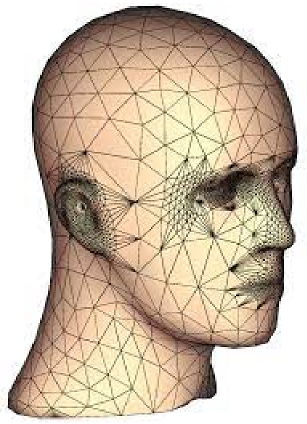
Triangulations
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/mpp_2023/triangulations/triangulations.pdf



Even 3D geometry can be suggested with triangles.

Triangulations

- *The triangle is the fundamental shape of 2D geometry;*
- *Every polygon can be decomposed into triangles;*
- *General 2D shapes can be approximated by triangles;*
- *Triangulated objects can be understood using properties of triangles.*
- *Partial differential equations are solved on triangular grids.*
- *Triangles create 3D images for games, movies, medicine, science.*

1 Representing a triangulation

A triangulation is a collection of triangles. Generally, these triangles are connected, and in fact, connected triangles will share a side. Triangulations commonly lie flat, in 2D, but they can also describe surfaces in 3D, and in fact, this is the way the computer graphics programs create scenes and characters in video games and animations.

Any triangle can be specified by listing the coordinates of its three vertices. Since the triangles in a triangulation will connect to each other, they share some vertices. This makes it efficient to represent a triangle by two arrays. The first is a real array of $node\_num \times 2$ or $node\_num \times 3$ list of xy or xyz coordinates. The second is an integer array of $tri\_num \times 3$ indices of the node array, pointing to the three vertices of each triangle.

Consider the following triangulation, with the nickname "tiny":

```
A-------B----D
 \  0  /| 2 /
  \   / |  /
   \ / 1| /
    C---E
```

Assuming the node numbering $A = 0, B = 1, C = 2, D = 3, E = 4$, the node coordinate array will be

```
  node_xy = [
    [ 0.0, 4.0 ]
    [ 8.0, 4.0 ]
    [ 4.0, 0.0 ]
    [13.0, 4.0 ]
    [ 8.0, 0.0 ] ]
```

and the triangle node array will be

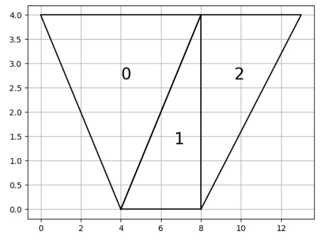
```
  triangle_node = [
    [ 2, 1, 0 ]
    [ 1, 2, 4 ]
    [ 1, 4, 3 ] ]
```

2 Plotting a triangulation

Given the node and triangle information, we can write a function to plot a triangulation:

```
def triangulation_plot ( node_xy, triangle_node ):
  import matplotlib.pyplot as plt
  triangle_num = triangle_node.shape[0]
  for t in range ( 0, triangle_num ):
    for j in range ( -1, 2 ):  # This weird statement works!  Write out what it does.
      n0 = triangle_node[n,j]
      n1 = triangle_node[n,j+1]
      x0 = node_xy[n0,0]
      y0 = node_xy[n0,1]
      x1 = node_xy[n1,0]
      y1 = node_xy[n1,1]
      plt.plot ( ( x0, x1 ), ( y0, y1 ), 'k-' )
  plt.show ( )
```

Here's the resulting plot for our tiny triangulation example:



3 Orientation

It is usually the case that every triangle in a triangulation has a common orientation. What this means in our situation is that, in the `triangle_node()` array, the vertices of each triangle are listed so that they move in a counterclockwise direction. For instance, triangle 0 in the tiny triangulation has vertices A, B, C, (numbered 0, 1, 2). However, in the `triangle_node()` array, we list them in the order C, B, A (numbered 2, 1, 0) to enforce the correct orientation.

In the previous discussion of triangles, a formula $AC(T)$ was presented, which returns the area of a triangle, with a sign. The area is negative if the triangle has the "wrong" clockwise orientation. Using this function, the orientation of each triangle in a triangulation's can be checked. If a triangle has the wrong orientation, this can be corrected by simply reversing the order in which its nodes are listed.

One of the exercises asks you to take the "screwy" triangulation, a badly oriented triangulation in which some triangles are described clockwise and some counterclockwise. Your task is to correct the bad `triangle_node()` array.

