Getting Started

A pure Python library for 2D geometric computations with zero dependencies.

Chapter 1

Installation

Chapter 2

Requirements

- Python 3.10 or higher
- pip or another package manager

Chapter 3

From PyPI

```
pip install planar-geometry
```

3.1 From Source

Clone the repository and install in development mode:

```
git clone https://github.com/yourusername/planar_geometry.git
cd planar_geometry
pip install -e .
```

3.2 Installing Development Tools

For development, testing, and documentation:

```
pip install -r requirements-dev.txt
pip install -r requirements-test.txt
pip install -r requirements-docs.txt
```

Or install all at once:

```
make install-all
```

3.3 Verifying Installation

To verify that planar_geometry is installed correctly:

```
python -c "import planar_geometry; print(planar_geometry.__version__)"
```

Or run a simple test:

```
python -c "from planar_geometry.point import Point; p = Point(3, 4);␣
↪print(p)"
```

Chapter 4

Quick Start

4.1 Basic Usage

4.1.1 Creating Points

```
from planar_geometry.point import Point

# Create points
p1 = Point(0, 0)
p2 = Point(3, 4)

print(f"Point: {p1}")
print(f"Distance: {p1.distance_to(p2)}")
```

4.1.2 Working with Vectors

```
from planar_geometry.vector import Vector

# Create vectors
v1 = Vector(1, 0)
v2 = Vector(0, 1)

# Vector operations
v3 = v1 + v2
dot_product = v1.dot(v2)
magnitude = v1.magnitude()
```

4.1.3 Lines and Circles

```
from planar_geometry.line import Line
from planar_geometry.circle import Circle
from planar_geometry.point import Point

# Create a line
line = Line(Point(0, 0), Point(1, 1))

# Create a circle
circle = Circle(center=Point(0, 0), radius=5)

# Check if point is on circle
point = Point(3, 4)
is_on_circle = circle.contains_point(point)
```

4.2 Next Steps

- *Basic Usage* - More detailed examples
- *Advanced Usage* - Advanced geometric operations
- *Points API* - Complete API reference for Points

Chapter 5

Basic Usage

This guide covers the fundamental operations with planar_geometry.

5.1 Points

5.1.1 Creating and Using Points

Points are the basic building blocks. They represent locations in 2D space.

```
from planar_geometry.point import Point

# Create points
origin = Point(0, 0)
p1 = Point(3, 4)
p2 = Point(1, 2)

# Access coordinates
x, y = p1.x, p1.y

# Distance between points
dist = p1.distance_to(p2)
print(f"Distance: {dist}")

# Midpoint
mid = p1.midpoint_to(p2)
print(f"Midpoint: {mid}")
```

5.2 Vectors

5.2.1 Vector Operations

Vectors represent directions and magnitudes.

```
from planar_geometry.vector import Vector
from planar_geometry.point import Point

# Create vectors
v1 = Vector(1, 0)  # Unit vector in x direction
v2 = Vector(0, 1)  # Unit vector in y direction

# Vector arithmetic
v3 = v1 + v2  # Addition
v4 = v1.scale(2)  # Scaling

# Dot product
dot = v1.dot(v2)  # Perpendicular vectors have dot product = 0

# Magnitude
mag = v1.magnitude()  # Length of vector

# Normalization
normalized = v1.normalize()
```

5.3 Lines

5.3.1 Working with Lines

```
from planar_geometry.line import Line
from planar_geometry.point import Point

# Create a line from two points
p1 = Point(0, 0)
p2 = Point(3, 4)
line = Line(p1, p2)

# Check if point is on line
p3 = Point(1.5, 2)
if line.contains_point(p3):
    print("Point is on the line")
```

(continues on next page)

(continued from previous page)

```
# Find intersection with another line
line2 = Line(Point(0, 4), Point(4, 0))
intersection = line.intersection(line2)
print(f"Intersection: {intersection}")
```

5.4 Circles

5.4.1 Circle Operations

```
from planar_geometry.circle import Circle
from planar_geometry.point import Point

# Create a circle
center = Point(0, 0)
circle = Circle(center, radius=5)

# Check if point is inside, on, or outside circle
p1 = Point(3, 4)  # On the circle (distance = 5)
p2 = Point(2, 2)  # Inside the circle
p3 = Point(10, 0)  # Outside the circle

print(f"p1 on circle: {circle.contains_point(p1)}")
print(f"Area: {circle.area()}")
print(f"Circumference: {circle.circumference()}")
```

5.5 Polygons

5.5.1 Working with Polygons

```
from planar_geometry.polygon import Polygon
from planar_geometry.point import Point

# Create a triangle
vertices = [
    Point(0, 0),
    Point(3, 0),
    Point(3, 4)
]
triangle = Polygon(vertices)
```

(continues on next page)

(continued from previous page)

```
# Basic properties
print(f"Perimeter: {triangle.perimeter()}")
print(f"Area: {triangle.area()}")

# Check if point is inside
p = Point(1, 1)
if triangle.contains_point(p):
    print("Point is inside the polygon")
```

Chapter 6

Advanced Usage

6.1 Geometric Transformations

6.1.1 Rotation

```
from planar_geometry.point import Point
from planar_geometry.vector import Vector
from math import pi

# Rotate a vector
v = Vector(1, 0)
rotated = v.rotate(pi / 4)  # Rotate 45 degrees

# Rotate a point around origin
p = Point(1, 0)
# Use vector representation
v = Vector(p.x, p.y)
rotated_v = v.rotate(pi / 4)
rotated_p = Point(rotated_v.x, rotated_v.y)
```

6.1.2 Reflection

```
from planar_geometry.line import Line
from planar_geometry.point import Point

# Reflect a point across a line
line = Line(Point(0, 0), Point(1, 0))  # x-axis
point = Point(1, 1)
reflected = point.reflect_across(line)
```

6.2 Composition of Operations

6.2.1 Working with Multiple Geometric Objects

```
from planar_geometry.point import Point
from planar_geometry.polygon import Polygon
from planar_geometry.circle import Circle

# Create complex shapes
triangle = Polygon([
    Point(0, 0),
    Point(4, 0),
    Point(2, 3)
])

# Find relationships
center = triangle.centroid()
circle = Circle(center, radius=5)

# Check intersections
for vertex in triangle.vertices:
    if not circle.contains_point(vertex):
        print(f"Vertex {vertex} is outside circle")
```

6.3 Intersection Detection

6.3.1 Line Intersections

```
from planar_geometry.line import Line
from planar_geometry.point import Point

line1 = Line(Point(0, 0), Point(2, 2))
line2 = Line(Point(0, 2), Point(2, 0))

intersection = line1.intersection(line2)
if intersection:
    print(f"Lines intersect at: {intersection}")
else:
    print("Lines are parallel")
```

6.3.2 Circle Intersections

```
from planar_geometry.circle import Circle
from planar_geometry.point import Point

circle1 = Circle(Point(0, 0), 5)
circle2 = Circle(Point(6, 0), 5)

intersections = circle1.intersection(circle2)
for point in intersections:
    print(f"Intersection point: {point}")
```

6.4 Polygon Operations

6.4.1 Advanced Polygon Methods

```
from planar_geometry.polygon import Polygon
from planar_geometry.point import Point

# Create a quadrilateral
quad = Polygon([
    Point(0, 0),
    Point(4, 0),
    Point(4, 3),
    Point(0, 3)
])

# Check if polygon is convex
is_convex = quad.is_convex()

# Get centroid
centroid = quad.centroid()

# Calculate area
area = quad.area()

# Perimeter
perimeter = quad.perimeter()
```

Chapter 7

Points API

planar_geometry/point.py

: : : 0.01 : wangheng <wangfaofao@gmail.com>

:

- Point2D:

:

- math:
- measurable:

:

from planar_geometry import Point2D

p1 = Point2D(1.0, 2.0) p2 = Point2D(4.0, 6.0) distance = p1.distance_to(p2)

class planar_geometry.point.point2d.**Point2D**(*x*, *y*)

Bases: `Measurable1D`

:

-
-
- 0

:

x: float - y: float -

:

- Cython

- Measurable1Dlength()0

:

p = Point2D(1.0, 2.0) print(p.x, p.y)

Parameters

- **x** (`float`) – float -
- **y** (`float`) – float -

`length()`

Return type
`float`

:

- 0

:
float: 0.0

`distance_to(`*other*`)`

:

-

- distance = sqrt((x1-x2)^2 + (y1-y2)^2)

Parameters
other (*Point2D*) – Point2D -

Return type
`float`

:
float:

`distance_squared_to(`*other*`)`

:

-

-

Parameters
other (*Point2D*) – Point2D -

Return type
`float`

:
float:

`midpoint_to(other)`

Parameters
other (*Point2D*) – Point2D -

Return type
Point2D

:
Point2D:

`add(dx, dy)`

Parameters

- **dx** (`float`) – float - x
- **dy** (`float`) – float - y

Return type
Point2D

:
Point2D:

`subtract(other)`
- = (dx, dy)

Parameters
other (*Point2D*) – Point2D -

Return type
`tuple`

:
tuple: (dx, dy)

`multiply(scalar)`
* =

Parameters
scalar (`float`) – float -

Return type
Point2D

:
Point2D:

negate()
- = (-x, -y)

Return type
Point2D

:
Point2D:

equals(*other*, *tolerance=1e-09*)

Parameters
- **other** (*Point2D*) – Point2D -
- **tolerance** (float) – float -

Return type
bool

:
bool:

is_zero(*tolerance=1e-09*)

Parameters
tolerance (float) – float -

Return type
bool

:
bool:

to_tuple()

Return type
tuple

:
tuple: (x, y)

static from_tuple(*data*)

Parameters
data (tuple) – tuple - (x, y)

Return type
Point2D

:
Point2D:

static origin()

Return type
Point2D

:
Point2D: (0, 0)

__add__(*other*)
+ (dx, dy) =

Parameters
other (tuple) – tuple - (dx, dy)

Return type
Point2D

:
Point2D:

__sub__(*other*)
- = (x, y)

Parameters
other (*Point2D*) – Point2D -

Return type
tuple

:
tuple: (dx, dy)

__mul__(*scalar*)
* =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

__rmul__(*scalar*)
* =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

__truediv__(*scalar*)
/ =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

:
ZeroDivisionError: 0

__eq__(*other*)

:

- math.isclose

Parameters
other (object) – object -

Return type
bool

:
bool:

__hash__()

Return type
int

:

- Point2D

:
int:

7.1 Point2D Class

class planar_geometry.point.point2d.**Point2D**(*x*, *y*)

Bases: Measurable1D

:

-
-
- 0

:
x: float - y: float -

:

- Cython
- Measurable1Dlength()0

:
p = Point2D(1.0, 2.0) print(p.x, p.y)

Parameters

- **x** (float) – float -
- **y** (float) – float -

length()

Return type
float

:

- 0

:

float: 0.0

distance_to(*other*)

:

-

- distance = sqrt((x1-x2)^2 + (y1-y2)^2)

Parameters
other (*Point2D*) – Point2D -

Return type
float

:

float:

distance_squared_to(*other*)

:

-

-

Parameters
other (*Point2D*) – Point2D -

Return type
float

:

float:

midpoint_to(*other*)

Parameters
other (*Point2D*) – Point2D -

Return type
Point2D

:

Point2D:

add(*dx*, *dy*)

Parameters

- **dx** (float) – float - x
- **dy** (float) – float - y

Return type
Point2D

:
Point2D:

subtract(*other*)
- = (dx, dy)

Parameters
other (*Point2D*) – Point2D -

Return type
tuple

:
tuple: (dx, dy)

multiply(*scalar*)
* =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

negate()
- = (-x, -y)

