolygon**(*point: Point*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds location of point in multipolygon.

Time complexity:

`O(sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)) for polygon in multipolygon.polygons))`

Memory complexity:

`O(1)`

Parameters

- **point** – point to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns
location of point in multipolygon.

```
>>> from ground.base import Location, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                          Point(4, 8)])
>>> (point_in_multipolygon(Point(6, 2),
...                        Multipolygon([Polygon(first_square, []),
...                                      Polygon(second_square, [])]))
...  is Location.EXTERIOR)
True
>>> (point_in_multipolygon(Point(4, 4),
...                        Multipolygon([Polygon(first_square, []),
...                                      Polygon(second_square, [])]))
...  is Location.BOUNDARY)
True
>>> (point_in_multipolygon(Point(2, 2),
...                        Multipolygon([Polygon(first_square, []),
...                                      Polygon(second_square, [])]))
...  is Location.INTERIOR)
True
```

orient.planar.**segment_in_multipolygon**(*segment: Segment*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between segment and multipolygon.

Time complexity:
`O(segments_count * log segments_count)`

Memory complexity:
`O(segments_count)`

where `segments_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **segment** – segment to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns
relation between segment and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
```

(continues on next page)

(continued from previous page)

```
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                          Point(4, 8)])
>>> (segment_in_multipolygon(Segment(Point(2, 5), Point(2, 9)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.DISJOINT)
True
>>> (segment_in_multipolygon(Segment(Point(2, 4), Point(2, 8)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.TOUCH)
True
>>> (segment_in_multipolygon(Segment(Point(2, 2), Point(2, 6)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.CROSS)
True
>>> (segment_in_multipolygon(Segment(Point(2, 4), Point(6, 4)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.COMPONENT)
True
>>> (segment_in_multipolygon(Segment(Point(3, 3), Point(5, 5)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.ENCLOSED)
True
>>> (segment_in_multipolygon(Segment(Point(1, 1), Point(3, 3)),
...                          Multipolygon([Polygon(first_square, []),
...                                        Polygon(second_square, [])]))
...  is Relation.WITHIN)
True
```

orient.planar.**multisegment_in_multipolygon**(*multisegment: Multisegment*, *multipolygon: Multipolygon*, *, *context: Optional[Context] = None*) → Relation

Finds relation between multisegment and multipolygon.

Time complexity:

`O(segments_count * log segments_count)`

Memory complexity:

`O(segments_count)`

where `segments_count = len(multisegment.segments) + multipolygon_segments_count`, `multipolygon_segments_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **multisegment** – multisegment to check for.

- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns

relation between multisegment and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Multisegment = context.multisegment_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Segment = context.segment_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                          Point(4, 8)])
>>> second_inner_square = Contour([Point(5, 5), Point(7, 5), Point(7, 7),
...                                Point(5, 7)])
>>> first_square_edges = [Segment(Point(0, 0), Point(4, 0)),
...                       Segment(Point(0, 0), Point(0, 4)),
...                       Segment(Point(4, 0), Point(4, 4)),
...                       Segment(Point(0, 4), Point(4, 4))]
>>> first_inner_square_edges = [Segment(Point(1, 1), Point(3, 1)),
...                             Segment(Point(1, 3), Point(1, 1)),
...                             Segment(Point(3, 1), Point(3, 3)),
...                             Segment(Point(1, 3), Point(3, 3))]
>>> first_square_diagonals = [Segment(Point(0, 0), Point(2, 2)),
...                           Segment(Point(2, 2), Point(4, 0)),
...                           Segment(Point(2, 2), Point(4, 4)),
...                           Segment(Point(0, 4), Point(2, 2))]
>>> (multisegment_in_multipolygon(
...      Multisegment(first_square_edges),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, [])]))
...  is Relation.DISJOINT)
True
>>> (multisegment_in_multipolygon(
...      Multisegment(first_square_edges + first_inner_square_edges),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, [])]))
...  is Relation.TOUCH)
True
>>> (multisegment_in_multipolygon(
...      Multisegment(first_square_diagonals),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, [])]))
...  is Relation.CROSS)
True
>>> (multisegment_in_multipolygon(
```

(continues on next page)

(continued from previous page)

```
...      Multisegment(first_square_edges),
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(second_square, [])]))
...  is Relation.COMPONENT)
True
>>> (multisegment_in_multipolygon(
...      Multisegment(first_inner_square_edges),
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(second_square, [])]))
...  is Relation.WITHIN)
True
```

orient.planar.**contour_in_multipolygon**(*contour: Contour*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between contour and multipolygon.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = len(contour.vertices) + multipolygon_vertices_count`, `multipolygon_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **contour** – contour to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns

relation between contour and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (contour_in_multipolygon(
...      first_square, Multipolygon([Polygon(first_inner_square, []),
```

(continues on next page)

(continued from previous page)

