贝塞尔·三角形模块

贝塞尔三角形帮助文档

class bezier.triangle.**Triangle**(*nodes*, *degree*, ***, *copy=True*, *verify=True*)

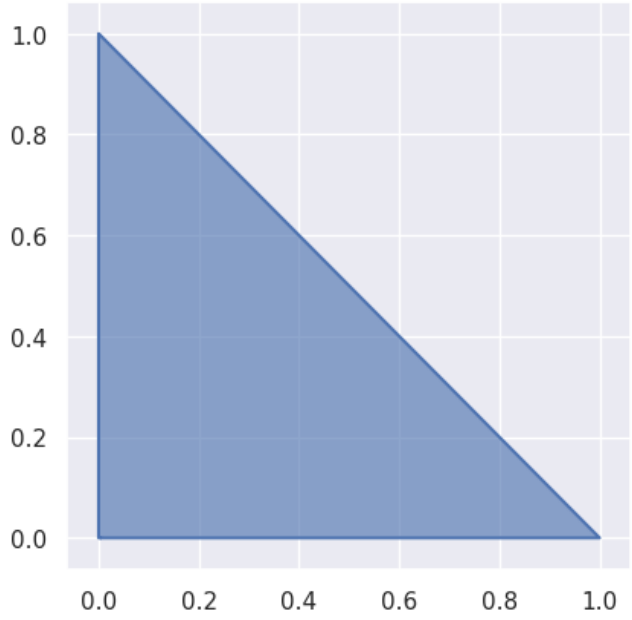
Bases: **Base**

代表一个贝塞尔三角形。

我们将贝塞尔三角形定义为从二维空间的unit simplex （即单位三角形）到任意维度三角形的映射。我们使用重心坐标

$$\lambda_1 = 1 - s - t,\ \lambda_2 = s,\ \lambda_3 = t$$

来表示单位三角形中的点 $\{(s, t) \mid 0 \le s, t, s + t \le 1\}$:



与曲线一样，利用这些权重，我们得到这些点$v_{i,j,k}$ 在向量空间中的凸组合:

$$B(\lambda_1, \lambda_2, \lambda_3) = \sum_{i+j+k=d} \binom{d}{i\,j\,k} \lambda_1^i \lambda_2^j \lambda_3^k \cdot v_{i,j,k}$$

备注:

我们假设nodes（节点）是从左到右、从下到上排序的。例如，线性三角形：

```
(0,0,1)

(1,0,0)  (0,1,0)
```

其排列为

$$\begin{bmatrix} v_{1,0,0} & v_{0,1,0} & v_{0,0,1} \end{bmatrix}$$

二次三角形：

```
(0,0,2)

(1,0,1)  (0,1,1)

(2,0,0)  (1,1,0)  (0,2,0)
```

其排列为

$$[v_{2,0,0} \quad v_{1,1,0} \quad v_{0,2,0} \quad v_{1,0,1} \quad v_{0,1,1} \quad v_{0,0,2}]$$

三次三角形：

```
    (0,0,3)

    (1,0,2)  (0,1,2)

    (2,0,1)  (1,1,1)  (0,2,1)

    (3,0,0)  (2,1,0)  (1,2,0)  (0,3,0)
```

其排列为

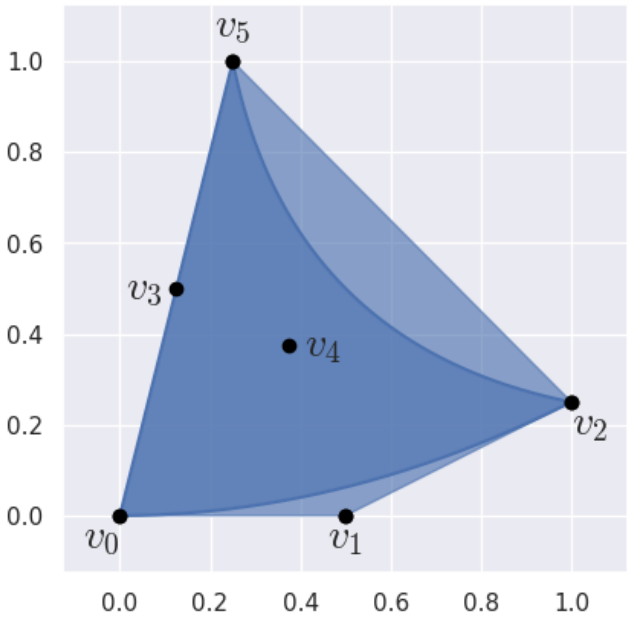
$$[v_{3,0,0} \quad v_{2,1,0} \quad v_{1,2,0} \quad v_{0,3,0} \quad v_{2,0,1} \quad v_{1,1,1} \quad v_{0,2,1} \quad v_{1,0,2} \quad v_{0,1,2} \quad v_{0,0,3}]$$

