Geometry: Triangles and Polygons
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/math1800_2023/geometry3/geometry3.pdf

There is an endless diversity of polygonal shapes.

”Triangles and Polygons”

- *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 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.

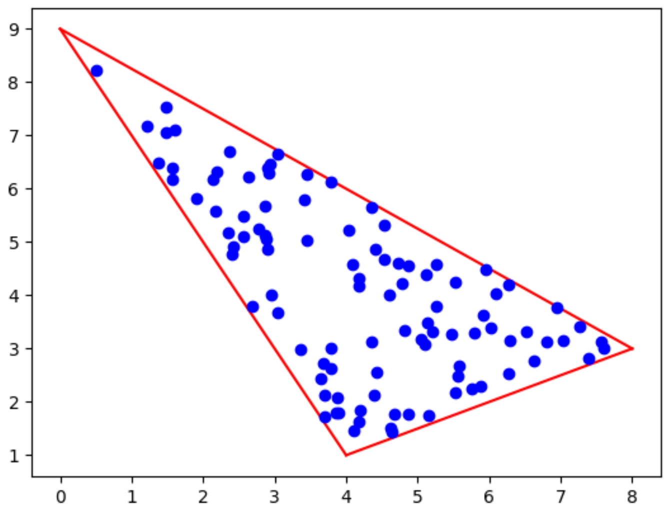
```
def triangle_barycentric ( t, p ):

  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
```

2 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:



3 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.

4 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.

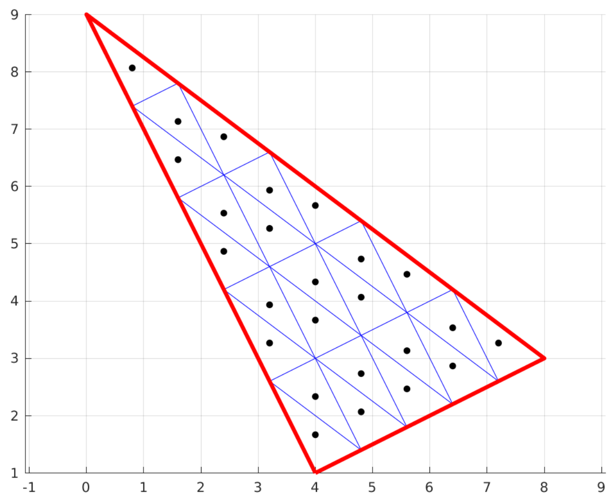
A rule which is precise for all polynomials of degree $0 \le d \le 1$ is the 1-point centroid rule. Here, we describe the

```
   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?

5 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.

6 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.

7 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.

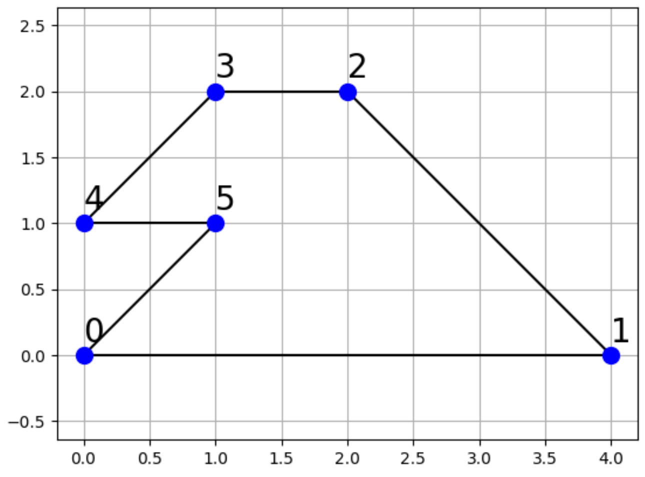
8 Representing a Polygon

A polygon in 2D can easily be represented by extending the way we represented a triangle. So we would create an array v which lists the (x, y) coordinates of the vertices in counter clockwise order.

Let us take for our example the following polygon:

```
v = np.array ( [ [0,0], [4,0], [2,2], [1,2], [0,1], [1,1] ] )
```

Here is an image of the polygon:



Outline of example polygon.

9 Representing a triangulated polygon

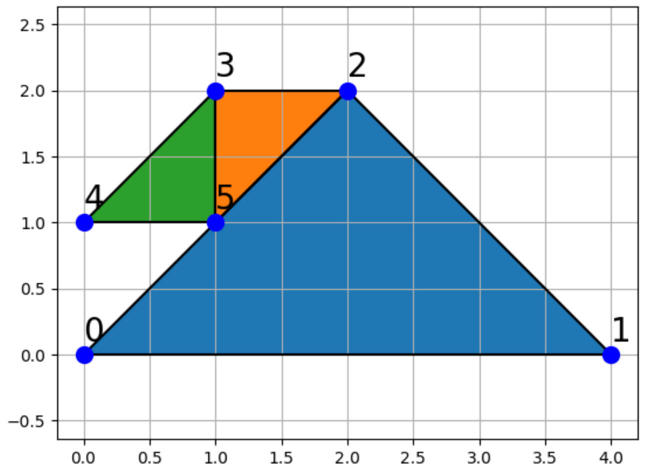
If we wish to apply our triangle knowledge to the study of polygons, then we will need to see a polygon as a collection of triangles. In that case, we need to decompose the polygon into triangles. We can do this by creating an array `tri` which describes each triangle in terms of the polygon vertices. For our example polygon, we could write