Return type
Point2D

:
Point2D:

equals(*other*, *tolerance=1e-09*)

Parameters

- **other** (*Point2D*) – Point2D -
- **tolerance** (float) – float -

Return type
bool

:
bool:

is_zero(*tolerance=1e-09*)

Parameters
tolerance (float) – float -

Return type
bool

:
bool:

to_tuple()

Return type
tuple

:
tuple: (x, y)

static from_tuple(*data*)

Parameters
data (tuple) – tuple - (x, y)

Return type
Point2D

:
Point2D:

static origin()

Return type
Point2D

:
Point2D: (0, 0)

__add__(*other*)
+ (dx, dy) =

Parameters
other (tuple) – tuple - (dx, dy)

Return type
Point2D

:
Point2D:

__sub__(*other*)
- = (x, y)

Parameters
other (*Point2D*) – Point2D -

Return type
tuple

:
tuple: (dx, dy)

__mul__(*scalar*)
* =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

__rmul__(*scalar*)
* =

Parameters
scalar (float) – float -

Return type
Point2D

:
Point2D:

__truediv__(*scalar*)

/ =

Parameters
scalar (float) – float -

Return type
Point2D

:

Point2D:

:

ZeroDivisionError: 0

__eq__(*other*)

:

- math.isclose

Parameters
other (object) – object -

Return type
bool

:

bool:

__hash__()

Return type
int

:

- Point2D

:

int:

Chapter 8

Vectors API

planar_geometry/curve/vector2_d.py

: Vector2D : Vector2D : 0.1.0 : wangheng <wangfaofao@gmail.com>

:

- planar_geometry.abstracts:
- planar_geometry.point: Point2D

:

from planar_geometry.curve import Vector2D

class planar_geometry.curve.vector2d.**Vector2D**(x, y)

Bases: `Curve`

:

$\vec{v} = (x, y) \in \mathbb{R}^2$

:

- : $|\vec{v}| = \sqrt{x^2 + y^2}$
- : $\theta = \arctan 2(y, x)$
- : $\hat{\vec{v}} = \frac{\vec{v}}{|\vec{v}|}$

:

x (float): x y (float): y

:

```
#
v1 = Vector2D(3, 4)
```

(continues on next page)

(continued from previous page)

```
v2 = Vector2D(1, 2)

#
dot_product = v1.dot(v2)         # : 11
cross_product = v1.cross(v2)     # : 2
normalized = v1.normalized()     # : (0.6, 0.8)

#
v_sum = v1.add(v2)               # (4, 6)
v_diff = v1.subtract(v2)         # (2, 2)
v_scaled = v1.multiply(2)        # (6, 8)
```

Parameters

- **x** (float) – float - x
- **y** (float) – float - y

length()

Return type
float

:

$\vec{v} = (x, y)$

$$|\vec{v}| = \sqrt{x^2 + y^2}$$

:

float:

:

O(1) -

:

```
v = Vector2D(3, 4)
length = v.length()  #  5.0
```

length_squared()

Return type
float

:

$$|\vec{v}|^2 = x^2 + y^2$$

:

length()

:

float: 0

:

O(1) -

:

```
v = Vector2D(3, 4)
len_sq = v.length_squared()  #  25.0
#  v.length()

#
if v1.length_squared() < v2.length_squared():
    print("v1 ")  #
```

angle()

Return type
float

:

x

$$\theta = \arctan 2(y, x) \cdot \frac{180}{\pi}$$

:

- atan2
- [0, 360)
- 0° x

:

float: [0, 360)

:

O(1)

:

```
v1 = Vector2D(1, 0)      #  0.0
v2 = Vector2D(0, 1)      #  90.0
v3 = Vector2D(-1, 0)     #  180.0
v4 = Vector2D(0, -1)     #  270.0
```

angle_rad()

Return type
float

:

x

$$\theta = \arctan 2(y, x)$$

:

- atan2
- [0, 2)
-

:

float: [0, 2)

:

O(1)

:

```
import math
v = Vector2D(1, 1)
angle = v.angle_rad()  #  /4  0.785
assert abs(angle - math.pi / 4) < 1e-9
```

normalized()

Return type
Vector2D

:

$\vec{v}$

$$\hat{\vec{v}} = \frac{\vec{v}}{|\vec{v}|} = \left(\frac{x}{|\vec{v}|}, \frac{y}{|\vec{v}|}\right)$$

:

- (0, 0)
- 1
-

:

Vector2D: 1 (0, 0)

:

O(1)

:

```
v = Vector2D(3, 4)
u = v.normalized()  #  (0.6, 0.8)
assert abs(u.length() - 1.0) < 1e-9

#
zero = Vector2D(0, 0)
zero_normalized = zero.normalized()  #  (0, 0)
```

dot(*other*)

Return type
float

Parameters
other (Vector2D)

:

$\vec{u} = (x_1, y_1)$ $\vec{v} = (x_2, y_2)$

$$\vec{u} \cdot \vec{v} = x_1 x_2 + y_1 y_2 = |\vec{u}||\vec{v}| \cos(\theta)$$

θ

:

- > 0 ($\theta < 90$ř)
- $= 0$ ($\theta = 90$ř)
- < 0 ($\theta > 90$ř)

:

other (Vector2D):

:

float:

:

O(1)

:

-
-
- Lambert's law
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 2)
dot_result = u.dot(v)  #  3*1 + 4*2 = 11

#
a = Vector2D(1, 0)
b = Vector2D(0, 1)
assert a.dot(b) == 0  #

#
c = Vector2D(1, 1)
if u.dot(c) > 0:
    print("")
```

cross(*other*)

2D

Return type
`float`

Parameters
other (Vector2D)

:

$\vec{u} = (x_1, y_1)$ $\vec{v} = (x_2, y_2)$2D

$$\vec{u} \times \vec{v} = x_1 y_2 - y_1 x_2$$

z$|\vec{u}||\vec{v}| \sin(\theta)$

:

-
- $> 0\vec{v}$ $\vec{u}$ $(\theta > 0)$

- $= 0$
- $< 0 \vec{v} \ \vec{u} \ \ (\theta < 0)$

:

other (Vector2D):

:

float:

:

O(1)

:

- /
- Graham Scan
-
-
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 2)
cross_result = u.cross(v)  #  3*2 - 4*1 = 2

#
a = Vector2D(1, 0)
b = Vector2D(0, 1)
assert a.cross(b) > 0  # ba

#
c = Vector2D(2, 0)
assert a.cross(c) == 0  # 0

#
#  (0,0), (3,0), (0,4)  = ||/2 = 6
edge1 = Vector2D(3, 0)
edge2 = Vector2D(0, 4)
area = abs(edge1.cross(edge2)) / 2  # 6.0
```

perpendicular()

90

Return type
Vector2D

:

$\vec{v} = (x, y)$

$$\vec{v}_{\perp} = (-y, x)$$

:

- $0\vec{v} \cdot \vec{v}_{\perp} = 0$
- $|\vec{v}_{\perp}| = |\vec{v}|$
- 90 rotated(90ř)

:

Vector2D:

:

O(1)

:

```
v = Vector2D(3, 4)
perp = v.perpendicular()  #  (-4, 3)

#
assert v.dot(perp) == 0
assert abs(v.length() - perp.length()) < 1e-9

#
u = Vector2D(1, 0)
u_perp = u.perpendicular()  #  (0, 1)
```

rotated(*angle_deg*)

Return type
Vector2D

Parameters
angle_deg (*float*)

:

2D $\vec{v} = (x, y)$ θ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

:

- $|\vec{v}'| = |\vec{v}|$
-
- -

:

angle_deg (float):

-
-

:

Vector2D:

:

O(1)

:

-
-
- 2D

:

```
v = Vector2D(1, 0)

# 90
v90 = v.rotated(90)  #  (0, 1)

# 45
v45 = v.rotated(-45)

# 360
v360 = v.rotated(360)
assert v360.equals(v)

#
```

(continues on next page)

(continued from previous page)

```
original_len = v.length()
rotated_len = v.rotated(30).length()
assert abs(original_len - rotated_len) < 1e-9
```

projection(*other*)

Return type
Vector2D

Parameters
other (Vector2D)

:

$\vec{u}$ $\vec{v}$ $\vec{u}$ $\vec{v}$

$$\text{proj}_{\vec{v}}(\vec{u}) = \left(\frac{\vec{u} \cdot \vec{v}}{|\vec{v}|^2}\right)\vec{v}$$

$$\text{proj}_{\vec{v}}(\vec{u}) = (|\vec{u}|\cos(\theta))\hat{\vec{v}}$$

θ $\hat{\vec{v}}$

:

- $\vec{u}$ $\vec{v}$
- $\vec{v}$
- $= |\vec{u}|\cos(\theta)$

:

other (Vector2D):

:

Vector2D: (0, 0)

:

O(1)

:

-
-
-
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 0)  # x

proj = u.projection(v)  #  (3, 0)
assert proj.x == 3

#
v_diag = Vector2D(1, 1)
proj_diag = u.projection(v_diag)
#  = (3*1 + 4*1) / sqrt(2) = 7/sqrt(2)

#
zero = Vector2D(0, 0)
proj_zero = u.projection(zero)  #  (0, 0)
```

component(*direction*)

Parameters
direction (*Vector2D*) – Vector2D -

Return type
float

:
float:

add(*other*)

Parameters
other (*Vector2D*) – Vector2D -

Return type
Vector2D

:
Vector2D:

subtract(*other*)

Parameters
other (*Vector2D*) – Vector2D -

Return type
Vector2D

:
Vector2D:

multiply(*scalar*)

Parameters
scalar (float) – float -

Return type
Vector2D

:
Vector2D:

divide(*scalar*)

Parameters
scalar (float) – float -

Return type
Vector2D

:
Vector2D:

:
ZeroDivisionError: 0

negate()

Return type
Vector2D

:
Vector2D:

is_zero(*tolerance=1e-09*)

Parameters
tolerance (float) – float -

Return type
bool

:
bool:

equals(*other*, *tolerance=1e-09*)

Parameters

- **other** (*Vector2D*) – Vector2D -

- **tolerance** (float) – float -

Return type
bool

:
bool:

`to_tuple()`

Return type
tuple

:
tuple: (x, y)

`static from_tuple(data)`

Parameters
data (tuple) – tuple - (x, y)

Return type
Vector2D

:
Vector2D:

`static zero()`

Return type
Vector2D

:
Vector2D: (0, 0)

`static unit_x()`
X

Return type
Vector2D

:
Vector2D: (1, 0)

static unit_y()

Y

Return type
Vector2D

:

Vector2D: (0, 1)

__add__(*other*)

Return type
Vector2D

Parameters
other (Vector2D)

:

$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$

:

```
v1 = Vector2D(1, 2)
v2 = Vector2D(3, 4)
result = v1 + v2  # Vector2D(4, 6)
```

__mul__(*scalar*)

Return type
Vector2D

Parameters
scalar (*float*)

:

$k \cdot \vec{v} = (k \cdot x, k \cdot y)$

:

- k>0
- k=0
- k<0

:

```
v = Vector2D(2, 3)
result = v * 2  # Vector2D(4, 6)
```

__rmul__(*scalar*)

Return type
Vector2D

Parameters
scalar (*float*)

:
$\vec{v} \cdot k = (x \cdot k, y \cdot k)$

:

-
- *scalar * vector*

:

```
v = Vector2D(2, 3)
result = 2 * v  # Vector2D(4, 6)
#  v * 2
```

__eq__(*other*)

Return type
bool

Parameters
other (*object*)

:

- *math.isclose*
-
- 1e-9

:
other: Vector2D

:
bool:

:

```
v1 = Vector2D(1.0, 2.0)
v2 = Vector2D(1.0, 2.0)
v3 = Vector2D(1.000000001, 2.0)
```

(continues on next page)

(continued from previous page)

```
assert v1 == v2  # True
assert v1 == v3  # True
assert v1 != Vector2D(1.1, 2.0)  # False
```

8.1 Vector2D Class

class planar_geometry.curve.vector2d.**Vector2D**(x, y)

Bases: Curve

:

$\vec{v} = (x, y) \in \mathbb{R}^2$

:

- : $|\vec{v}| = \sqrt{x^2 + y^2}$
- : $\theta = \arctan 2(y, x)$
- : $\hat{\vec{v}} = \frac{\vec{v}}{|\vec{v}|}$

:

x (float): x y (float): y

:

```
#
v1 = Vector2D(3, 4)
v2 = Vector2D(1, 2)

#
dot_product = v1.dot(v2)          # : 11
cross_product = v1.cross(v2)      # : 2
normalized = v1.normalized()      # : (0.6, 0.8)

#
v_sum = v1.add(v2)                # (4, 6)
v_diff = v1.subtract(v2)          # (2, 2)
v_scaled = v1.multiply(2)         # (6, 8)
```

Parameters

- **x** (float) – float - x

- **y** (float) – float - y

length()

Return type
float

:

$\vec{v} = (x, y)$

$$|\vec{v}| = \sqrt{x^2 + y^2}$$

:

float:

:

O(1) -

:

```
v = Vector2D(3, 4)
length = v.length()  #  5.0
```

length_squared()

Return type
float

:

$$|\vec{v}|^2 = x^2 + y^2$$

:

length()

:

float: 0

:

O(1) -

:

```
v = Vector2D(3, 4)
len_sq = v.length_squared()  #  25.0
#  v.length()
```

(continues on next page)

(continued from previous page)

```
#
if v1.length_squared() < v2.length_squared():
    print("v1 ")  #
```

angle()

Return type
float

:

x

$$\theta = \arctan 2(y, x) \cdot \frac{180}{\pi}$$

:

- atan2
- [0, 360)
- 0° x

:

float: [0, 360)

:

O(1)

:

```
v1 = Vector2D(1, 0)      #  0.0
v2 = Vector2D(0, 1)      #  90.0
v3 = Vector2D(-1, 0)     #  180.0
v4 = Vector2D(0, -1)     #  270.0
```

angle_rad()

Return type
float

:

x

$$\theta = \arctan 2(y, x)$$

:

- atan2
- [0, 2)
-

:

float: [0, 2)

:

O(1)

:

```
import math
v = Vector2D(1, 1)
angle = v.angle_rad()  #  /4  0.785
assert abs(angle - math.pi / 4) < 1e-9
```

normalized()

Return type
Vector2D

:

$\vec{v}$

$$\hat{\vec{v}} = \frac{\vec{v}}{|\vec{v}|} = \left(\frac{x}{|\vec{v}|}, \frac{y}{|\vec{v}|}\right)$$

:

- (0, 0)
- 1
-

:

Vector2D: 1 (0, 0)

:

O(1)

:

```
v = Vector2D(3, 4)
u = v.normalized()  #  (0.6, 0.8)
assert abs(u.length() - 1.0) < 1e-9
```

(continues on next page)

(continued from previous page)

```
#
zero = Vector2D(0, 0)
zero_normalized = zero.normalized()  #  (0, 0)
```

dot(*other*)

Return type
float

Parameters
other (Vector2D)

:

$\vec{u} = (x_1, y_1)$ $\vec{v} = (x_2, y_2)$

$$\vec{u} \cdot \vec{v} = x_1x_2 + y_1y_2 = |\vec{u}||\vec{v}| \cos(\theta)$$

θ

:

- > 0 ($\theta < 90$ř)
- $= 0$ ($\theta = 90$ř)
- < 0 ($\theta > 90$ř)

:

other (Vector2D):

:

float:

:

O(1)

:

-
-
- Lambert's law
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 2)
dot_result = u.dot(v)  #  3*1 + 4*2 = 11

#
a = Vector2D(1, 0)
b = Vector2D(0, 1)
assert a.dot(b) == 0  #

#
c = Vector2D(1, 1)
if u.dot(c) > 0:
    print("")
```

cross(*other*)

2D

Return type
`float`

Parameters
other (Vector2D)

:

$\vec{u} = (x_1, y_1)$ $\vec{v} = (x_2, y_2)$2D

$$\vec{u} \times \vec{v} = x_1 y_2 - y_1 x_2$$

z$|\vec{u}||\vec{v}| \sin(\theta)$

:

-
- $> 0\vec{v}$ $\vec{u}$ $(\theta > 0)$
- $= 0$
- $< 0\vec{v}$ $\vec{u}$ $(\theta < 0)$

:

other (Vector2D):

:

float:

:

O(1)

:

- /
- Graham Scan
-
-
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 2)
cross_result = u.cross(v)  #  3*2 - 4*1 = 2

#
a = Vector2D(1, 0)
b = Vector2D(0, 1)
assert a.cross(b) > 0  # ba

#
c = Vector2D(2, 0)
assert a.cross(c) == 0  # 0

#
#  (0,0), (3,0), (0,4)  = ||/2 = 6
edge1 = Vector2D(3, 0)
edge2 = Vector2D(0, 4)
area = abs(edge1.cross(edge2)) / 2  # 6.0
```

perpendicular()

90

Return type
Vector2D

:

$\vec{v} = (x, y)$

$$\vec{v}_\perp = (-y, x)$$

:

- $0\vec{v} \cdot \vec{v}_\perp = 0$
- $|\vec{v}_\perp| = |\vec{v}|$
- 90 rotated(90ř)

:

Vector2D:

:

O(1)

:

```
v = Vector2D(3, 4)
perp = v.perpendicular()  #  (-4, 3)

#
assert v.dot(perp) == 0
assert abs(v.length() - perp.length()) < 1e-9

#
u = Vector2D(1, 0)
u_perp = u.perpendicular()  #  (0, 1)
```

rotated(*angle_deg*)

Return type

Vector2D

Parameters

angle_deg (*float*)

:

2D $\vec{v} = (x, y)$ θ

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

:

- $|\vec{v'}| = |\vec{v}|$
-
- -

:

angle_deg (float):

-
-

:

Vector2D:

:

O(1)

:

-
-
- 2D

:

```
v = Vector2D(1, 0)

# 90
v90 = v.rotated(90)  #  (0, 1)

# 45
v45 = v.rotated(-45)

# 360
v360 = v.rotated(360)
assert v360.equals(v)

#
original_len = v.length()
rotated_len = v.rotated(30).length()
assert abs(original_len - rotated_len) < 1e-9
```

projection(*other*)

Return type
Vector2D

Parameters
other (Vector2D)

:

$\vec{u}$ $\vec{v}$ $\vec{u}$ $\vec{v}$

$$\text{proj}_{\vec{v}}(\vec{u}) = \left(\frac{\vec{u} \cdot \vec{v}}{|\vec{v}|^2}\right) \vec{v}$$

$$\text{proj}_{\vec{v}}(\vec{u}) = (|\vec{u}|\cos(\theta))\hat{\vec{v}}$$

θ $\hat{\vec{v}}$

:

- $\vec{u}$ $\vec{v}$
- $\vec{v}$
- $= |\vec{u}|\cos(\theta)$

:

other (Vector2D):

:

Vector2D: (0, 0)

:

O(1)

:

-
-
-
-

:

```
u = Vector2D(3, 4)
v = Vector2D(1, 0)  # x

proj = u.projection(v)  #  (3, 0)
assert proj.x == 3

#
v_diag = Vector2D(1, 1)
proj_diag = u.projection(v_diag)
#  = (3*1 + 4*1) / sqrt(2) = 7/sqrt(2)

#
zero = Vector2D(0, 0)
proj_zero = u.projection(zero)  #  (0, 0)
```

component(*direction*)

Parameters
direction (*Vector2D*) – Vector2D -

Return type
float

:
float:

add(*other*)

Parameters
other (*Vector2D*) – Vector2D -

Return type
Vector2D

:
Vector2D:

subtract(*other*)

Parameters
other (*Vector2D*) – Vector2D -

Return type
Vector2D

:
Vector2D:

multiply(*scalar*)

Parameters
scalar (float) – float -

Return type
Vector2D

:
Vector2D:

divide(*scalar*)

Parameters
scalar (float) – float -

Return type
Vector2D

:

Vector2D:

:

ZeroDivisionError: 0

negate()

Return type
Vector2D

:

Vector2D:

is_zero(*tolerance=1e-09*)

Parameters
tolerance (float) – float -

Return type
bool

:

bool:

equals(*other*, *tolerance=1e-09*)

Parameters

- **other** (*Vector2D*) – Vector2D -
- **tolerance** (float) – float -

Return type
bool

:

bool:

to_tuple()

Return type
tuple

:

tuple: (x, y)

static from_tuple(*data*)

Parameters
data (tuple) – tuple - (x, y)

Return type
Vector2D

:
Vector2D:

static zero()

Return type
Vector2D

:
Vector2D: (0, 0)

static unit_x()
X

Return type
Vector2D

:
Vector2D: (1, 0)

static unit_y()
Y

Return type
Vector2D

:
Vector2D: (0, 1)

__add__(*other*)

Return type
Vector2D

Parameters
other (Vector2D)

:
$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$

:

```
v1 = Vector2D(1, 2)
v2 = Vector2D(3, 4)
result = v1 + v2  # Vector2D(4, 6)
```

__mul__(*scalar*)

Return type
Vector2D

Parameters
scalar (*float*)

:
$k \cdot \vec{v} = (k \cdot x, k \cdot y)$

:

- k>0
- k=0
- k<0

:

```
v = Vector2D(2, 3)
result = v * 2  # Vector2D(4, 6)
```

__rmul__(*scalar*)

Return type
Vector2D

Parameters
scalar (*float*)

:
$\vec{v} \cdot k = (x \cdot k, y \cdot k)$

:

-
- *scalar * vector*

:

```
v = Vector2D(2, 3)
result = 2 * v  # Vector2D(4, 6)
#  v * 2
```

__eq__(*other*)

Return type
bool

Parameters
other (*object*)

:

- *math.isclose*
-
- 1e-9

:

other: Vector2D

:

bool:

:

```
v1 = Vector2D(1.0, 2.0)
v2 = Vector2D(1.0, 2.0)
v3 = Vector2D(1.000000001, 2.0)

assert v1 == v2  # True
assert v1 == v3  # True
assert v1 != Vector2D(1.1, 2.0)  # False
```

Chapter 9

Lines API

9.1 Line Class

class planar_geometry.curve.line.**Line**(*point*, *direction*)

Bases: Curve

:

-
-

:

point: Point2D - direction: Vector2D -

:

line = Line(Point2D(0, 0), Vector2D(1, 1))

Parameters

- **point** (*Point2D*) – Point2D -
- **direction** (*Vector2D*) – Vector2D -

length()

Return type

float

:

float:

get_intersection(*other*, *tolerance=1e-09*)

Return type
Point2D | None

Parameters

- **other** (Line)
- **tolerance** (*float*)

:

:

1 P1 d12 P2 d2

$$L_1 : \vec{r} = P_1 + t \cdot \vec{d1}$$
$$L_2 : \vec{r} = P_2 + s \cdot \vec{d2}$$

P1 + t*d1 = P2 + s*d2

$$t = \frac{(P_2 - P_1) \times \vec{d2}}{\vec{d1} \times \vec{d2}}$$

× 2D

:

- d1 × d2 = 0
-
-

:

other (Line): tolerance (float): 1e-9

:

Point2D:

:

ValueError:

:

O(1) -

:

- Ray Tracing
-
-

:

```
from planar_geometry import Line, Point2D, Vector2D

# y = 2
line1 = Line(Point2D(0, 2), Vector2D(1, 0))

# x = 3
line2 = Line(Point2D(3, 0), Vector2D(0, 1))

#  (3, 2)
intersection = line1.get_intersection(line2)
assert abs(intersection.x - 3) < 1e-9
assert abs(intersection.y - 2) < 1e-9

#  -
line3 = Line(Point2D(0, 0), Vector2D(1, 0))
line4 = Line(Point2D(0, 1), Vector2D(1, 0))
try:
    line3.get_intersection(line4)
except ValueError:
    print("Lines are parallel")
```

get_distance_to_point(*point*)

Return type
float

Parameters
point (Point2D)

:

:

P A d AP = P - A

$$d = |\vec{AP} \times \vec{d}|$$

d

$$d = \frac{|\vec{AP} \times \vec{d}|}{|\vec{d}|} = |\vec{AP} \times \hat{d}|$$

$\hat{d}$

:

point (Point2D):

:

float:

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Line, Point2D, Vector2D

# y = 0 x
line = Line(Point2D(0, 0), Vector2D(1, 0))

#  (0, 5)  5
dist = line.get_distance_to_point(Point2D(0, 5))
assert abs(dist - 5.0) < 1e-9

#  (3, 4)  y = 0  4
dist2 = line.get_distance_to_point(Point2D(3, 4))
assert abs(dist2 - 4.0) < 1e-9
```

get_closest_point(*point*)

Return type
Point2D

Parameters
point (Point2D)

:

P L A d F L P

:

AP = P - A

$$t = A\vec{P} \cdot \hat{d}$$
$$F = A + t \cdot \hat{d}$$

$\hat{d}$ t

t A d

:

point (Point2D):

:

Point2D:

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Line, Point2D, Vector2D

# y = 0
line = Line(Point2D(0, 0), Vector2D(1, 0))

#  (3, 5)  (3, 0)
foot = line.get_closest_point(Point2D(3, 5))
assert abs(foot.x - 3.0) < 1e-9
assert abs(foot.y - 0.0) < 1e-9

#  (0, 0)
foot2 = line.get_closest_point(Point2D(0, 0))
assert abs(foot2.x - 0.0) < 1e-9
assert abs(foot2.y - 0.0) < 1e-9
```

contains_point(*point*, *tolerance=1e-09*)

Return type
bool

Parameters

- **point** (Point2D)
- **tolerance** (*float*)

:

P L 0

:

1. P L F

2. P F

$$P \text{ on } L \iff |P - F| < \text{tolerance}$$

:

point (Point2D): tolerance (float): 1e-9

:

bool: True

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Line, Point2D, Vector2D

#  (0,0) (1,1) y = x
line = Line(Point2D(0, 0), Vector2D(1, 1))

#  (3, 3)  y = x
assert line.contains_point(Point2D(3, 3))

#  (3, 4)
assert not line.contains_point(Point2D(3, 4))

#
assert line.contains_point(Point2D(0, 0))
```

__repr__()

Return type
str

:

- Line(point, direction=unit_direction_vector)
-

:

str:

:

```
from planar_geometry import Line, Point2D, Vector2D

line = Line(Point2D(0, 0), Vector2D(3, 4))
print(repr(line))
# : Line(Point2D(0.0, 0.0), direction=Vector2D(0.6, 0.8))
```

9.2 LineSegment Class

class planar_geometry.curve.line_segment.**LineSegment**(*start*, *end*)

Bases: Curve

:

-
-

:

start: Point2D - end: Point2D -

:

s = LineSegment(Point2D(0, 0), Point2D(3, 4)) print(s.length())

Parameters

- **start** (*Point2D*) – Point2D -
- **end** (*Point2D*) – Point2D -

length()

Return type
float

:

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

:

float:

:

O(1) -

:

```
from planar_geometry import LineSegment, Point2D

#  (0,0)  (3,4) 5
seg = LineSegment(Point2D(0, 0), Point2D(3, 4))
assert abs(seg.length() - 5.0) < 1e-9
```

midpoint()

Return type
Point2D

:

Point2D:

direction()

Return type
Vector2D

:

Vector2D:

contains_point(*point*, *tolerance=1e-09*)

Return type
bool

Parameters

- **point** (Point2D)
- **tolerance** (*float*)

:

P AB 1. P AB 0 2. P A B 0 <= t <= 1

:

P = A + t*(B - A)

$$t = \frac{(P - A) \cdot (B - A)}{|B - A|^2}$$

$$0 \le t \le 1 \quad \text{and} \quad \text{distance}(P, \text{line}) < \text{tolerance}$$

:

point (Point2D): tolerance (float): 1e-9

:

bool: True

:

O(1) -

:

-
-
-

:

```
from planar_geometry import LineSegment, Point2D

#  (0,0)  (4,0)
seg = LineSegment(Point2D(0, 0), Point2D(4, 0))

#  (2,0)
assert seg.contains_point(Point2D(2, 0))

#
assert seg.contains_point(Point2D(0, 0))
assert seg.contains_point(Point2D(4, 0))

#  (5,0)
assert not seg.contains_point(Point2D(5, 0))

#  (2,1)
assert not seg.contains_point(Point2D(2, 1))
```

get_parameter(*point*)

t

Return type
float

Parameters
point (Point2D)

:

t

$$P = A + t \cdot (B - A)$$

A B P

$$t = \frac{(P - A) \cdot (B - A)}{|B - A|^2}$$

:

- t = 0 A
- t = 1 B
- 0 < t < 1
- t < 0
- t > 1

:

point (Point2D):

:

float: t

:

O(1) -

:

-
-
-

:

```
from planar_geometry import LineSegment, Point2D

seg = LineSegment(Point2D(0, 0), Point2D(4, 0))

#
t_start = seg.get_parameter(Point2D(0, 0))
assert abs(t_start - 0.0) < 1e-9

#
t_end = seg.get_parameter(Point2D(4, 0))
assert abs(t_end - 1.0) < 1e-9

#
t_mid = seg.get_parameter(Point2D(2, 0))
```

(continues on next page)

(continued from previous page)

```
assert abs(t_mid - 0.5) < 1e-9

#
t_outside = seg.get_parameter(Point2D(5, 0))
assert abs(t_outside - 1.25) < 1e-9  # > 1
```

get_closest_point(*point*)

Return type
Point2D

Parameters
point (Point2D)

:

P AB P

1. P AB F t
2. 0 <= t <= 1 F
3. t < 0 A
4. t > 1 B

$$F = A + t \cdot (B - A), \quad t = \text{clamp}(t, 0, 1)$$

:

point (Point2D):

:

Point2D:

:

O(1) -

:

-
- -
-

:

```
from planar_geometry import LineSegment, Point2D

seg = LineSegment(Point2D(0, 0), Point2D(4, 0))
```

(continues on next page)

(continued from previous page)

```
# (2, 3) (2, 0)
closest = seg.get_closest_point(Point2D(2, 3))
assert abs(closest.x - 2.0) < 1e-9
assert abs(closest.y - 0.0) < 1e-9

# (5, 0) (4, 0)
closest2 = seg.get_closest_point(Point2D(5, 0))
assert abs(closest2.x - 4.0) < 1e-9

# (-1, 0) (0, 0)
closest3 = seg.get_closest_point(Point2D(-1, 0))
assert abs(closest3.x - 0.0) < 1e-9
```

get_distance_to_point(*point*)

Return type
float

Parameters
point (Point2D)

:

P AB P

$$d = \min_{F \in AB} |P - F|$$

F t

:

1. t
2. t [0, 1]
3.

:

point (Point2D):

:

float:

:

O(1) -

:

•

-
-

:

```
from planar_geometry import LineSegment, Point2D

seg = LineSegment(Point2D(0, 0), Point2D(4, 0))

#  (2, 3)  3
dist = seg.get_distance_to_point(Point2D(2, 3))
assert abs(dist - 3.0) < 1e-9

#  0
dist2 = seg.get_distance_to_point(Point2D(2, 0))
assert abs(dist2 - 0.0) < 1e-9

#  (5, 0)  (4, 0) 1
dist3 = seg.get_distance_to_point(Point2D(5, 0))
assert abs(dist3 - 1.0) < 1e-9
```

__eq__(*other*)

Parameters
other (object) – object -

Return type
bool

:
bool:

Chapter 10

Circles API

class planar_geometry.surface.circle.**Circle**(*center*, *radius*)

Bases: Surface

:

-
-

:

center: Point2D - radius: float -

:

circle = Circle(Point2D(0, 0), 5.0) print(circle.area()) # 78.54

Parameters

- **center** (*Point2D*) – Point2D -
- **radius** (float) – float -

:

ValueError:

TOLERANCE: float = 1e-06

static from_diameter(*p1*, *p2*)

Parameters

- **p1** (*Point2D*) – Point2D -
- **p2** (*Point2D*) – Point2D -

Return type
Circle

:

Circle:

area()

Return type
float

:

$$A = \pi r^2$$

r

:

float:

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Circle, Point2D
import math

#  1
circle = Circle(Point2D(0, 0), 1.0)
assert abs(circle.area() - math.pi) < 1e-9

#  5  25
circle2 = Circle(Point2D(0, 0), 5.0)
assert abs(circle2.area() - 25 * math.pi) < 1e-9
```

perimeter()

Return type
float

:

$$C = 2\pi r = \pi d$$

r d = 2r

:

-
-

:

float:

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Circle, Point2D
import math

#  1  2
circle = Circle(Point2D(0, 0), 1.0)
assert abs(circle.perimeter() - 2 * math.pi) < 1e-9

#  5  10
circle2 = Circle(Point2D(0, 0), 5.0)
assert abs(circle2.perimeter() - 10 * math.pi) < 1e-9
```

get_bounds()

(AABB)

Return type

tuple

:

tuple: (x_min, y_min, x_max, y_max)

`get_center()`

Return type
Point2D

:

Point2D:

`contains_point(`*point*`)`

Return type
`bool`

Parameters
point (Point2D)

:

P

C r P

$$|P - C| \leq r$$

tolerance

$$|P - C| \leq r + \text{tolerance}$$

:

- $<$ r
- $=$ r
- $>$ r

:

point (Point2D):

:

bool: True

:

O(1) -

:

-
-
-

:

```
from planar_geometry import Circle, Point2D

#  5
circle = Circle(Point2D(0, 0), 5.0)

#
assert circle.contains_point(Point2D(0, 0))

#  (5, 0)
assert circle.contains_point(Point2D(5, 0))

#  (3, 4)
assert circle.contains_point(Point2D(3, 4))

#  (6, 0)
assert not circle.contains_point(Point2D(6, 0))
```

get_circumference()

Return type
float

:

perimeter() """"

:

perimeter()

$$C = 2\pi r$$

:

float: perimeter()

:

O(1) -

:

```
from planar_geometry import Circle, Point2D
import math

circle = Circle(Point2D(0, 0), 3.0)
```

(continues on next page)

(continued from previous page)

```
#
assert abs(circle.get_circumference() - circle.perimeter()) < 1e-9

#  6
assert abs(circle.get_circumference() - 6 * math.pi) < 1e-9
```

equals(*other*, *tolerance=1e-06*)

Parameters

- **other** (object) – object -
- **tolerance** (float) – float -

Return type
bool

:
bool:

10.1 Ellipse Class

class planar_geometry.surface.ellipse.**Ellipse**(*center*, *semi_major*, *semi_minor*, *rotation=0.0*)