诸如此类。

指标公式

$$j + \frac{k}{2}(2(i + j) + k + 3)$$

可以用来将三元(i,j,k)映射到相应的线性索引上，但并没有带来特别深刻或实质性的见解或帮助。




```
>>> import bezier
>>> import numpy as np
>>> nodes = np.asfortranarray([
...     [0.0, 0.5, 1.0 , 0.125, 0.375, 0.25],
...     [0.0, 0.0, 0.25, 0.5  , 0.375, 1.0 ],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle
<Triangle (degree=2, dimension=2)>
```

参数说明：

- **nodes** (**SequenceSequencenumbers.Number**) – 三角形中的节点。这些节点必须能够被转换为一个二维 NumPy 浮点数数组，其中每一列代表每个节点，每一行代表环境空间的维度。
- **degree** (*int*) – 三角形的阶数。这个阶数假定与节点（nodes）数正确对应。如果阶数尚未计算，则使用 **from_nodes()** 函数。

- **copy** (*bool*) – 指示在存储之前是否应该复制节点的一个标志。由于调用者可能在传入后自由更改节点（nodes），因此默认为**True**。
- **verify** (*bool*) – 指示是否应该验证阶数与节点数是否相符的一个标志。默认为**True**。

classmethod **from_nodes**(*nodes*, *copy=True*)

从节点创造一个三角形（**Triangle**）。

基于给定节点（nodes）的形状计算阶数（degree）。

参数说明：

- **nodes** (**SequenceSequencenumbers.Number**) – 三角形中的节点。这些节点必须能够被转换为一个二维 NumPy 浮点数数组，其中每一列代表每个节点，每一行代表环境空间的维度。
- **copy** (*bool*) – 指示在存储之前是否应该复制节点的一个标志。由于调用者可能在传入后自由更改节点（nodes），因此默认为**True**。

返回值： 已构建的三角形。

返回类型： Triangle

property **area**

当前三角形的面积。

通过格林公式（Green’ s Theorem）计算二维空间中三角形的面积。对于一个向量场 $F = [-y, x]^T$，由于 $\partial_x(x) - \partial_y(-y) = 2$ 这暗示面积的两倍等于

$$\int_{B(\mathcal{U})} 2\,d\mathbf{x} = \int_{\partial B(\mathcal{U})} -y\,dx + x\,dy.$$

其依赖于一个前提条件，即当前三角形是有效的，这意味着在贝塞尔映射—B (U)—下单位三角形的图像的边缘正好对应于当前三角形的边界。

针对给定带有控制点x_j, y_j的边 C(r)，积分可以被简化：

$$\int_C -y\,dx + x\,dy = \int_0^1 (xy' - yx')\,dr = \sum_{i<j}(x_i y_j - y_i x_j)\int_0^1 b_{i,d} b'_{j,d}\,dr$$

其中$b_{i,d}$, $b_{j,d}$ 是Bernstein basis polynomials（伯恩斯坦基多项式）。

返回值： 当前三角形的面积。

返回类型： float

异常情况： **NotImplementedError** – 如果当前三角形不在二维空间里。

property **edges**

三角形边缘。

```
>>> nodes = np.asfortranarray([
...     [0.0,  0.5   , 1.0, 0.1875, 0.625, 0.0],
...     [0.0, -0.1875, 0.0, 0.5   , 0.625, 1.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> edge1, _, _ = triangle.edges
>>> edge1
<Curve (degree=2, dimension=2)>
>>> edge1.nodes
array([[ 0. ,  0.5   , 1. ],
       [ 0. , -0.1875, 0. ]])
```

