Polygon Cut

February 13, 2023

[1]:
```
from IPython import display
```

1 Polygon Cut

L'obiettivo di questo progetto è implementare una funzione chiamata `CutPoly`, la quale, presi in input un poligono con i vertici numerati e un segmento, restituisce gli N poligoni derivati dal taglio e la collezione dei nuovi vertici concatenati con i precedenti.

A tal fine, si è deciso di creare una libreria all'interno della quale i poligoni vengono trattati come una collezione di vertici ordinati in senso antiorario e il segmento come una coppia di vertici (i suoi estremi). Prolungando il segmento dato in input, si cercano i punti di intersezione tra quest'ultimo e i singoli lati del poligono, i quali vengono poi utilizzati per creare i nuovi poligoni.

All'interno della libreria è, inoltre, presente la function `CreateMesh` la quale, prendendo in input un dominio rettangolare ed un elemento di riferimento, ricopre il rettangolo iterando l'elemento di riferimento.

1.1 Classi

In questo progetto è stata utilizzata la Programmazione orientata agli oggetti (OOP) e, per questo, sono state create le seguenti classi:

- **Point**: è caratterizzata da una coppia di attributi di tipo double (*_x* e *_y*) che rappresentano le coordinate cartesiane del punto (x,y) e dall'attributo* id* che identifica il punto tramite un numero intero;

sono presenti due diversi costruttori: il primo prende in input una coppia di double quali le coordinate del punto, mentre il secondo costruisce l'oggetto sfruttando la classe Vector2d della libreria Eigen. Inoltre, è stato ridefinito l'operatore di uguaglianza tra due punti (due istanze della stessa classe) attraverso un operatore di overload

- **Segment**: il costruttore di questa classe riempie i due attributi *primo_estremo* e *secondo_estremo* (vettori 2-dimensionali che rappresentano gli estremi del segmento) prendendo in input due Vector2d denotati con origin e end

Queste due classi sono state raggruppate all'interno del namespace Elementi_geometrici

- **Polygon**: gli oggetti di questa classe sono identificati da una collezione di vertici salvata nell'attributo *_vertici* di tipo vector< Vector2d> e dall'alias *_index_vert*, vettore della classe Point, che rappresenta l'indice dei vertici;

oltre al costruttore, che prende in input un vettore di punti (vector2d), sono presenti due metodi: la funzione `N_vertici` che ritorna l'intero dato dal numero dei vertici dell'oggetto poligono e la funzione `Area`, la quale, tramite la formula di Gauss, restituisce l'area della figura

- **Intersector1D1D**: calcola le intersezioni tra due oggetti geometrici 1-dimensionali: linee rette o segmenti. Le rette sono espresse attraverso l'equazione vettoriale r: x0+s*t, mentre i segmenti con r: x0+s*(x1-x0), con s ascissa curvilinea.

La classe riconosce il tipo di intersezione e la posizione dell'intersezione in ogni oggetto geometrico tramite due enumerazioni. L'enumerazione **Position** contiene: Begin, End, Inner, Outer e fornisce informazioni sulla posizione del punto di intersezione; nell'enumerazione **Type** sono presenti: IntersectionOnSegment, IntersectionOnLine, ParallelIntersectionOnSegment, ParallelIntersectionOnLine, NoIntersection e restituisce indicazioni riguardo il tipo di intersezione riferito alla prima retta o segmento.

Per riconoscere se la coordinata curvilinea equivale a 0 oppure a 1, viene introdotta una tolleranza di 1.0e-7 (attributo di tipo double *toleranceParallelism*); mentre per individuare se due segmenti o rette sono paralleli, si utilizza una tolleranza pari a 1.0e-7 (attributo double *toleranceIntersection*). Entrambe le tolleranze sono modificabili tramite i metodi `SetToleranceIntersection` e `SetToleranceParallelism`.