Bases: Surface

:

-
-

:
center: Point2D - semi_major: float - semi_minor: float - rotation: float -

:
ellipse = Ellipse(Point2D(0, 0), 5.0, 3.0) print(ellipse.area()) # 47.12

Parameters

- **center** (*Point2D*) – Point2D -
- **semi_major** (float) – float - >= semi_minor
- **semi_minor** (float) – float -

- **rotation** (float) – float -

:

ValueError: semi_major < semi_minor

TOLERANCE: float = 1e-06

static from_center_and_axes(*center*, *major_axis*, *minor_axis*, *rotation=0.0*)

Parameters

- **center** (*Point2D*) – Point2D -
- **major_axis** (float) – float -
- **minor_axis** (float) – float -
- **rotation** (float) – float -

Return type
Ellipse

:

Ellipse:

static from_foci_and_point(*focus1*, *focus2*, *point*)

Parameters

- **focus1** (*Point2D*) – Point2D -
- **focus2** (*Point2D*) – Point2D -
- **point** (*Point2D*) – Point2D -

Return type
Ellipse

:

Ellipse:

area()

Return type
float

:

float: (* a * b)

perimeter()

Return type
float

:

- Ramanujan
-

:
float:

eccentricity()

Return type
float

:
float: e = sqrt(1 - b^2/a^2)

focal_distance()

Return type
float

:
float: c = sqrt(a^2 - b^2)

foci()

Return type
Tuple[*Point2D*, *Point2D*]

:
Tuple[Point2D, Point2D]: (focus1, focus2)

get_bounds()
(AABB)

Return type
tuple

:
tuple: (x_min, y_min, x_max, y_max)

get_center()

Return type
Point2D

:
Point2D:

contains_point(*point*)

:

-
-

Parameters
point (*Point2D*) – Point2D -

Return type
bool

:
bool: True

get_major_axis_endpoints()

Return type
Tuple[*Point2D*, *Point2D*]

:
Tuple[Point2D, Point2D]: (end1, end2)

get_minor_axis_endpoints()

Return type
Tuple[*Point2D*, *Point2D*]

:
Tuple[Point2D, Point2D]: (end1, end2)

equals(*other*, *tolerance=1e-06*)

Parameters

- **other** (object) – object -
- **tolerance** (float) – float -

Return type
`bool`

:

bool:

Chapter 11

Polygons API

11.1 Polygon Class

class planar_geometry.surface.polygon.**Polygon**(*vertices*)

Bases: Surface

:

-
-
-

:

vertices: List[Point2D] -

:

```
#
tri = Polygon([
    Point2D(0, 0),
    Point2D(3, 0),
    Point2D(0, 4)
])
print(tri.area())  # 6.0

#
quad = Polygon([
    Point2D(0, 0),
    Point2D(4, 0),
```

(continues on next page)

(continued from previous page)

```
    Point2D(4, 3),
    Point2D(0, 3)
])
print(quad.area())  # 12.0
```

:

- vertices 3
-

Parameters
vertices (List[*Point2D*]) – List[Point2D] -

:
ValueError: 3

TOLERANCE: float = 1e-06

static from_points(*points*)

Parameters
points (List[*Point2D*]) – List[Point2D] -

Return type
Polygon

:
Polygon:

static regular(*n*, *center*, *radius*, *rotation=0.0*)

Parameters

- **n** (int) – int -
- **center** (*Point2D*) – Point2D -
- **radius** (float) – float -
- **rotation** (float) – float -

Return type
Polygon

:
Polygon:

:

ValueError: 3

static triangle(*p1*, *p2*, *p3*)

Parameters

- **p1** (*Point2D*) – Point2D -
- **p2** (*Point2D*) – Point2D -
- **p3** (*Point2D*) – Point2D -

Return type

Polygon

:

Polygon:

static rectangle(*p1*, *p2*, *p3*, *p4*)

Parameters

- **p1-p4** – Point2D -
- **p1** (Point2D)
- **p2** (Point2D)
- **p3** (Point2D)
- **p4** (Point2D)

Return type

Polygon

:

Polygon:

area()

Return type

float

:

Shoelace Formula Gauss Area Formula

:

n P0 = (x0, y0), P1 = (x1, y1), . . . , Pn-1 = (xn-1, yn-1)

$$A = \frac{1}{2}\left|\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right|$$

$P_n = P_0$

$$A = \frac{1}{2}|x_0(y_1 - y_{n-1}) + x_1(y_2 - y_0) + ... + x_{n-1}(y_0 - y_{n-2})|$$

:

-
- O(n)
-
-

:

float:

:

O(n) - n

:

-
-
-

:

```
# (0,0), (4,0), (4,3), (0,3)
rect = Polygon([Point2D(0, 0), Point2D(4, 0),
                Point2D(4, 3), Point2D(0, 3)])
assert abs(rect.area() - 12.0) < 1e-9

# (0,0), (3,0), (0,4)
tri = Polygon([Point2D(0, 0), Point2D(3, 0), Point2D(0, 4)])
assert abs(tri.area() - 6.0) < 1e-9
```

test_simple_math()

: $E = mc^2$: $A = \frac{1}{2}|\sum(x_i y_{i+1} - x_{i+1} y_i)|$

Return type
float

perimeter()

Return type
float

:

n P0, P1, ..., Pn-1

$$L = \sum_{i=0}^{n-1} |P_i P_{i+1}|$$

$|P_i P_{i+1}|$

:

float:

:

O(n) - n

:

-
- ""
-

:

```
#  = 4
square = Polygon([Point2D(0, 0), Point2D(1, 0),
                  Point2D(1, 1), Point2D(0, 1)])
assert abs(square.perimeter() - 4.0) < 1e-9

#  (3-4-5)  = 12
tri = Polygon([Point2D(0, 0), Point2D(3, 0), Point2D(0, 4)])
assert abs(tri.perimeter() - 12.0) < 1e-9
```

get_bounds()

(AABB)

Return type
tuple

:

tuple: (x_min, y_min, x_max, y_max)

get_center()

Return type
Point2D

:

-

- :

  Point2D:

centroid()

Return type
Point2D

:

- :
- :

Point2D:

get_edges()

Return type
List[tuple]

:

List[Tuple[Point2D, Point2D]]:

get_edge_count()

Return type
int

:

int:

get_vertex_count()

Return type
int

:

int:

get_vertex(*index*)

Parameters
index (int) – int -

Return type
Point2D

:

Point2D:

`get_edge`(*index*)

Parameters
index (int) – int -

Return type
tuple

:

Tuple[Point2D, Point2D]:

`contains_point`(*point*)

Return type
bool

Parameters
point (Point2D)

:

Ray Casting Algorithm

:

P

-
-
-

1. 0
2. Pi -> Pi+1 - y y - x - 1
3. $$\text{inside} = (\text{intersection\_count} \mod 2) == 1$$

:

bool: True

:

O(n) - n

:

-
-
-

:

```
#
square = Polygon([Point2D(0, 0), Point2D(1, 0),
                  Point2D(1, 1), Point2D(0, 1)])

#
assert square.contains_point(Point2D(0.5, 0.5))

#
assert not square.contains_point(Point2D(2, 2))

#
assert square.contains_point(Point2D(0.5, 0))
```

is_convex()

Return type
bool

:

180

:

Cross Product

P0, P1, P2

$$\vec{v1} = P_1 - P_0, \quad \vec{v2} = P_2 - P_1$$
$$\text{cross} = v1_x \cdot v2_y - v1_y \cdot v2_x$$

-
-
- 0

$$\text{convex} = \forall i : \text{sign}(\text{cross}_i) = \text{constant}$$

:

-
-
-

:

bool:

:

O(n) - n

:

- SAT
-
-

:

```
#  -
square = Polygon([Point2D(0, 0), Point2D(1, 0),
                  Point2D(1, 1), Point2D(0, 1)])
assert square.is_convex()

#  -
star = Polygon([Point2D(0, 1), Point2D(0.2, 0.2),
                Point2D(1, 0), Point2D(0.3, 0.4),
                Point2D(0.5, -0.5)])
assert not star.is_convex()
```

is_simple()

Return type
bool

:

Self-intersection

:

Edge_i Edge_j $|i - j| \geq 2$

- - Edge_0 Edge_{n-1}

$$\text{simple} = \forall i, j : |i - j| \geq 2 \wedge \text{intersection}(E_i, E_j) = \emptyset$$

:

bool:

:

O(n^2) -

:

-
-
-

:

```
#  -
square = Polygon([Point2D(0, 0), Point2D(1, 0),
                  Point2D(1, 1), Point2D(0, 1)])
assert square.is_simple()

#  -
# butterfly = Polygon([...])  #
# assert not butterfly.is_simple()
```

is_regular()

Return type
bool

:

Regular Polygon n

$$\theta = \frac{(n-2) \times 180\check{r}}{n}$$

:

1.

$$|P_i - P_{i+1}| = L, \quad \forall i$$

2.

$$\angle P_{i-1} P_i P_{i+1} = \theta, \quad \forall i$$

:

1. TOLERANCE
2.
3.

:

bool:

:

O(n) - n

:

-
-
-

:

```
#
tri_eq = Polygon.regular(3, Point2D(0, 0), 1.0)
assert tri_eq.is_regular()

#
square = Polygon.regular(4, Point2D(0, 0), 1.0)
assert square.is_regular()

#  -
rect = Polygon([Point2D(0, 0), Point2D(2, 0),
                Point2D(2, 1), Point2D(0, 1)])
assert not rect.is_regular()
```

get_convex_hull()

Return type
Polygon

:

Convex Hull

:

Graham Scan O(n log n)

1. x y
2.
3.

4.

Cross Product

$$\text{cross}(O, A, B) = (A_x - O_x)(B_y - O_y) - (A_y - O_y)(B_x - O_x)$$

- cross > 0
- cross < 0
- cross $= 0$

:

Polygon:

:

O(n log n) -

:

-
-
-

:

```
#
points = Polygon([Point2D(0, 0), Point2D(2, 2),
                  Point2D(2, 0), Point2D(1, 1)])  # (1,1)
hull = points.get_convex_hull()
#
assert hull.get_vertex_count() == 3
```

11.2 Triangle Class

class planar_geometry.surface.triangle.**Triangle**(*vertices*)

Bases: *Polygon*

:

- Polygon
-
- from_points(), from_sides()

:

vertices: List[Point2D] - 3

:

```
#
tri = Triangle.from_points([
    Point2D(0, 0),
    Point2D(3, 0),
    Point2D(0, 4)
])
print(tri.area())  # 6.0

#
tri = Triangle.from_sides(3.0, 4.0, 5.0)
```

:

- vertices 3
-

Parameters
vertices (List[*Point2D*]) – List[Point2D] - 3

:
ValueError: 3

static from_points(*points*)

Parameters
points (List[*Point2D*]) – List[Point2D] - 3

Return type
Triangle

:
Triangle:

static from_sides(*a*, *b*, *c*)

Return type
Triangle

Parameters

- **a** (*float*)
- **b** (*float*)

- **c** (*float*)

:

a, b, cHeron's formula

$$s = \frac{a+b+c}{2}()$$

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

$$a+b>c, \quad a+c>b, \quad b+c>a$$

:

a (float): b (float): c (float):

:

Triangle:

:

ValueError: 0

:

O(1) -

:

```
#  (3-4-5)
tri_345 = Triangle.from_sides(3.0, 4.0, 5.0)
assert abs(tri_345.area() - 6.0) < 1e-9

#
tri_eq = Triangle.from_sides(1.0, 1.0, 1.0)

#
try:
    Triangle.from_sides(1.0, 2.0, 5.0)  # 1 + 2  5
except ValueError:
    print("")
```

get_side_lengths()