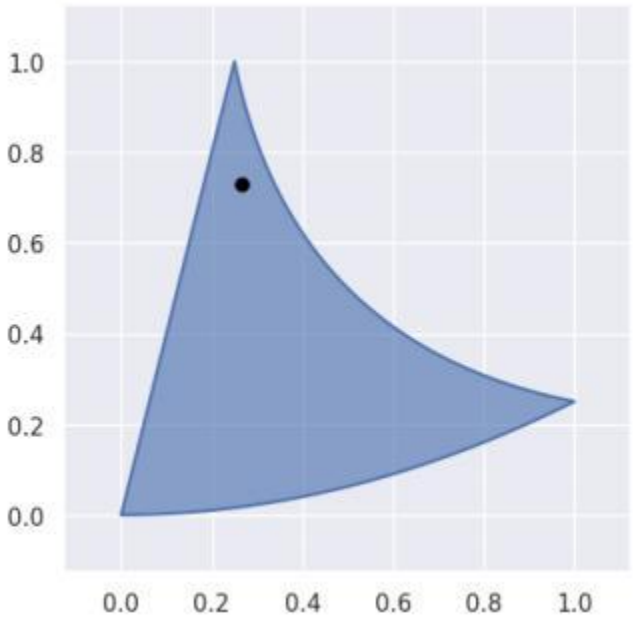
返回值： 三角形的边缘。

返回类型： **TupleCurve**, **Curve**, **Curve**

evaluate_barycentric(*lambda1*, *lambda2*, *lambda3*, *verify=True*)

计算三角形中的点。

求得 B $(\lambda_1, \lambda_2, \lambda_3)$




```
>>> nodes = np.asfortranarray([
...     [0.0, 0.5, 1.0 , 0.125, 0.375, 0.25],
...     [0.0, 0.0, 0.25, 0.5  , 0.375, 1.0 ],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> point = triangle.evaluate_barycentric(0.125, 0.125, 0.75)
>>> point
array([[0.265625  ],
       [0.73046875]])
```

然而，这不能用于位于参考三角外部的点：

```
>>> triangle.evaluate_barycentric(-0.25, 0.75, 0.5)
Traceback (most recent call last):
  ...
ValueError: ('Weights must be positive', -0.25, 0.75, 0.5)
```

或非重心坐标；

```
>>> triangle.evaluate_barycentric(0.25, 0.25, 0.25)
Traceback (most recent call last):
  ...
ValueError: ('Weights do not sum to 1', 0.25, 0.25, 0.25)
```

然而，如果`verify`是 **`False`**，那么这些“ 无效 ”的输入就可以被接受和使用。

```
>>> triangle.evaluate_barycentric(-0.25, 0.75, 0.5, verify=False)
array([[0.6875  ],
       [0.546875]])
>>> triangle.evaluate_barycentric(0.25, 0.25, 0.25, verify=False)
array([[0.203125],
       [0.1875  ]])
```

参数说明：

- **lambda1** (*float*) – 在参考三角形上的参数。
- **lambda2** (*float*) – 在参考三角形上的参数。
- **lambda3** (*float*) – 在参考三角形上的参数。
- **verify** (**`Optionalbool`**) – 表示是否应该验证重心坐标，确保它们加和总为一且均为非负数（即验证为重心坐标）。这个参数可以用来在定义域外部进行求值，或者在调用者已经知道输入已被验证的情况下节省时间。默认为 **`True`** 。

返回值： 三角形上的某个点（作为一个具有单列的二维 NumPy 数组）。

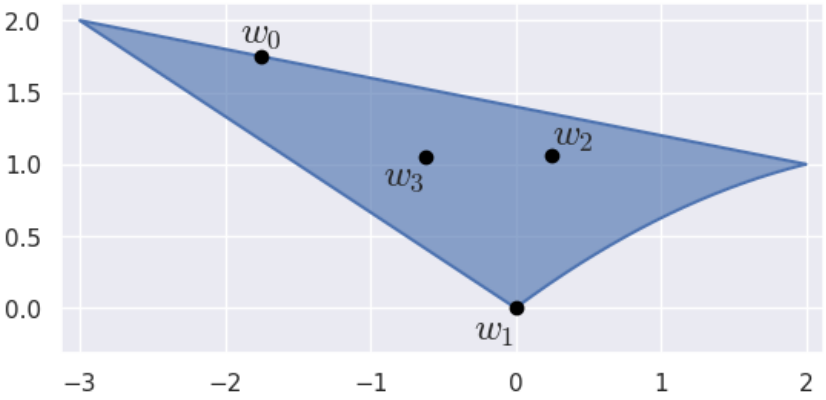
返回类型： numpy.ndarray

异常情况：
- **ValueError** – 如果权重不是有效的重心坐标，也就是它们的总和不等于1（如果 `verify=False` ，则不引发异常）。
- **ValueError** – 如果一些权重是负数（如果 `verify=False` ，则不引发异常）。