```
...                                     Polygon(second_inner_square, [])]))
...  is Relation.DISJOINT)
True
>>> (contour_in_multipolygon(
...      second_square, Multipolygon([Polygon(first_square, []),
...                                   Polygon(third_square, [])]))
...  is Relation.TOUCH)
True
>>> (contour_in_multipolygon(
...      first_inner_square, Multipolygon([Polygon(triangle, []),
...                                        Polygon(second_square, [])]))
...  is Relation.CROSS)
True
>>> (contour_in_multipolygon(
...      first_square, Multipolygon([Polygon(first_square, []),
...                                  Polygon(third_square, [])]))
...  is Relation.COMPONENT)
True
>>> (contour_in_multipolygon(
...      triangle, Multipolygon([Polygon(first_square, []),
...                              Polygon(third_square, [])]))
...  is Relation.ENCLOSED)
True
>>> (contour_in_multipolygon(
...      first_inner_square, Multipolygon([Polygon(first_square, []),
...                                        Polygon(third_square, [])]))
...  is Relation.WITHIN)
True
```

orient.planar.**region_in_multipolygon**(*region: Contour*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between region and multipolygon.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = len(region.vertices) + multipolygon_vertices_count`, `multipolygon_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **region** – region to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns

relation between region and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
```

(continues on next page)

(continued from previous page)

```
>>> Contour = context.contour_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> Multipolygon = context.multipolygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> outer_square = Contour([Point(0, 0), Point(8, 0), Point(8, 8),
...                         Point(0, 8)])
>>> triangle = Contour([Point(0, 0), Point(4, 0), Point(0, 4)])
>>> (region_in_multipolygon(
...      third_square, Multipolygon([Polygon(first_inner_square, []),
...                                  Polygon(second_inner_square, [])]))
...  is Relation.DISJOINT)
True
>>> (region_in_multipolygon(
...      second_square, Multipolygon([Polygon(first_square, []),
...                                   Polygon(third_square, [])]))
...  is Relation.TOUCH)
True
>>> (region_in_multipolygon(
...      first_square, Multipolygon([Polygon(first_inner_square, []),
...                                  Polygon(second_inner_square, [])]))
...  is Relation.OVERLAP)
True
>>> (region_in_multipolygon(
...      outer_square, Multipolygon([Polygon(first_inner_square, []),
...                                  Polygon(second_inner_square, [])]))
...  is Relation.COVER)
True
>>> (region_in_multipolygon(
...      outer_square, Multipolygon([Polygon(first_square, []),
...                                  Polygon(third_square, [])]))
...  is Relation.ENCLOSES)
True
>>> (region_in_multipolygon(
...      triangle, Multipolygon([Polygon(first_square, []),
...                              Polygon(third_square, [])]))
...  is Relation.ENCLOSED)
True
>>> (region_in_multipolygon(
...      first_inner_square, Multipolygon([Polygon(first_square, []),
...                                        Polygon(third_square, [])]))
...  is Relation.WITHIN)
True
```

orient.planar.**multiregion_in_multipolygon**(*multiregion: Sequence[Contour]*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between multiregion and multipolygon.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = multiregion_vertices_count + multipolygon_vertices_count`, `multiregion_vertices_count = sum(len(region.vertices) for region in multiregion)`, `multipolygon_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **multiregion** – multiregion to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns
relation between multiregion and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> fourth_square = Contour([Point(0, 4), Point(4, 4), Point(4, 8),
...                          Point(0, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> third_inner_square = Contour([Point(5, 5), Point(7, 5), Point(7, 7),
...                               Point(5, 7)])
>>> fourth_inner_square = Contour([Point(1, 5), Point(3, 5), Point(3, 7),
...                                Point(1, 7)])
>>> (multiregion_in_multipolygon(
...      [first_square, third_square],
...      Multipolygon([Polygon(second_inner_square, []),
...                    Polygon(fourth_inner_square, [])]))
...  is multiregion_in_multipolygon(
...          [first_inner_square, third_inner_square],
...          Multipolygon([Polygon(second_square, [second_inner_square]),
...                        Polygon(fourth_square, [fourth_inner_square])]))
```

(continues on next page)

(continued from previous page)