I due segmenti tra cui si calcola l'intersezione vengono configurati tramite le due funzioni `SetFirstSegment` e `SetSecondSegment`. Il metodo `ComputeIntersection` calcola l'intersezione tramite gli attributi *matrixTangentVector*, matrice 2-dimensionale, e *rightHandSide*, vettore 2d, definiti a partire dalle coordinate cartesiane degli estremi dei due segmenti in questione. Per richiamare gli attributi al di fuori della classe sono stati implementate diverse funzioni, tra le quali `PositionIntersectionInFirstEdge`, `PositionIntersectionInSecondEdge` e `TypeIntersection`

- **IntersectorPolySegment**: effettua il taglio tra un poligono e un segmento e restituisce gli n oggetti geometrici 2-dimensionali derivanti da esso.

A tale scopo, sono state create tre funzioni, ovvero `Compute`, `Listasottovertici` e `CutPoly`. Il primo metodo calcola le intersezioni tra un segmento e i vari lati del poligono; il secondo crea le figure derivanti dal taglio muovendosi lungo i lati del poligono; infine, `CutPoly` , sfruttando le prime due funzioni, gestisce l'output in modo che coincida con i requirements

All'interno della classe è, inoltre, presente l'enumerazione **Right_or_left** che individua la corretta numerazione dei vertici e dei punti di interesse calcolati

1.2 Analisi del codice

In questa sezione verranno trattate le funzioni fondamentali del progetto. Il metodo `Compute` della classe **IntersectorPolySegment** si occupa del calcolo delle intersezioni tra il segmento e i vari lati del poligono. In particolare, sfruttando il metodo `ComputeIntersection` della classe **Intersector1D1D** e scorrendo con un ciclo for lungo tutti i lati del poligono di partenza, si cerca se ci sono intersezioni tra il segmento e il lato (preso come secondo segmento) lungo il primo segmento o lungo il suo prolungamento. Nello speficico, usando `ComputeIntersection` lato per lato, si scorrono tutti i vertici del poligono di partenza e li si inserisce nell'attributo *vertici_complessivi* (vector< vector2d») finchè non viene trovato un punto di intersezione. A questo punto, viene memorizzata la sua coordinata parametrica (sfruttando la funzione `FirstParametricCoordinate`) nell'attributo *coordinatepara* (che conterrà poi le coordinate parametriche di tutti i punti di intersezione) e, a

partire dal suo valore, si divide in Sinistra e Destra in base a se l'intersezione si trovi dietro il primo estremo del segmento o meno. Il punto di intersezione (se non è già stato trovato) viene poi anch'esso salvato nell'attributo *vertici_complessivi* (in cui saranno contenuti i vertici di tutti i punti di interesse) e, inoltre, se questo non corrisponde al vertice successivo del poligono iniziale, si aggiunge in *vertici_complessivi* anche quest'ultimo. Si procede in questo modo per tutti i lati del poligono, ordinando di volta in volta in senso crescente *coordinatepara* e *ascisse_curv_con_estremi* il quale oltre a contenere le coordinate parametriche dei punti di intersezione, contiene anche quelle degli estremi del segmetno (ovvero 0 e 1). A questo punto vengono ordinati tutti i punti di intersezione sfruttando le coordinate parametriche precedentemente ordinate in senso crescente e vengono così salvati nell'attributo *puntiordinati*; vengono inoltre distinti in punti di ingresso e punti di uscita (i punti di intersezione, una volta ordinati, si alternano tra punti di ingresso e uscita). Procedimento analogo per l'attributo *point_compresi_esseg* il quale sfruttando *ascisse_curv_con_estremi*, ordina in base al valore delle coordinate parametriche i punti di intersezione e gli estremi del segmento. Si va, infine a riempire l'attributo *index_vert* (che già contiene i vertici del poligono iniziale) aggiungendogli questi punti ordinati e si ordinano i vertici totali con la giusta numerazione per poi salvare il risultato in *alias_vertici*.