`evaluate_barycentric_multi(param_vals, verify=True)`

计算三角形上的多个点。

假设 `param_vals` 具有三列重心坐标。参见**evaluate_barycentric()**来获得每一行参数值计算的更多细节。




```
>>> nodes = np.asfortranarray([
...     [0.0, 1.0 , 2.0, -1.5, -0.5, -3.0],
...     [0.0, 0.75, 1.0,  1.0,  1.5,  2.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle
<Triangle (degree=2, dimension=2)>
>>> param_vals = np.asfortranarray([
...     [0.   , 0.25, 0.75 ],
...     [1.   , 0.  , 0.   ],
...     [0.25 , 0.5 , 0.25 ],
...     [0.375, 0.25, 0.375],
... ])
>>> points = triangle.evaluate_barycentric_multi(param_vals)
>>> points
array([[-1.75 , 0. , 0.25   , -0.625   ],
       [ 1.75 , 0. , 1.0625 ,  1.046875]])
```

参数说明：
- **param_vals** (*numpy.ndarray*) – 参数值数组（作为一个 `N x 3` 数组）。
- **verify** (**Optionalbool**) – 指示是否应该验证坐标。参见**evaluate_barycentric()**。默认为**True**。也会检查 `param_vals` 是否是正确的形状。

返回值： 三角形上的点。

返回类型： numpy.ndarray

异常情况： **ValueError** – 如果 `param_vals` 不是二维数组且 `verify=True`。

`evaluate_cartesian(s, t, verify=True)`

计算三角形上的点。

通过 **evaluate_barycentric()** 计算 B (1 − s − t, s, t)：

这个方法充当了**locate()**的（部分）逆操作。

```
>>> nodes = np.asfortranarray([
...     [0.0, 0.5, 1.0  , 0.0, 0.5, 0.25],
```

```
...      [0.0, 0.5, 0.625, 0.5, 0.5, 1.0 ],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> point = triangle.evaluate_cartesian(0.125, 0.375)
>>> point
array([[0.16015625],
       [0.44726562]])
>>> triangle.evaluate_barycentric(0.5, 0.125, 0.375)
array([[0.16015625],
       [0.44726562]])
```

参数说明：

- **s** (*float*) – 参考三角形上的参数。
- **t** (*float*) – 参考三角形上的参数。
- **verify** (**Optionalbool**) – 指示坐标是否应该在参考三角形内部进行验证。 默认为**True**。

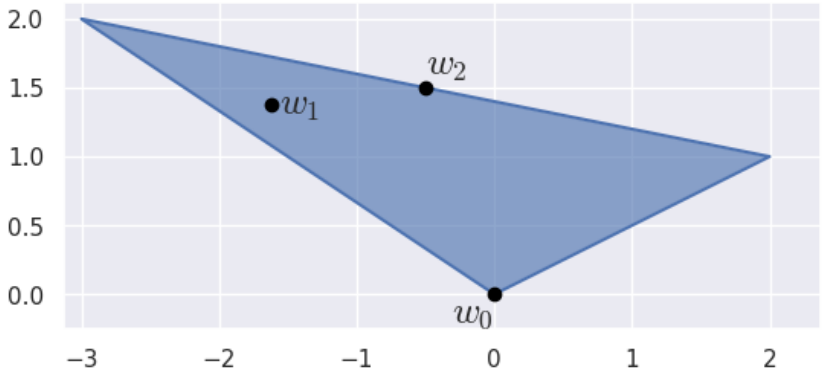
返回值： 三角形上的点（作为二维NumPy 数 组 ） 。

返回类型： numpy.ndarray

evaluate_cartesian_multi(*param_vals*, *verify=True*)

计算三角形上的多个点。

假设 `param_vals` 具有两列笛卡尔坐标。参见**evaluate_cartesian()** 来获得每一行参数值的更多细节。




```
>>> nodes = np.asfortranarray([
...      [0.0, 2.0, -3.0],
...      [0.0, 1.0,  2.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=1)
>>> triangle
<Triangle (degree=1, dimension=2)>
>>> param_vals = np.asfortranarray([
...      [0.0  , 0.0  ],
...      [0.125, 0.625],
...      [0.5  , 0.5  ],
... ])
>>> points = triangle.evaluate_cartesian_multi(param_vals)
>>> points
array([[ 0.  , -1.625, -0.5 ],
       [ 0.  ,  1.375,  1.5 ]])
```

参数说明：

- **param_vals** (*numpy.ndarray*) – 参数值数组（作为一个 `N x 2` 数组）。
- **verify** (**Optionalbool**) – 指示坐标是否应该进行验证。参见**evaluate_cartesian()** 。默认为**True**。还会再次检查 `param_vals` 是否具有正确的形状。