Here is the information for the screwy triangulation:

```
S_T-U
|\|\|
P-Q-R
|\|\|
K-L-M-N-O
|\|\|\|\|
F-G-H-I-J
|\|\|\|\|
A-B-C-D-E
```

Assuming the node numbering $A = 0, B = 1, C = 2, D = 3, E = 4$, and so on, the node coordinate array will be

```
node_xy = [
0.0   0.0
1.0   0.0
2.0   0.0
3.0   0.0
4.0   0.0
0.0   1.0
1.0   1.0
2.0   1.0
3.0   1.0
4.0   1.0
0.0   2.0
1.0   2.0
2.0   2.0
3.0   2.0
4.0   2.0
0.0   3.0
1.0   3.0
2.0   3.0
0.0   4.0
1.0   4.0
2.0   4.0  ]
```

and the triangle node array will be

```
triangle_node = [
 0   1   5
 1   5   6
 1   2   6
 7   6   2
 7   3   2
 3   7   8
 3   4   8
 9   8   4
 5   6  10
 6  10  11
 6   7  11
12  11   7
 7   8  12
```

```
  13  12   8
  13   9   8
   9  13  14
  10  11  15
  16  15  11
  11  12  16
  12  16  17
  15  16  18
  19  18  16
  19  17  16
  20  19  17
 ]
```

4 The boundary

When the triangles in a triangulation have the proper orientation, this fact can be relied on to simplify various calculations. In particular, we can discover the triangle edges that form the boundary of the triangulation.

For instance, we can list the edges of a triangle as consecutive pairs of vertices. Triangle 0 in the tiny triangulation has edges (2,1), (1,0), and (0,2), where we take consecutive pairs of nodes in counterclockwise order, finishing with the edge formed by the last and first nodes.
An edge of a triangulation can be interior or exterior. An interior edge is shared by two triangles, while an exterior edge is only part of one triangle. Assuming the triangulation has a counterclockwise orientation, then

- every interior edge appears twice in `triangle_node()`, once as (a,b) and once as (b,a).
- every exterior edge appears only once in `triangle_node()`.
- The set of exterior edges forms the boundary of the triangulation.
- The set of exerior edges can be listed as a closed path (a,b), (b,c), .., (z,a).
- This closed path traverses the boundary of the triangulation in counterclockwise direction.

In the exercises, you are asked to explore and verify some of these statements using programming.

5 Area of a triangulation

In a moment, we will try to randomly sample points inside a triangulated region. To do that, we will need first to compute the area of each triangle, and the total area of the triangulation. From our discussion of triangles, we already know how to compute the area of a triangle. So now we simply want to compute an area A containing the area of each triangle. In our discussion of triangles, we represented a triangle by a 3×2 array v, containing the x, y coordinates of vertex i in row i. Suppose we have available a corresponding function *triangle_area(v)* and we want to use that with our triangulation data. How do we extract information from the triangulation representation and put it into an array v?

A straightforward Python implementation would look something like this:

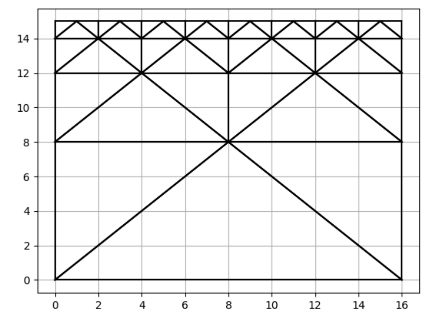
```
v = np.zeros ( [ 3, 2 ] )
for t in range ( 0, triangle_num ):
  for vertex in range ( 0, 3 ):
    node_index = triangle_node[t,vertex]
    x = node_xy[node_index,0]
    y = node_xy[node_index,1]
    v[vertex,0] = x
    v[vertex,1] = y
  triangle_area[t] = triangle_area ( v )
```

There is a more adventurous way to do this. It relies on the fact that in Python, a two-dimensional array is really a list of lists. Thus, if refer to such an array with a single index i, then you get the entire row i of the array.

```
  for t in range ( 0, triangle_num ):
    v = node_xy[triangle_node[t]]
    triangle_area[t] = triangle_area ( v )
```

You may find this second version obscure, or delightful. But stop for a moment and try to understand why these two versions are equivalent.

In the exercises, you will be asked to compute the areas of the triangles in the "shrink" triangulation, which involves 36 nodes and 45 triangles in a 16×15 rectangle.



6 Uniform samples of a triangulation

In order to uniformly sample points in a triangulation, we need to know the area of each triangle. That's because a uniform sampling means that, on average, two regions of equal area should be sampled equally often. Hence, random sampling starts by picking a triangle from the triangulation using area as a weight; once we have picked a triangle, we know how to sample it because of our previous discussion on triangles.

To see what is going on, suppose that our triangulation involves just two triangles, T_0 of area $A_0 = 6$ and T_1 having area $A_1 = 12$, so the triangulation has total area $A = 18$. We see that T_0 involves $\frac{6}{18} \approx 0.33$ of the total area and T_1 has $\frac{12}{18 \approx 0.66}$. So sampling begins by picking a random number r. If $0 \leq r \leq 0.33$, we sample T_0, otherwise T_1.

