Triangles
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/triangles/triangles.pdf

---

This paradoxical monument celebrates the fundamental geometric shape.

Triangles

- *The triangle is the fundamental shape of 2D geometry;*
- *Given the location of the vertices, many triangular properties can be computed;*
- *Examples include area, orientation, linear functions, random sampling, quadrature;*
- *Every polygon can be decomposed into triangles;*
- *General 2D shapes can be approximated by triangles;*
- *Triangulated objects can be understood using properties of triangles.*

1 Representing a triangle

We can specify a triangle by listing the (x, y) coordinates of its vertices. In Python, it makes sense to create a 3×2 array `v`, as in this case, known as the *reference triangle*:

```
v = np.array ( [ \
  [ 0.0, 0.0 ], \
  [ 1.0, 0.0 ], \
  [ 0.0, 1.0 ] ] )
```

Let's immediately try to draw this object to verify what we have done:

```
plt.plot ( v[:,0], v[:,1], 'r.', markersize = 10 )
I = [ 0, 1, 2, 0 ]
plt.plot ( v[I,0], v[I,1], 'b-', linewidth = 3 )
plt.axis ( 'equal' )
```

Here, we created the index list `I` to allow us to "wrap around" the vertex list, in order to consider each consecutive pair of vertices.

2 Properties of a triangle

The **side lengths** of a triangle can be computed:

```
s = np.zeros ( 3 )
for i in range ( 0, 3 ):
  if ( i < 2 ):
    s[i] = np.linalg.norm ( v[i+1] - v[i] )
  else
    s[i] = np.linalg.norm ( v[0] - v[i] )
```

Alternatively, we can use Python's modulus function to convert the index 3 to 0:

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

The **perimeter** is simply the sum of the side lengths.

The **angles** can be computed using the law of cosines. Suppose the vertices 0, 1, and 2 are labeled A, B, and C, and that we want to determine the angle α associated with vertex A. The Law of cosines says:

$$||B-C||^2 = ||A-B||^2 + ||A-C||^2 - 2 * ||A-B|| ||A-C|| \cos(\alpha)$$

or, using our array `s` of side lengths:

$$s_1^2 = s_0^2 + s_2^2 - 2 s_0 s_2 \cos(\alpha)$$

So we can compute

```
frac = - 0.5 * ( s[1]**2 - s[0]**2 - s[2]**2 ) / s[0] / s[2]
alpha = np.arccos ( frac )
```

We can write the formula for all angles if we are careful to use the modulus function for indices greater than `i`:

```
Purpose:
  Compute angles of a triangle
Compute:
  s = array of triangle side lengths
  for i in range ( 0, 3 ):
    angle[i] = - 0.5 * ( s[(i+1)%3]**2 - s[i]**2 - s[(i+2)%3]**2 ) / s[i] / s[(i+2)%3]
  angle = np.arccos ( angle )
Return
  angle
```

The **centroid** of a triangle can be thought of as the center of mass, assuming the triangle has a uniform density. However, more correctly, it is the center of the triangle's geometry. It is the point c such that any line through c divides the triangle into two regions of equal area. The formula for the centroid is quite simple:

$$c = \sum_i \frac{v_i}{3}$$

It's easy to do the arithmetic mentally to decude that the reference triangle has centroid $(\frac{1}{3}, \frac{1}{3})$; for general triangles we may appreciate having a function that will compute this value for us. The Python `sum()` function will add up all the entries of our array and give a single number as the result. In order to average the x and y entries separately, we need to include the `axis = 0` argument to return a separate sum over each column:

```
c = np.sum ( v[:], axis = 0 ) / 3.0
```

3 Area of a triangle

The area is one of the most important measurements we can make. We will look at three different ways of computing it, and then choose the one that is reasonably simple and gives extra information.

The classical procedure goes back to Euclid, and computes half the base times the height. Here the base is defined by two vertices, and the height is the distance from the third vertex to the line through the base. If you are drawing a diagram, and the triangle "leans" to one side or the other, you may have to extend the base in that direction in order to make an accurate measurement. Well the length of the base is easy to compute, the length of the height takes some work. We create a vector from v_0 to v_2. We know we can use projection to find the component of the vector that is parallel to the base line. Subtracting this component from the vector gives us the height vector. Taking the norm of this vector gives us the height.

