# Triangulation d'un polygone simple

## 1. Introduction

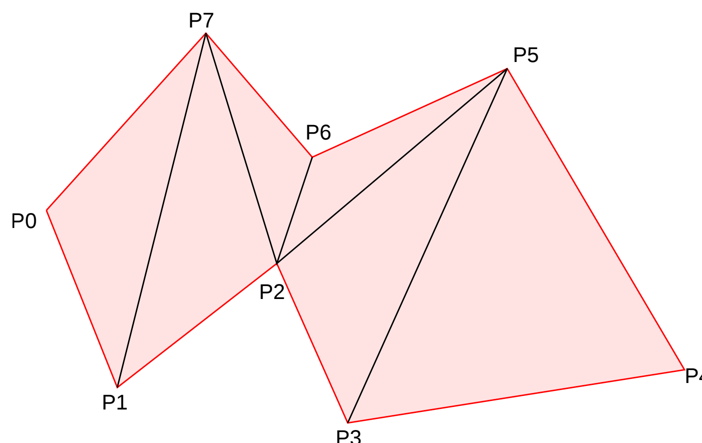
La triangulation d'une surface plane polygonale est une décomposition de la surface en triangles. La triangulation est en particulier utilisée pour la représentation graphique des surfaces, car le coloriage d'une surface triangulaire est une opération élémentaire simple à programmer. Les processeurs graphiques sont programmés pour colorier des surfaces élémentaires triangulaires (les triangles sont traités en parallèle). La résolution numérique des équations différentielles à dérivées partielles fait aussi appel à la triangulation des surfaces.

On sait par ailleurs qu'une courbe plane, par exemple une courbe de Bézier, peut être approchée par une ligne polygonale. On pourra donc trianguler la surface délimitée par une courbe fermée en triangulant cette ligne polygonale.

On s'intéresse ici à un algorithme permettant de trianguler un polygone simple, dont le temps d'exécution est quadratique par rapport au nombre de sommets du polygone. Il convient pour les polygones comportant peu de sommets. Il existe aussi un algorithme dont le temps d'exécution est $O(n\ ln(n))$ [1], préférable pour les grands polygones.

## 2. Définitions

On considère une région polygonale du plan, délimitée par une chaîne polygonale fermée qui ne se coupe pas elle-même. On appelera par la suite *polygone* cette région. La figure suivante montre un polygone dont les sommets sont $P_0, P_1, ...P_7$.




Par convention, l'intérieur du polygone se trouve à gauche des segments lorsqu'on parcourt les sommets dans l'ordre. Autrement dit, le contour polygonal est orienté dans le sens trigonométrique.

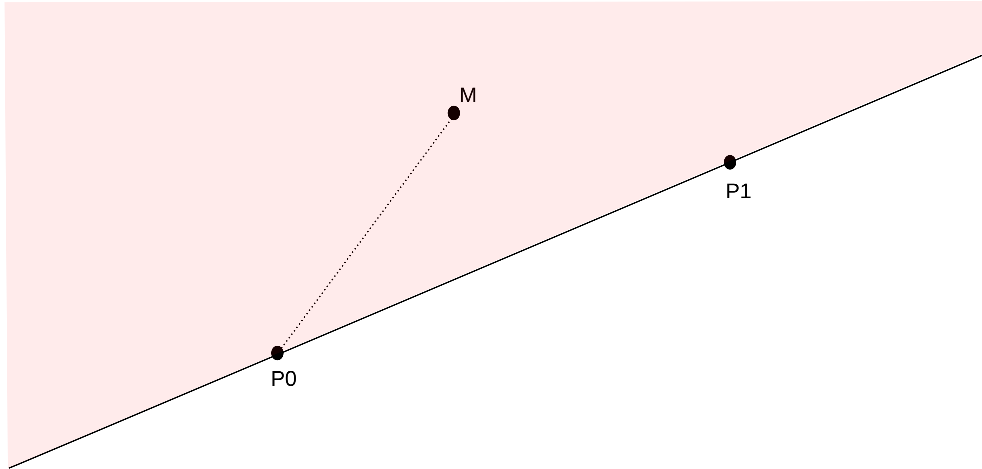
Une *diagonale* est un segment qui relie deux sommets et qui se trouve entièrement dans le polygone.

