pygeos

Casper van der Wel

Sep 28, 2020

CONTENTS:

1 What is a ufunc? 3

2 Multithreading 5

3 The Geometry object 7

4 Examples 9

5 Relationship to Shapely 11

6 References 13

7 Copyright & License 15

Python Module Index 59

Index 61

PyGEOS is a C/Python library with vectorized geometry functions. The geometry operations are done in the open-source geometry library GEOS. PyGEOS wraps these operations in NumPy ufuncs providing a performance improvement when operating on arrays of geometries.

Note: PyGEOS is a very young package. While the available functionality should be stable and working correctly, it's still possible that APIs change in upcoming releases. But we would love for you to try it out, give feedback or contribute!

CHAPTER

ONE

WHAT IS A UFUNC?

A universal function (or ufunc for short) is a function that operates on n-dimensional arrays in an element-by-element fashion, supporting array broadcasting. The for-loops that are involved are fully implemented in C diminishing the overhead of the Python interpreter.

CHAPTER

TWO

MULTITHREADING

PyGEOS functions support multithreading. More specifically, the Global Interpreter Lock (GIL) is released during function execution. Normally in Python, the GIL prevents multiple threads from computing at the same time. PyGEOS functions internally releases this constraint so that the heavy lifting done by GEOS can be done in parallel, from a single Python process.

CHAPTER

THREE

THE GEOMETRY OBJECT

The *pygeos.Geometry* object is a container of the actual GEOSGeometry object. The Geometry object keeps track of the underlying GEOSGeometry and allows the python garbage collector to free memory when it is not used anymore.

Geometry objects are immutable. This means that after constructed, they cannot be changed inplace. Every PyGEOS operation will result in a new object being returned.

Construct a Geometry from a WKT (Well-Known Text):

```
>>> from pygeos import Geometry

>>> geometry = Geometry("POINT (5.2 52.1)")
```

Or using one of the provided (vectorized) functions:

```
>>> from pygeos import points

>>> point = points([(5.2, 52.1), (5.1, 52.2)]]
```

CHAPTER

FOUR

EXAMPLES

Compare an grid of points with a polygon:

```
>>> geoms = points(*np.indices((4, 4)))
>>> polygon = box(0, 0, 2, 2)

>>> contains(polygon, geoms)

  array([[False, False, False, False],
         [False,  True, False, False],
         [False, False, False, False],
         [False, False, False, False]])
```

Compute the area of all possible intersections of two lists of polygons:

```
>>> from pygeos import box, area, intersection

>>> polygons_x = box(range(5), 0, range(10, 15), 10)
>>> polygons_y = box(0, range(5), 10, range(10, 15))

>>> area(intersection(polygons_x[:, np.newaxis], polygons_y[np.newaxis, :]))

array([[100.,  90.,  80.,  70.,  60.],
       [ 90.,  81.,  72.,  63.,  54.],
       [ 80.,  72.,  64.,  56.,  48.],
       [ 70.,  63.,  56.,  49.,  42.],
       [ 60.,  54.,  48.,  42.,  36.]])
```

See the documentation for more: https://pygeos.readthedocs.io

CHAPTER

FIVE

RELATIONSHIP TO SHAPELY

Both Shapely and PyGEOS are exposing the functionality of the GEOS C++ library to Python. While Shapely only deals with single geometries, PyGEOS provides vectorized functions to work with arrays of geometries, giving better performance and convenience for such usecases.

There is active discussion and work toward integrating PyGEOS into Shapely:

- latest proposal: https://github.com/shapely/shapely-rfc/pull/1
- prior discussion: https://github.com/Toblerity/Shapely/issues/782

For now PyGEOS is developed as a separate project.

CHAPTER

SIX

REFERENCES

- GEOS: http://trac.osgeo.org/geos
- Shapely: https://shapely.readthedocs.io/en/latest/
- Numpy ufuncs: https://docs.scipy.org/doc/numpy/reference/ufuncs.html
- Joris van den Bossche's blogpost: https://jorisvandenbossche.github.io/blog/2017/09/19/geopandas-cython/
- Matthew Rocklin's blogpost: http://matthewrocklin.com/blog/work/2017/09/21/accelerating-geopandas-1

CHAPTER

SEVEN

COPYRIGHT & LICENSE

7.1 API Reference

7.1.1 Installation

Installation from PyPI

PyGEOS is available as a binary distribution (wheel) for Linux, OSX and Windows platforms. Install as follows:

```
$ pip install pygeos
```

Installation using conda

PyGEOS is available on the conda-forage channel. Install as follows:

```
$ conda install pygeos --channel conda-forge
```

Installation with custom GEOS libary

On Linux:

```
$ sudo apt install libgeos-dev
```

On OSX:

```
$ brew install geos
```

Make sure *geos-config* is available from you shell; append PATH if necessary:

```
$ export PATH=$PATH:/path/to/dir/having/geos-config
$ pip install pygeos --no-binary
```

We do not have a recipe for Windows platforms. The following steps should enable you to build PyGEOS yourself:

- Get a C compiler applicable to your Python version (https://wiki.python.org/moin/WindowsCompilers)
- Download and install a GEOS binary (https://trac.osgeo.org/osgeo4w/)

- Set GEOS_INCLUDE_PATH and GEOS_LIBRARY_PATH environment variables
- Run `pip install pygeos --no-binary`

Installation from source

The same as above, but then instead of installing pygeos with pip, you clone the package from Github:

```
$ git clone git@github.com:pygeos/pygeos.git
```

Install it in development mode using *pip*:

```
$ pip install -e .[test]
```

Run the unittests:

```
$ pytest pygeos
```

Notes on GEOS discovery

If GEOS is installed, normally the `geos-config` command line utility will be available, and `pip install` will find GEOS automatically. But if needed, you can specify where PyGEOS should look for the GEOS library before installing it:

On Linux / OSX:

```
$ export GEOS_INCLUDE_PATH=$CONDA_PREFIX/Library/include
$ export GEOS_LIBRARY_PATH=$CONDA_PREFIX/Library/lib
```

On Windows (assuming you are in a Visual C++ shell):

```
$ set GEOS_INCLUDE_PATH=%CONDA_PREFIX%\Library\include
$ set GEOS_LIBRARY_PATH=%CONDA_PREFIX%\Library\lib
```

7.1.2 Input/Output

pygeos.io.**from_shapely**(*geometry*, ***kwargs*)

Creates geometries from shapely Geometry objects.

This function requires the GEOS version of PyGEOS and shapely to be equal.

Parameters **geometry** : shapely Geometry object or array_like

Examples

```
>>> from shapely.geometry import Point
>>> from_shapely(Point(1, 2))
<pygeos.Geometry POINT (1 2)>
```

pygeos.io.**from_wkb**(*geometry*, ***kwargs*)

Creates geometries from the Well-Known Binary (WKB) representation.

The Well-Known Binary format is defined in the OGC Simple Features Specification for SQL.

Parameters **geometry** : str or array_like

The WKB byte object(s) to convert.

Examples

```
>>> from_wkb(b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\
↪x00\x00\xf0?')
<pygeos.Geometry POINT (1 1)>
```

pygeos.io.**from_wkt**(*geometry*, ***kwargs*)

Creates geometries from the Well-Known Text (WKT) representation.

The Well-known Text format is defined in the OGC Simple Features Specification for SQL.

Parameters **geometry** : str or array_like

The WKT string(s) to convert.

Examples

```
>>> from_wkt('POINT (0 0)')
<pygeos.Geometry POINT (0 0)>
```

pygeos.io.**to_wkb**(*geometry*, *hex=False*, *output_dimension=3*, *byte_order=- 1*, *include_srid=False*, ***kwargs*)

Converts to the Well-Known Binary (WKB) representation of a Geometry.

The Well-Known Binary format is defined in the OGC Simple Features Specification for SQL.

Parameters **geometry** : Geometry or array_like

hex : bool, default False

If true, export the WKB as a hexidecimal string. The default is to return a binary bytes object.

output_dimension : int, default 3

The output dimension for the WKB. Supported values are 2 and 3. Specifying 3 means that up to 3 dimensions will be written but 2D geometries will still be represented as 2D in the WKB represenation.

byte_order : int

Defaults to native machine byte order (-1). Use 0 to force big endian and 1 for little endian.

include_srid : bool, default False

Whether the SRID should be included in WKB (this is an extension to the OGC WKB specification).

Examples

```
>>> to_wkb(Geometry("POINT (1 1)"))
b'\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\xf0?'
>>> to_wkb(Geometry("POINT (1 1)"), hex=True)
'0101000000000000000000F03F000000000000F03F'
```

pygeos.io.**to_wkt**(*geometry*, *rounding_precision=6*, *trim=True*, *output_dimension=3*, *old_3d=False*, ***kwargs*)

Converts to the Well-Known Text (WKT) representation of a Geometry.

The Well-known Text format is defined in the OGC Simple Features Specification for SQL.

Parameters **geometry** : Geometry or array_like

rounding_precision : int, default 6

The rounding precision when writing the WKT string. Set to a value of -1 to indicate the full precision.

trim : bool, default True

Whether to trim unnecessary decimals (trailing zeros).

output_dimension : int, default 3

The output dimension for the WKT string. Supported values are 2 and 3. Specifying 3 means that up to 3 dimensions will be written but 2D geometries will still be represented as 2D in the WKT string.

old_3d : bool, default False

Enable old style 3D/4D WKT generation. By default, new style 3D/4D WKT (ie. "POINT Z (10 20 30)") is returned, but with `old_3d=True` the WKT will be formatted in the style "POINT (10 20 30)".

Notes

The defaults differ from the default of the GEOS library. To mimic this, use:

```
to_wkt(geometry, rounding_precision=-1, trim=False, output_dimension=2)
```

Examples

```
>>> to_wkt(Geometry("POINT (0 0)"))
'POINT (0 0)'
>>> to_wkt(Geometry("POINT (0 0)"), rounding_precision=3, trim=False)
'POINT (0.000 0.000)'
>>> to_wkt(Geometry("POINT (0 0)"), rounding_precision=-1, trim=False)
'POINT (0.0000000000000000 0.0000000000000000)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True)
'POINT Z (1 2 3)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True, output_dimension=2)
'POINT (1 2)'
>>> to_wkt(Geometry("POINT (1 2 3)"), trim=True, old_3d=True)
'POINT (1 2 3)'
```

7.1.3 Geometry creation

pygeos.creation.**box**(*x1*, *y1*, *x2*, *y2*)
Create box polygons.