This procedure can be written up as an algorithm `AE(T)` and a Python function we might call `triangle_area_euclid()`: awkward to formulate.

```
  base = np.linalg.norm ( v[1] - v[0] )
  base_unit = ( v[1] - v[0] ) / base

  vector = ( v[2] - v[0] )
  dot = np.dot ( vector , base_unit )
  vector_parallel = dot * base_unit
  vector_perp = vector - vector_parallel
  height = np.linalg.norm ( vector_perp )

  area = 0.5 * base * height
```

Heron's formula computes the area from the side lengths. We might refer to the algorithm as `AH(T)`:

$$AH(T) = \frac{1}{4}\sqrt{(s_0 + s + 1 + s_2)(-s_0 + s_1 + s_2)(s_0 - s_1 + s_2)(s_0 + s_1 - s_2)}$$

and the corresponding Python code is easy to write.

For our third area definition, recall the definition of the *cross product* of 2-dimensional vectors p and q:

$$p \times v = p_x q_y - p_y q_x$$

The cross product is equal to the area of a parallelogram with two sides formed by p and q. If we connect the ends of the vectors p and q, we form a triangle of half the area of the parallelogram. This gives us another formula for the area. To apply it, we can assume that $p = v_2 - v_1$ and $q = v_0 - v_1$. If we write $v_0 = (x_0, x_1)$ and so on, our algorithm `AC(T)` would be:

$$AC(T) = \frac{1}{2}(x_2 - x_1)(y_0 - y_1) - (y_2 - y_1)(x_0 - x_1)$$

The value returned by this formula is signed; that is, its absolute value is indeed the area, but the value itself might be positive or negative. It turns out that the sign of $AC(T)$ tells us something useful.

The **orientation** of a triangle is positive if the vertices are listed in counter-clockwise order, and negative otherwise. The sign of AC(T) matches the orientation of the triangle.

If we aren't careful about the order in which we list our triangle vertices, and we use AC(T) to compute areas, then of course we want to take the absolute value of the result. But in applications involving a mesh of triangles, it is important that the triangle all share the same (usually positive) orientation. In that case, it can be important to be able to check each triangle in the mesh in advance, so that we are able to correct the ordering of the vertices if necessary.

4 Is a point left or right of a line?

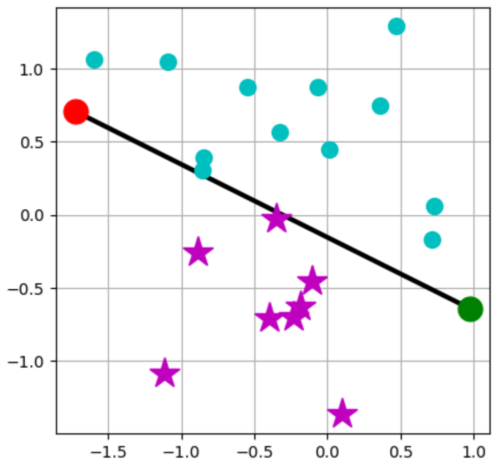
Teaching a computer to "see" geometric facts that are obvious to the eye can be tricky. A fact that the computer really needs to learn is to say whether a point r is to the left or right of a line passing from points p to point q. To be clear about left and right, we imagine that we are walking along the line in the direction from p to q, so that half the plane is on our left hand side, and half on our right.

Surprisingly, to answer this question, we simply have to evaluate the following cross product:

$$left_right(p, q, r) = (r - p) \times (q - p)$$

The position of r relative to the line $p \rightarrow q$ is

- to the left, if negative;
- on the line, if zero (never happens!);
- to the right, if positive



P green, Q red, Magenta = negative/left, Cyan = positive/right.

5 Is a point inside a triangle?

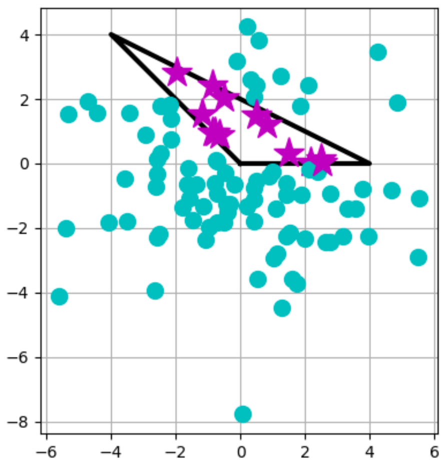
The question we really want to answer is, is a given point r inside or outside a triangle T. But now we can answer this question. Suppose that the vertices of T are given in counterclockwise order. (Remember, I told you orientation would be important!). Pretend you are walking along the perimeter, from vertex v_0, v_1, v_2 back to v_0. If point r is inside the triangle, then as you walk along each side of the triangle, r will **always** be to your left. And on the other hand, if r is outside the triangle, then there will be at least one side of the triangle that you traverse, for which r will be to the right, not the left.