L'altra funzione fondamentale è `Listasottovertici` la quale, dopo una prima distinzione tra caso Destra e caso Sinistra, crea i vari sottopoligoni derivati dal taglio. Nello specifico, scorrendo lungo tutti i vertici del poligono iniziale e i punti di intersezione, quindi attraverso l'attributo *vertici_complessivi*, si cerca di volta in volta se ci sono vertici che coincidono con un punto di intersezione; in caso contrario, vengono aggiunti all'attributo *sottovertici* della classe **IntersectorPolySegment** il quale conterrà i vertici della figura principale (ovvero l'ultima che viene a crearsi). Quando si incontra un punto di uscita, viene aggiunto anche questo ai vertici della figura principale, ovvero a *sottovertici* e, inoltre, da questo si capisce che si inizierà a creare un'altra figura e quindi viene aumentata la dimensione di *verticirestanti* che è un vettore che contiene i vertici degli altri poligoni (a parte quello principale). Poichè incontrare un punto di ingresso significa dover chiudere il nuovo sottopoligono che si sta creando, finchè non lo si incontra, si aggiungono di volta in volta i *vertici_complessivi* a *verticirestanti*, ovvero come vertici del nuovo sottopoligono. Una volta trovato un punto di ingresso (in particolare quello corrispondente al punto di uscita in esame), viene aggiunto anche questo a *verticirestanti* e, a questo punto, si controlla se gli estremi del segmento vanno considerati anch'essi tra i vertici del nuovo sottopoligono e cioè se sono compresi all'interno del poligono di partenza. Quando si incontra un punto di ingresso, anche questo va aggiunto ai vertici della figura principale, cioè a *sottovertici*, oltre che il punto di uscita. Infine si controlla se il punto di ingresso in esame è anche un punto in uscita (come ad esempio accade nel test aggiuntivo), perchè in quel caso vuol dire che ci sarà ancora un'altra sottofigura, che verrà creata a partire da quel vertice.

Infine, si ha la funzione `CutPoly` la quale, sfruttando i due metodi richiamati sopra, effettua effettivamente il taglio del poligono di partenza in tutte le nuove sottofigure. In particolare, qualora non ci fosse mai intersezione (ovvero non avvenisse nessun taglio) questa funzione ritorna il poligono di partenza; in caso contrario viene aggiunto a *listapolygon* la figura principale e poi, di volta in volta, scorrendo lungo *verticirestanti*, ci aggiunge anche tutti gli altri sottopoligoni dovuti al taglio. Infine, viene utilizzato l'attributo <vetctor< vector< int» *alias_pol* il quale, sfruttando *alias_vertici* che contiene i vertici complessivi numerati correttamente (da `Compute`), conterrà infine gli indici corretti dei vari sottopoligoni.

1.3 Unit Test

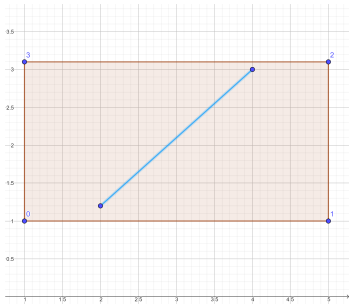
Ogni classe di questa libreria è stata testata tramite degli unit test: tutti i metodi presenti vengono verificati attraverso dei valori arbitrari.

In particolare, vengono qui riportati quelli richiesti dal cliente, ovvero quelli relativi alla classe **IntersectorPolySegment**

1. **Rettangolo**:

[2]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test1.png")
```

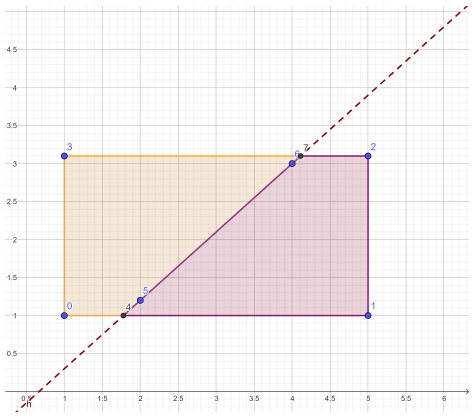
[2]:



```
Vector2d A(2, 1.2);
    Vector2d B(4,3);
    Segment AB(A,B);

    vector<Vector2d> Vertici{{1,1},{5,1},{5,3.1},{1,3.1}};
    Poligon P(Vertici);