```
tri = np.array ( [ [ 0, 1, 2 ], [2, 3, 5], [3,4,5] ] )
```

Think of each entry of `tri` as `ti`, a list of indices in the vertex array, that form a triangle. For any list `ti`, there is a corresponding triangle `t=v[ti]`. Using the information from the `v` and `tri` arrays, we could plot our polygon as follows:

```
for ti in tri:
  x = [ v[t[0],0], v[ti[1],0], v[ti[2],0] ]
  y = [ v[t[0],1], v[ti[1],1], v[ti[2],1] ]
  plt.fill ( x, y )

for ti in tri:
  for i in range ( 0, 3 ):
    ip1 = ( i + 1 ) % 3
    x = [ v[ti[i],0], v[ti[ip1],0] ]
    y = [ v[ti[i],1], v[ti[ip1],1] ]
    plt.plot ( x, y, 'k-' )

i = 0
for xy in v:
  plt.plot ( xy[0], xy[1], 'b.', markersize = 20 )
  plt.text ( xy[0], xy[1] + 0.10, str ( i ), fontsize = 20 )
  i = i + 1
```

with the result:



Outline of example polygon.

10 Two ways of computing the area of a polygon

Earlier in one of our homework assignments, we saw the “shoelace” algorithm for computing the area of a polygon.

```
def polygon_area_shoelace ( v ):

  import numpy as np
  n = v.shape[0]
  np.append ( v, v[0] )
  area = 0.5 * ( np.sum ( v[0:n-1,0] * v[1:n,1] ) \
               - np.sum ( v[1:n,0]   * v[0:n-1,1] ) )
  return area
```

If we have triangulated our polygon, we can compute its area as the sum of the areas of the triangles:

```
  polygon_area = 0
  for ti in tri:
    t = np.array ( v[ti] )
    polygon_area = polygon_area + triangle_area ( t )
```

11 Centroid of a polygon

There is an interesting property of centroids. If a shape S of area A can be divided into smaller shapes s_i with areas a_i, then the centroid C of the big shape is the average of the area a_i times the centroid c_i of each of the little shapes:

$$C = \frac{\sum a_i c_i}{A}$$

The only reason this helps us is that if we have divided our big shape into triangles, then we know how to compute the areas and centroids of the smaller shapes. Thus, for a triangulated polygon, we have

```
def polygon_centroid ( v, tri ):

  import numpy as np
  t_num = tri.shape[0]
  centroid = np.zeros ( 2 )
  area = 0.0
  for ti in tri:
    t = np.array ( v[ti] )
    t_area = triangle_area ( t )
    t_centroid = triangle_centroid ( t )
    area = area + t_area
    centroid = centroid + t_area * t_centroid
  centroid = centroid / area
  return centroid
```

12 Does a polygon contain a point?

Because a polygon is not guaranteed to be convex, our trick that worked for triangles may not work for general polygons. For nonconvex polygons, while we walk around the boundary, a point that is actually inside the polygon may appear sometimes on our left and sometimes our right. The way to answer this question is to work with a polygon that has been subdivided into triangles. If the point is in the polygon, it must be in one of the triangles, and so we simply have to start assuming the answer is "No", check every triangle, and change our answer to "Yes" if we spot the point.

```
def polygon_contains_point ( v, tri, q ):

  import numpy as np
  t_num = tri.shape[0]
  value = False
  for ti in tri:
    t = np.array ( v[ti] )
    if ( triangle_contains_point ( t, q ) ):
      value = True
      break
  return value
```

13 Integral over a polygon

If we are interested in estimating the integral of a function over a polygon, and we have divided the polygon into triangles, then we can simply compute the integral estimate over each triangle, and sum them.

For example, we can estimate integrals using the very low accuracy 1 point triangle rule as follows:

```
def polygon_quad1 ( v, tri, f ):
  import numpy as np
  Q = 0.0
  for ti in tri:
    t = np.array ( v[ti] )
    Q = Q + triangle_quad1 ( t, f )
  return Q
```

And since we have also defined a six point rule called `triangle_quad6()`, we can easily improve our estimate for the polygon integral by using that function instead.

14 Sampling a polygon

To uniformly sample points from a polygon takes a little extra care. We will assume the polygon has been triangulated. Now we know how to sample a point from any of these triangles, but how should we pick which triangle to sample? The answer is that the choice must be made at random, but with a probability that is proportional to the area of each triangle. Unless all the triangles have the same area, this requires some extra work; we need to make a sort of ruler that runs from 0 to 1, marked off at certain points that indicate which triangle to sample. We will save the details of this process to the next discussion.

Here is a partial code that suggests how the sampling would work:

```
def polygon_sample ( v, tri ):
  import numpy as np
  tri_num = tri.shape[0]
  Choose triangle index 0 <= i < tri_num in the right way
  t = np.array ( v[ti[i]] )
  p = triangle_sample ( t )
  return p
```