In other words, our `triangle_inside` algorithm is

```
Purpose:
  Determine if point r is inside triangle T with vertices v
Iterate:
  inside = True
  for i in range ( 0, 3 )
```

```
    p = v[i]
    q = v[(i+1)%3)
    if 0 < left_right ( p, q, r )
      inside = False
      break
Return
  inside
```

Using this idea, we can draw a triangle (in counterclockwise order), create a bunch of random points nearby, and determine which ones are contained in the triangle:



Magenta = inside the triangle, Cyan = outside.

This function really becomes useful in situations where we have a mesh of many triangles. We will often need to evaluate a quantity whose definition depends on which triangle contains it. Our `triangle_contains()` function is a simple approach to answering that question.

6 Barycentric coordinates

Consider any triangle T with vertices v, and s be some point. Assume s is inside the triangle. The point s can be used to subdivide T into three smaller triangles, where s replaces one of the original vertices. We might describe these subtriangles as:

$$T_0 = \{s, v_1, v_2\}$$
$$T_1 = \{v_0, s, v_2\}$$
$$T_2 = \{v_0, v_1, s\}$$

Obviously, the area $A(T)$ of the original triangle equals the sum $A(T_0) + A(T_1) + A(T_2)$. Each of these areas is computable. For instance, to get $A(T_0)$ we compute one half the product of the length of the base $||v_1 - v_1||$ and the height of s above the base, just like we did earlier, applying the Euclidean area formula.

Now suppose we normalize these three values, by dividing by $A(T)$, getting values we will symbolize as

$$(\xi_0, \xi_1, \xi_2) = (\frac{A(T_0)}{A(T)}, \frac{A(T_1)}{A(T)}, \frac{A(T_2)}{A(T)})$$

The quantities (ξ_0, ξ_1, ξ_2) are known as the *barycentric coordinates* of the point s with respect to the triangle T. They can be regarded as the areas of each subtriangle relative to the original triangle, or the height of each subtriangle relative to the corresponding height of the original triangle. That is, ξ_i measures how close point s is to vertex v_i, on a scale of 0 to 1.

It should be obvious that the ξ values satisfy the following conditions:

- Assuming s is strictly inside T, then $0 < \xi < 1$ for each i;
- If $\xi_i == 1$ for some i, then s is on vertex i;
- If $\xi_i == 0$ for some i, then s lies on the triangle side opposite vertex i;
- $\xi_0 + \xi_1 + \xi_2 = 1$.
- Every point s inside T has a unique set of barycentric coordinates ξ;
- Every triple of nonnegative values ξ that sum to 1 identifies a unique point s in T.

We began this discussion by assuming that the point s was inside the triangle T, but this is not necessary. Every point in the plane can have its barycentric coordinates calculated. However, we must note that if the point is outside the triangle, then the height computation must result in a negative value for at least one component of the coordinates.

In fact, this gives us a second way to answer the question, "Is point s inside a triangle?". If we correctly compute the barycentric coordinates, then s is

- strictly inside the triangle if $0 < \xi < 1$ for every i;
- on the boundary of the triangle if one ξ is 1, and the other two are 0;
- outside the triangle if any ξ is negative (s is below the base) or greater than 1 (s is above the vertex).

We can also use barycentric coordinates to randomly sample points inside a given triangle.

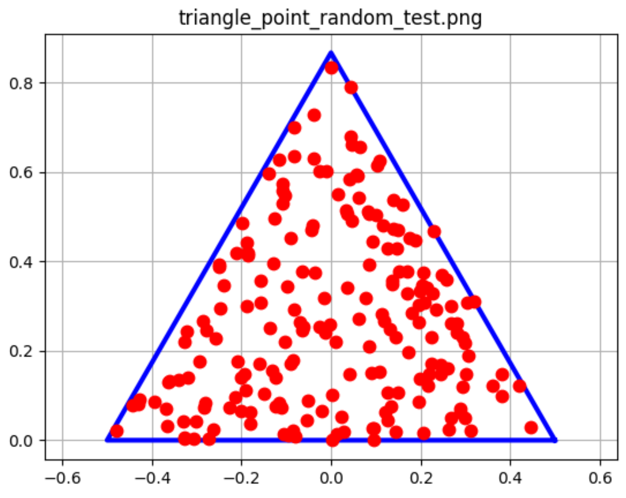
7 Random sampling

For our next "trick", we will need to be able to choose points p at random inside a triangle T. We want this choice to be done uniformly, which roughly speaking means that if we choose 1,000 points and plot them, they should be spread over the area of the triangle with no obvious pattern of clustering or clumping.