```
...  is Relation.DISJOINT)
True
>>> (multiregion_in_multipolygon(
...      [first_square, third_square],
...      Multipolygon([Polygon(second_square, []),
...                    Polygon(fourth_square, [])]))
...  is multiregion_in_multipolygon(
...          [first_inner_square, third_inner_square],
...          Multipolygon([Polygon(first_square, [first_inner_square]),
...                        Polygon(third_square, [third_inner_square])]))
...  is Relation.TOUCH)
True
>>> (multiregion_in_multipolygon(
...      [first_square, third_inner_square],
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.OVERLAP)
True
>>> (multiregion_in_multipolygon(
...      [first_square, third_square],
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.COVER)
True
>>> (multiregion_in_multipolygon(
...      [first_square, third_inner_square],
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is multiregion_in_multipolygon(
...          [first_square, third_square],
...          Multipolygon([Polygon(first_square, [first_inner_square]),
...                        Polygon(third_square, [third_inner_square])]))
...  is Relation.ENCLOSES)
True
>>> (multiregion_in_multipolygon(
...      [first_inner_square, second_inner_square, third_inner_square],
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.COMPOSITE)
True
>>> (multiregion_in_multipolygon(
...      [first_square, third_square],
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.EQUAL)
True
>>> (multiregion_in_multipolygon(
...      [first_inner_square, second_inner_square],
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.COMPONENT)
```

(continues on next page)

(continued from previous page)

```
True
>>> (multiregion_in_multipolygon(
...      [first_inner_square, third_inner_square],
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.ENCLOSED)
True
>>> (multiregion_in_multipolygon(
...      [first_inner_square, third_inner_square],
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.WITHIN)
True
```

orient.planar.**polygon_in_multipolygon**(*polygon: Polygon*, *multipolygon: Multipolygon*, ***, *context: Optional[Context] = None*) → Relation

Finds relation between polygon and multipolygon.

Time complexity:

`O(vertices_count * log vertices_count)`

Memory complexity:

`O(vertices_count)`

where `vertices_count = polygon_vertices_count + multipolygon_vertices_count`, `polygon_vertices_count = len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes)`, `multipolygon_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in multipolygon.polygons)`.

Parameters

- **polygon** – polygon to check for.
- **multipolygon** – multipolygon to check in.
- **context** – geometric context.

Returns

relation between polygon and multipolygon.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> outer_square = Contour([Point(0, 0), Point(7, 0), Point(7, 7),
...                         Point(0, 7)])
>>> inner_square = Contour([Point(1, 1), Point(6, 1), Point(6, 6),
...                         Point(1, 6)])
>>> innermore_square = Contour([Point(2, 2), Point(5, 2), Point(5, 5),
...                             Point(2, 5)])
>>> innermost_square = Contour([Point(3, 3), Point(4, 3), Point(4, 4),
...                             Point(3, 4)])
>>> (polygon_in_multipolygon(Polygon(outer_square, [inner_square]),
```

(continues on next page)

(continued from previous page)

```
...                           Multipolygon([Polygon(innermore_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(innermore_square, []),
...                 Multipolygon([Polygon(outer_square, [inner_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [inner_square]),
...                 Multipolygon([Polygon(innermore_square,
...                                       [innermost_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(innermore_square, [innermost_square]),
...                 Multipolygon([Polygon(outer_square, [inner_square])]))
...  is Relation.DISJOINT)
True
>>> (polygon_in_multipolygon(
...         Polygon(inner_square, []),
...         Multipolygon([Polygon(outer_square, [inner_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [inner_square]),
...                 Multipolygon([Polygon(inner_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [inner_square]),
...                 Multipolygon([Polygon(inner_square,
...                                       [innermore_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(inner_square, [innermore_square]),
...                 Multipolygon([Polygon(outer_square, [inner_square])]))
...  is Relation.TOUCH)
True
>>> (polygon_in_multipolygon(
...         Polygon(inner_square, []),
...         Multipolygon([Polygon(outer_square, [innermore_square])]))
...  is polygon_in_multipolygon(Polygon(outer_square, [innermore_square]),
...                             Multipolygon([Polygon(inner_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [innermore_square]),
...                 Multipolygon([Polygon(inner_square,
...                                       [innermost_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(inner_square, [innermost_square]),
...                 Multipolygon([Polygon(outer_square,
...                                       [innermore_square])]))
...  is Relation.OVERLAP)
True
>>> (polygon_in_multipolygon(Polygon(outer_square, []),
...                          Multipolygon([Polygon(inner_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [innermost_square]),
...                 Multipolygon([Polygon(inner_square,
...                                       [innermore_square])]))
...  is Relation.COVER)
True
>>> (polygon_in_multipolygon(
```

(continues on next page)