Parameters **x1** : array_like

y2 : array_like

x1 : array_like

y2 : array_like

pygeos.creation.**geometrycollections**(*geometries*)
Create geometrycollections from arrays of geometries

Parameters **geometries** : array_like

An array of geometries

pygeos.creation.**linearrings**(*coords*, *y=None*, *z=None*)
Create an array of linearrings.

If the provided coords do not constitute a closed linestring, the first coordinate is duplicated at the end to close the ring.

Parameters **coords** : array_like

An array of lists of coordinate tuples (2- or 3-dimensional) or, if *y* is provided, an array of lists of x coordinates

y : array_like

z : array_like

pygeos.creation.**linestrings**(*coords*, *y=None*, *z=None*)
Create an array of linestrings.

Parameters **coords** : array_like

An array of lists of coordinate tuples (2- or 3-dimensional) or, if *y* is provided, an array of lists of x coordinates

y : array_like

z : array_like

pygeos.creation.**multilinestrings**(*geometries*)
Create multilinestrings from arrays of linestrings

Parameters **geometries** : array_like

An array of linestrings or coordinates (see linestrings).

pygeos.creation.**multipoints**(*geometries*)
Create multipoints from arrays of points

Parameters **geometries** : array_like

An array of points or coordinates (see points).

pygeos.creation.**multipolygons**(*geometries*)
Create multipolygons from arrays of polygons

Parameters **geometries** : array_like

An array of polygons or coordinates (see polygons).

pygeos.creation.**points**(*coords*, *y=None*, *z=None*)
Create an array of points.

Parameters **coords** : array_like

An array of coordinate tuples (2- or 3-dimensional) or, if *y* is provided, an array of x coordinates.

y : array_like

z : array_like

pygeos.creation.**polygons**(*shells*, *holes=None*)
Create an array of polygons.

Parameters **shell** : array_like

An array of linearrings that constitute the out shell of the polygons. Coordinates can also be passed, see linearrings.

holes : array_like

An array of lists of linearrings that constitute holes for each shell.

7.1.4 Geometry properties

pygeos.geometry.**get_coordinate_dimension**(*geometry*)
Returns the dimensionality of the coordinates in a geometry (2 or 3).

Returns -1 for not-a-geometry values.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> get_coordinate_dimension(Geometry("POINT (0 0)"))
2
>>> get_coordinate_dimension(Geometry("POINT Z (0 0 0)"))
3
>>> get_coordinate_dimension(None)
-1
```

pygeos.geometry.**get_dimensions**(*geometry*)
Returns the inherent dimensionality of a geometry.

The inherent dimension is 0 for points, 1 for linestrings and linearrings, and 2 for polygons. For geometrycollections it is the max of the containing elements. Empty and None geometries return -1.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> get_dimensions(Geometry("POINT (0 0)"))
0
>>> get_dimensions(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
2
>>> get_dimensions(Geometry("GEOMETRYCOLLECTION (POINT(0 0), LINESTRING(0 0, 1 1))
↪"))
1
>>> get_dimensions(Geometry("GEOMETRYCOLLECTION EMPTY"))
-1
>>> get_dimensions(None)
-1
```

pygeos.geometry.**get_exterior_ring**(*geometry*)

Returns the exterior ring of a polygon.

Parameters **geometry** : Geometry or array_like

See also:

get_interior_ring

Examples

```
>>> get_exterior_ring(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
<pygeos.Geometry LINEARRING (0 0, 0 10, 10 10, 10 0, 0 0)>
>>> get_exterior_ring(Geometry("POINT (1 1)")) is None
True
```

pygeos.geometry.**get_geometry**(*geometry*, *index*)

Returns the nth geometry from a collection of geometries.

Parameters **geometry** : Geometry or array_like

index : int or array_like

Negative values count from the end of the collection backwards.

See also:

get_num_geometries

Notes

- simple geometries act as length-1 collections
- out-of-range values return None

Examples

```
>>> multipoint = Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)")
>>> get_geometry(multipoint, 1)
<pygeos.Geometry POINT (1 1)>
>>> get_geometry(multipoint, -1)
<pygeos.Geometry POINT (3 3)>
>>> get_geometry(multipoint, 5) is None
True
>>> get_geometry(Geometry("POINT (1 1)"), 0)
<pygeos.Geometry POINT (1 1)>
>>> get_geometry(Geometry("POINT (1 1)"), 1) is None
True
```

pygeos.geometry.**get_interior_ring**(*geometry*, *index*)

Returns the nth interior ring of a polygon.

Parameters **geometry** : Geometry or array_like

index : int or array_like

Negative values count from the end of the interior rings backwards.

See also:

get_exterior_ring, *get_num_interior_rings*

Examples

```
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2␣
↪4, 4 4, 4 2, 2 2))")
>>> get_interior_ring(polygon_with_hole, 0)
<pygeos.Geometry LINEARRING (2 2, 2 4, 4 4, 4 2, 2 2)>
>>> get_interior_ring(Geometry("POINT (1 1)"), 0) is None
True
```

pygeos.geometry.**get_num_coordinates**(*geometry*)

Returns the total number of coordinates in a geometry.

Returns -1 for not-a-geometry values.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> get_num_coordinates(Geometry("POINT (0 0)"))
1
>>> get_num_coordinates(Geometry("POINT Z (0 0 0)"))
1
>>> get_num_coordinates(Geometry("GEOMETRYCOLLECTION (POINT(0 0), LINESTRING(0 0,␣
↪1 1))"))
3
>>> get_num_coordinates(None)
-1
```

pygeos.geometry.**get_num_geometries**(*geometry*)

Returns number of geometries in a collection.

Parameters **geometry** : Geometry or array_like

The number of geometries in points, linestrings, linearrings and polygons equals one.

See also:

get_num_points, *get_geometry*

Examples

```
>>> get_num_geometries(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"))
4
>>> get_num_geometries(Geometry("POINT (1 1)"))
1
```

pygeos.geometry.**get_num_interior_rings**(*geometry*)

Returns number of internal rings in a polygon

Parameters **geometry** : Geometry or array_like

The number of interior rings in non-polygons equals zero.

See also:

get_exterior_ring, *get_interior_ring*

Examples

```
>>> polygon = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))")
>>> get_num_interior_rings(polygon)
0
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2␣
↪4, 4 4, 4 2, 2 2))")
>>> get_num_interior_rings(polygon_with_hole)
1
>>> get_num_interior_rings(Geometry("POINT (1 1)"))
0
```

pygeos.geometry.**get_num_points**(*geometry*)

Returns number of points in a linestring or linearring.

Parameters **geometry** : Geometry or array_like

The number of points in geometries other than linestring or linearring equals zero.

See also:

get_point, *get_num_geometries*

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)")
>>> get_num_points(line)
4
>>> get_num_points(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"))
0
```

pygeos.geometry.**get_point**(*geometry*, *index*)

Returns the nth point of a linestring or linearring.

Parameters **geometry** : Geometry or array_like

index : int or array_like

Negative values count from the end of the linestring backwards.

See also:

get_num_points

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)")
>>> get_point(line, 1)
<pygeos.Geometry POINT (1 1)>
>>> get_point(line, -2)
<pygeos.Geometry POINT (2 2)>
>>> get_point(line, [0, 3]).tolist()
[<pygeos.Geometry POINT (0 0)>, <pygeos.Geometry POINT (3 3)>]
>>> get_point(Geometry("LINEARRING (0 0, 1 1, 2 2, 0 0)"), 1)
<pygeos.Geometry POINT (1 1)>
>>> get_point(Geometry("MULTIPOINT (0 0, 1 1, 2 2, 3 3)"), 1) is None
True
>>> get_point(Geometry("POINT (1 1)"), 0) is None
True
```

pygeos.geometry.**get_srid**(*geometry*)

Returns the SRID of a geometry.

Returns -1 for not-a-geometry values.

Parameters **geometry** : Geometry or array_like

See also:

set_srid

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> with_srid = set_srid(point, 4326)
>>> get_srid(point)
0
>>> get_srid(with_srid)
4326
```

pygeos.geometry.**get_type_id**(*geometry*)

Returns the type ID of a geometry.

- None is -1
- POINT is 0
- LINESTRING is 1
- LINEARRING is 2
- POLYGON is 3
- MULTIPOINT is 4
- MULTILINESTRING is 5
- MULTIPOLYGON is 6
- GEOMETRYCOLLECTION is 7

Parameters **geometry** : Geometry or array_like

See also:

GeometryType

Examples

```
>>> get_type_id(Geometry("LINESTRING (0 0, 1 1, 2 2, 3 3)"))
1
>>> get_type_id([Geometry("POINT (1 2)"), Geometry("POINT (1 2)")]).tolist()
[0, 0]
```

pygeos.geometry.**get_x**(*point*)

Returns the x-coordinate of a point

Parameters **point** : Geometry or array_like

Non-point geometries will result in NaN being returned.

See also:

get_y, *get_z*

Examples

```
>>> get_x(Geometry("POINT (1 2)"))
1.0
>>> get_x(Geometry("MULTIPOINT (1 1, 1 2)"))
nan
```

pygeos.geometry.**get_y**(*point*)

Returns the y-coordinate of a point

Parameters **point** : Geometry or array_like

Non-point geometries will result in NaN being returned.

See also:

get_x, *get_z*

Examples

```
>>> get_y(Geometry("POINT (1 2)"))
2.0
>>> get_y(Geometry("MULTIPOINT (1 1, 1 2)"))
nan
```

pygeos.geometry.**get_z**(*point*)

Returns the z-coordinate of a point.

Requires at least GEOS 3.7.0.

Parameters **point** : Geometry or array_like

Non-point geometries or geometries without 3rd dimension will result in NaN being returned.

See also:

get_x, *get_y*

Examples

```
>>> get_z(Geometry("POINT Z (1 2 3)"))
3.0
>>> get_z(Geometry("POINT (1 2)"))
nan
>>> get_z(Geometry("MULTIPOINT Z (1 1 1, 2 2 2)"))
nan
```

pygeos.geometry.**set_srid**(*geometry*, *srid*)

Returns a geometry with its SRID set.