Return type
Tuple[float, float, float]

:

Tuple[float, float, float]: (a, b, c)

get_angles()

Return type
Tuple[float, float, float]

:

Law of Cosines a, b, c A, B, C

$$\cos(A) = \frac{b^2 + c^2 - a^2}{2bc}$$
$$\cos(B) = \frac{a^2 + c^2 - b^2}{2ac}$$
$$\cos(C) = \frac{a^2 + b^2 - c^2}{2ab}$$

$$A = \arccos(\cos(A)), \quad B = \arccos(\cos(B)), \quad C = \arccos(\cos(C))$$

:

- (degree)
- 180°
- cos clamp

:

Tuple[float, float, float]: (A, B, C)

:

O(1)

:

```
#  (3-4-5)
tri = Triangle.from_sides(3.0, 4.0, 5.0)
angles = tri.get_angles()
A, B, C = angles

# 90
assert any(abs(angle - 90.0) < 1e-6 for angle in angles)

# 180
assert abs(sum(angles) - 180.0) < 1e-6

#
tri_eq = Triangle.from_sides(1.0, 1.0, 1.0)
angles_eq = tri_eq.get_angles()
assert all(abs(angle - 60.0) < 1e-6 for angle in angles_eq)
```

`circumcenter()`

Return type
Point2D

:

:

- R
-
-

:

$P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_3 = (x_3, y_3)$

$$O_x = \frac{|(P_1|^2(P_2^y - P_3^y) + |P_2|^2(P_3^y - P_1^y) + |P_3|^2(P_1^y - P_2^y)|}{2D}$$
$$O_y = \frac{|(P_1|^2(P_3^x - P_2^x) + |P_2|^2(P_1^x - P_3^x) + |P_3|^2(P_2^x - P_1^x)|}{2D}$$

D

$$D = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

D 0

:

Point2D:

:

O(1)

:

-
-
-

:

```
#  -
tri = Triangle.from_sides(3.0, 4.0, 5.0)
circumcenter = tri.circumcenter()

#
v1, v2, v3 = tri.vertices
```

(continues on next page)

(continued from previous page)

```
d1 = circumcenter.distance_to(v1)
d2 = circumcenter.distance_to(v2)
d3 = circumcenter.distance_to(v3)
assert abs(d1 - d2) < 1e-9
assert abs(d2 - d3) < 1e-9
```

incenter()

Return type
Point2D

:

:

- r
-
-

:

P1, P2, P3 a, b, c (a = |P2P3|, b = |P3P1|, c = |P1P2|)

$$I = \frac{a \cdot P_1 + b \cdot P_2 + c \cdot P_3}{a + b + c}$$
$$I_x = \frac{a \cdot x_1 + b \cdot x_2 + c \cdot x_3}{a + b + c}$$
$$I_y = \frac{a \cdot y_1 + b \cdot y_2 + c \cdot y_3}{a + b + c}$$

$p = a + b + c$

:

Point2D:

:

O(1) -

:

-
-
-

:

```
#
tri = Triangle.from_sides(3.0, 4.0, 5.0)
incenter = tri.incenter()

#
r = tri.inradius()
#
assert tri.contains_point(incenter)
```

orthocenter()

Return type
Point2D

:

:

-
-
-

:

P1 = (x1, y1), P2 = (x2, y2), P3 = (x3, y3) P1P2 = (A, B) P1P3 = (C, D)

$$
\begin{aligned}
A = x_2 - x_1, \quad B = y_2 - y_1 \\
C = x_3 - x_1, \quad D = y_3 - y_1 \\
E = A(x_1 + x_2) + B(y_1 + y_2) \\
F = C(x_1 + x_3) + D(y_1 + y_3) \\
G = 2(A(x_2 - x_3) + B(y_2 - y_3)) \\
H_x = \frac{DE - BF}{G} \\
H_y = \frac{AF - CE}{G}
\end{aligned}
$$

G $G \approx 0$

:

Point2D:

:

O(1) -

:

-

-
-

:

```
#  -
tri = Triangle.from_sides(3.0, 4.0, 5.0)
orthocenter = tri.orthocenter()

#  -
tri_eq = Triangle.from_sides(1.0, 1.0, 1.0)
orthocenter_eq = tri_eq.orthocenter()
centroid_eq = tri_eq.centroid()
# assert orthocenter_eq  centroid_eq
```

centroid()

Return type
Point2D

:

- 2:1

:

Point2D:

circumradius()

Return type
`float`

:

-:

a, b, c A

$$R = \frac{a}{2\sin(A)} = \frac{b}{2\sin(B)} = \frac{c}{2\sin(C)}$$

$A = \frac{1}{2}bc\sin(A)$

$$R = \frac{abc}{4A}$$

A

$$s = \frac{a+b+c}{2} \quad ()$$

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

:

-
- R
-

:

float: 0 inf

:

O(1) -

:

-
-
-

:

```
#  (3-4-5) -  5
tri_345 = Triangle.from_sides(3.0, 4.0, 5.0)
R = tri_345.circumradius()
assert abs(R - 2.5) < 1e-9  # R = 5/2

#  a
tri_eq = Triangle.from_sides(1.0, 1.0, 1.0)
R_eq = tri_eq.circumradius()
# R = a / sqrt(3)  0.577
assert abs(R_eq - 1.0 / math.sqrt(3)) < 1e-9
```

inradius()

Return type
float

:

:

a, b, c A s

$$s = \frac{a+b+c}{2}$$

$$r = \frac{A}{s}$$

$$A = A_1 + A_2 + A_3 = \frac{1}{2}ar + \frac{1}{2}br + \frac{1}{2}cr = \frac{1}{2}(a+b+c)r = sr$$
$$\therefore r = \frac{A}{s}$$

:

-
- r
- r = (a - 2 · c) / 2 c

:

float: 0 0

:

O(1) -

:

-
-
-

:

```
#  (3-4-5)
tri_345 = Triangle.from_sides(3.0, 4.0, 5.0)
r = tri_345.inradius()
# 66 r = 6/6 = 1
assert abs(r - 1.0) < 1e-9

#  a
tri_eq = Triangle.from_sides(1.0, 1.0, 1.0)
r_eq = tri_eq.inradius()
# r = a / (2*sqrt(3))  0.289
assert abs(r_eq - 1.0 / (2 * math.sqrt(3))) < 1e-9
```

is_right_angled(*tolerance=1e-06*)

Parameters
tolerance (float) – float -

Return type
bool

:

bool:

is_equilateral(*tolerance=1e-06*)

Parameters
tolerance (float) – float -

Return type
bool

:

bool:

is_isosceles(*tolerance=1e-06*)

Parameters
tolerance (float) – float -

Return type
bool

:

bool:

get_circumcircle()

Return type
Circle

:

Circle:

get_incicle()

Return type
Circle

:

Circle:

__repr__()

Return type
str

:

- Triangle(Point2D(...), Point2D(...), Point2D(...))
-

:

str:

:

O(1) -

:

```
tri = Triangle.from_sides(3.0, 4.0, 5.0)
print(repr(tri))
# : Triangle(Point2D(0.0, 0.0), Point2D(3.0, 0.0), Point2D(...))
```

11.3 Rectangle Class

class planar_geometry.surface.rectangle.**Rectangle**(*vertices*)

Bases: Surface

:

- 4
-
-

:

vertices: List[Point2D] - 4

:

- Cython
-

:

```
#
rect = Rectangle.from_center_and_size(
    center=Point2D(0, 0),
    size=2.0,
    direction=Vector2D(1, 0)
)
```

(continues on next page)

(continued from previous page)

```
#
print(rect.area())
print(rect.perimeter())
```

:

- vertices 4
- : [, , ,]

Parameters
vertices (List[*Point2D*]) – List[Point2D] - 4

:
ValueError: 4

TOLERANCE: float = 1e-06

static from_center_and_size(*center*, *size*, *direction*)

:

-
- direction

Parameters

- **center** (*Point2D*) – Point2D -
- **size** (float) – float -
- **direction** (*Vector2D*) – Vector2D -

Return type
Rectangle

:
Rectangle:

static from_bounds(*x_min*, *y_min*, *x_max*, *y_max*)

Parameters

- **x_min** (float) – float - x
- **y_min** (float) – float - y

- **x_max** (float) – float - x
- **y_max** (float) – float - y

Return type
Rectangle

:
Rectangle:

area()

Return type
float

:
float:

perimeter()

Return type
float

:
float:

get_bounds()
(AABB)

Return type
tuple

:
tuple: (x_min, y_min, x_max, y_max)

get_edges()
4

Return type
List[tuple]

:
List[Tuple[Point2D, Point2D]]: [(v0,v1), (v1,v2), (v2,v3), (v3,v0)]

`get_edge_count()`

Return type
`int`

:
int: 4

`get_vertex_count()`

Return type
`int`

:
int: 4

`get_center()`

Return type
Point2D

:
Point2D:

`contains_point(point)`

:

-
- AABB

Parameters
point (*Point2D*) – Point2D -

Return type
`bool`

:
bool: True

`is_square(tolerance=1e-06)`

Parameters
tolerance (`float`) – float -

Return type
`bool`

:

bool:

Chapter 12

Utility Functions API

12.1 Geometry Utilities

planar_geometry/geometry_utils.py

: : : 0.01 : wangheng <wangfaofao@gmail.com>

:

- : line_segment_intersection
- : line_intersection
- : rectangle_intersection_points
- : polygon_intersection_points
- : point_to_segment_distance, point_to_line_distance
- : point_to_rectangle_distance, point_to_polygon_distance
- : angle_between, are_perpendicular, are_parallel
- : segments_distance, segments_closest_points

:

- math:
- typing:
- point:
- curve:
- surface:

:

from planar_geometry import Point2D, LineSegment, line_segment_intersection

s1 = LineSegment(Point2D(0, 0), Point2D(2, 2)) s2 = LineSegment(Point2D(0, 2), Point2D(2, 0)) intersection = line_segment_intersection(s1, s2)

planar_geometry.utils.geometry_utils.**line_segment_intersection**(*s1*, *s2*, *tolerance=1e-09*)

:

-
- 1: P(t) = p1 + t(p2-p1), t [0,1]
- 2: Q(s) = p3 + s(p4-p3), s [0,1]
- P(t) = Q(s)

Parameters

- **s1** (*LineSegment*) – LineSegment -
- **s2** (*LineSegment*) – LineSegment -
- **tolerance** (float) – float -

Return type
Point2D | None

:
Optional[Point2D]: None

:

-
-

planar_geometry.utils.geometry_utils.**line_intersection**(*l1*, *l2*, *tolerance=1e-09*)

:

-
-

Parameters

- **l1** (*Line*) – Line -
- **l2** (*Line*) – Line -
- **tolerance** (float) – float -

Return type
Point2D | None

:
Optional[Point2D]: None

:
ValueError:

planar_geometry.utils.geometry_utils.**rectangle_intersection_points**(*r1*, *r2*, *tolerance=1e-06*)

:

- 8
- 16
-

Parameters

- **r1** (*Rectangle*) – Rectangle -
- **r2** (*Rectangle*) – Rectangle -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:
List[Point2D]:

planar_geometry.utils.geometry_utils.**polygon_intersection_points**(*poly1*, *poly2*, *tolerance=1e-06*)

:

-
-
-

Parameters

- **poly1** (*Polygon*) – Polygon -
- **poly2** (*Polygon*) – Polygon -

- **tolerance** (float) – float -

Return type
List[*Point2D*]