The barycentric coordinates provide a simple way to do this. Essentially, we need to choose three nonnegative numbers that add up to 1. We can regard this task, in turn, as randomly dividing a unit stick into three pieces. Let's say the first piece stops at point r_1, the second at r_2, and the third, obviously at 1.

```
Purpose:
  Select a point p at random in triangle T with vertices v
Compute:
  r1, r2 = uniform random values in [0,1]
  r1, r2 = min ( r1, r2 ), max ( r1, r2 )
  (Compute lengths of each segment)
  xi0 = r1 - 0
  xi1 = r2 - r1
  xi3 = 1 - r2
  (Compute random point in triangle)
  p = xi0 * v[0] + xi1 * v[1] + xi2 * v[2]
Return
  p
```

We can try out this algorithm by plotting a bunch of points inside one of our test triangles:



Random points inside a triangle.

8 Integrals by random sampling

Consider the integral $I(T, f(x,y))$ of some function $f()$ over the interior of triangle T. Recall that the mean value theorem essentially regards the integral as the average value of the function times the length or area of the domain. That means that, if we have n sample values of $f(x,y)$ at points inside T, we can estimate

$$I \approx A(T) * \frac{1}{n} \sum_{i=0}^{i<n} f(x_i, y_i)$$

We know how to compute areas, and we know how to sample a triangle uniformly. We don't know how many values n we need to get a good integral estimate, but we can try increasing n and watching how the results change.

Here is an example of such a calculation, in which we take $T = [(0,3),(1,1),(5,3)]$ and $f(x,y) = 6x^2 - 40y$. The area is 5. The exact value of the integral is $-311.666...$ although we should pretend we don't know that!

n	estimate	error
10	-335.0826933257193	23.41602665905259
100	-310.96273075260945	0.7039359140572401
1000	-318.7311974184857	7.064530751819007
10000	-313.5388492778983	1.872182611231608
10000	-310.90616615257994	0.7605005140867434

Notice that even with 10,000 points, there is still considerable error. Also, we see that using $n = 100$ points magically gives us a much better answer than we get for $n = 1,000$ and $10,000$. That's just a feature of this approach; sometimes a crude random sample will get a lucky set of sample values. However, if we repeated this calculation, the good estimate at $n = 100$ would very likely disappear.

This method of estimating integrals is known as the "Monte Carlo" method. Although there are methods for exactly computing the integral of a polynomial over any triangle T, these methods are little known and somewhat cumbersome. Moreover, for a general function $f(x,y)$, no exact approach is known for evaluating integrals. Thus, the Monte Carlo approach is very attractive, since it requires little fancy programming, and the estimates can generally be improved by increasing the number of sample points.

9 Integral over a triangle

An alternative approach to estimating integrals over triangles is to use a *quadrature rule*. A quadrature rule is a set of n points p and weights w. The rule is usually defined for the reference triangle $T = [(0,0),(1,0),(0,1)]$. We apply it by the formula

$$I(T, f(x,y)) \approx A(T) * \sum_{i=0}^{i<n} w_i f(p_i)$$

Typically, a quadrature rule is designed so that it returns the exact answer if the function $f(x,y)$ is a polynomial of degree less than or equal to some power. Often, a family of quadrature rules is available, in which rules with greater number of points n are exact for polynomials of higher and higher degree.

The simplest quadrature rule has $n = 1$, $w_0 = 1$ and $p_0 = (\frac{1}{3}, \frac{1}{3})$, known as the *centroid rule*. Surprisingly, this one-point rule is exact for any constant or linear function defined over the reference triangle.

As it turns out, the next rule, known as the *vertex rule*, uses $n = 3$ points, but is no more exact than the centroid rule. It uses $w = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ and $p = [(0,0),(1,0),(0,1)]$.

In order to precisely integrate functions up to quadratic order, we can use a three point rule of Strang and Fix. $n = 3$, $w = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, $p = [(\frac{4}{6}, \frac{1}{6}), (\frac{1}{6}, \frac{4}{6}), (\frac{1}{6}, \frac{1}{6})]$.

We can try all three rules on our example from the previous section:

name	estimate	error
Centroid	-6.333	0.1666
Vertex	-6.666	0.5000
Strang & Fix	-6.166	0.0

The Strang rule was able to get the result exactly, because the integrand was a polynomial of degree 2 or less.