Parameters **geometry** : Geometry or array_like

srid : int

See also:

get_srid

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> with_srid = set_srid(point, 4326)
>>> get_srid(point)
0
>>> get_srid(with_srid)
4326
```

7.1.5 Measurement

pygeos.measurement.**area**(*geometry*, ***kwargs*)

Computes the area of a (multi)polygon.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> area(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
100.0
>>> area(Geometry("MULTIPOLYGON (((0 0, 0 10, 10 10, 0 0)), ((0 0, 0 10, 10 10, 0␣
↪0)))"))
100.0
>>> area(Geometry("POLYGON EMPTY"))
0.0
>>> area(None)
nan
```

pygeos.measurement.**bounds**(*geometry*, ***kwargs*)

Computes the bounds (extent) of a geometry.

For each geometry these 4 numbers are returned: min x, min y, max x, max y.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> bounds(Geometry("POINT (2 3)")).tolist()
[2.0, 3.0, 2.0, 3.0]
>>> bounds(Geometry("LINESTRING (0 0, 0 2, 3 2)")).tolist()
[0.0, 0.0, 3.0, 2.0]
>>> bounds(Geometry("POLYGON EMPTY")).tolist()
[nan, nan, nan, nan]
>>> bounds(None).tolist()
[nan, nan, nan, nan]
```

pygeos.measurement.**distance**(*a*, *b*, ***kwargs*)

Computes the Cartesian distance between two geometries.

Parameters **a, b** : Geometry or array_like

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> distance(Geometry("POINT (10 0)"), point)
10.0
>>> distance(Geometry("LINESTRING (1 1, 1 -1)"), point)
1.0
>>> distance(Geometry("POLYGON ((3 0, 5 0, 5 5, 3 5, 3 0))"), point)
3.0
>>> distance(Geometry("POINT EMPTY"), point)
nan
>>> distance(None, point)
nan
```

pygeos.measurement.**frechet_distance**(*a*, *b*, *densify=None*, ***kwargs*)

Compute the discrete Fréchet distance between two geometries.

The Fréchet distance is a measure of similarity: it is the greatest distance between any point in A and the closest point in B. The discrete distance is an approximation of this metric: only vertices are considered. The parameter 'densify' makes this approximation less coarse by splitting the line segments between vertices before computing the distance.

Fréchet distance sweep continuously along their respective curves and the direction of curves is significant. This makes it a better measure of similarity than Hausdorff distance for curve or surface matching.

Parameters **a, b** : Geometry or array_like

densify : float, array_like or None

The value of densify is required to be between 0 and 1.

Examples

```
>>> line_1 = Geometry("LINESTRING (0 0, 100 0)")
>>> line_2 = Geometry("LINESTRING (0 0, 50 50, 100 0)")
>>> frechet_distance(line_1, line_2)
70.71...
>>> frechet_distance(line_1, line_2, densify=0.5)
50.0
>>> frechet_distance(line_1, Geometry("LINESTRING EMPTY"))
nan
>>> frechet_distance(line_1, None)
nan
```

pygeos.measurement.**hausdorff_distance**(*a*, *b*, *densify=None*, ***kwargs*)

Compute the discrete Hausdorff distance between two geometries.

The Hausdorff distance is a measure of similarity: it is the greatest distance between any point in A and the closest point in B. The discrete distance is an approximation of this metric: only vertices are considered. The parameter 'densify' makes this approximation less coarse by splitting the line segments between vertices before computing the distance.

Parameters **a, b** : Geometry or array_like

densify : float, array_like or None

The value of densify is required to be between 0 and 1.

Examples

```
>>> line_1 = Geometry("LINESTRING (130 0, 0 0, 0 150)")
>>> line_2 = Geometry("LINESTRING (10 10, 10 150, 130 10)")
>>> hausdorff_distance(line_1, line_2)
14.14...
>>> hausdorff_distance(line_1, line_2, densify=0.5)
70.0
>>> hausdorff_distance(line_1, Geometry("LINESTRING EMPTY"))
nan
>>> hausdorff_distance(line_1, None)
nan
```

pygeos.measurement.**length**(*geometry*, ***kwargs*)

Computes the length of a (multi)linestring or polygon perimeter.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> length(Geometry("LINESTRING (0 0, 0 2, 3 2)"))
5.0
>>> length(Geometry("MULTILINESTRING ((0 0, 1 0), (0 0, 1 0))"))
2.0
>>> length(Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0))"))
40.0
>>> length(Geometry("LINESTRING EMPTY"))
0.0
>>> length(None)
nan
```

pygeos.measurement.**total_bounds**(*geometry*, ***kwargs*)

Computes the total bounds (extent) of the geometry.

Parameters **geometry** : Geometry or array_like

Returns numpy ndarray of [xmin, ymin, xmax, ymax]

```
>>> total_bounds(Geometry("POINT (2 3)")).tolist()
```

[2.0, 3.0, 2.0, 3.0]

```
>>> total_bounds([Geometry("POINT (2 3)"), Geometry("POINT (4 5)")]).
↪tolist()
```

[2.0, 3.0, 4.0, 5.0]

```
>>> total_bounds([Geometry("LINESTRING (0 1, 0 2, 3 2)"),Geometry(
↪"LINESTRING (4 4, 4 6, 6 7)")]).tolist()
```

[0.0, 1.0, 6.0, 7.0]

```
>>> total_bounds(Geometry("POLYGON EMPTY")).tolist()
```

[nan, nan, nan, nan]

```
>>> total_bounds([Geometry("POLYGON EMPTY"), Geometry("POINT (2 3)
↪")]).tolist()
```

[2.0, 3.0, 2.0, 3.0]

```
>>> total_bounds(None).tolist()
```

[nan, nan, nan, nan]

7.1.6 Coordinate operations

pygeos.coordinates.**apply**(*geometry*, *transformation*)

Returns a copy of a geometry array with a function applied to its coordinates.

All returned geometries will be two-dimensional; the third dimension will be discarded, if present.

Parameters **geometry** : Geometry or array_like

transformation : function

A function that transforms a (N, 2) ndarray of float64 to another (N, 2) ndarray of float64.

Examples

```
>>> apply(Geometry("POINT (0 0)"), lambda x: x + 1)
<pygeos.Geometry POINT (1 1)>
>>> apply(Geometry("LINESTRING (2 2, 4 4)"), lambda x: x * [2, 3])
<pygeos.Geometry LINESTRING (4 6, 8 12)>
>>> apply(None, lambda x: x) is None
True
>>> apply([Geometry("POINT (0 0)"), None], lambda x: x).tolist()
[<pygeos.Geometry POINT (0 0)>, None]
```

pygeos.coordinates.**count_coordinates**(*geometry*)

Counts the number of coordinate pairs in a geometry array.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> count_coordinates(Geometry("POINT (0 0)"))
1
>>> count_coordinates(Geometry("LINESTRING (2 2, 4 4)"))
2
>>> count_coordinates(None)
0
>>> count_coordinates([Geometry("POINT (0 0)"), None])
1
```

pygeos.coordinates.**get_coordinates**(*geometry*, *include_z=False*)

Gets coordinates from a geometry array as an array of floats.

The shape of the returned array is (N, 2), with N being the number of coordinate pairs. With the default of *include_z=False*, three-dimensional data is ignored. When specifying *include_z=True*, the shape of the returned array is (N, 3).

Parameters **geometry** : Geometry or array_like

include_z : bool, default False

Whether to include the third dimension in the output. If True, and a geometry has no third dimension, the z-coordinates will be NaN.

Examples

```
>>> get_coordinates(Geometry("POINT (0 0)")).tolist()
[[0.0, 0.0]]
>>> get_coordinates(Geometry("LINESTRING (2 2, 4 4)")).tolist()
[[2.0, 2.0], [4.0, 4.0]]
>>> get_coordinates(None)
array([], shape=(0, 2), dtype=float64)
```

By default the third dimension is ignored:

```
>>> get_coordinates(Geometry("POINT Z (0 0 0)")).tolist()
[[0.0, 0.0]]
>>> get_coordinates(Geometry("POINT Z (0 0 0)"), include_z=True).tolist()
[[0.0, 0.0, 0.0]]
```

pygeos.coordinates.**set_coordinates**(*geometry*, *coordinates*)

Returns a copy of a geometry array with different coordinates.

All returned geometries will be two-dimensional; the third dimension will be discarded, if present.

Parameters **geometry** : Geometry or array_like

coordinates: array_like

Examples

```
>>> set_coordinates(Geometry("POINT (0 0)"), [[1, 1]])
<pygeos.Geometry POINT (1 1)>
>>> set_coordinates([Geometry("POINT (0 0)"), Geometry("LINESTRING (0 0, 0 0)")],␣
↪[[1, 2], [3, 4], [5, 6]]).tolist()
[<pygeos.Geometry POINT (1 2)>, <pygeos.Geometry LINESTRING (3 4, 5 6)>]
>>> set_coordinates([None, Geometry("POINT (0 0)")], [[1, 2]]).tolist()
[None, <pygeos.Geometry POINT (1 2)>]
```

7.1.7 Predicates

pygeos.predicates.**contains**(*a*, *b*, ***kwargs*)

Returns True if geometry B is completely inside geometry A.

A contains B if no points of B lie in the exterior of A and at least one point of the interior of B lies in the interior of A.