    Intersector_segment_polygon Int(P,AB);

    vector<Vector2d> Vertici_1_{{1.0,1.0},{1.777777777777778,1},{2.0,1.2},{4.0,3.0},{4.1111111
    vector<Vector2d> Vertici_2_{{1.777777777777778,1.0},{5.0,1.0},{5.0,3.1},{4.111111111111111
    vector<Poligon> Poligoni_aspettati {{Poligon (Vertici_1_)},{Poligon (Vertici_2_)}};
    vector<vector<int>> Vertici_predetti {{0,4,5,6,7,3},{4,1,2,7,6,5}};

    EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));

    EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[3]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test1.1.png")
```

[3]:

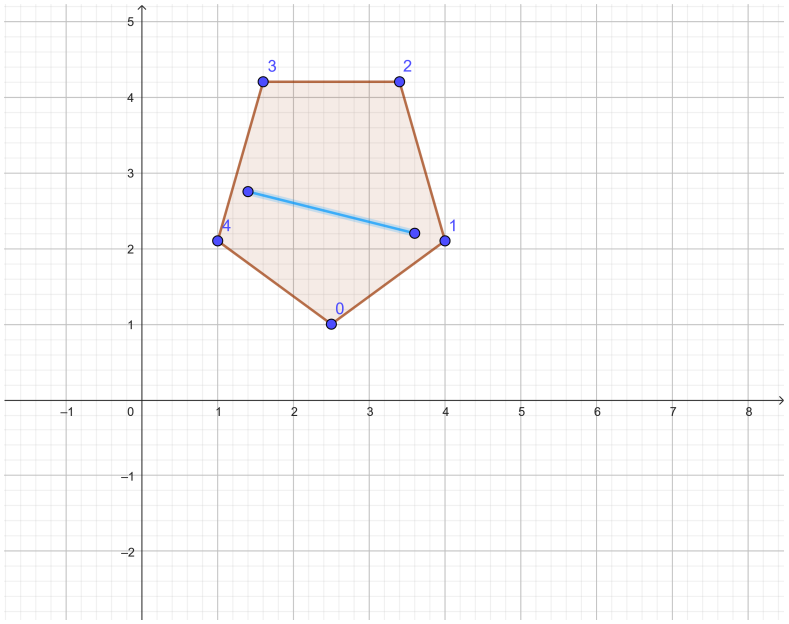


In questo primo esempio, il rettangolo, a seguito del taglio con il segmento AB è stato correttamente diviso in due nuovi poligoni

2. **Poligono convesso 1**:

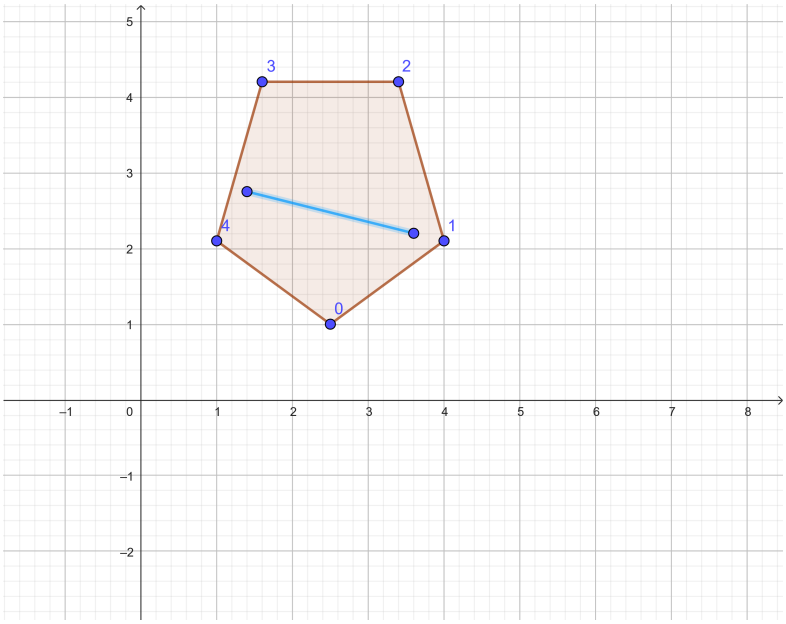
[4]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test2.png")
```