Une triangulation est une décomposition du polygone en surfaces triangulaires. Elle est obtenue en déterminant un ensemble maximal de diagonales. La figure ci-dessus montre un exemple. Il existe à l'évidence plusieurs triangulations d'un polygone.

# 3. Algorithme de triangulation

## 3.a. Position d'un point par rapport à un segment

Les points sont définis par leurs coordonnées $(x, y)$ dans un repère orthogonal $(0, \overrightarrow{u_x}, \overrightarrow{u_y})$. Soit $P_0P_1$ un segment (par exemple un côté du polygone) et $M$ un point quelconque.



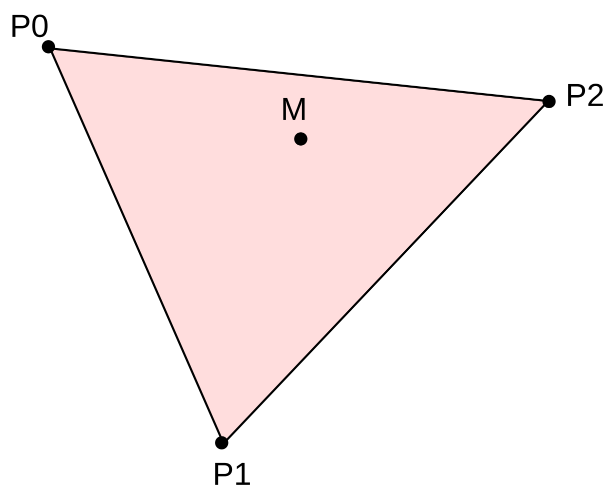

Pour savoir de quel côté de la droite $P_0P_1$ le point $M$ se trouve, on considère le produit vectoriel suivant :

$$L_z = (\overrightarrow{P_0P_1} \wedge \overrightarrow{P_0M}) \cdot \overrightarrow{u_z} \tag{1}$$

Si $L_z > 0$ alors le point se trouve du côté gauche (région coloriée sur la figure). Si $L_z = 0$, le point se trouve sur la droite.

## 3.b. Point dans un triangle

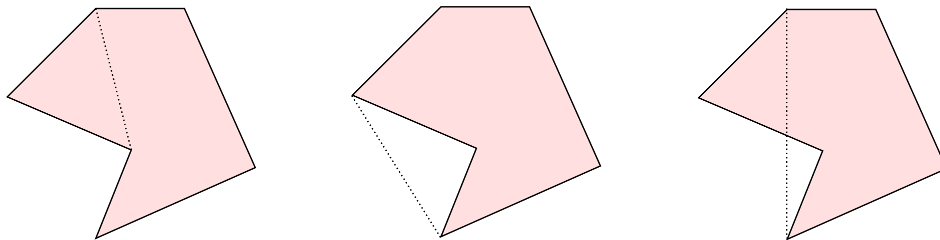
Pour savoir si un point se trouve à l'intérieur d'un triangle, supposons que les sommets $(P_0, P_1, P_2)$ soient définis dans le sens trigonométrique.



Pour que le point $M$ soit dans le triangle (sans être sur un de ses côtés), il faut et il suffit que les trois produits $L_z$ considérés ci-dessus soient strictement positifs.

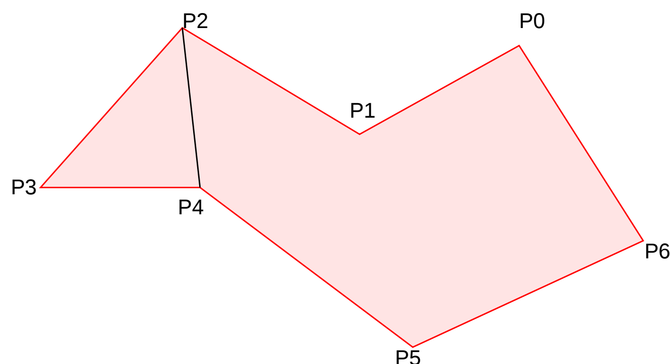
### 3.c. Segment à l'intérieur d'un polygone

