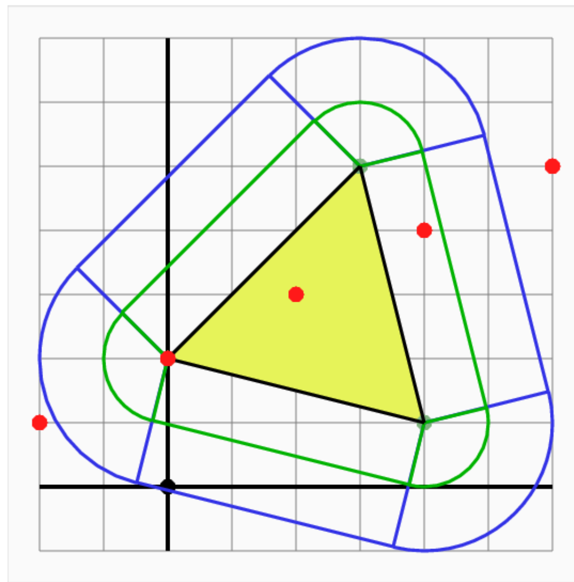
Geometry: Triangles
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/geometry2/geometry2.pdf

---



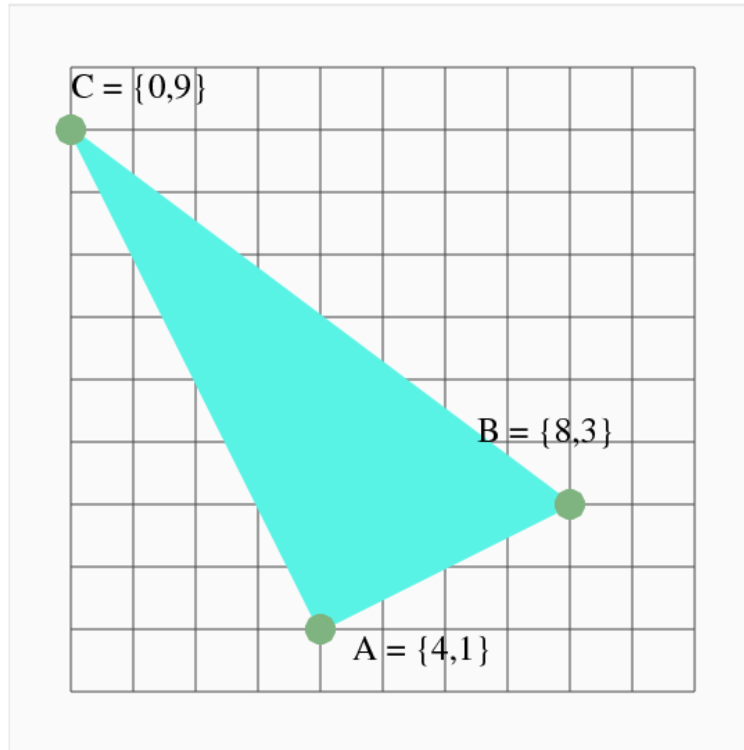
We can compute the distance from a point to a triangle.

> "Computational Geometry in 2D"
>
> - *In 2D, the basic object is the triangle;*
> - *Algorithms can be found for area, containment, distance, and so on;*
> - *Triangles can be combined to form polygons;*
> - *Polygonal properties can be derived from the elemental triangles.*
> - *Given data on a rectangular mesh, graphics packages interpolate a function over triangles;*
> - *A function can be integrated over a polygon by working with elemental triangles.*
> - *Partial differential equations (PDEs) may be posed over a 2D region;*
> - *To approximate the PDE solution, If the region is filled with sample points, organized into triangles, then a system of equations can be generated and solved..*

1 Representing triangles

To represent a triangle in a 2D region, we need the (x,y) coordinates of its three vertices. By mathematical convention, we also prefer that the vertices are listed in counterclockwise order. Consider the triangle #1, with vertices A, B, C, we might define a corresponding Python array T:

```
t = np.array ( [ [ 4, 1 ], [ 8, 3 ], [ 0, 9 ] ] )
```



Example triangle #1.