Parameters **a, b** : Geometry or array_like

See also:

within contains(A, B) == within(B, A)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> contains(line, Geometry("POINT (0 0)"))
False
>>> contains(line, Geometry("POINT (0.5 0.5)"))
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> contains(area, Geometry("POINT (0 0)"))
False
>>> contains(area, line)
True
>>> contains(area, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4␣
↪2, 4 4, 2 4, 2 2))")
>>> contains(polygon_with_hole, Geometry("POINT(1 1)"))
True
>>> contains(polygon_with_hole, Geometry("POINT(2 2)"))
False
>>> contains(polygon_with_hole, Geometry("LINESTRING(1 1, 5 5)"))
False
>>> contains(area, area)
True
>>> contains(area, None)
False
```

pygeos.predicates.**covered_by**(*a*, *b*, ***kwargs*)

Returns True if no point in geometry A is outside geometry B.

Parameters **a, b** : Geometry or array_like

See also:

covers covered_by(A, B) == covers(B, A)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> covered_by(Geometry("POINT (0 0)"), line)
True
>>> covered_by(Geometry("POINT (0.5 0.5)"), line)
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> covered_by(Geometry("POINT (0 0)"), area)
True
>>> covered_by(line, area)
True
>>> covered_by(Geometry("LINESTRING(0 0, 2 2)"), area)
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4␣
↪2, 4 4, 2 4, 2 2))")  # NOQA
>>> covered_by(Geometry("POINT(1 1)"), polygon_with_hole)
True
>>> covered_by(Geometry("POINT(2 2)"), polygon_with_hole)
True
```

(continues on next page)

(continued from previous page)

```
>>> covered_by(Geometry("LINESTRING(1 1, 5 5)"), polygon_with_hole)
False
>>> covered_by(area, area)
True
>>> covered_by(None, area)
False
```

pygeos.predicates.**covers**(*a*, *b*, ***kwargs*)

Returns True if no point in geometry B is outside geometry A.

Parameters **a, b** : Geometry or array_like

See also:

covered_by covers(A, B) == covered_by(B, A)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> covers(line, Geometry("POINT (0 0)"))
True
>>> covers(line, Geometry("POINT (0.5 0.5)"))
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> covers(area, Geometry("POINT (0 0)"))
True
>>> covers(area, line)
True
>>> covers(area, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4␣
↪2, 4 4, 2 4, 2 2))")  # NOQA
>>> covers(polygon_with_hole, Geometry("POINT(1 1)"))
True
>>> covers(polygon_with_hole, Geometry("POINT(2 2)"))
True
>>> covers(polygon_with_hole, Geometry("LINESTRING(1 1, 5 5)"))
False
>>> covers(area, area)
True
>>> covers(area, None)
False
```

pygeos.predicates.**crosses**(*a*, *b*, ***kwargs*)

Returns True if the intersection of two geometries spatially crosses.

That is: the geometries have some, but not all interior points in common. The geometries must intersect and the intersection must have a dimensionality less than the maximum dimension of the two input geometries. Additionally, the intersection of the two geometries must not equal either of the source geometries.

Parameters **a, b** : Geometry or array_like

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> crosses(line, Geometry("POINT (0.5 0.5)"))
False
>>> crosses(line, Geometry("MULTIPOINT ((0 1), (0.5 0.5))"))
True
>>> crosses(line, Geometry("LINESTRING(0 1, 1 0)"))
True
>>> crosses(line, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> crosses(area, line)
False
>>> crosses(area, Geometry("LINESTRING(0 0, 2 2)"))
True
>>> crosses(area, Geometry("POINT (0.5 0.5)"))
False
>>> crosses(area, Geometry("MULTIPOINT ((2 2), (0.5 0.5))"))
True
```

pygeos.predicates.**disjoint**(*a*, *b*, ***kwargs*)

Returns True if A and B do not share any point in space.

Disjoint implies that overlaps, touches, within, and intersects are False. Note missing (None) values are never disjoint.

Parameters **a, b** : Geometry or array_like

See also:

intersects disjoint(A, B) == ~intersects(A, B)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> disjoint(line, Geometry("POINT (0 0)"))
False
>>> disjoint(line, Geometry("POINT (0 1)"))
True
>>> disjoint(line, Geometry("LINESTRING(0 2, 2 0)"))
False
>>> empty = Geometry("GEOMETRYCOLLECTION EMPTY")
>>> disjoint(line, empty)
True
>>> disjoint(empty, empty)
True
>>> disjoint(empty, None)
False
>>> disjoint(None, None)
False
```

pygeos.predicates.**equals**(*a*, *b*, ***kwargs*)

Returns True if A and B are spatially equal.

If A is within B and B is within A, A and B are considered equal. The ordering of points can be different.

Parameters **a, b** : Geometry or array_like

See also:

equals_exact Check if A and B are structurally equal given a specified tolerance.

Examples

```
>>> line = Geometry("LINESTRING(0 0, 5 5, 10 10)")
>>> equals(line, Geometry("LINESTRING(0 0, 10 10)"))
True
>>> equals(Geometry("POLYGON EMPTY"), Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> equals(None, None)
False
```

pygeos.predicates.**equals_exact**(*a*, *b*, *tolerance=0.0*, ***kwargs*)

Returns True if A and B are structurally equal.

This method uses exact coordinate equality, which requires coordinates to be equal (within specified tolerance) and and in the same order for all components of a geometry. This is in contrast with the *equals* function which uses spatial (topological) equality.

Parameters **a, b** : Geometry or array_like

tolerance : float or array_like

See also:

equals Check if A and B are spatially equal.

Examples

```
>>> point1 = Geometry("POINT(50 50)")
>>> point2 = Geometry("POINT(50.1 50.1)")
>>> equals_exact(point1, point2)
False
>>> equals_exact(point1, point2, tolerance=0.2)
True
>>> equals_exact(point1, None, tolerance=0.2)
False
```

Difference between structucal and spatial equality:

```
>>> polygon1 = Geometry("POLYGON((0 0, 1 1, 0 1, 0 0))")
>>> polygon2 = Geometry("POLYGON((0 0, 0 1, 1 1, 0 0))")
>>> equals_exact(polygon1, polygon2)
False
>>> equals(polygon1, polygon2)
True
```

pygeos.predicates.**has_z**(*geometry*, ***kwargs*)

Returns True if a geometry has a Z coordinate.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> has_z(Geometry("POINT (0 0)"))
False
>>> has_z(Geometry("POINT Z (0 0 0)"))
True
```

pygeos.predicates.**intersects**(*a*, *b*, ***kwargs*)

Returns True if A and B share any portion of space.

Intersects implies that overlaps, touches and within are True.

Parameters **a, b** : Geometry or array_like

See also:

disjoint intersects(A, B) == ~disjoint(A, B)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> intersects(line, Geometry("POINT (0 0)"))
True
>>> intersects(line, Geometry("POINT (0 1)"))
False
>>> intersects(line, Geometry("LINESTRING(0 2, 2 0)"))
True
>>> intersects(None, None)
False
```

pygeos.predicates.**is_closed**(*geometry*, ***kwargs*)

Returns True if a linestring's first and last points are equal.

Parameters **geometry** : Geometry or array_like

This function will return False for non-linestrings.

See also:

is_ring Checks additionally if the geometry is simple.

Examples

```
>>> is_closed(Geometry("LINESTRING (0 0, 1 1)"))
False
>>> is_closed(Geometry("LINESTRING(0 0, 0 1, 1 1, 0 0)"))
True
>>> is_closed(Geometry("POINT (0 0)"))
False
```

pygeos.predicates.**is_empty**(*geometry*, ***kwargs*)

Returns True if a geometry is an empty point, polygon, etc.

Parameters **geometry** : Geometry or array_like

Any geometry type is accepted.

See also:

is_missing checks if the object is a geometry

Examples

```
>>> is_empty(Geometry("POINT EMPTY"))
True
>>> is_empty(Geometry("POINT (0 0)"))
False
>>> is_empty(None)
False
```

pygeos.predicates.**is_geometry**(*geometry*, ***kwargs*)

Returns True if the object is a geometry

Parameters **geometry** : any object or array_like

See also:

is_missing check if an object is missing (None)

is_valid_input check if an object is a geometry or None

Examples

```
>>> is_geometry(Geometry("POINT (0 0)"))
True
>>> is_geometry(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_geometry(None)
False
>>> is_geometry("text")
False
```

pygeos.predicates.**is_missing**(*geometry*, ***kwargs*)

Returns True if the object is not a geometry (None)

Parameters **geometry** : any object or array_like

See also:

is_geometry check if an object is a geometry

is_valid_input check if an object is a geometry or None

is_empty checks if the object is an empty geometry

Examples

```
>>> is_missing(Geometry("POINT (0 0)"))
False
>>> is_missing(Geometry("GEOMETRYCOLLECTION EMPTY"))
False
>>> is_missing(None)
True
>>> is_missing("text")
False
```

pygeos.predicates.**is_ring**(*geometry*, ***kwargs*)

Returns True if a linestring is closed and simple.

Parameters **geometry** : Geometry or array_like

This function will return False for non-linestrings.

See also:

is_closed Checks only if the geometry is closed.

is_simple Checks only if the geometry is simple.

Examples

```
>>> is_ring(Geometry("POINT (0 0)"))
False
>>> geom = Geometry("LINESTRING(0 0, 1 1)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(False, True, False)
>>> geom = Geometry("LINESTRING(0 0, 0 1, 1 1, 0 0)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, True, True)
>>> geom = Geometry("LINESTRING(0 0, 1 1, 0 1, 1 0, 0 0)")
>>> is_closed(geom), is_simple(geom), is_ring(geom)
(True, False, False)
```

pygeos.predicates.**is_simple**(*geometry*, ***kwargs*)

Returns True if a Geometry has no anomalous geometric points, such as self-intersections or self tangency.

Parameters **geometry** : Geometry or array_like

This function will return False for geometrycollections.

See also:

is_ring Checks additionally if the geometry is closed.

Examples

```
>>> is_simple(Geometry("POLYGON((1 1, 2 1, 2 2, 1 1))"))
True
>>> is_simple(Geometry("LINESTRING(0 0, 1 1, 0 1, 1 0, 0 0)"))
False
>>> is_simple(None)
False
```

pygeos.predicates.**is_valid**(*geometry*, ***kwargs*)

Returns True if a geometry is well formed.

Parameters **geometry** : Geometry or array_like

Any geometry type is accepted. Returns False for missing values.

See also:

is_valid_reason Returns the reason in case of invalid.

Examples

```
>>> is_valid(Geometry("LINESTRING(0 0, 1 1)"))
True
>>> is_valid(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
False
>>> is_valid(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_valid(None)
False
```

pygeos.predicates.**is_valid_input**(*geometry*, ***kwargs*)

Returns True if the object is a geometry or None

Parameters **geometry** : any object or array_like

See also:

is_geometry checks if an object is a geometry

is_missing checks if an object is None

Examples

```
>>> is_valid_input(Geometry("POINT (0 0)"))
True
>>> is_valid_input(Geometry("GEOMETRYCOLLECTION EMPTY"))
True
>>> is_valid_input(None)
True
>>> is_valid_input(1.0)
False
>>> is_valid_input("text")
False
```

pygeos.predicates.**is_valid_reason**(*geometry*, ***kwargs*)

Returns a string stating if a geometry is valid and if not, why.

Parameters **geometry** : Geometry or array_like

Any geometry type is accepted. Returns None for missing values.

See also:

is_valid returns True or False

Examples

```
>>> is_valid_reason(Geometry("LINESTRING(0 0, 1 1)"))
'Valid Geometry'
>>> is_valid_reason(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
'Self-intersection[0 0]'
>>> is_valid_reason(None) is None
True
```

pygeos.predicates.**overlaps**(*a*, *b*, ***kwargs*)

Returns True if A and B intersect, but one does not completely contain the other.

Parameters **a, b** : Geometry or array_like

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> overlaps(line, line)
False
>>> overlaps(line, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> overlaps(line, Geometry("LINESTRING(0.5 0.5, 2 2)"))
True
>>> overlaps(line, Geometry("POINT (0.5 0.5)"))
False
>>> overlaps(None, None)
False
```

pygeos.predicates.**relate**(*a*, *b*, ***kwargs*)

Returns a string representation of the DE-9IM intersection matrix.