Considérons un segment formé par deux sommets quelconques du polygone (non consécutifs). Le segment est soit à l'intérieur du polygone, soit complètement à l'extérieur, soit coupé par (au moins) un côté du polygone. La figure suivante illustre ces trois cas :



### 3.d. Triangulation récursive

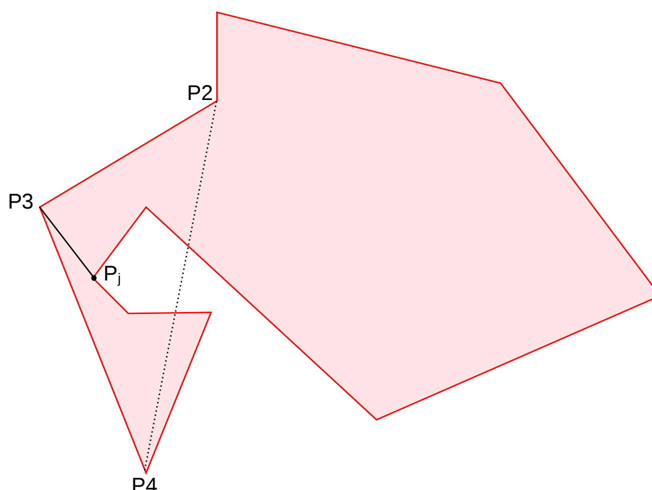
Considérons le sommet le plus à gauche du polygone, $P_3$ sur la figure suivante, et le segment formé des deux sommets voisins $P_2$ et $P_4$. Supposons que ce segment soit dans le polygone, ce qui en fait une diagonale :




Le triangle $P_2P_3P_4$ peut être ajouté à la liste des triangles. Le sommet $P_3$ est enlevé du polygone, et on applique récursivement la triangulation au polygone restant $P_0P_1P_2P_4P_5P_6$.

Considérons à présent le cas où le segment formé des deux sommets voisins n'est pas à l'intérieur du polygone. Comme nous l'avons vu plus faut, il peut être alors soit complètement hors du polygone, soit coupé par au moins un côté du polygone (la distinction

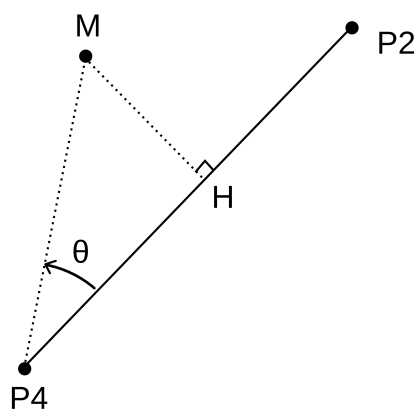
de ces deux cas n'importe pas ici). Il existe nécessairement un ou plusieurs sommets à l'intérieur du triangle $P_2P_3P_4$. Soit $P_j$ le sommet situé dans le triangle $P_2P_3P_4$ le plus loin de $P_2P_4$.




Le segment $P_3P_j$ constitue une diagonale, car si un côté coupait ce segment, il aurait une extrémité dans le triangle $P_2P_3P_4$, plus éloigné de $P_2P_4$ que ne l'est $P_j$ (en contradiction avec l'hypothèse).

La diagonale $P_3P_j$ permet de diviser le polygone initial en deux polygones. On applique récursivement la diagonalisation à ces deux polygones.

Pour chercher le sommet $P_j$ appartenant au triangle et le plus éloigné du segment $P_2P_4$, on utilise à nouveau le produit vectoriel (1).




La valeur absolue de ce produit est en effet :

$$|L_z| = |P_4P_2||P_4M|\sin(\theta) = |P_4P_2||HM| \quad (2)$$

où $H$ est le projeté orthogonal de $M$ sur la droite $P_4P_2$.

Le sommet $P_j$ est donc le sommet qui appartient au triangle et dont le produit $|L_z|$ est le plus grand.

Si aucun sommet $P_j$ ne vérifie cette condition, c'est-à-dire si le triangle $P_2P_3P_4$ ne contient aucun sommet, alors le triangle $P_2P_3P_4$ peut être ajouté à la liste des triangles et le polygone restant est triangulé récursivement.