返回值： 三角形上的点。

返回类型： numpy.ndarray

异常情况： **ValueError** – 如果`param_vals` 不是二维数组且 `verify= True`。

`plot`(*pts_per_edge*, *color=None*, *ax=None*, *with_nodes=False*, *alpha=0.625*)

绘制当前的三角形。

参数说明：

- **pts_per_edge** (*int*) – 绘制每条边时点的数目。
- **color** (`Optional`[`Tuple`[`float`, `float`, `float`]]) – 由RGB确定的颜色。
- **ax** (`Optional`[`matplotlib.artist.Artist`]) – 用于添加绘图内容的 Matplotlib 坐标轴对象。
- **with_nodes** (`Optional`[`bool`]) – 确定是否将控制点添加到绘图中。默认关闭。
- **alpha** (`Optional`[`float`]) – 图形中心区域的透明度值，取值范围为 0 到 1（包含边界值）。

返回值： 包含绘图的坐标轴。可能是一个新创建的坐标轴。

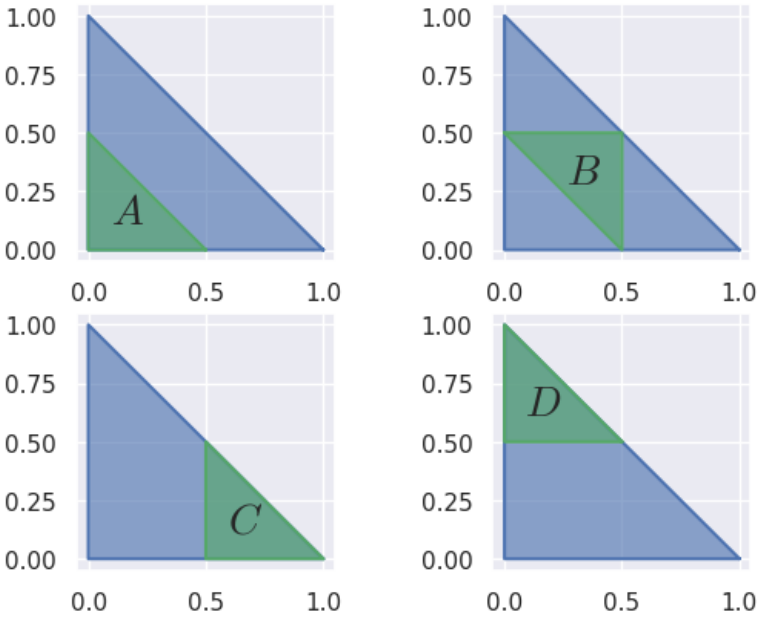
返回类型： matplotlib.artist.Artist

异常情况： **NotImplementedError** – 如果三角形的维度不是2。

`subdivide`()

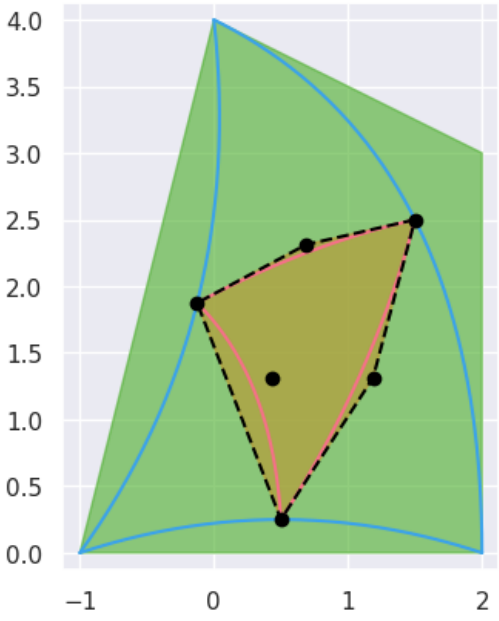
将三角形分割成四个子三角形。

这是通过将单位三角形（即三角形的范围）分割成四个子三角形来完成的。



然后，通过映射到或从给定的子三角形和单位三角形映射，重新参数化了三角形。

例如，当一个二次三角形被细分时：



```
>>> nodes = np.asfortranarray([
...     [-1.0, 0.5, 2.0, 0.25, 2.0, 0.0],
...     [ 0.0, 0.5, 0.0, 1.75, 3.0, 4.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> _, sub_triangle_b, _, _ = triangle.subdivide()
>>> sub_triangle_b
<Triangle (degree=2, dimension=2)>
>>> sub_triangle_b.nodes
array([[ 1.5 ,  0.6875, -0.125 , 1.1875, 0.4375, 0.5  ],
       [ 2.5 ,  2.3125,  1.875 , 1.3125, 1.3125, 0.25 ]])
```