Parameters **a, b** : Geometry or array_like

Examples

```
>>> point = Geometry("POINT (0 0)")
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> relate(point, line)
'F0FFFF102'
```

pygeos.predicates.**touches**(*a*, *b*, ***kwargs*)

Returns True if the only points shared between A and B are on the boundary of A and B.

Parameters **a, b** : Geometry or array_like

Examples

```
>>> line = Geometry("LINESTRING(0 2, 2 0)")
>>> touches(line, Geometry("POINT(0 2)"))
True
>>> touches(line, Geometry("POINT(1 1)"))
False
>>> touches(line, Geometry("LINESTRING(0 0, 1 1)"))
True
>>> touches(line, Geometry("LINESTRING(0 0, 2 2)"))
False
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> touches(area, Geometry("POINT(0.5 0)"))
True
>>> touches(area, Geometry("POINT(0.5 0.5)"))
False
>>> touches(area, line)
True
>>> touches(area, Geometry("POLYGON((0 1, 1 1, 1 2, 0 2, 0 1))"))
True
```

pygeos.predicates.**within**(*a*, *b*, ***kwargs*)

Returns True if geometry A is completely inside geometry B.

A is within B if no points of A lie in the exterior of B and at least one point of the interior of A lies in the interior of B.

Parameters **a, b** : Geometry or array_like

See also:

contains within(A, B) == contains(B, A)

Examples

```
>>> line = Geometry("LINESTRING(0 0, 1 1)")
>>> within(Geometry("POINT (0 0)"), line)
False
>>> within(Geometry("POINT (0.5 0.5)"), line)
True
>>> area = Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))")
>>> within(Geometry("POINT (0 0)"), area)
False
>>> within(line, area)
True
>>> within(Geometry("LINESTRING(0 0, 2 2)"), area)
False
>>> polygon_with_hole = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0), (2 2, 4␣
↪2, 4 4, 2 4, 2 2))")  # NOQA
>>> within(Geometry("POINT(1 1)"), polygon_with_hole)
True
>>> within(Geometry("POINT(2 2)"), polygon_with_hole)
False
>>> within(Geometry("LINESTRING(1 1, 5 5)"), polygon_with_hole)
False
>>> within(area, area)
True
>>> within(None, area)
False
```

7.1.8 Set operations

pygeos.set_operations.**coverage_union**(*a*, *b*, ***kwargs*)

Merges multiple polygons into one. This is an optimized version of union which assumes the polygons to be non-overlapping.

Requires at least GEOS 3.8.0.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

See also:

coverage_union_all

Examples

```
>>> from pygeos.constructive import normalize
>>> polygon = Geometry("POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))")
>>> normalize(coverage_union(polygon, Geometry("POLYGON ((1 0, 1 1, 2 1, 2 0, 1␣
↪0))")))
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 2 1, 2 0, 1 0, 0 0))>
```

Union with None returns same polygon >>> normalize(coverage_union(polygon, None)) <pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))>

pygeos.set_operations.**coverage_union_all**(*geometries*, *axis=0*, ***kwargs*)

Returns the union of multiple polygons of a geometry collection. This is an optimized version of union which assumes the polygons to be non-overlapping.

Requires at least GEOS 3.8.0.

Parameters **geometries** : array_like

axis : int

Axis along which the operation is performed. The default (zero) performs the operation over the first dimension of the input array. axis may be negative, in which case it counts from the last to the first axis.

See also:

coverage_union

Examples

```
>>> from pygeos.constructive import normalize
>>> polygon_1 = Geometry("POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))")
>>> polygon_2 = Geometry("POLYGON ((1 0, 1 1, 2 1, 2 0, 1 0))")
>>> normalize(coverage_union_all([polygon_1, polygon_2]))
<pygeos.Geometry POLYGON ((0 0, 0 1, 1 1, 2 1, 2 0, 1 0, 0 0))>
```

pygeos.set_operations.**difference**(*a*, *b*, ***kwargs*)

Returns the part of geometry A that does not intersect with geometry B.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

Examples

```
>>> line = Geometry("LINESTRING (0 0, 2 2)")
>>> difference(line, Geometry("LINESTRING (1 1, 3 3)"))
<pygeos.Geometry LINESTRING (0 0, 1 1)>
>>> difference(line, Geometry("LINESTRING EMPTY"))
<pygeos.Geometry LINESTRING (0 0, 2 2)>
>>> difference(line, None) is None
True
```

pygeos.set_operations.**intersection**(*a*, *b*, ***kwargs*)

Returns the geometry that is shared between input geometries.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

See also:

intersection_all

Examples

```
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> intersection(line, Geometry("LINESTRING(1 1, 3 3)"))
<pygeos.Geometry LINESTRING (1 1, 2 2)>
```

pygeos.set_operations.**intersection_all**(*geometries*, *axis=0*, ***kwargs*)

Returns the intersection of multiple geometries.

Parameters **geometries** : array_like

axis : int

Axis along which the operation is performed. The default (zero) performs the operation over the first dimension of the input array. axis may be negative, in which case it counts from the last to the first axis.

See also:

intersection

Examples

```
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(1 1, 3 3)")
>>> intersection_all([line_1, line_2])
<pygeos.Geometry LINESTRING (1 1, 2 2)>
>>> intersection_all([[line_1, line_2, None]], axis=1).tolist()
[None]
```

pygeos.set_operations.**symmetric_difference**(*a*, *b*, ***kwargs*)

Returns the geometry that represents the portions of input geometries that do not intersect.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

See also:

symmetric_difference_all

Examples

```
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> symmetric_difference(line, Geometry("LINESTRING(1 1, 3 3)"))
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>
```

pygeos.set_operations.**symmetric_difference_all**(*geometries*, *axis=0*, ***kwargs*)

Returns the symmetric difference of multiple geometries.

Parameters **geometries** : array_like

axis : int

Axis along which the operation is performed. The default (zero) performs the operation over the first dimension of the input array. axis may be negative, in which case it counts from the last to the first axis.

See also:

symmetric_difference

Examples

```
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(1 1, 3 3)")
>>> symmetric_difference_all([line_1, line_2])
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (2 2, 3 3))>
>>> symmetric_difference_all([[line_1, line_2, None]], axis=1).tolist()
[None]
```

pygeos.set_operations.**union**(*a*, *b*, ***kwargs*)

Merges geometries into one.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

See also:

union_all

Examples

```
>>> line = Geometry("LINESTRING(0 0, 2 2)")
>>> union(line, Geometry("LINESTRING(2 2, 3 3)"))
<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union(line, None) is None
True
```

pygeos.set_operations.**union_all**(*geometries*, *axis=0*, ***kwargs*)

Returns the union of multiple geometries.

Parameters **geometries** : array_like

axis : int

Axis along which the operation is performed. The default (zero) performs the operation over the first dimension of the input array. axis may be negative, in which case it counts from the last to the first axis.

See also:

union

Examples

```
>>> line_1 = Geometry("LINESTRING(0 0, 2 2)")
>>> line_2 = Geometry("LINESTRING(2 2, 3 3)")
>>> union_all([line_1, line_2])
<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>
>>> union_all([[line_1, line_2, None]], axis=1).tolist()
[<pygeos.Geometry MULTILINESTRING ((0 0, 2 2), (2 2, 3 3))>]
```

7.1.9 Constructive operations

pygeos.constructive.**boundary**(*geometry*, ***kwargs*)

Returns the topological boundary of a geometry.

Parameters **geometry** : Geometry or array_like

This function will return None for geometrycollections.

Examples

```
>>> boundary(Geometry("POINT (0 0)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
>>> boundary(Geometry("LINESTRING(0 0, 1 1, 1 2)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 2)>
>>> boundary(Geometry("LINEARRING (0 0, 1 0, 1 1, 0 1, 0 0)"))
<pygeos.Geometry MULTIPOINT EMPTY>
>>> boundary(Geometry("POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))"))
<pygeos.Geometry LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)>
>>> boundary(Geometry("MULTIPOINT (0 0, 1 2)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
>>> boundary(Geometry("MULTILINESTRING ((0 0, 1 1), (0 1, 1 0))"))
<pygeos.Geometry MULTIPOINT (0 0, 0 1, 1 0, 1 1)>
>>> boundary(Geometry("GEOMETRYCOLLECTION (POINT (0 0))")) is None
True
```

pygeos.constructive.**buffer**(*geometry*, *radius*, *quadsegs=8*, *cap_style='round'*, *join_style='round'*, *mitre_limit=5.0*, *single_sided=False*, ***kwargs*)

Computes the buffer of a geometry for positive and negative buffer radius.

The buffer of a geometry is defined as the Minkowski sum (or difference, for negative width) of the geometry with a circle with radius equal to the absolute value of the buffer radius.

The buffer operation always returns a polygonal result. The negative or zero-distance buffer of lines and points is always empty.