## 4. Implémentation en python

```
# -*- coding: utf-8 -*-
import math
from matplotlib.pyplot import *
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
import numpy
```

Les points sont représentés par une liste `[x,y]`. Un polygone est une liste de points.

La fonction `voisin_sommet(n,i,di)` renvoit l'indice $i+di$ pour un polygone à $n$ sommets, en tenant compte des conditions limites circulaires $P_n=P_0$ et $P_{-1}=P_{n-1}$.

```
def voisin_sommet(n,i,di):
    return (i+di)%n
```

La fonction `equation_droite(P0,P1,M)` calcule le produit $L_z$ pour un point $M$ par rapport à la droite $P_0,P_1$. Cette fonction permettra de savoir de quel côté de la droite le point se situe, et de comparer les distances de différents points à cette droite.

```
def equation_droite(P0,P1,M):
    return (P1[0]-P0[0])*(M[1]-P0[1])-(P1[1]-P0[1])*(M[0]-P0[0])
```

La fonction `point_dans_triangle(triangle,M)` renvoit `True` si le point M est à l'intérieur (strictement) du triangle. Les points du triangles doivent être donnés dans le sens trigonométrique.

```
def point_dans_triangle(triangle,M):
    P0 = triangle[0]
    P1 = triangle[1]
    P2 = triangle[2]
    return equation_droite(P0,P1,M) > 0 and equation_droite(P1,P2,M) > 0 and equation_
```

La fonction `sommet_distance_maximale(polygone,P0,P1,P2,indices)` détermine le sommet du polygone appartenant au triangle `P0,P1,P2`, qui est à la plus grande distance du côté `P1,P2`. Les indices des points du triangle dans la liste `polygone` sont fournis dans `indices`. Par exemple, `indices=[4,5,6]` si les points `P0,P1,P2` sont `polygone[4]`, `polygone[5]` et `polygone[6]`. La fonction renvoit l'indice $j$ du point $P_j$. Si le triangle ne contient aucun sommet, l'objet `None` est renvoyé.

```
def sommet_distance_maximale(polygone,P0,P1,P2,indices):
    n = len(polygone)
    distance = 0.0
    j = None
    for i in range(n):
        if not(i in indices):
            M = polygone[i]
            if point_dans_triangle([P0,P1,P2],M):
                d = abs(equation_droite(P1,P2,M))
                if d > distance:
                    distance = d
                    j = i
    return j
```

La fonction `sommet_gauche(polygone)` renvoit l'indice du sommet le plus à gauche du polygone. Si plusieurs sommets ont la même abscisse, l'un quelconque de ces sommets peut être choisi.

```
def sommet_gauche(polygone):
    n = len(polygone)
    x = polygone[0][0]
    j = 0
    for i in range(1,n):
        if polygone[i][0] < x:
            x = polygone[i][0]
            j = i
    return j
```

La fonction `nouveau_polygone(polygone,i_debut,i_fin)` génère un nouveau polygone en allant de l'indice `i_debut` à l'indice `i_fin`. Il faut bien sûr tenir compte de la condition cyclique lors de l'incrément de l'indice.

```
def nouveau_polygone(polygone,i_debut,i_fin):
    n = len(polygone)
    p = []
    i = i_debut
    while i!=i_fin:
        p.append(polygone[i])
        i = voisin_sommet(n,i,1)
    p.append(polygone[i_fin])
    return p
```