If our integration triangle is not the reference triangle, what do we do? Let's return to the problem we solved with the Monte Carlo method. In order to apply our quadrature rules, we simply need to determine the new locations of the quadrature points. The easiest way to do that is to compute the barycentric coordinates of the quadrature points. But since the quadrature points have been defined in the reference triangle, the barycentric coordinates are trivial to determine. For any point p, we have

$$\xi(p) = (p_0, p_1, 1.0 - p_0 - p_1)$$

Then to determine the image of and point p when mapped to a triangle U, we can apply these barycentric coordinates to the vertices w of U:

$$q = \xi_0 * w_0 + \xi_1 * w_1 + \xi_2 * w_2$$

This means that the coordinates of our Strang rule become:

$$p_0 = (\frac{10}{6}, \frac{16}{6})$$
$$p_1 = (\frac{9}{6}, \frac{10}{6})$$
$$p_2 = (\frac{21}{6}, \frac{16}{6})$$

and now we can estimate the integral of $f(x,y) = 6x^2 - 40y$ over triangle U using our three rules:

name	estimate	error
Centroid	-346.666	35.000
Vertex	-206.666	105.000
Strang & Fix	-311.666	0

Once again, we see that Strang is exact for this integrand, and once again, we should notice that the 1 point centroid rule seems to have better accuracy than the 3 point vertex rule.

10 Mapping one triangle to another

Assume that triangle T has vertices v, and we want to find a map that transforms points p in T into points q in a new triangle U, with vertices w.

Actually, we are looking for an *affine* map $\mathcal{M}(T,U)$. To simplify our problem to a linear map, let $T - v_0$ and $U - w_0$ represent the triangles T and U shifted so that the first vertex is at (0,0). Now if we ask for a *linear* map A that transforms $T - v_0$ to $U - w_0$, we can do the necessary adjustments before and after we apply A in order to define our affine map.

We only have to check that we map the second and third vertices correctly. This means we need to satisfy the following equations:

$$A * (v_1 - v_0) = w_1 - w_0$$
$$A * (v_2 - v_0) = w_2 - w_0$$

We can rewrite this as

$$A * [(v_1 - v_0], (v_2 - v_0)] = [(w_1 - w_0), (w_2 - w_0)]$$

Now we have to look at this equation as a slightly nontypical linear system to be solved. It has the form $A * B = C$, where A is the unknown, and B and C are matrices. By taking the transpose of our equationwe get $B' * A' = C'$ and now we have something more familiar. B' is the coefficient matrix. We can think of A' as a pair of unknown column vectors, and C' as a pair of known right hand sides. We can define these transposes using Python expressions like `AT = np.transpose(A)` or `AT = A.T`). Then we can call a linear equation solver, as in `AT = np.linalg.solve(BT, CT)` to get A', and then transpose again to get our desired matrix $A = np.transpose(AT)$.

Finally, this means our affine mapping $\mathcal{M}$ of a point p has the form

$$\mathcal{M}(T,U)(p) = A * (p - v_0) + w_0$$

As a simple example, let T be the reference triangle with vertices [(0,0),(1,0),(0,1)] and let U be the triangle with vertices [(4,1),(7,2),(6,5)]. Our linear system for A has the form

$$A \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

This is a linear system so simple we can solve it immediately,

$$A = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix}$$

and so our mapping is

$$\mathcal{M}(T,U)(p) = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} * \left(\begin{bmatrix} p_0 \\ p_1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) + \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

Recall that the centroid of the reference triangle is $(\frac{1}{3}, \frac{1}{3})$. The mapping $\mathcal{M}$ will send this to

$$q = \mathcal{M}(T,U)(p) = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} * \left(\begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) + \begin{bmatrix} 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 5\frac{2}{3} \\ 2\frac{2}{3} \end{bmatrix}$$

and it should be no suprise that if we compute the centroid of the triangle U directly, we get this same result.