[4]:



[5]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test2.png")
```

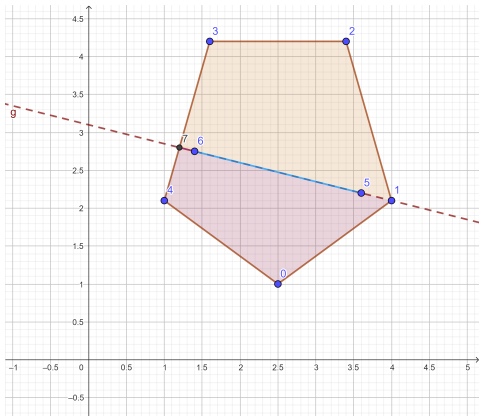
[5]:




```
Vector2d A(1.4,2.75);
    Vector2d B(3.6,2.2);
    Segment AB(A,B);

    vector<Vector2d> Vertici{{2.5,1.0},{4.0,2.1},{3.4,4.2},{1.6,4.2},{1.0,2.1}};
    Poligon P(Vertici);

    Intersector_segment_polygon Int(P,AB);

    vector<Vector2d> Vertici_1_{{2.5,1.0},{4.0,2.1},{3.6,2.2},{1.4,2.75},{1.2,2.8},{1.0,2.1}};
    vector<Vector2d> Vertici_2_{{4.0,2.1},{3.4,4.2},{1.6,4.2},{1.2,2.8},{1.4,2.75},{3.6,2.2}};
    vector<Poligon> Poligoni_aspettati {{Poligon (Vertici_1_)},{Poligon (Vertici_2_)}};
    vector<vector<int>> Vertici_predetti {{0,1,5,6,7,4},{1,2,3,7,6,5}};

    EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));

    EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[6]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test2.1.png")
```

[6]:

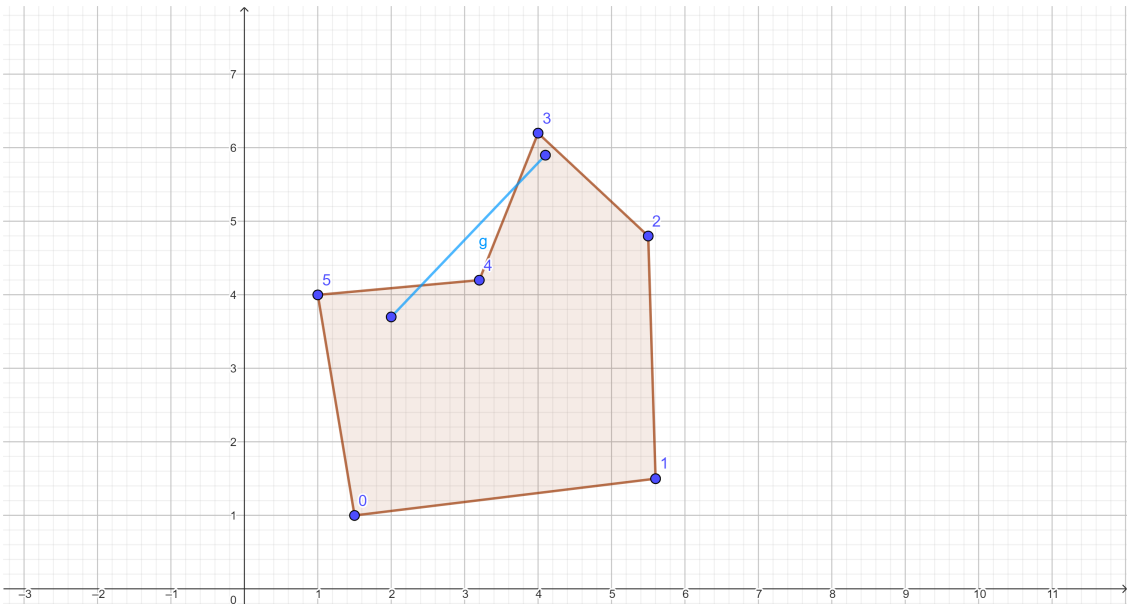


In questo esempio, il pentagono di partenza, a seguito del taglio con il segmento AB è stato diviso esattamente nei due poligoni predetti

3. **Poligono concavo 1**:

[7]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test3.png")
```