返回值： 下左、中心、下右和上左的子三角形（按照这个顺序）。

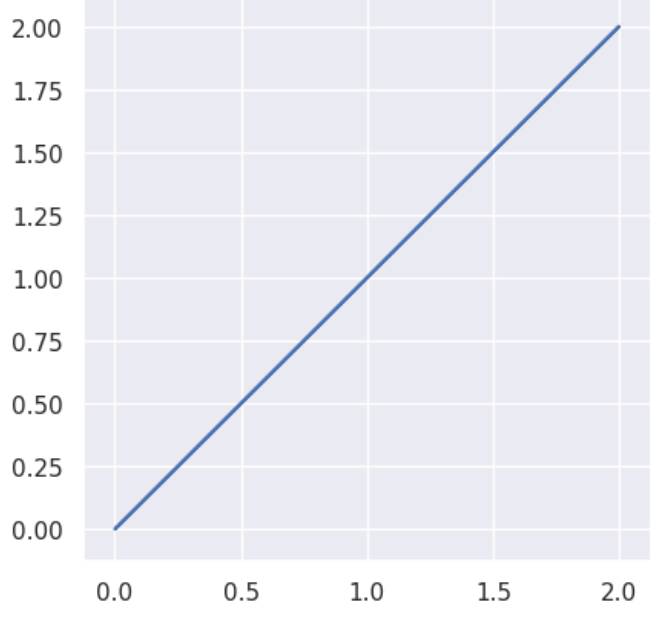
返回类型： **TupleTriangle**, **Triangle**, **Triangle**, **Triangle**

property **is_valid**

指示三角形是否“有效”。

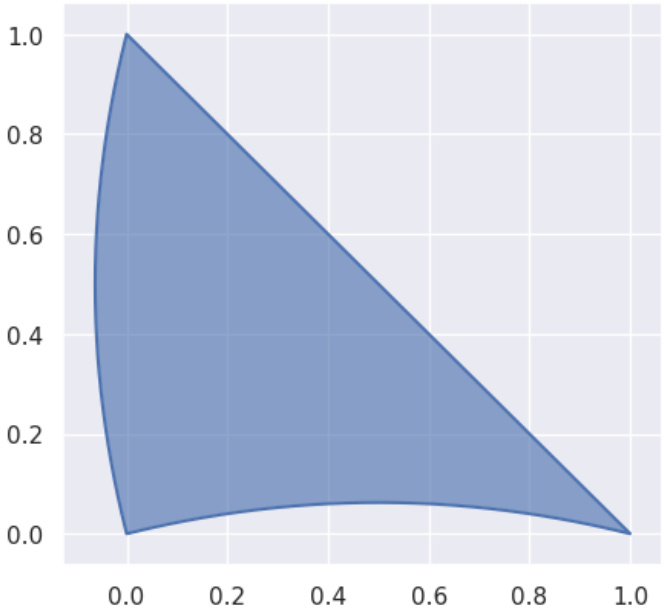
在这里，“有效”指的是没有自相交或奇异点，并且边缘的方向与内部一致（也就是说，向左旋转切向量90度指向内部）。

这个检查是针对从参考三角形到其他三角形的映射的雅可比（Jacobian）矩阵是否处处为正。例如，具有共线点的线性“三角形”是无效的：



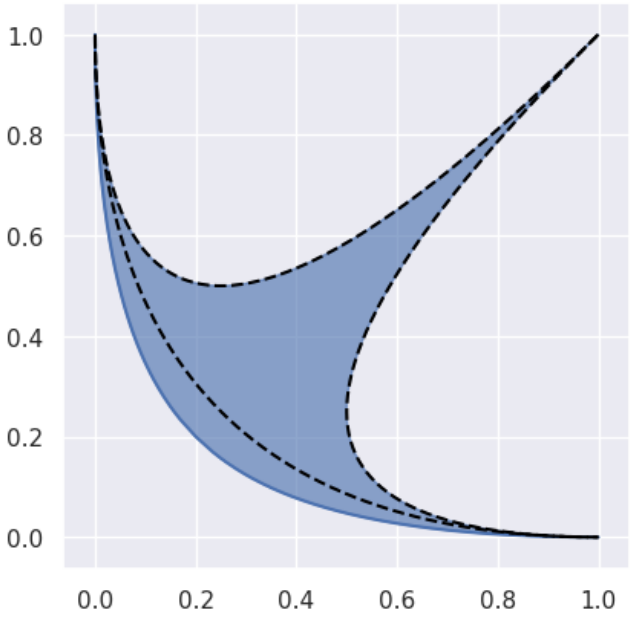
```
>>> nodes = np.asfortranarray([
...     [0.0, 1.0, 2.0],
...     [0.0, 1.0, 2.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=1)
>>> triangle.is_valid
False
```