La fonction principale est `trianguler_polygone_recursif(polygone,liste_triangles)`. Elle effectue la division du polygone en deux parties, en partant de son sommet le plus à gauche. Elle s'appelle récursivement pour les nouveaux polygones qui ne sont pas réduits à un triangle. La liste des triangles est complétée.

```
def trianguler_polygone_recursif(polygone,liste_triangles):
    n = len(polygone)
    print(polygone)
    j0 = sommet_gauche(polygone)
    j1 = voisin_sommet(n,j0,1)
    j2 = voisin_sommet(n,j0,-1)
    P0 = polygone[j0]
    P1 = polygone[j1]
    P2 = polygone[j2]
    j = sommet_distance_maximale(polygone,P0,P1,P2,[j0,j1,j2])
    if j==None:
        liste_triangles.append([P0,P1,P2])
        polygone_1=nouveau_polygone(polygone,j1,j2)
        if len(polygone_1)==3:
            liste_triangles.append(polygone_1)
        else:
            trianguler_polygone_recursif(polygone_1,liste_triangles)
    else:
        polygone_1 = nouveau_polygone(polygone,j0,j)
        polygone_2 = nouveau_polygone(polygone,j,j0)
        if len(polygone_1)==3:
            liste_triangles.append(polygone_1)
        else:
            trianguler_polygone_recursif(polygone_1,liste_triangles)
        if len(polygone_2)==3:
            liste_triangles.append(polygone_2)
        else:
            trianguler_polygone_recursif(polygone_2,liste_triangles)
    return liste_triangles
```

La fonction `trianguler_polygone(polygone)` effectue la recherche des triangles et renvoit la liste des triangles.

```
def trianguler_polygone(polygone):
    liste_triangles = []
    trianguler_polygone_recursif(polygone,liste_triangles)
    return liste_triangles
```

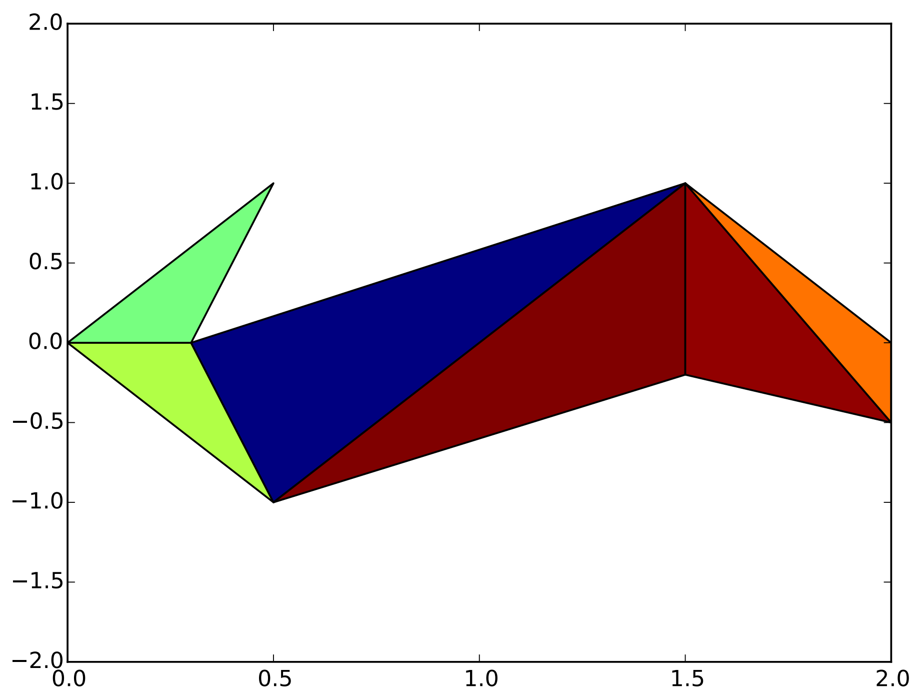
La fonction suivante sera utilisée pour colorier les polygones avec une couleur aléatoire :

```
def draw_liste_triangles(liste_triangles):
    fig,ax = subplots()
    patches = []
    for triangle in liste_triangles:
        patches.append(Polygon(triangle))
    p = PatchCollection(patches, cmap=matplotlib.cm.jet, alpha=1.0)
```

```
    colors = 100*numpy.random.rand(len(patches))
    p.set_array(numpy.array(colors))
    ax.add_collection(p)
```

Voici un exemple :

```
polygone = [[0,0],[0.5,-1],[1.5,-0.2],[2,-0.5],[2,0],[1.5,1],[0.3,0,],[0.5,1]]
liste_triangles = trianguler_polygone(polygone)
draw_liste_triangles(liste_triangles)
axis([0,2,-2,2])
```




**Référence :**

[1] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, *Computational geometry*, (Springer, 2010)