(continued from previous page)

```
...         Polygon(outer_square, []),
...         Multipolygon([Polygon(outer_square, [inner_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [innermore_square]),
...                 Multipolygon([Polygon(outer_square, [inner_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [innermore_square]),
...                 Multipolygon([Polygon(inner_square,
...                                       [innermore_square])]))
...  is Relation.ENCLOSES)
True
>>> (polygon_in_multipolygon(Polygon(outer_square, []),
...                          Multipolygon([Polygon(outer_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [inner_square]),
...                 Multipolygon([Polygon(outer_square, [inner_square])]))
...  is Relation.EQUAL)
True
>>> (polygon_in_multipolygon(Polygon(innermore_square, []),
...                          Multipolygon([Polygon(outer_square,
...                                                [inner_square]),
...                                        Polygon(innermore_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(innermore_square, [innermost_square]),
...                 Multipolygon([Polygon(outer_square, [inner_square]),
...                               Polygon(innermore_square,
...                                       [innermost_square])]))
...  is Relation.COMPONENT)
True
>>> (polygon_in_multipolygon(Polygon(outer_square, [inner_square]),
...                          Multipolygon([Polygon(outer_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(outer_square, [inner_square]),
...                 Multipolygon([Polygon(outer_square,
...                                       [innermore_square])]))
...  is polygon_in_multipolygon(
...                 Polygon(inner_square, [innermore_square]),
...                 Multipolygon([Polygon(outer_square,
...                                       [innermore_square])]))
...  is Relation.ENCLOSED)
True
>>> (polygon_in_multipolygon(Polygon(inner_square, []),
...                          Multipolygon([Polygon(outer_square, [])]))
...  is polygon_in_multipolygon(
...                 Polygon(inner_square, [innermore_square]),
...                 Multipolygon([Polygon(outer_square,
...                                       [innermost_square])]))
...  is Relation.WITHIN)
True
```

orient.planar.**multipolygon_in_multipolygon**(*left: Multipolygon*, *right: Multipolygon*, *, *context: Optional[Context] = None*) → Relation

Finds relation between multipolygons.

Time complexity:
`O(vertices_count * log vertices_count)`

Memory complexity:
`O(vertices_count)`

where `vertices_count = left_vertices_count + right_vertices_count`, `left_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in left.polygons)`, `right_vertices_count = sum(len(polygon.border.vertices) + sum(len(hole.vertices) for hole in polygon.holes) for polygon in right.polygons)`.

Parameters

- **left** – multipolygon to check for.
- **right** – multipolygon to check in.
- **context** – geometric context.

Returns
relation between multipolygons.

```
>>> from ground.base import Relation, get_context
>>> context = get_context()
>>> Contour = context.contour_cls
>>> Multipolygon = context.multipolygon_cls
>>> Point = context.point_cls
>>> Polygon = context.polygon_cls
>>> first_square = Contour([Point(0, 0), Point(4, 0), Point(4, 4),
...                         Point(0, 4)])
>>> second_square = Contour([Point(4, 0), Point(8, 0), Point(8, 4),
...                          Point(4, 4)])
>>> third_square = Contour([Point(4, 4), Point(8, 4), Point(8, 8),
...                         Point(4, 8)])
>>> fourth_square = Contour([Point(0, 4), Point(4, 4), Point(4, 8),
...                          Point(0, 8)])
>>> first_inner_square = Contour([Point(1, 1), Point(3, 1), Point(3, 3),
...                               Point(1, 3)])
>>> second_inner_square = Contour([Point(5, 1), Point(7, 1), Point(7, 3),
...                                Point(5, 3)])
>>> third_inner_square = Contour([Point(5, 5), Point(7, 5), Point(7, 7),
...                               Point(5, 7)])
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(second_square, []),
...                    Polygon(fourth_square, [])]))
...  is Relation.DISJOINT)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]),
...      Multipolygon([Polygon(second_square, []),
...                    Polygon(fourth_square, [])]))
```

(continues on next page)

(continued from previous page)

```
...  is multipolygon_in_multipolygon(
...          Multipolygon([Polygon(first_inner_square, []),
...                        Polygon(third_inner_square, [])]),
...          Multipolygon([Polygon(first_square, [first_inner_square]),
...                        Polygon(third_square, [third_inner_square])]))
...  is Relation.TOUCH)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.OVERLAP)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.COVER)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is multipolygon_in_multipolygon(
...          Multipolygon([Polygon(first_square, []),
...                        Polygon(third_square, [])]),
...          Multipolygon([Polygon(first_square, [first_inner_square]),
...                        Polygon(third_square, [third_inner_square])]))
...  is Relation.ENCLOSES)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.COMPOSITE)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]),
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.EQUAL)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(second_inner_square, [])]),
...      Multipolygon([Polygon(first_inner_square, []),
```

(continues on next page)

(continued from previous page)

```
...                     Polygon(second_inner_square, []),
...                     Polygon(third_inner_square, [])]))
...  is Relation.COMPONENT)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_inner_square, [])]))
...  is Relation.ENCLOSED)
True
>>> (multipolygon_in_multipolygon(
...      Multipolygon([Polygon(first_inner_square, []),
...                    Polygon(third_inner_square, [])]),
...      Multipolygon([Polygon(first_square, []),
...                    Polygon(third_square, [])]))
...  is Relation.WITHIN)
True
```

PYTHON MODULE INDEX

INDEX