To see the general case, suppose our triangulation involves three triangles, T_0, T_1, T_2. Then we sample

- T_0 if $0 \leq r < \frac{A_0}{A}$
- T_1 if $\frac{A_0}{A} \leq r < \frac{A_0+A_1}{A}$
- T_2 if $\frac{A_0+A_1}{A} \leq r \leq \frac{A_0+A_1+A_2}{A} = 1$

If you think about this pattern, you will see how to choose which triangle to sample for the general case where the triangulation involves $triangle_num$ triangles:

```
Purpose:
  Uniformly sample n points from a triangulation
Compute:

  For each triangle t
    A(t) = area
  AT = sum ( A )

  for each sample point p
    pick r at random in [0,1]
    find triangle t such that A(0:t-1)/AT <= r < A(0:t)/AT
    p = random sample of triangle t
```

The exercises ask you to write such a program, and apply it to the "shrink" triangulation.

7 Estimating an integral

Suppose that we wish to estimate the integral of some function $f(x, y)$ over some region R. In some cases, we can use calculus to determine the integral $\int_R f(x, y)\,dx\,dy$. Often, however, the function or region is so irregular that calculus will not be sufficient. Suppose, instead, that we have a triangulation Δ which exactly covers the region R. (In some cases, such as when R is a circle, a triangulation cannot exactly cover the region. Even then, we might be willing to try a triangulation that comes close to covering!)

Let's see if we can use the triangulation to estimate the integral we are interested in.

The simplest approach uses random sampling:

```
Purpose:
  Estimate integral of f(x,y) over a triangulation DELTA using n random samples
Compute:
  Compute the areas of the triangles
  Set AREA_DELTA as the sum of the triangle areas
  Initialize Q = 0.0
  Iterate n times on index i
    Compute random sample point p(i) = x(i), y(i)
    Q = Q + f(x(i),y(i))
  Normalize Q = Q * Area(DELTA) / n
Return
  Q
```

The second simplest approach uses the centroid quadrature rule on each triangle:

```
Purpose:
  Estimate integral of f(x,y) over a triangulation DELTA using centroid rule
Compute:
  Compute the areas of the triangles
  Initialize Q = 0.0
  For each triangle t
    Compute centroid = x(i), y(i)
    Q = Q + Area(t) * f(x(i),y(i))
Return
  Q
```

The third approach uses the Strang quadrature rule on each triangle:

```
Purpose:
  Estimate integral of f(x,y) over a triangulation DELTA using Strang rule
Compute:
  Compute the areas of the triangles
  Initialize Q = 0.0
  For each triangle t
    For three Strang points p - x(i),y(i)
      Q = Q + Area(t) * f(x(i),y(i)) / 3
Return
  Q
```

Note that there are many quadrature rules for triangles that do even better than the Strang rule. However, these rules use more points in each triangle, and the values of the points and weights become more complicated to work with.

8 Exercises

1. Implement the function *triangulation_plot()*. Compute the centroid of each triangle, and call the `matplotlib()` function ittext() to plot the index of each triangle. Use your function to plot the "tiny" triangulation.
2. In most triangulations, the vertices of each triangle are listed in counterclockwise order. Write a function which accepts a `triangle_node()` array, checks that each triangle's nodes are listed in the proper orientation, and if not, modifies `triangle_node()` so that that order is corrected. Try your function on the "screwy" triangulation.
3. Using only the `triangle_node()` data for the tiny triangulation, try to explain why the boundary of the triangulation must be the sequence of edges (0,2), (2,4), (4,3), (3,1), (1,0).
4. Write a code which accepts a `triangle_node()` array for a triangulation, and returns the edges that form the triangulation boundary.
5. Do the previous exercise, but in addition, list the boundary edges or boundary nodes in order. For the tiny triangulation, for instance, the boundary node sequence would be 0, 2, 4, 4, 1, 0.
6. Write a code which accepts a `triangle_node()` array for a triangulation, and plots the interior edges in blue, and the exterior edges in red.
7. Write a code *triangulation_area(node_xy,triangle_node* which returns the area of each triangle in a triangulation. Test it on the "shrink" triangulaton.
8. Write a code *triangulation_sample(node_xy,triangle_node,point_num* which returns `point_num` uniformly random sample points from a triangulation. Use your code to sample the "shrink" triangulation. Do your sample points seem evenly distributed over this region?
9. Estimate the integral of $1 + x + y$ over the 16×15 rectangle, using the "shrink" triangulation and random sampling. The exact answer is 3960.
10. Estimate the integral of $1 + x + y$ over the 16×15 rectangle, using the "shrink" triangulation and the centroid rule on each triangle. The exact answer is 3960.
11. Estimate the integral of $1 + x + y$ over the 16×15 rectangle, using the "shrink" triangulation and the Strang rule on each triangle. The exact answer is 3960.