:
List[Point2D]:

planar_geometry.utils.geometry_utils.**point_to_segment_distance**(*point*, *segment*)

Parameters

- **point** (*Point2D*) – Point2D -
- **segment** (*LineSegment*) – LineSegment -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**point_to_segment_closest_point**(*point*, *segment*)

Parameters

- **point** (*Point2D*) – Point2D -
- **segment** (*LineSegment*) – LineSegment -

Return type
Point2D

:
Point2D:

planar_geometry.utils.geometry_utils.**point_to_line_distance**(*point*, *line*)

Parameters

- **point** (*Point2D*) – Point2D -
- **line** (*Line*) – Line -

Return type
float

:

float:

planar_geometry.utils.geometry_utils.**point_to_line_closest_point**(*point*, *line*)

Parameters

- **point** (*Point2D*) – Point2D -
- **line** (*Line*) – Line -

Return type
Point2D

:

Point2D:

planar_geometry.utils.geometry_utils.**point_to_rectangle_distance**(*point*, *rect*)

:

- 0
-

Parameters

- **point** (*Point2D*) – Point2D -
- **rect** (*Rectangle*) – Rectangle -

Return type
float

:

float:

planar_geometry.utils.geometry_utils.**point_to_polygon_distance**(*point*, *poly*)

:

- 0
-

Parameters

- **point** (*Point2D*) – Point2D -
- **poly** (*Polygon*) – Polygon -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**angle_between**(*v1*, *v2*)

:

- [0, 180]

Parameters

- **v1** (*Vector2D*) – Vector2D -
- **v2** (*Vector2D*) – Vector2D -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**angle_between_rad**(*v1*, *v2*)

:

- [0,]

Parameters

- **v1** (*Vector2D*) – Vector2D -
- **v2** (*Vector2D*) – Vector2D -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**are_perpendicular**(*v1*, *v2*, *tolerance=1e-06*)

Parameters

- **v1** (*Vector2D*) – Vector2D -
- **v2** (*Vector2D*) – Vector2D -

- **tolerance** (float) – float -

Return type
bool

:
bool:

planar_geometry.utils.geometry_utils.**are_parallel**(*v1*, *v2*, *tolerance=1e-06*)

Parameters

- **v1** (*Vector2D*) – Vector2D -
- **v2** (*Vector2D*) – Vector2D -
- **tolerance** (float) – float -

Return type
bool

:
bool:

planar_geometry.utils.geometry_utils.**segments_distance**(*s1*, *s2*)

:

- 0
-

Parameters

- **s1** (*LineSegment*) – LineSegment -
- **s2** (*LineSegment*) – LineSegment -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**segments_closest_points**(*s1*, *s2*)

Parameters

- **s1** (*LineSegment*) – LineSegment -
- **s2** (*LineSegment*) – LineSegment -

Return type
Tuple[*Point2D*, *Point2D*]

:
Tuple[Point2D, Point2D]: (s1, s2)

planar_geometry.utils.geometry_utils.**point_line_distance_squared**(*px*, *py*, *line_point*, *line_dir*)

Parameters
- **px** (float) – float -
- **py** (float) – float -
- **line_point** (*Point2D*) – Point2D -
- **line_dir** (*Vector2D*) – Vector2D -

Return type
float

:
float:

planar_geometry.utils.geometry_utils.**bounding_box**(*points*)

Parameters
points (List[*Point2D*]) – List[Point2D] -

Return type
Tuple[float, float, float, float]

:
Tuple[float, float, float, float]: (x_min, y_min, x_max, y_max)

:
ValueError:

planar_geometry.utils.geometry_utils.**centroid**(*points*)

Parameters
points (List[*Point2D*]) – List[Point2D] -

Return type
Point2D

:
Point2D:

:

ValueError:

12.2 Intersection Operations

planar_geometry/utils/intersection_ops.py

: : --- : 0.2.0 : wangheng <wangfaofao@gmail.com>

:

- circle_line_intersection:
- circle_segment_intersection:
- circles_intersection:
- line_polygon_intersection_points:
- segment_polygon_intersection_points:
- ellipse_line_intersection:
- ellipse_circle_intersection:

:

- math:
- typing:
- point: Point2D
- curve: Line, LineSegment, Vector2D
- surface: Circle, Polygon, Ellipse
- geometry_utils:

:

- SOLID :
- : Vector2D.cross/dot
- :
- : List[Point2D]

:

from planar_geometry import Circle, Line, Vector2D, Point2D from planar_geometry.utils import circle_line_intersection

circle = Circle(Point2D(0, 0), 5) line = Line(Point2D(-10, 0), Vector2D(1, 0)) intersections = circle_line_intersection(circle, line)

planar_geometry.utils.intersection_ops.**circle_line_intersection**(*circle*, *line*, *tolerance=1e-10*)

:

-
- : $d = \frac{|ax+by+c|}{\sqrt{a^2+b^2}}$
- d > r: (0)
- d = r: (1)
- d < r: (2)
- x

Parameters

- **circle** (*Circle*) – Circle -
- **line** (*Line*) – Line -
- **tolerance** (float) – float - 1e-10

Return type
List[*Point2D*]

:

List[Point2D]: 0, 1, 2 x

:

-
-
-

:

1.
2.
3.

planar_geometry.utils.intersection_ops.**circle_segment_intersection**(*circle*, *segment*, *tolerance=1e-10*)

:

- circle_line_intersection
-
- /

Parameters

- **circle** (*Circle*) – Circle -
- **segment** (*LineSegment*) – LineSegment -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:
List[Point2D]: 0, 1, 2

:

-
-
-

planar_geometry.utils.intersection_ops.**circles_intersection**(*circle1*, *circle2*, *tolerance=1e-10*)

:

-
- 0, 1, 2
- /

Parameters

- **circle1** (*Circle*) – Circle -
- **circle2** (*Circle*) – Circle -
- **tolerance** (float) – float -

Return type
List[*Point2D*] | str

:
Union[List[Point2D], str]: - List[Point2D]: 0, 1, 2 - “tangent_external”: - “tangent_internal”: - “concentric”: - “no_intersection”:

:

-
-
-

:

1.
2.
3.

planar_geometry.utils.intersection_ops.**line_polygon_intersection_points**(*line*, *polygon*, *tolerance=1e-10*)

:

-
-
-
- t

Parameters

- **line** (*Line*) – Line -
- **polygon** (*Polygon*) – Polygon -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:

List[Point2D]:

:

-
-
-

:

1.
2.
3.
4. t

planar_geometry.utils.intersection_ops.**segment_polygon_intersection_points**(*segment*, *polygon*, *tolerance=1e-10*)

:

- line_polygon_intersection_points
-

Parameters

- **segment** (*LineSegment*) – LineSegment -
- **polygon** (*Polygon*) – Polygon -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:
List[Point2D]:

:

-
-

planar_geometry.utils.intersection_ops.**ellipse_line_intersection**(*ellipse*, *line*, *tolerance=1e-10*)

:

-
- : x = cx + a*cos(t), y = cy + b*sin(t)
- t
- 2

Parameters

- **ellipse** (*Ellipse*) – Ellipse -
- **line** (*Line*) – Line -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:

List[Point2D]: 0, 1, 2

:

-
-

planar_geometry.utils.intersection_ops.**ellipse_circle_intersection**(*ellipse*, *circle*, *tolerance=1e-10*)

:

-
-
-
- 4

Parameters

- **ellipse** (*Ellipse*) – Ellipse -
- **circle** (*Circle*) – Circle -
- **tolerance** (float) – float -

Return type
List[*Point2D*]

:

List[Point2D]: 0 4

:

-
-

12.3 Projection Operations

planar_geometry/utils/projection_ops.py

: : : 0.2.0 : wangheng <wangfaofao@gmail.com>

:

- nearest_point_on_geometry:
- polygon_nearest_points:
- point_to_circle_nearest:

:

- math:
- typing:
- geometry_utils:

:

from planar_geometry import Point2D, Circle, Polygon from planar_geometry.utils import nearest_point_on_geometry

point = Point2D(10, 0) circle = Circle(Point2D(0, 0), 5) nearest, dist = nearest_point_on_geometry(point, circle)

planar_geometry.utils.projection_ops.**nearest_point_on_geometry**(*point*, *geometry*, *tolerance=1e-10*)

:

- Line, LineSegment, Circle, Polygon, Ellipse
- (,)
- Line
- Circle, Polygon

Parameters

- **point** (*Point2D*) – Point2D -
- **geometry** – (Line, LineSegment, Circle, Polygon, Ellipse, Rectangle)
- **tolerance** (float) – float -

Return type

Tuple[*Point2D*, float]

:

Tuple[Point2D, float]: (,)

:

-
-
-

:

```
>>> from planar_geometry import Point2D, Circle
>>> point = Point2D(10, 0)
>>> circle = Circle(Point2D(0, 0), 5)
>>> nearest, dist = nearest_point_on_geometry(point, circle)
>>> print(f": {nearest}, : {dist}")
```

planar_geometry.utils.projection_ops.**polygon_nearest_points**(*poly1*, *poly2*, *tolerance=1e-10*)

:

- (1, 2,)
- ---
- AABB

Parameters

- **poly1** (*Polygon*) – Polygon -
- **poly2** (*Polygon*) – Polygon -
- **tolerance** (float) – float -

Return type

Tuple[*Point2D*, *Point2D*, float]

:

Tuple[Point2D, Point2D, float]: (1, 2,)

:

-
-
-

:

1.
2. poly1 poly2
3. poly2 poly1
4.

planar_geometry.utils.projection_ops.**point_to_circle_nearest**(*point*, *circle*, *tolerance=1e-10*)

:

- (, ,)
- [0, 360)
- 0°

Parameters

- **point** (*Point2D*) – Point2D -
- **circle** (*Circle*) – Circle -
- **tolerance** (float) – float -

Return type
Tuple[*Point2D*, float, float]

:

Tuple[Point2D, float, float]: (, ,)

:

-
-
-

:

```
>>> from planar_geometry import Point2D, Circle
>>> point = Point2D(10, 0)
>>> circle = Circle(Point2D(0, 0), 5)
>>> nearest, dist, angle = point_to_circle_nearest(point, circle)
>>> print(f" {angle}°  {dist}")
```

12.4 Angle Operations

planar_geometry/utils/angle_ops.py

: : : 0.2.0 : wangheng <wangfaofao@gmail.com>

:

- angle_between_three_points: ()
- polygon_vertex_angles:
- point_polar_angle: (0-360)

:

- math:
- typing:
- point: Point2D
- curve: Vector2D
- surface: Polygon

:

- SOLID :
- : Vector2D
- : (0-360)
- :

:

from planar_geometry import Point2D, Polygon from planar_geometry.utils import angle_between_three_points, polygon_vertex_angles

p1 = Point2D(0, 0) p2 = Point2D(1, 0) p3 = Point2D(1, 1)

angle = angle_between_three_points(p1, p2, p3) # 90

poly = Polygon([Point2D(0, 0), Point2D(2, 0), Point2D(2, 2), Point2D(0, 2)]) angles = polygon_vertex_angles(poly) # [90, 90, 90, 90]

`planar_geometry.utils.angle_ops.`**`angle_between_three_points`**`(`*p1*, *p2*, *p3*, *tolerance=1e-10*`)`

(p2)

:

- p2->p1 p2->p3
- [0, 360)

-
- p1=p2 p3=p2 0
- p1-p2-p3
- (360 - angle) polygon_vertex_angles()

Parameters

- **p1** (*Point2D*) – Point2D -
- **p2** (*Point2D*) – Point2D - ()
- **p3** (*Point2D*) – Point2D -
- **tolerance** (float) – float - (1e-10)