[7]:

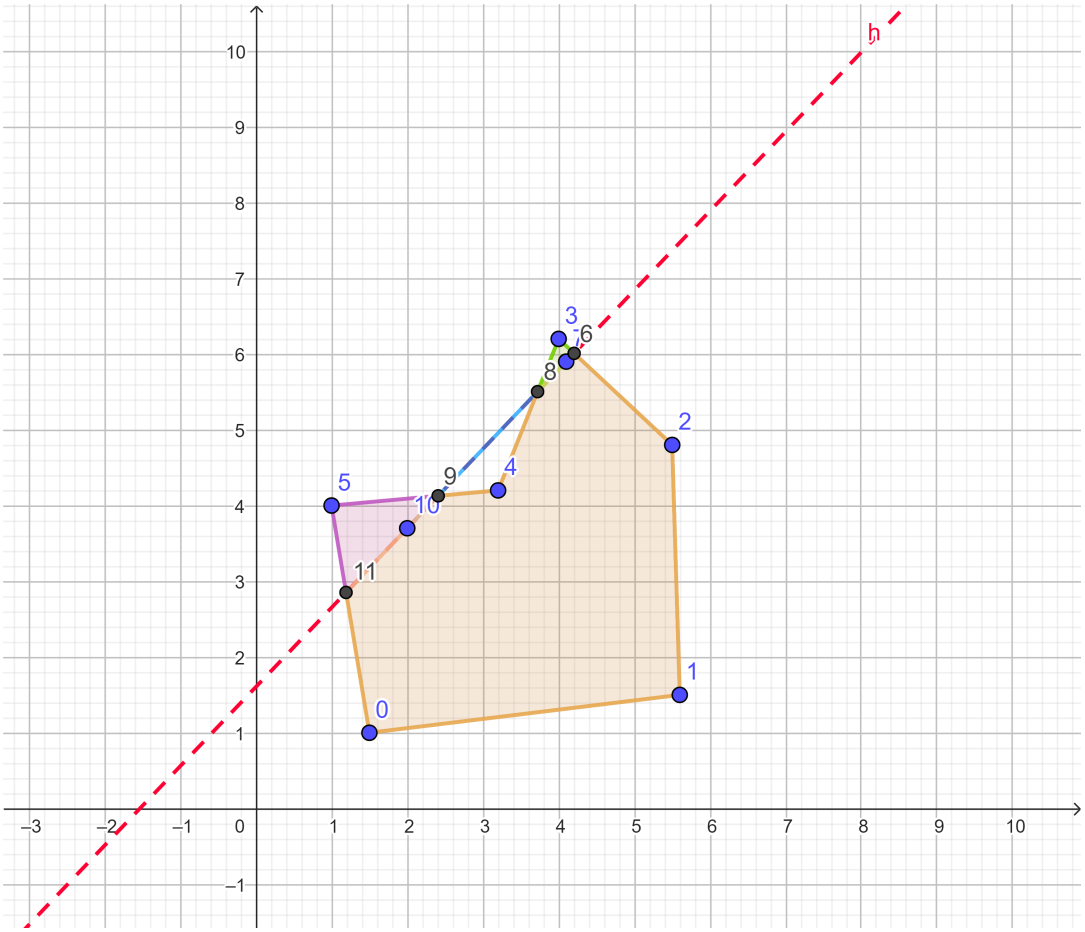



```
//creo segmento
Vector2d A(2, 3.7);
    Vector2d B(4.1, 5.9);
    Segment AB(A,B);

//creo poligono
```

```
    vector<Vector2d> Vertici{{1.5,1.0},{5.6,1.5},{5.5,4.8},{4.0, 6.2}, {3.2, 4.2}, {1.0, 4.0}}
    Poligon P(Vertici);

//creo oggetto della classe Intersector_segment_polygon
  Intersector_segment_polygon Int(P,AB);

//creo poligoni e vertici attesi
   vector<Vector2d> Vertici_1_{{1.5,1.0},{5.6,1.5},{5.5,4.8},{4.204326923076923,6.0092948717948

    vector<Vector2d> Vertici_2_{{4.204326923076923,6.009294871794872},{4,6.2},{3.7213114754098
    vector<Vector2d> Vertici_3_{{2.408597285067873,4.128054298642534},{1,4},{1.191216216216216
    vector<Poligon> Poligoni_aspettati {{Poligon (Vertici_1_)},{Poligon (Vertici_2_)},{Poligon
   vector<vector<int>> Vertici_predetti {{0, 1, 2, 6, 7, 8, 4, 9, 10, 11}, {6, 3, 8, 7}, {9, 5


//verifico la corrispondenza tra risultato atteso e risultato ottenuto
   EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));

   EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[8]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test3.1.png")
```