Parameters **geometry** : Geometry or array_like

width : float or array_like

Specifies the circle radius in the Minkowski sum (or difference).

quadsegs : int

Specifies the number of linear segments in a quarter circle in the approximation of circular arcs.

cap_style : {'round', 'square', 'flat'}

Specifies the shape of buffered line endings. 'round' results in circular line endings (see `quadsegs`). Both 'square' and 'flat' result in rectangular line endings, only 'flat' will end at the original vertex, while 'square' involves adding the buffer width.

join_style : {'round', 'bevel', 'sharp'}

Specifies the shape of buffered line midpoints. 'round' results in rounded shapes. 'bevel' results in a beveled edge that touches the original vertex. 'mitre' results in a single vertex that is beveled depending on the `mitre_limit` parameter.

mitre_limit : float

Crops of 'mitre'-style joins if the point is displaced from the buffered vertex by more than this limit.

single_sided : bool

Only buffer at one side of the geometry.

Examples

```
>>> buffer(Geometry("POINT (10 10)"), 2, quadsegs=1)
<pygeos.Geometry POLYGON ((12 10, 10 8, 8 10, 10 12, 12 10))>
>>> buffer(Geometry("POINT (10 10)"), 2, quadsegs=2)
<pygeos.Geometry POLYGON ((12 10, 11.4 8.59, 10 8, 8.59 8.59, 8 10, 8.59 11.4, 10␣
↪12, 11.4 11.4, 12 10))>
>>> buffer(Geometry("POINT (10 10)"), -2, quadsegs=1)
<pygeos.Geometry POLYGON EMPTY>
>>> line = Geometry("LINESTRING (10 10, 20 10)")
>>> buffer(line, 2, cap_style="square")
<pygeos.Geometry POLYGON ((20 12, 22 12, 22 8, 10 8, 8 8, 8 12, 20 12))>
>>> buffer(line, 2, cap_style="flat")
<pygeos.Geometry POLYGON ((20 12, 20 8, 10 8, 10 12, 20 12))>
>>> buffer(line, 2, single_sided=True, cap_style="flat")
<pygeos.Geometry POLYGON ((20 10, 10 10, 10 12, 20 12, 20 10))>
>>> line2 = Geometry("LINESTRING (10 10, 20 10, 20 20)")
>>> buffer(line2, 2, cap_style="flat", join_style="bevel")
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 22 10, 20 8, 10 8, 10 12, 18 12))>
>>> buffer(line2, 2, cap_style="flat", join_style="mitre")
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 22 8, 10 8, 10 12, 18 12))>
>>> buffer(line2, 2, cap_style="flat", join_style="mitre", mitre_limit=1)
<pygeos.Geometry POLYGON ((18 12, 18 20, 22 20, 21.8 9, 21 8.17, 10 8, 10 12, 18␣
↪12))>
>>> square = Geometry("POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))")
>>> buffer(square, 2, join_style="mitre")
<pygeos.Geometry POLYGON ((-2 -2, -2 12, 12 12, 12 -2, -2 -2))>
>>> buffer(square, -2, join_style="mitre")
<pygeos.Geometry POLYGON ((2 2, 2 8, 8 8, 8 2, 2 2))>
>>> buffer(square, -5, join_style="mitre")
<pygeos.Geometry POLYGON EMPTY>
>>> buffer(line, float("nan")) is None
True
```

`pygeos.constructive.`**`build_area`**(*geometry*, ***kwargs*)

Creates an areal geometry formed by the constituent linework of given geometry.

Equivalent of the PostGIS ST_BuildArea() function.

Requires at least GEOS 3.8.0.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> build_area(Geometry("GEOMETRYCOLLECTION(POLYGON((0 0, 3 0, 3 3, 0 3, 0 0)),␣
↪POLYGON((1 1, 1 2, 2 2, 1 1)))"))
<pygeos.Geometry POLYGON ((0 0, 0 3, 3 3, 3 0, 0 0), (1 1, 2 2, 1 2, 1 1))>
```

pygeos.constructive.**centroid**(*geometry*, ***kwargs*)

Computes the geometric center (center-of-mass) of a geometry.

For multipoints this is computed as the mean of the input coordinates. For multilinestrings the centroid is weighted by the length of each line segment. For multipolygons the centroid is weighted by the area of each polygon.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> centroid(Geometry("POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("LINESTRING (0 0, 2 2, 10 10)"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("MULTIPOINT (0 0, 10 10)"))
<pygeos.Geometry POINT (5 5)>
>>> centroid(Geometry("POLYGON EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

pygeos.constructive.**convex_hull**(*geometry*, ***kwargs*)

Computes the minimum convex geometry that encloses an input geometry.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> convex_hull(Geometry("MULTIPOINT (0 0, 10 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 10, 10 0, 0 0))>
>>> convex_hull(Geometry("POLYGON EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

pygeos.constructive.**delaunay_triangles**(*geometry*, *tolerance=0.0*, *only_edges=False*, ***kwargs*)

Computes a Delaunay triangulation around the vertices of an input geometry.

The output is a geometrycollection containing polygons (default) or linestrings (see only_edges). Returns an None if an input geometry contains less than 3 vertices.

Parameters **geometry** : Geometry or array_like

tolerance : float or array_like

Snap input vertices together if their distance is less than this value.

only_edges : bool or array_like

If set to True, the triangulation will return a collection of linestrings instead of polygons.

Examples

```
>>> points = Geometry("MULTIPOINT (50 30, 60 30, 100 100)")
>>> delaunay_triangles(points)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(points, only_edges=True)
<pygeos.Geometry MULTILINESTRING ((50 30, 100 100), (50 30, 60 30), (60 30, 100␣
↪100))>
>>> delaunay_triangles(Geometry("MULTIPOINT (50 30, 51 30, 60 30, 100 100)"),␣
↪tolerance=2)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("POLYGON ((50 30, 60 30, 100 100, 50 30))"))
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("LINESTRING (50 30, 60 30, 100 100)"))
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((50 30, 60 30, 100 100, 50 30)))>
>>> delaunay_triangles(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

pygeos.constructive.**envelope**(*geometry*, ***kwargs*)

Computes the minimum bounding box that encloses an input geometry.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> envelope(Geometry("LINESTRING (0 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(Geometry("MULTIPOINT (0 0, 10 0, 10 10)"))
<pygeos.Geometry POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))>
>>> envelope(Geometry("POINT (0 0)"))
<pygeos.Geometry POINT (0 0)>
>>> envelope(Geometry("GEOMETRYCOLLECTION EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

pygeos.constructive.**extract_unique_points**(*geometry*, ***kwargs*)

Returns all distinct vertices of an input geometry as a multipoint.

Note that only 2 dimensions of the vertices are considered when testing for equality.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> extract_unique_points(Geometry("POINT (0 0)"))
<pygeos.Geometry MULTIPOINT (0 0)>
>>> extract_unique_points(Geometry("LINESTRING(0 0, 1 1, 1 1)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 1)>
>>> extract_unique_points(Geometry("POLYGON((0 0, 1 0, 1 1, 0 0))"))
<pygeos.Geometry MULTIPOINT (0 0, 1 0, 1 1)>
>>> extract_unique_points(Geometry("MULTIPOINT (0 0, 1 1, 0 0)"))
<pygeos.Geometry MULTIPOINT (0 0, 1 1)>
```

(continues on next page)

(continued from previous page)

```
>>> extract_unique_points(Geometry("LINESTRING EMPTY"))
<pygeos.Geometry MULTIPOINT EMPTY>
```

pygeos.constructive.**make_valid**(*geometry*, ***kwargs*)

Repairs invalid geometries.

Requires at least GEOS 3.8.0.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> make_valid(Geometry("POLYGON((0 0, 1 1, 1 2, 1 1, 0 0))"))
<pygeos.Geometry MULTILINESTRING ((0 0, 1 1), (1 1, 1 2))>
```

pygeos.constructive.**normalize**(*geometry*, ***kwargs*)

Converts Geometry to normal form (or canonical form).

This method orders the coordinates, rings of a polygon and parts of multi geometries consistently. Typically useful for testing purposes (for example in combination with *equals_exact*).

Parameters **geometry** : Geometry or array_like

Examples

```
>>> p = Geometry("MULTILINESTRING((0 0, 1 1),(2 2, 3 3))")
>>> normalize(p)
<pygeos.Geometry MULTILINESTRING ((2 2, 3 3), (0 0, 1 1))>
```

pygeos.constructive.**point_on_surface**(*geometry*, ***kwargs*)

Returns a point that intersects an input geometry.

Parameters **geometry** : Geometry or array_like

Examples

```
>>> point_on_surface(Geometry("POLYGON ((0 0, 10 0, 10 10, 0 10, 0 0))"))
<pygeos.Geometry POINT (5 5)>
>>> point_on_surface(Geometry("LINESTRING (0 0, 2 2, 10 10)"))
<pygeos.Geometry POINT (2 2)>
>>> point_on_surface(Geometry("MULTIPOINT (0 0, 10 10)"))
<pygeos.Geometry POINT (0 0)>
>>> point_on_surface(Geometry("POLYGON EMPTY"))
<pygeos.Geometry POINT EMPTY>
```

pygeos.constructive.**simplify**(*geometry*, *tolerance*, *preserve_topology=False*, ***kwargs*)

Returns a simplified version of an input geometry using the Douglas-Peucker algorithm.

Parameters **geometry** : Geometry or array_like

tolerance : float or array_like

The maximum allowed geometry displacement. The higher this value, the smaller the number of vertices in the resulting geometry.

preserve_topology : bool

If set to True, the operation will avoid creating invalid geometries.

Examples

```
>>> line = Geometry("LINESTRING (0 0, 1 10, 0 20)")
>>> simplify(line, tolerance=0.9)
<pygeos.Geometry LINESTRING (0 0, 1 10, 0 20)>
>>> simplify(line, tolerance=1)
<pygeos.Geometry LINESTRING (0 0, 0 20)>
>>> polygon_with_hole = Geometry("POLYGON((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2␣
↪4, 4 4, 4 2, 2 2))")
>>> simplify(polygon_with_hole, tolerance=4, preserve_topology=True)
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0), (2 2, 2 4, 4 4, 4 2, 2␣
↪2))>
>>> simplify(polygon_with_hole, tolerance=4, preserve_topology=False)
<pygeos.Geometry POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))>
```

pygeos.constructive.**snap**(*geometry*, *reference*, *tolerance*, ***kwargs*)

Snaps an input geometry to reference geometry's vertices.