Return type
float

:

float: [0, 360)

:

- ()
-
-

:

1. v1 = p1 - p2, v2 = p3 - p2
2. atan2
3. [0, 360)

planar_geometry.utils.angle_ops.**polygon_vertex_angles**(*polygon*, *tolerance=1e-10*)

:

-
-
- (n-2)*180
- ""180

Parameters

- **polygon** (*Polygon*) – Polygon -
- **tolerance** (float) – float - (1e-10)

Return type
List[float]

:

List[float]:

:

-
-
- ()

:

1.
2.
3.
4. = 360 -

planar_geometry.utils.angle_ops.**point_polar_angle**(*point*, *reference=None*, *tolerance=1e-10*)

:

- point - reference
- [0, 360)
- 0°
- point = reference 0

Parameters

- **point** (*Point2D*) – Point2D -
- **reference** (*Point2D*) – Point2D - ()
- **tolerance** (float) – float - (1e-10)

Return type
float

:

float: [0, 360)

:

-
-
-

:

1. v = point - reference
2. atan2
3. [0, 360)

12.5 Coordinate Operations

planar_geometry/utils/coordinate_ops.py

: : : 0.2.0 : wangheng <wangfaofao@gmail.com>

:

- cartesian_to_polar:
- polar_to_cartesian:
- sort_points_by_angle:
- are_collinear:

:

- math:
- typing:
- point: Point2D
- curve: Vector2D

:

- SOLID :
- : Vector2D
- : (x, y) (,)
- : atan2

:

from planar_geometry import Point2D from planar_geometry.utils import cartesian_to_polar, polar_to_cartesian

distance, angle = cartesian_to_polar(Point2D(1, 0), Point2D(0, 0)) # distance=1.0, angle=0.0

point = polar_to_cartesian(1.0, 90.0, reference=Point2D(0, 0)) # point Point2D(0, 1)

planar_geometry.utils.coordinate_ops.**cartesian_to_polar**(*point*, *reference=None*, *tolerance=1e-10*)

:

- (,)
- >= 0
- [0, 360)
- 0°

Parameters

- **point** (*Point2D*) – Point2D -
- **reference** (*Point2D*) – Point2D - ()
- **tolerance** (float) – float - (1e-10)

Return type
Tuple[float, float]

:

Tuple[float, float]: (,)

:

-
-
- /

:

1. $\vec{v}$ = point − reference
2. = $|\vec{v}|$
3. = atan2($\vec{v}_y$, $\vec{v}_x$)

planar_geometry.utils.coordinate_ops.**polar_to_cartesian**(*distance*, *angle_deg*, *reference=None*)

:

-

- 0°
-

Parameters

- **distance** (float) – float - (>= 0)
- **angle_deg** (float) – float - ()
- **reference** (*Point2D*) – Point2D - ()

Return type
Point2D

:

Point2D:

:

-
-
-

:

1.
2. x = reference.x + distance * cos(angle)
3. y = reference.y + distance * sin(angle)

planar_geometry.utils.coordinate_ops.**sort_points_by_angle**(*points*, *reference=None*, *clockwise=False*)

:

-
- (0°)
- clockwise=True
-

Parameters

- **points** (List[*Point2D*]) – List[Point2D] -
- **reference** (*Point2D*) – Point2D - ()
- **clockwise** (bool) – bool - ()

Return type
List[*Point2D*]

:

List[Point2D]:

:

- (Graham Scan)
-
-

:

1. (,)
2. ()
3.

planar_geometry.utils.coordinate_ops.**are_collinear**(*points*, *tolerance=1e-10*)

:

- True
- < 2 True ()
- =0

Parameters

- **points** (List[*Point2D*]) – List[Point2D] -
- **tolerance** (float) – float - (1e-10)

Return type
bool

:

bool: True False

:

-
-
-

:

1. < 3 True

2.

3.

4. 0

12.6 Query Operations

planar_geometry/utils/query_ops.py

: : : 0.2.0 : wangheng <wangfaofao@gmail.com>

:

- point_side_of_segment: //
- circle_polygon_intersect:
- minimum_distance:
- within_distance:

:

- typing:
- projection_ops:

:

from planar_geometry import Point2D, LineSegment from planar_geometry.utils import point_side_of_segment

point = Point2D(1, 1) segment = LineSegment(Point2D(0, 0), Point2D(2, 0)) side = point_side_of_segment(point, segment) print(side) # : 1 ()

`planar_geometry.utils.query_ops.`**`point_side_of_segment`**`(`*point*, *segment*, *tolerance=1e-10*`)`

//

:

-
- 1 -1 0
-

Parameters

- **point** (*Point2D*) – Point2D -
- **segment** (*LineSegment*) – LineSegment -
- **tolerance** (float) – float -

Return type
int

:

int: 1 (), -1 (), 0 ()

:

-
-
-
- Graham

:

v1 = segment.end - segment.start v2 = point - segment.start cross = v1.cross(v2) - cross > 0: - cross < 0: - cross = 0:

:

```
>>> from planar_geometry import Point2D, LineSegment
>>> segment = LineSegment(Point2D(0, 0), Point2D(2, 0))
>>> left_point = Point2D(1, 1)
>>> right_point = Point2D(1, -1)
>>> on_line_point = Point2D(1, 0)
>>> print(point_side_of_segment(left_point, segment))  # 1
>>> print(point_side_of_segment(right_point, segment))  # -1
>>> print(point_side_of_segment(on_line_point, segment))  # 0
```

planar_geometry.utils.query_ops.**circle_polygon_intersect**(*circle*, *polygon*, *tolerance=1e-10*)

:

- ””
-
- True

Parameters

- **circle** (*Circle*) – Circle -
- **polygon** (*Polygon*) – Polygon -
- **tolerance** (float) – float -

Return type
bool

:

bool: True False

:

-
-
-

:

1.
2.
3.
4. <=

planar_geometry.utils.query_ops.**minimum_distance**(*geom1*, *geom2*, *tolerance=1e-10*)

:

-
- 0

Parameters

- **geom1** –
- **geom2** –
- **tolerance** (float) – float -

Return type

float

:

float:

:

-
-
-

:

```
>>> from planar_geometry import Point2D, Circle
>>> point = Point2D(10, 0)
>>> circle = Circle(Point2D(0, 0), 5)
>>> dist = minimum_distance(point, circle)
>>> print(dist)  # 5.0
```

planar_geometry.utils.query_ops.**within_distance**(*geom1*, *geom2*, *distance*, *tolerance=1e-10*)

:

- /
-
-

Parameters

- **geom1** –
- **geom2** –
- **distance** (float) – float -
- **tolerance** (float) – float -

Return type
bool

:

bool: True <= distance

:

- AABB
-
-

:

```
>>> from planar_geometry import Point2D, Circle
>>> point = Point2D(7, 0)
>>> circle = Circle(Point2D(0, 0), 5)
>>> print(within_distance(point, circle, 3))  # True ( 2 < 3)
>>> print(within_distance(point, circle, 1))  # False ( 2 > 1)
```

Chapter 13

Contributing

We welcome contributions to planar_geometry! This guide will help you get started.

13.1 Development Setup

1. Fork the repository on GitHub
2. Clone your fork locally:

```
git clone https://github.com/yourusername/planar_geometry.git
cd planar_geometry
```

3. Set up development environment:

```
make dev
```

This will: - Install the package in development mode - Install all development tools - Set up pre-commit hooks

13.2 Running Tests

Run all tests:

```
make test
```

Run with coverage:

```
make test-coverage
```

Run specific test file:

```
pytest tests/test_point.py -v
```

13.3 Code Quality

Format your code:

```
make format
```

Run linters:

```
make lint
```

Type checking:

```
make type
```

Or run all checks:

```
make check
```

13.4 Pre-commit Hooks

Pre-commit hooks automatically run checks before each commit:

```
make pre-commit-install
make pre-commit-run  # Run on all files
```

13.5 Submitting Changes

1. Create a new branch for your changes:

```
git checkout -b feature/your-feature-name
```

2. Make your changes and commit them:

```
git commit -m "Add your commit message"
```

3. Push to your fork:

```
git push origin feature/your-feature-name
```

4. Create a Pull Request on GitHub

13.6 Coding Standards

- Follow PEP 8 style guide
- Use type hints for all functions
- Write docstrings for all public functions
- Maintain 100% test coverage for new code
- Use black for formatting
- Use ruff for linting

13.7 Documentation

Build documentation locally:

```
make docs
```

View documentation:

```
make serve-docs
```

Open http://localhost:8000 in your browser.

13.8 Release Process

1. Update version in pyproject.toml
2. Update CHANGELOG.md
3. Create a git tag:

```
git tag -a v0.3.0 -m "Version 0.3.0"
```

4. Push tag to GitHub:

```
git push origin v0.3.0
```

5. Publish to PyPI:

```
make publish
```

Chapter 14

Architecture

14.1 Project Structure

```
planar_geometry/
├── src/planar_geometry/       # Source code
│   ├── __init__.py
│   ├── point.py               # Point class
│   ├── vector.py              # Vector class
│   ├── line.py                # Line class
│   ├── circle.py              # Circle class
│   ├── polygon.py             # Polygon class
│   └── ...
├── tests/                     # Unit tests
├── docs/                      # Documentation
├── pyproject.toml             # Project metadata
├── Makefile                   # Development tasks
├── tox.ini                    # Testing configuration
└── ...
```

14.2 Design Principles

14.2.1 SOLID Principles

The planar_geometry library follows SOLID principles:

Single Responsibility - Each class has one reason to change - Point handles point operations - Vector handles vector operations - etc.

Open/Closed - Open for extension via inheritance - Closed for modification of existing code - New geometric shapes can extend base classes

Liskov Substitution - Derived classes can substitute base classes - Consistent interface across all geometric objects

Interface Segregation - Clients depend only on interfaces they use - Each class provides focused, cohesive methods

Dependency Inversion - Depend on abstractions, not concrete classes - Use abstract base classes where appropriate

14.3 Core Concepts

14.3.1 Basic Types

Point - Represents a location in 2D space (x, y) - Immutable value object - Supports distance calculations

Vector - Represents direction and magnitude - Supports arithmetic operations - Can be rotated, scaled, normalized

Line - Represents an infinite line through two points - Supports intersection tests - Can contain points

Circle - Represents a circle with center and radius - Supports containment and distance tests - Can intersect with other circles

Polygon - Represents a closed shape with vertices - Supports area and perimeter calculations - Can contain points and check convexity

14.3.2 Mathematical Operations

Distance calculations use the Euclidean distance formula:

```
distance = sqrt((x2 - x1)² + (y2 - y1)²)
```

Area calculations use standard formulas:

- Triangle: Shoelace formula
- Polygon: Shoelace formula
- Circle: r^2

Intersection detection uses geometric algorithms:

- Line-line: Parametric form intersection
- Circle-circle: Distance-based intersection
- Point-in-polygon: Ray casting algorithm

14.4 Zero Dependencies Philosophy

planar_geometry intentionally has zero external dependencies:

- Uses only Python standard library
- Improves reliability and maintainability
- Reduces installation size
- Simplifies dependency management

This is achieved by:

- Using pure Python implementations
- Focusing on geometric fundamentals
- Avoiding matrix libraries (numpy)
- Not using plotting libraries (matplotlib)

14.5 Performance Considerations

For better performance:

- Avoid repeated distance calculations
- Cache expensive computations
- Use appropriate data structures
- Consider using Cython for critical paths (future)

14.6 Testing Strategy

- Unit tests for all public methods
- Edge case coverage
- Integration tests for complex operations
- 100% code coverage target

See *Contributing* for testing details.

Python Module Index

p