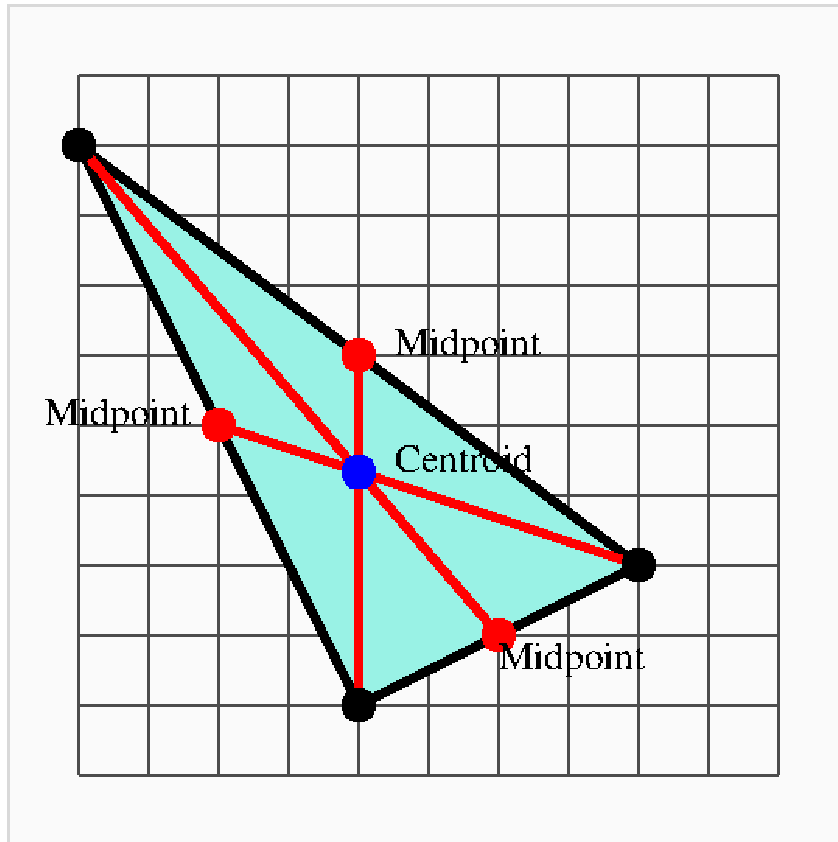
2 Triangle centroid

The **centroid** or, loosely speaking, the "center of mass", of a triangle, is the unique point **CM** with the property that *any* line through **CM** divides the triangle into two pieces of equal area.

You can locate the centroid in a drawing of a triangle very easily: Connect each edge midpoint to the opposite vertex. All three lines intersect at the centroid.

```
def triangle_centroid ( t ):
  import numpy as np
  centroid = np.mean ( t, axis = 0 )
  return centroid
```

For our example, `centroid = [ 4.0, 4.333 ]`.



Locating the centroid of example triangle #1.

3 Triangle side lengths

Another important property is the length of the sides. Typically, the side opposite A, B, C are labeled BC, CA, AB. If we compute all the side lengths in an array s, our first entry, $s[0]$, involves the side described by the vector `t[2] - t[1]`, and so on.

```
s = np.zeros ( 3 )
for i in range ( 0, 3 ):
  ip1 = ( i + 1 ) % 3
  ip2 = ( i + 2 ) % 3
  s[i] = np.linalg.norm ( t[ip2,:] - t[ip1,:] )
```

For our example, `s = [ 10, 8.94, 4.47 ]`, so we see in particular that side c is the shortest, as the diagram suggests.

4 Triangle angles

Another important property is the measurement of the angles. By tradition, the angles at vertices A, B, C are denoted by α, β, γ respectively. While software will work in radians, we can convert to degrees by multiplying by $\frac{180}{\pi}$.

The law of cosines for angle α says:

$$BC^2 = AB^2 + AC^2 - 2 * AB * AC * \cos(\alpha)$$

Using the inverse cosine function, we can derive formulas for all three angles:

$$\alpha = \cos^{-1}\left(\frac{AB^2 + AC^2 - BC^2}{2 * AB * AC}\right)$$
$$\beta = \cos^{-1}\left(\frac{AB^2 + BC^2 - AC^2}{2 * AB * BC}\right)$$
$$\gamma = \cos^{-1}\left(\frac{AC^2 + BC^2 - AB^2}{2 * AC * BC}\right)$$

Assuming that the array `s` has already been computed to hold the side lengths, we can produce the angles as follows:

```
angle = np.zeros ( 3 )
for i in range ( 0, 3 ):
  ip1 = ( i + 1 ) % 3
  ip2 = ( i + 2 ) % 3
  top = s[i]**2 + s[ip2]**2 - s[ip1]**2
  bot = 2.0 * s[i] * s[ip2]
  angle[i] = np.arccos ( top / bot )
```