11 A linear function

Suppose we want to define a linear (affine, actually) function $z = f(x, y)$ over a triangle T. It is enough to specify the values of this function at the vertices v, so that we are given $z_i = f(v_i), 0 \leq i \leq 2$. Now suppose we want to evaluate $f()$ at some point $p = (x, y)$ inside the triangle. It is a happy fact that all we have to do is compute the barycentric coordinates $\xi(p)$, after which we may write

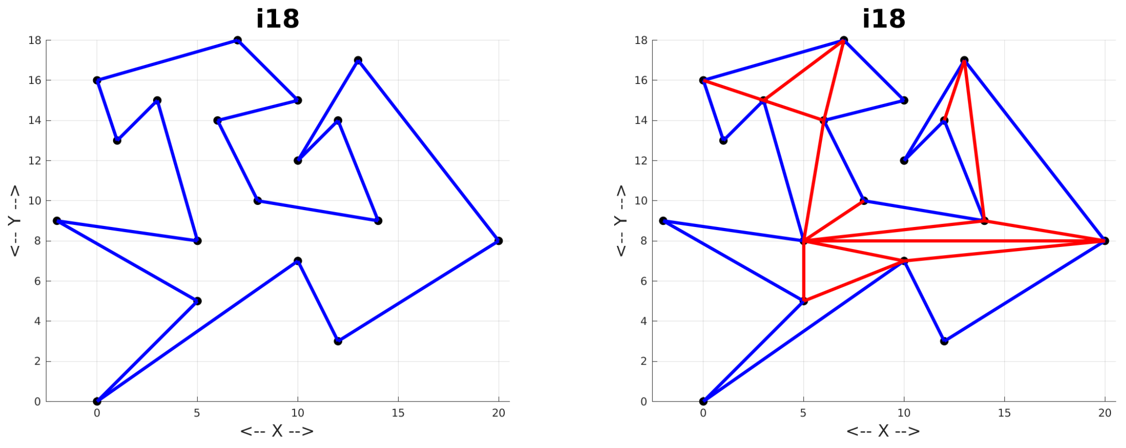
$$f(p) = \xi_0 z_0 + \xi_1 z_1 + \xi_2 z_2$$

This fact will be very useful later, when we have values of data $z = f(x, y)$ at many scattered points. If we can group the points into triangles, we can then construct a piecewise linear function that can be evaulated at points where we have no data.

12 Decomposing a polygon into triangles

Finally, we want to suggest that knowing about triangles gives us the ability to analyze more complicated figures. Polygons are of great importance mathematically. Can we reduce any polygon to a collection of triangles? If the polygon is convex, we could just put a point in its center and make triangles by connecting each vertex to this center point. For nonconvex polygons, however, this won't work at all. A general approach uses the fact that every polygon with more than three sides must have at least two "ears", that is, three consecutive vertices a, b and c, such that the line from a to c does not leave the polygon. We can add triangle a,b,c to our collection, and consider the simplified polygon that has lost vertex b. If the remaining polygon is a triangle, add that to our collection and we are done. Otherwise, find another ear, lop it off and add it to our collection.

As an example, here is a polygon before and after it has been reduced to a collection of triangles.



The triangulation of a polygon.

Once the polygon has been triangulated, we can easily compute its area, and we can estimate integrals over it. To do random sampling or compute the centroid, we have to use a procedure that is weighted by the areas of each triangle. This is a nice challenge for you!

13 Exercises

1. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, compute the area, angles, centroid, and side lengths.
2. Consider the line that passes through (5,4) to (9,1). Which of the following points are to the left, and which to the right, of this line: (5,2), (6,5), (7,4), (8,1). Determine your answer computationall.y
3. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, which points are contained in the triangle? (0,0), (1,6), (2,3), (3,5), (4,3).
4. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, what are the barycentric coordinates of each of the points (0,0), (1,6), (2,3), (3,5), (4,3).
5. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, generate and plot 100 random sample points inside the triangle.
6. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, use the Monte Carlo method to estimate the integral of $f(x,y) = x^2 + 3y^2$.
7. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, use the Strang rule to estimate the integral of $f(x,y) = x^2 + 3y^2$.
8. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, what is the form of the affine mapping to the obtuse triangle U with vertices $w = [(0,0),(1,0),(-1,1)]$.
9. For the **scalene** triangle T with vertices $v = [(0,6),(1,1),(5,3)]$, suppose we associate values $z_0 = 18, z_1 = 5, z_2 = 19$ to the three vertices. Define a linear function $f(x,y)$ whose values are $f(v_i) = z_i$. What is the value of $f(2,4)$?
10. Suppose a polygon P has been decomposed into 3 triangles T_0, T_1, T_2. Describe a procedure for computing the centroid of P.
11. Suppose a polygon P has been decomposed into 3 triangles T_0, T_1, T_2. Describe a procedure for sampling 100 points uniformly over P, using what we know about triangles. You should imagine that one of the triangles might be very small, one average, and one big. So there should be many more sample points in the big triangle.