The tolerance is used to control where snapping is performed. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged.

Parameters **geometry** : Geometry or array_like

reference : Geometry or array_like

tolerance : float or array_like

Examples

```
>>> point = Geometry("POINT (0 2)")
>>> snap(Geometry("POINT (0.5 2.5)"), point, tolerance=1)
<pygeos.Geometry POINT (0 2)>
>>> snap(Geometry("POINT (0.5 2.5)"), point, tolerance=0.49)
<pygeos.Geometry POINT (0.5 2.5)>
>>> polygon = Geometry("POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))")
>>> snap(polygon, Geometry("POINT (8 10)"), tolerance=5)
<pygeos.Geometry POLYGON ((0 0, 0 10, 8 10, 10 0, 0 0))>
>>> snap(polygon, Geometry("LINESTRING (8 10, 8 0)"), tolerance=5)
<pygeos.Geometry POLYGON ((0 0, 0 10, 8 10, 8 0, 0 0))>
```

pygeos.constructive.**voronoi_polygons**(*geometry*, *tolerance=0.0*, *extend_to=None*, *only_edges=False*, ***kwargs*)

Computes a Voronoi diagram from the vertices of an input geometry.

The output is a geometrycollection containing polygons (default) or linestrings (see only_edges). Returns empty if an input geometry contains less than 2 vertices or if the provided extent has zero area.

Parameters **geometry** : Geometry or array_like

tolerance : float or array_like

Snap input vertices together if their distance is less than this value.

extend_to : Geometry or array_like

If provided, the diagram will be extended to cover the envelope of this geometry (unless this envelope is smaller than the input geometry).

only_edges : bool or array_like

If set to True, the triangulation will return a collection of linestrings instead of polygons.

Examples

```
>>> points = Geometry("MULTIPOINT (2 2, 4 2)")
>>> voronoi_polygons(points)
<pygeos.Geometry GEOMETRYCOLLECTION (POLYGON ((3 0, 0 0, 0 4, 3 4, 3 0)), POLYGON␣
↪((3 4, 6 4, 6 0, 3 0, 3 4)))>
>>> voronoi_polygons(points, only_edges=True)
<pygeos.Geometry LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(Geometry("MULTIPOINT (2 2, 4 2, 4.2 2)"), 0.5, only_
↪edges=True)
<pygeos.Geometry LINESTRING (3 4.2, 3 -0.2)>
>>> voronoi_polygons(points, extend_to=Geometry("LINESTRING (0 0, 10 10)"), only_
↪edges=True)
<pygeos.Geometry LINESTRING (3 10, 3 0)>
>>> voronoi_polygons(Geometry("LINESTRING (2 2, 4 2)"), only_edges=True)
<pygeos.Geometry LINESTRING (3 4, 3 0)>
>>> voronoi_polygons(Geometry("POINT (2 2)"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

7.1.10 Linestring operations

pygeos.linear.**line_interpolate_point**(*line*, *distance*, *normalize=False*)

Returns a point interpolated at given distance on a line.

Parameters **line** : Geometry or array_like

distance : float or array_like

Negative values measure distance from the end of the line. Out-of-range values will be clipped to the line endings.

normalize : bool

If normalize is set to True, the distance is a fraction of the total line length instead of the absolute distance.

Examples

```
>>> line = Geometry("LINESTRING(0 2, 0 10)")
>>> line_interpolate_point(line, 2)
<pygeos.Geometry POINT (0 4)>
>>> line_interpolate_point(line, 100)
<pygeos.Geometry POINT (0 10)>
>>> line_interpolate_point(line, -2)
<pygeos.Geometry POINT (0 8)>
>>> line_interpolate_point(line, [0.25, -0.25], normalize=True).tolist()
[<pygeos.Geometry POINT (0 4)>, <pygeos.Geometry POINT (0 8)>]
>>> line_interpolate_point(Geometry("LINESTRING EMPTY"), 1)
<pygeos.Geometry POINT EMPTY>
```

pygeos.linear.**line_locate_point**(*line*, *other*, *normalize=False*)

Returns the distance to the line origin of given point.

If given point does not intersect with the line, the point will first be projected onto the line after which the distance is taken.

Parameters **line** : Geometry or array_like

point : Geometry or array_like

normalize : bool

If normalize is set to True, the distance is a fraction of the total line length instead of the absolute distance.

Examples

```
>>> line = Geometry("LINESTRING(0 2, 0 10)")
>>> line_locate_point(line, Geometry("POINT(4 4)"))
2.0
>>> line_locate_point(line, Geometry("POINT(4 4)"), normalize=True)
0.25
>>> line_locate_point(line, Geometry("POINT(0 18)"))
8.0
>>> line_locate_point(Geometry("LINESTRING EMPTY"), Geometry("POINT(4 4)"))
nan
```

pygeos.linear.**line_merge**(*line*)

Returns (multi)linestrings formed by combining the lines in a multilinestrings.

Parameters **line** : Geometry or array_like

Examples

```
>>> line_merge(Geometry("MULTILINESTRING((0 2, 0 10), (0 10, 5 10))"))
<pygeos.Geometry LINESTRING (0 2, 0 10, 5 10)>
>>> line_merge(Geometry("MULTILINESTRING((0 2, 0 10), (0 11, 5 10))"))
<pygeos.Geometry MULTILINESTRING ((0 2, 0 10), (0 11, 5 10))>
>>> line_merge(Geometry("LINESTRING EMPTY"))
<pygeos.Geometry GEOMETRYCOLLECTION EMPTY>
```

pygeos.linear.**shared_paths**(*a*, *b*, ***kwargs*)

Returns the shared paths between geom1 and geom2.

Both geometries should be linestrings or arrays of linestrings. A geometrycollection or array of geometrycollections is returned with two elements in each geometrycollection. The first element is a multilinestring containing shared paths with the same direction for both inputs. The second element is a multilinestring containing shared paths with the opposite direction for the two inputs.

Parameters **a** : Geometry or array_like

b : Geometry or array_like

Examples

```
>>> geom1 = Geometry("LINESTRING (0 0, 1 0, 1 1, 0 1, 0 0)")
>>> geom2 = Geometry("LINESTRING (1 0, 2 0, 2 1, 1 1, 1 0)")
>>> shared_paths(geom1, geom2)
<pygeos.Geometry GEOMETRYCOLLECTION (MULTILINESTRING EMPTY, MULTILINESTRING ((1 0,
↪ 1 1)))>
```

7.1.11 STRTree

class pygeos.strtree.**STRtree**(*geometries*, *leafsize=5*)

A query-only R-tree created using the Sort-Tile-Recursive (STR) algorithm.

For two-dimensional spatial data. The actual tree will be constructed at the first query.

Parameters **geometries** : array_like

leafsize : int

the maximum number of child nodes that a node can have

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> # Query geometries that overlap envelope of input geometries:
>>> tree.query(pygeos.box(2, 2, 4, 4)).tolist()
[2, 3, 4]
>>> # Query geometries that are contained by input geometry:
>>> tree.query(pygeos.box(2, 2, 4, 4), predicate='contains').tolist()
[3]
>>> # Query geometries that overlap envelopes of `geoms`
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).tolist()
[[0, 0, 0, 1, 1], [2, 3, 4, 5, 6]]
```

Methods

query(*geometry*, *predicate=None*)

Return the index of all geometries in the tree with extents that intersect the envelope of the input geometry.

If predicate is provided, a prepared version of the input geometry is tested using the predicate function against each item whose extent intersects the envelope of the input geometry: predicate(geometry, tree_geometry).

If geometry is None, an empty array is returned.

Parameters **geometry** : Geometry

The envelope of the geometry is taken automatically for querying the tree.

predicate : {None, 'intersects', 'within', 'contains', 'overlaps', 'crosses','touches', 'covers', 'covered_by', 'contains_properly'}, optional

The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

Returns ndarray

Indexes of geometries in tree

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.query(pygeos.box(1,1, 3,3)).tolist()
[1, 2, 3]
>>> # Query geometries that are contained by input geometry
>>> tree.query(pygeos.box(2, 2, 4, 4), predicate='contains').tolist()
[3]
```

query_bulk(*geometry*, *predicate=None*)

Returns all combinations of each input geometry and geometries in the tree where the envelope of each input geometry intersects with the envelope of a tree geometry.

If predicate is provided, a prepared version of each input geometry is tested using the predicate function against each item whose extent intersects the envelope of the input geometry: predicate(geometry, tree_geometry).

This returns an array with shape (2,n) where the subarrays correspond to the indexes of the input geometries and indexes of the tree geometries associated with each. To generate an array of pairs of input geometry index and tree geometry index, simply transpose the results.

In the context of a spatial join, input geometries are the "left" geometries that determine the order of the results, and tree geometries are "right" geometries that are joined against the left geometries. This effectively performs an inner join, where only those combinations of geometries that can be joined based on envelope overlap or optional predicate are returned.

Any geometry that is None or empty in the input geometries is omitted from the output.

Parameters **geometry** : Geometry or array_like

Input geometries to query the tree. The envelope of each geometry is automatically calculated for querying the tree.

predicate : {None, 'intersects', 'within', 'contains', 'overlaps', 'crosses','touches', 'covers', 'covered_by', 'contains_properly'}, optional

The predicate to use for testing geometries from the tree that are within the input geometry's envelope.

Returns ndarray with shape (2, n)

The first subarray contains input geometry indexes. The second subarray contains tree geometry indexes.

Examples

```
>>> import pygeos
>>> tree = pygeos.STRtree(pygeos.points(np.arange(10), np.arange(10)))
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).tolist()
[[0, 0, 0, 1, 1], [2, 3, 4, 5, 6]]
>>> # Query for geometries that contain tree geometries
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)],␣
↪predicate='contains').tolist()
[[0], [3]]
```

(continues on next page)

(continued from previous page)