5 Triangle area

For the triangle area, we will rely on Heron's formula, where $S = \frac{1}{2}(AB + BC + CA)$:

$$\text{Area} = \sqrt{S\,(S - AB)\,(S - BC)\,(S - CA)}$$

which we can program as

```
abc = 0.5 * np.sum ( s )
area = np.sqrt ( abc * ( abc - s[0] ) * ( abc - s[1] ) * ( abc - s[2] ) )
```

Because example triangle #1 is a right triangle, with sides of length $\sqrt{20}$ and $\sqrt{80}$, we can actually predict in advance that the area is $1/2\sqrt{20 * 80} = 20$.

6 Triangle signed area

There is a second formula, for the signed area of a triangle, which we will find useful.

$$\begin{aligned}\text{Signed area} = (1/2)\,(&\\ &+ xa\,(yb - yc)\\ &- xb\,(yc - ya)\\ &+ xc\,(ya - yb))\end{aligned}$$

The signed area can also be computed using the vector cross product. For our calculation

$$\text{Signed area} = (1/2)(B - A) \times (C - A)$$

which can become a function

```
def triangle_signed_area ( t ):
  import numpy as np
  value = np.cross ( t[:,1] - t[:,0], t[:,2] - t[:,0] )
  value = 0.5 * float ( value )
  return value
```

7 Triangle orientation

If the vertices of a triangle are given in counterclockwise order, then the signed area is the same as Heron's area. But if the vertices are given in clockwise order, the signed area is the negative of Heron's area. Oddly enough, this is a very useful fact. We can ensure counterclockwise order if we are in charge, but if we get the vertex coordinates of a triangle from some other source, we need to verify that they are in the proper order, and if not, reverse them.

In fact, this can be so useful that we can turn this idea into a function:

```
def triangle orientation ( t ):
  sa = triangle_signed_area ( t )
  if ( sa < 0.0 )
    value = -1
  else
    value = +1
  return value
```

8 Triangle contains point

Now we are ready to ask whether we can determine whether a point p is inside a triangle with vertices t. This is one of those questions that would be trivial to answer from a picture; however, it's really not obvious how to go about training a computer program to answer it correctly. We need to find a way of answering this question that is an algorithm, that is, *computational* and *automatic*.

We will suppose our triangle vertices are listed in counterclockwise order. Imagine then, walking from vertex **A** to **B**, then to **C** and on back to **A**. What can we say about our left hand?

It's always pointing into the triangle. Moreover, any point p inside the triangle will always be to our left. And any point in the triangle will be "to the left of" the edges **AB**, **BC**, and **CA**.

And if a point p is *not* inside the triangle, what can we say? It might be to the left of one or two of the sides, but it will always be to the right of at least one side. Now we can characterize the difference between points inside and outside the triangle:

Theorem 1 Triangle Contains Point: *A point* **p** *is inside a triangle* **t** *if, and only if, it is "to the left" of all three sides of* **t**.

Recall our function for the location of a point relative to a line.

```
def line_side ( p0, p1, q ):
  import numpy as np
  v0 = p1 - p0
  v1 = q - p0
  v0xv1 = float ( np.cross ( v0, v1 ) )
  return v0xv1
```

For a given line from `p0` to `p1` and a test point `q`, the value returned by **line_side(p0,p1,q)** is:

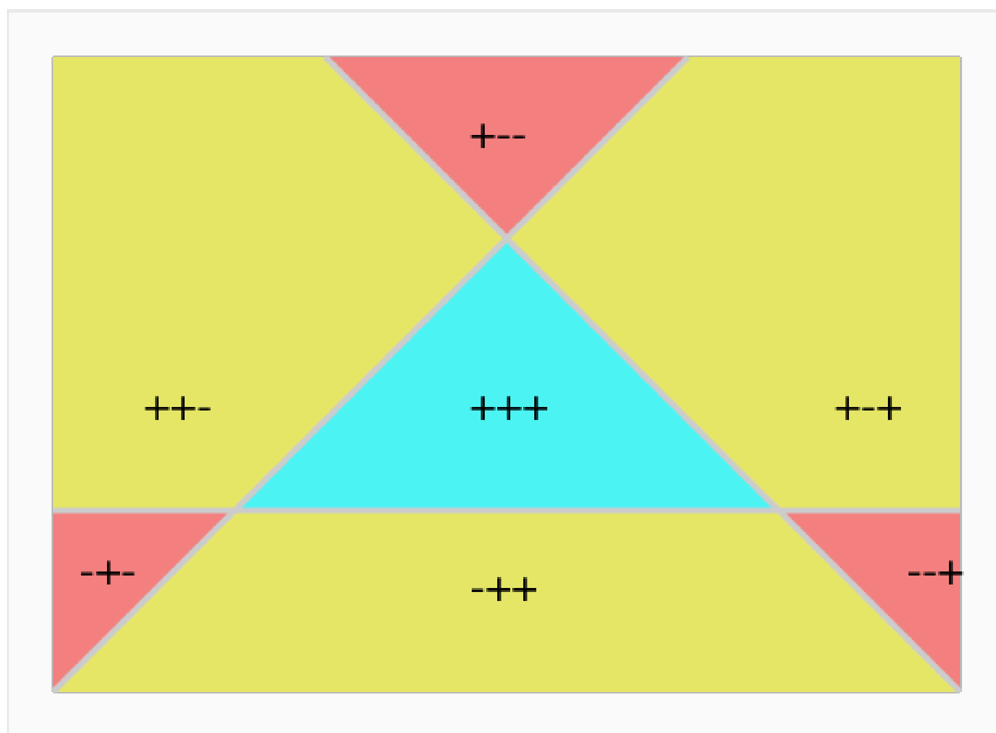
- +1 if **q** is to the left of the line;
- 0 if **q** is on the line;
- -1 if **q** is to the right of the line.

Suppose we call this function three times, to check the status of **q** with respect to the lines through the segments **p0–p1**, **p1–p2** and **p2–p0**?

If we test `q` against all three lines, with these results, where is `q`? (Note that there is something a little odd (or even impossible) about one result.)

```
     p0--p1  p1--p2  p2--p0  Where is q?

 q1    +1      +1      +1
 q2    -1      +1      +1
 q3    +1      -1      +1
 q4    +1      +1      -1
 q5    +1      -1      -1
 q6    -1      +1      -1
 q7    -1      +1      +1
 q8    -1      -1      -1
```

Technically, zero values are also very interesting, but unlikely computationally.

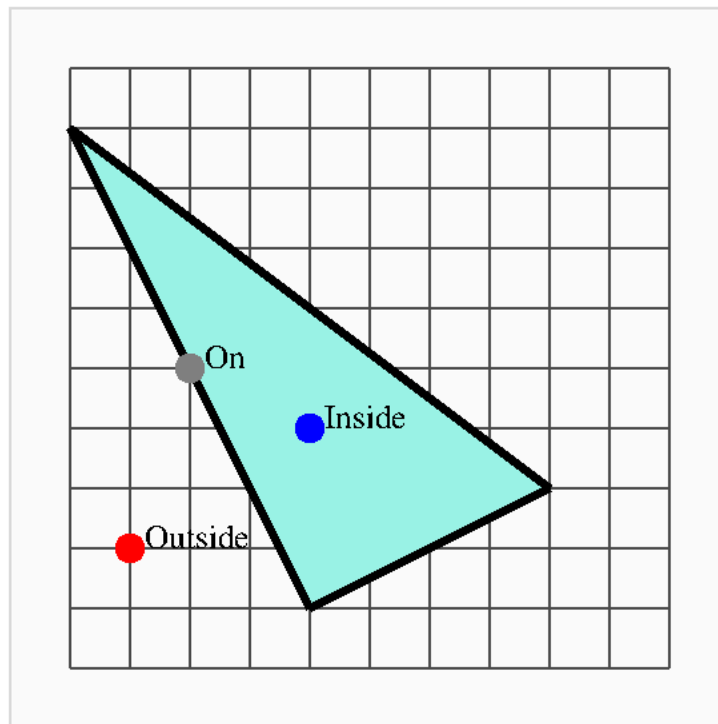


And now we know enough to be able to determine if a point is contained inside a triangle! If the point is to the right of any one of the three sides, it is not in the triangle.

```
def triangle_contains_point ( t, q ):
```

```
  value = True
  for i in range ( 0, 3 ):
    ip1 = ( i + 1 ) % 3
    if ( line_side( t[i,:], t[ip1,:], q ) < 0.0 ):
      value = False
      break

  return value
```

Let's test our method using our example triangle and the points $q1 = (1, 2), q2 = (2, 5), q3 = (4, 4)$.




```
 X    Y     Contains?

1.0  2.0    No
2.0  5.0    Yes
4.0  4.0    Yes
```

9 Triangle coordinate system

Every point p in a triangle can be represented as a linear combination of the vertices A, B, C, of the form

$$p = \alpha * A + \beta * B + \gamma * C$$

where the coefficients α, β, γ are nonnegative, and $\alpha + \beta + \gamma = 1$.