和具有一条直边的二次三角形：




```
>>> nodes = np.asfortranarray([
...     [0.0, 0.5  , 1.0, -0.125, 0.5, 0.0],
...     [0.0, 0.125, 0.0,  0.5  , 0.5, 1.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle.is_valid
True
```

然而，并不是所有高阶三角形都是有效的：




```
>>> nodes = np.asfortranarray([
...     [1.0, 0.0, 1.0, 0.0, 0.0, 0.0],
...     [0.0, 0.0, 1.0, 0.0, 0.0, 1.0],
```

```
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle.is_valid
False
```

类型： bool

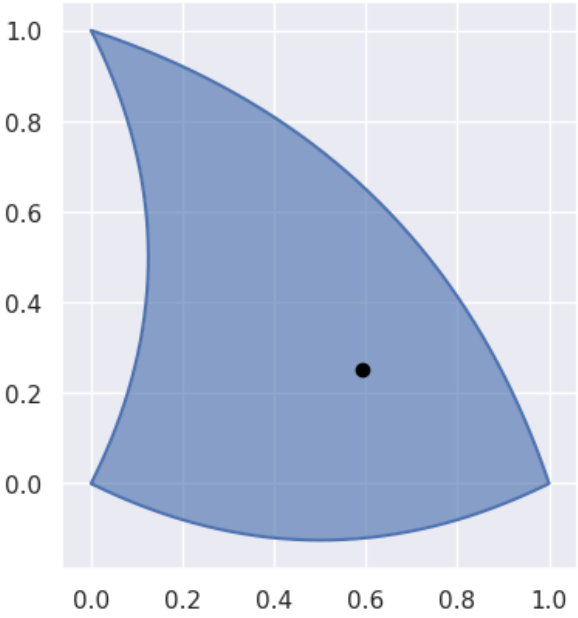
locate(*point*, *verify=True*)

在当前三角形寻找一个点。

解出方程 B(s, t) = p 中的 s 和 t。

这个方法充当了**evaluate_cartesian()**的（部分）逆操作。

> **警告：**
>
> 只有在当前三角形有效的情况下才能保证有唯一解。这段代码假设了一个有效的三角形，但并未进行检查。




```
>>> nodes = np.asfortranarray([
...     [0.0,  0.5 , 1.0, 0.25, 0.75, 0.0],
...     [0.0, -0.25, 0.0, 0.5 , 0.75, 1.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> point = np.asfortranarray([
...     [0.59375],
...     [0.25   ],
... ])
>>> s, t = triangle.locate(point)
>>> s
0.5
>>> t
0.25
```

参数说明：

- **point** (*numpy.ndarray*) – 一个 三角形上的`(D x 1)`点，其中 D 是三角形的维度。
- **verify** (**Optional**[**bool**]) – 指示是否应该对输入的假设使用额外的谨慎验证。可以禁用以加快执行速度。
  默认为 **True** 。

返回值： 与 `point` 对应的 **s** 和 **t** 值，如果不在三角形上则为 **None** 。

返回类型： **Optional**[**Tuple**[**float**, **float**]]

异常情况：

- **NotImplementedError** – 如果该三角形不在二维空间里。
- **ValueError** – 如果 `point` 的维度和当前三角形的维度不相符。

`intersect(other, strategy=IntersectionStrategy.GEOMETRIC, verify=True)`

寻找与另一个三角形的公共交点。

参数说明：

- **other** (*Triangle*) – 要进行交点计算的另一个三角形。
- **strategy** (**OptionalIntersectionStrategy**) – 要使用的交点计算算法。默认为几何算法。
- **verify** (**Optionalbool**) – 指示是否应该对算法进行进行额外的谨慎验证。可以禁用以加快执行速度。默认为**True**。

返回值： 交点列表（可能为空）。

返回类型： **ListUnionCurvedPolygon**, **Triangle**

异常情况：

- **TypeError** – 如果 `other` 不是一个三角形（且 `verify=True`）。
- **NotImplementedError** – 如果至少一个三角形不是二次的（并且`verify=True`）。
- **ValueError** – 如果`strategy` 不是有效的d **IntersectionStrategy**。