```
>>> # To get an array of pairs of index of input geometry, index of tree␣
↪geometry,
>>> # transpose the output:
>>> tree.query_bulk([pygeos.box(2, 2, 4, 4), pygeos.box(5, 5, 6, 6)]).T.
↪tolist()
[[0, 2], [0, 3], [0, 4], [1, 5], [1, 6]]
```

7.1.12 Changelog

Version 0.8 (unreleased)

Highlights of this release

- Handle multi geometries in `boundary` (#188)
- Handle empty points in to_wkb by conversion to POINT (nan, nan) (#179)
- Prevent segfault in to_wkt (and repr) with empty points in multipoints (#171)
- Fixed bug in `multilinestrings()`, it now accepts linearrings again (#168)
- Release the GIL to allow for multithreading in functions that do not create geometries (#144) and in the STRtree `query_bulk()` method (#174)
- Addition of a `frechet_distance()` function for GEOS >= 3.7 (#144)
- Addition of `coverage_union()` and `coverage_union_all()` functions for GEOS >= 3.8 (#142)
- Fixed segfaults when adding empty geometries to the STRtree (#147)
- Addition of `include_z=True` keyword in the `get_coordinates()` function to get 3D coordinates (#178)
- Addition of a `build_area()` function for GEOS >= 3.8 (#141)
- Addition of a `normalize()` function (#136)
- Addition of a `make_valid()` function for GEOS >= 3.8 (#107)
- Addition of a `get_z()` function for GEOS >= 3.7 (#175)
- Addition of a `relate()` function (#186)
- The `get_coordinate_dimensions()` function was renamed to `get_coordinate_dimension()` for consistency with GEOS (#176)
- Addition of `covers`, `covered_by`, `contains_properly` predicates to STRtree `query` and `query_bulk` (#157)

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward
- Casper van der Wel
- Joris Van den Bossche
- Krishna Chaitanya +
- Martin Fleischmann +

- Tom Clancy +

Version 0.7 (2020-03-18)

Highlights of this release

- STRtree improvements for spatial indexing: * Directly include predicate evaluation in `STRtree.query()` (#87) * Query multiple input geometries (spatial join style) with `STRtree.query_bulk` (#108)
- Addition of a `total_bounds()` function (#107)
- Geometries are now hashable, and can be compared with == or `!=` (#102)
- Fixed bug in `create_collections()` with wrong types (#86)
- Fixed a reference counting bug in STRtree (#97, #100)
- Start of a benchmarking suite using ASV (#96)
- This is the first release that will provide wheels!

Acknowledgments

Thanks to everyone who contributed to this release! People with a "+" by their names contributed a patch for the first time.

- Brendan Ward +
- Casper van der Wel
- Joris Van den Bossche
- Mike Taves +

Version 0.6 (2020-01-31)

Highlights of this release:

- Addition of the STRtree class for spatial indexing (#58)
- Addition of a `bounds` function (#69)
- A new `from_shapely` function to convert Shapely geometries to pygeos.Geometry (#61)
- Reintroduction of the `shared_paths` function (#77)

Contributors:

- Casper van der Wel
- Joris Van den Bossche
- mattijn +

Version 0.5 (2019-10-25)

Highlights of this release:

- Moved to the pygeos GitHub organization.
- Addition of functionality to get and transform all coordinates (eg for reprojections or affine transformations) [#44]
- Ufuncs for converting to and from the WKT and WKB formats [#45]
- `equals_exact` has been added [PR #57]

Version 0.4 (2019-09-16)

This is a major release of PyGEOS and the first one with actual release notes. Most important features of this release are:

- `buffer` and `haussdorff_distance` were completed [#15]
- `voronoi_polygons` and `delaunay_triangles` have been added [#17]
- The PyGEOS documentation is now mostly complete and available on http://pygeos.readthedocs.io .
- The concepts of "empty" and "missing" geometries have been separated. The `pygeos.Empty` and `pygeos.NaG` objects has been removed. Empty geometries are handled the same as normal geometries. Missing geometries are denoted by `None` and are handled by every pygeos function. `NaN` values cannot be used anymore to denote missing geometries. [PR #36]
- Added `pygeos.__version__` and `pygeos.geos_version`. [PR #43]

7.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pygeos.constructive, 45
pygeos.coordinates, 30
pygeos.creation, 19
pygeos.geometry, 20
pygeos.io, 16
pygeos.linear, 51
pygeos.measurement, 27
pygeos.predicates, 31
pygeos.set_operations, 41
pygeos.strtree, 53

INDEX

A

apply() (*in module pygeos.coordinates*), 30
area() (*in module pygeos.measurement*), 27

B

boundary() (*in module pygeos.constructive*), 45
bounds() (*in module pygeos.measurement*), 27
box() (*in module pygeos.creation*), 19
buffer() (*in module pygeos.constructive*), 45
build_area() (*in module pygeos.constructive*), 46

C

centroid() (*in module pygeos.constructive*), 47
contains() (*in module pygeos.predicates*), 31
convex_hull() (*in module pygeos.constructive*), 47
count_coordinates() (*in module pygeos.coordinates*), 30
coverage_union() (*in module pygeos.set_operations*), 41
coverage_union_all() (*in module pygeos.set_operations*), 42
covered_by() (*in module pygeos.predicates*), 32
covers() (*in module pygeos.predicates*), 33
crosses() (*in module pygeos.predicates*), 33

D

delaunay_triangles() (*in module pygeos.constructive*), 47
difference() (*in module pygeos.set_operations*), 42
disjoint() (*in module pygeos.predicates*), 34
distance() (*in module pygeos.measurement*), 27

E

envelope() (*in module pygeos.constructive*), 48
equals() (*in module pygeos.predicates*), 34
equals_exact() (*in module pygeos.predicates*), 35
extract_unique_points() (*in module pygeos.constructive*), 48

F

frechet_distance() (*in module pygeos.measurement*), 27
from_shapely() (*in module pygeos.io*), 16
from_wkb() (*in module pygeos.io*), 16
from_wkt() (*in module pygeos.io*), 17

G

geometrycollections() (*in module pygeos.creation*), 19
get_coordinate_dimension() (*in module pygeos.geometry*), 20
get_coordinates() (*in module pygeos.coordinates*), 30
get_dimensions() (*in module pygeos.geometry*), 20
get_exterior_ring() (*in module pygeos.geometry*), 21
get_geometry() (*in module pygeos.geometry*), 21
get_interior_ring() (*in module pygeos.geometry*), 22
get_num_coordinates() (*in module pygeos.geometry*), 22
get_num_geometries() (*in module pygeos.geometry*), 22
get_num_interior_rings() (*in module pygeos.geometry*), 23
get_num_points() (*in module pygeos.geometry*), 23
get_point() (*in module pygeos.geometry*), 24
get_srid() (*in module pygeos.geometry*), 24
get_type_id() (*in module pygeos.geometry*), 24
get_x() (*in module pygeos.geometry*), 25
get_y() (*in module pygeos.geometry*), 25
get_z() (*in module pygeos.geometry*), 26

H

has_z() (*in module pygeos.predicates*), 35
hausdorff_distance() (*in module pygeos.measurement*), 28

I

intersection() (*in module pygeos.set_operations*), 42
intersection_all() (*in module pygeos.set_operations*), 43
intersects() (*in module pygeos.predicates*), 36

is_closed() (*in module pygeos.predicates*), 36
is_empty() (*in module pygeos.predicates*), 36
is_geometry() (*in module pygeos.predicates*), 37
is_missing() (*in module pygeos.predicates*), 37
is_ring() (*in module pygeos.predicates*), 37
is_simple() (*in module pygeos.predicates*), 38
is_valid() (*in module pygeos.predicates*), 38
is_valid_input() (*in module pygeos.predicates*), 39
is_valid_reason() (*in module pygeos.predicates*), 39

L

length() (*in module pygeos.measurement*), 28
line_interpolate_point() (*in module pygeos.linear*), 51
line_locate_point() (*in module pygeos.linear*), 51
line_merge() (*in module pygeos.linear*), 52
linearrings() (*in module pygeos.creation*), 19
linestrings() (*in module pygeos.creation*), 19

M

make_valid() (*in module pygeos.constructive*), 49
module
    pygeos.constructive, 45
    pygeos.coordinates, 30
    pygeos.creation, 19
    pygeos.geometry, 20
    pygeos.io, 16
    pygeos.linear, 51
    pygeos.measurement, 27
    pygeos.predicates, 31
    pygeos.set_operations, 41
    pygeos.strtree, 53
multilinestrings() (*in module pygeos.creation*), 19
multipoints() (*in module pygeos.creation*), 19
multipolygons() (*in module pygeos.creation*), 19

N

normalize() (*in module pygeos.constructive*), 49

O

overlaps() (*in module pygeos.predicates*), 39

P

point_on_surface() (*in module pygeos.constructive*), 49
points() (*in module pygeos.creation*), 19
polygons() (*in module pygeos.creation*), 20
pygeos.constructive
    module, 45
pygeos.coordinates
    module, 30
pygeos.creation
    module, 19
pygeos.geometry
    module, 20
pygeos.io
    module, 16
pygeos.linear
    module, 51
pygeos.measurement
    module, 27
pygeos.predicates
    module, 31
pygeos.set_operations
    module, 41
pygeos.strtree
    module, 53

Q

query() (*pygeos.strtree.STRtree method*), 53
query_bulk() (*pygeos.strtree.STRtree method*), 54

R

relate() (*in module pygeos.predicates*), 40

S

set_coordinates() (*in module pygeos.coordinates*), 31
set_srid() (*in module pygeos.geometry*), 26
shared_paths() (*in module pygeos.linear*), 52
simplify() (*in module pygeos.constructive*), 49
snap() (*in module pygeos.constructive*), 50
STRtree (*class in pygeos.strtree*), 53
symmetric_difference() (*in module pygeos.set_operations*), 43
symmetric_difference_all() (*in module pygeos.set_operations*), 44

T

to_wkb() (*in module pygeos.io*), 17
to_wkt() (*in module pygeos.io*), 18
total_bounds() (*in module pygeos.measurement*), 29
touches() (*in module pygeos.predicates*), 40

U

union() (*in module pygeos.set_operations*), 44
union_all() (*in module pygeos.set_operations*), 44

V

voronoi_polygons() (*in module pygeos.constructive*), 50

W

within() (*in module pygeos.predicates*), 40