Here are the coordinates of a few special triangle points:

```
alpha beta gamma
```

A	1	0	0
B	0	1	0
C	0	0	1
AB Mid	1/2	1/2	0
BC Mid	0	1/2	1/2
CA Mid	1/2	0	1/2
Centroid	1/3	1/3	1/3

Given any point $p = (x, y)$ in the plane, and a triangle ABC, there is a procedure for computing α, β, γ, and of course, knowing these coefficients, we can easily compute p. Moreover, a point p is inside the triangle if and only if all the coefficients are positive.

This useful system is known as the set of *barycentric* coordinates, and in fact corresponds to the fact that every point in the triangle must be a convex combination of the vertices.

We will see this coordinate system in action for random sampling, and for defining quadrature rules.

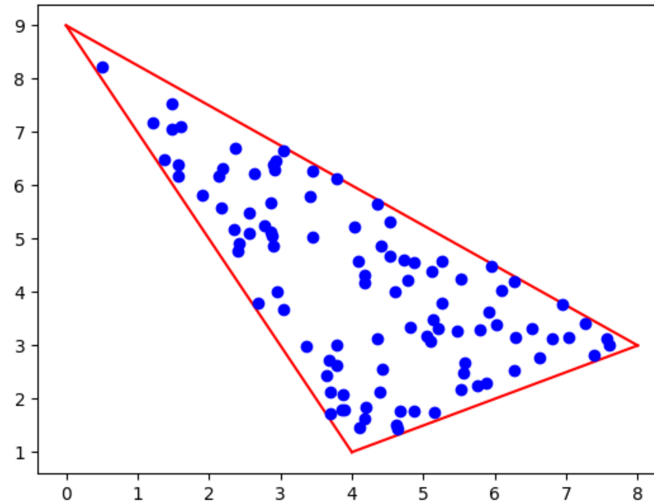
```
def triangle_barycentric ( t, p ):

  import numpy as np

  A = np.zeros ( [ 2, 2 ] )
  b = np.zeros ( 2 )

  A[0,0] = t[0,0] - t[2,0]
  A[0,1] = t[1,0] - t[2,0]
  b[0] =      p[0] - t[2,0]

  A[1,0] = t[0,1] - t[2,1]
  A[1,1] = t[1,1] - t[2,1]
  b[1] =      p[1] - t[2,1]

  sol = np.linalg.solve ( A, b );
  xsi = np.array ( [ sol[0], sol[1], 1.0 - sol[0] - sol[1] ] )

  return xsi
```

10 Triangle sampling

To pick a point p inside an arbitrary triangle uniformly at random, we compute the barycentric coefficients in the following way:

```
def triangle_sample ( t ):
  import numpy as np
  r1 = np.random.rand ( )
  r2 = np.random.rand ( )
  alpha = 1 - np.sqrt ( r1 )
  beta = np.sqrt( r1 ) * r2
  gamma = np.sqrt ( r1 ) * ( 1 - r2 )
  p = alpha * t[0,:] + beta * t[1,:] + gamma * t[2,:]
  return p
```

Requesting 100 sample points this way produces the following results:



11 Triangle Monte Carlo method

If we can uniformly sample n points (x_i, y_i) in a triangle T, then we can use the Monte Carlo method to estimate integrals, as follows:

$$I(f) = \int_T f(x,y)\,dx\,dy \approx Q(f) = \frac{\text{Area(T)}}{n} \sum_{i=0}^{i<n} f(x_i, y_i)$$

We can formalize this approach as a Python function:

```
def triangle_monte_carlo ( t, f, n ):

  A = triangle_area ( t )

  Q = 0.0
  for i in range ( 0, n ):
    p = triangle_sample ( t )
    Q = Q + f ( p[0], p[1] )
  Q = ( A / n ) * Q

  return Q
```

We could use this approach to estimate the integral of $f(x,y) = x^2 + y^2$ over our triangle. Since we don't know the answer, and we don't know an appropriate value of n, we can just experiment by increasing n and hoping to see the results settle down.

```
    10  856.01
   100  808.75
  1000  794.77
 10000  807.04
100000  808.06
```

From these results, we might guess that $I(f) \approx 808$ but the results really haven't settled down enough to be reasonably confident.