`elevate()`

返回当前三角形的升阶版本。

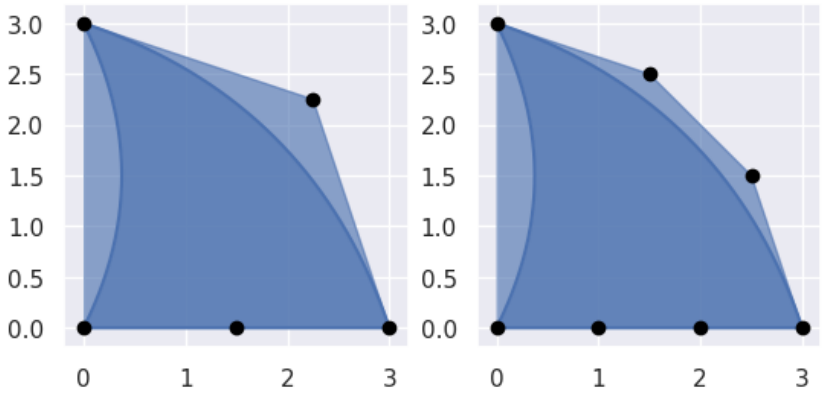
通过将当前节点$\{v_{i,j,k}\}_{i+j+k=d}$ 转换为新节点$\{w_{i,j,k}\}_{i+j+k=d+1}$来完成此操作。通过重新书写

$$E\left(\lambda_1, \lambda_2, \lambda_3\right) = \left(\lambda_1 + \lambda_2 + \lambda_3\right) B\left(\lambda_1, \lambda_2, \lambda_3\right) = \sum_{i+j+k=d+1} \binom{d+1}{i\,j\,k} \lambda_1^i \lambda_2^j \lambda_3^k \cdot w_{i,j,k}$$

以这种形式，必须有

$$\begin{aligned} &\binom{d+1}{i\,j\,k} \cdot w_{i,j,k} = \binom{d}{i-1\,j\,k} \cdot v_{i-1,j,k} + \binom{d}{i\,j-1\,k} \cdot v_{i,j-1,k} + \binom{d}{i\,j\,k-1} \cdot v_{i,j,k-1} \\ \iff &(d+1) \cdot w_{i,j,k} = i \cdot v_{i-1,j,k} + j \cdot v_{i,j-1,k} + k \cdot v_{i,j,k-1} \end{aligned}$$

我们在其中定义，比如 $v_{i,j,k-1} = 0$ if $k = 0$。




```
>>> nodes = np.asfortranarray([
...     [0.0, 1.5, 3.0, 0.75, 2.25, 0.0],
...     [0.0, 0.0, 0.0, 1.5 , 2.25, 3.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> elevated = triangle.elevate()
>>> elevated
<Triangle (degree=3, dimension=2)>
>>> elevated.nodes
 array([[0. , 1. , 2. , 3. , 0.5 , 1.5 , 2.5 , 0.5 , 1.5 , 0. ],
        [0. , 0. , 0. , 0. , 1.  , 1.25, 1.5 , 2.  , 2.5 , 3. ]])
```

返回值： 升阶三角形。
返回类型： Triangle

to_symbolic()

转换为一个表示B(s, t)的 SymPy 矩阵。

> **备注：**
>
> 这个方法需要SymPy。

```
>>> nodes = np.asfortranarray([
...     [0.0, 0.5, 1.0, -0.5, 0.0, -1.0],
...     [0.0, 0.0, 1.0,  0.0, 0.0,  0.0],
...     [0.0, 0.0, 0.0,  0.0, 0.0,  1.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle.to_symbolic()
Matrix([
[s - t],
[ s**2],
[ t**2]])
```

返回值： 三角形B(s, t)
返回类型： sympy.Matrix

implicitize()

将三角形转化为隐式形式。

> **备注：**
>
> 这个方法需要SymPy。

```
>>> nodes = np.asfortranarray([
...     [0.0, 0.5, 1.0, -0.5, 0.0, -1.0],
...     [0.0, 0.0, 1.0,  0.0, 0.0,  0.0],
...     [0.0, 0.0, 0.0,  0.0, 0.0,  1.0],
... ])
>>> triangle = bezier.Triangle(nodes, degree=2)
>>> triangle.implicitize()
(x**4 - 2*x**2*y - 2*x**2*z + y**2 - 2*y*z + z**2)**2
```

返回值： 通过f(x, y, z) = 0定义三维空间中三角形的函数。
返回类型： sympy.Expr
异常情况： **ValueError** – 如果三角形的维度不是 3。

property degree

当前形状的阶数。

类型： int

property dimension

形状所在的维度。

比如，若是三维空间中的，那么维度即为 3。

类型： int

property nodes

定义当前形状的节点。

类型： numpy.ndarray