12 Triangle quadrature rules

There are families of quadrature rules for triangles, which specify a set of n points (x_i, y_i) and weights w_i, and guarantee a precise answer for all polynomial integrands up to a particular degree d.

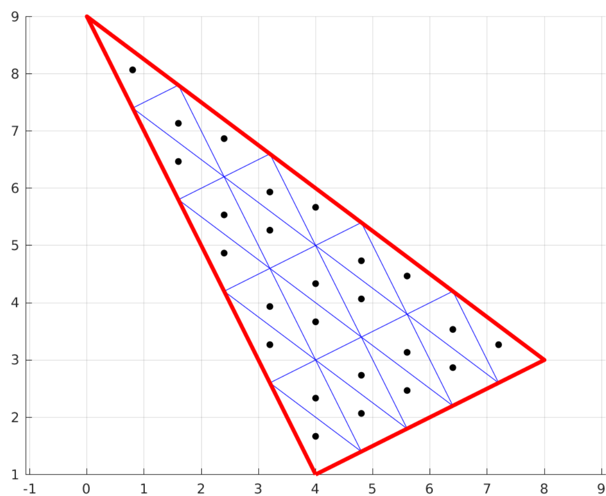
A rule which is precise for all polynomials of degree $0 \leq d \leq 1$ is the 1-point centroid rule. Here, we describe the

```
   x                        y              w

[ 1/3  1/3  1/3]  [ 1/3  1/3  1/3 ]  [ 1.0 ]
```

If we use this rule on our integral, we get the pretty crude estimate of 695.55. If we don't know the exact answer, then we need some other way to get some better estimates. What are our options?

13 Triangle refinement

One way to improve the estimate produced by a quadrature rule is to split the triangle up into smaller triangles, and apply the rule over each subtriangle. Even the centroid rule rule can be improved this way. Here is an illustration of the decomposition of our triangle:



After step 5, refinement into 25 subtriangles.

and here are the integral estimates using n subtriangles

```
   C     N   Estimate

   1     1   695.556
   2     4   778.889
   3     9   794.321
   4    16   799.722
   5    25   802.222
   6    36   803.58
   7    49   804.399
```

```
 8   64  804.931
 9   81  805.295
10  100  805.556
```

So, it looks like, with just 100 points, the centroid rule is already converging in a regular fashion, whereas our Monte Carlo rule was still producing significantly varying estimates.

14 Triangle rule of higher degree

Another way to improve an estimate for the integral of a function over a triangle rule is simply to find a stronger quadrature rule. We happen to know of a six point quadrature rule for triangles, which is exact for integrands through degree 4.

To define the rule, we need to set up some parameter values first:

```
a = 0.816847572980459;  b = 0.091576213509771;  u = 0.109951743655322;
c = 0.108103018168070;  d = 0.445948490915965;  v = 0.223381589678011;
```

Then a, b, u define the first three points and weights, and c, d, v the remaining three, as follows:

```
     x            y        w
barycentric  barycentric
[ a, b, b ]  [ b, a, b ]  u
[ b, a, b ]  [ a, b, b ]  u
[ b, b, a ]  [ b, b, a ]  u
[ c, d, d ]  [ d, c, d ]  v
[ d, c, d ]  [ c, d, d ]  v
[ d, d, c ]  [ d, d, c ]  v
```

To be clear, the (x, y) coordinates of the first quadrature point are computed using $[a, b, b]$ and $[b, a, b]$ as barycentric coordinates:

```
x[0] = a*4 + b*8 + b*0 = 4.00...
y[0] = b*1 + a*3 + b*9 = 3.36...
```

and their contribution to the integral estimate is $u * f(x[0], y[0])$. Using these six quadrature points, the rule gets an integral estimate of 806.66... which is the exact answer, since the integrand degree is 2.

15 Triangles as building blocks

The reason for working so hard on triangles is that we will now be able to handle almost any other shape that is polygonal, or that can be approximated by polygons. This is so because, as we shall see, we can take polygons or general sets of points, and organize them into triangles.

Using simple methods, we can then compute area, angles, centroids, integral estimates and so on for these polygonal structures. We will next suggest how this triangularization process can be done in a computational fashion.