[8]:

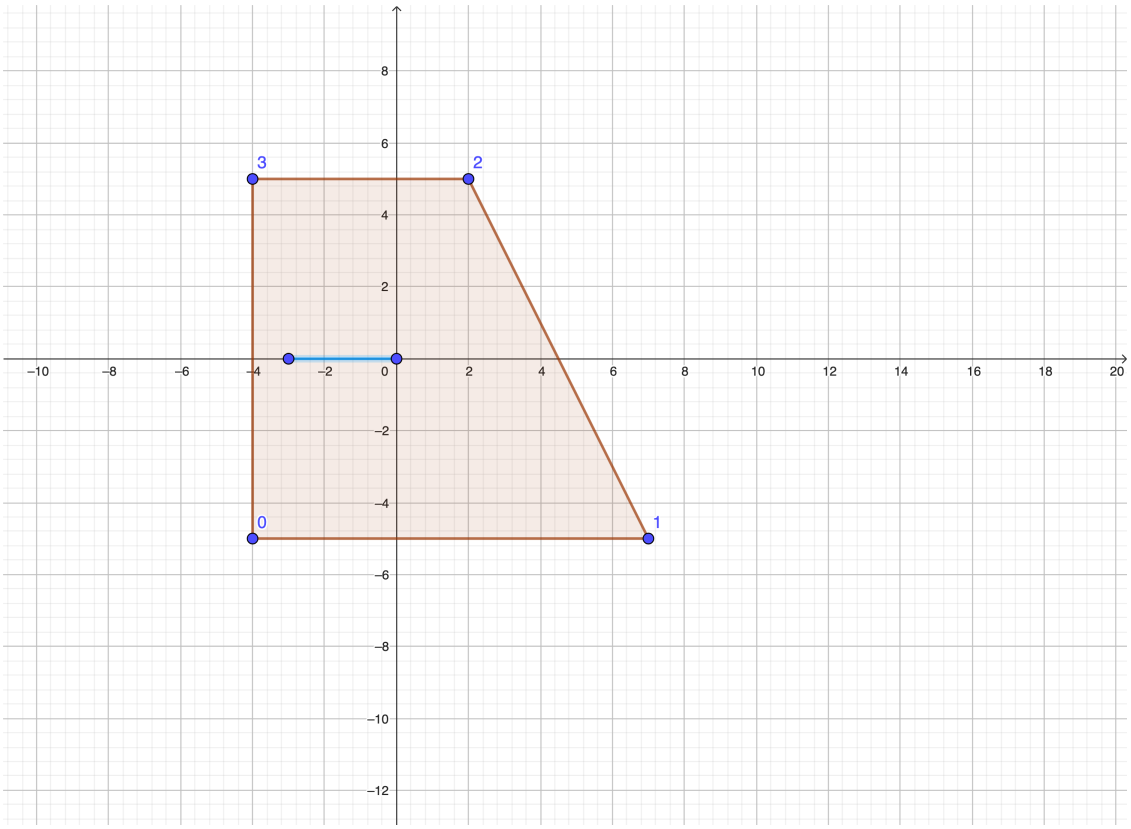


In questo esempio, il taglio del poligono ha prodotto tre nuovi poligoni, corrispondenti a quelli attesi

4. **Poligono convesso 2**:

[14]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test4.png")
```

[14]:

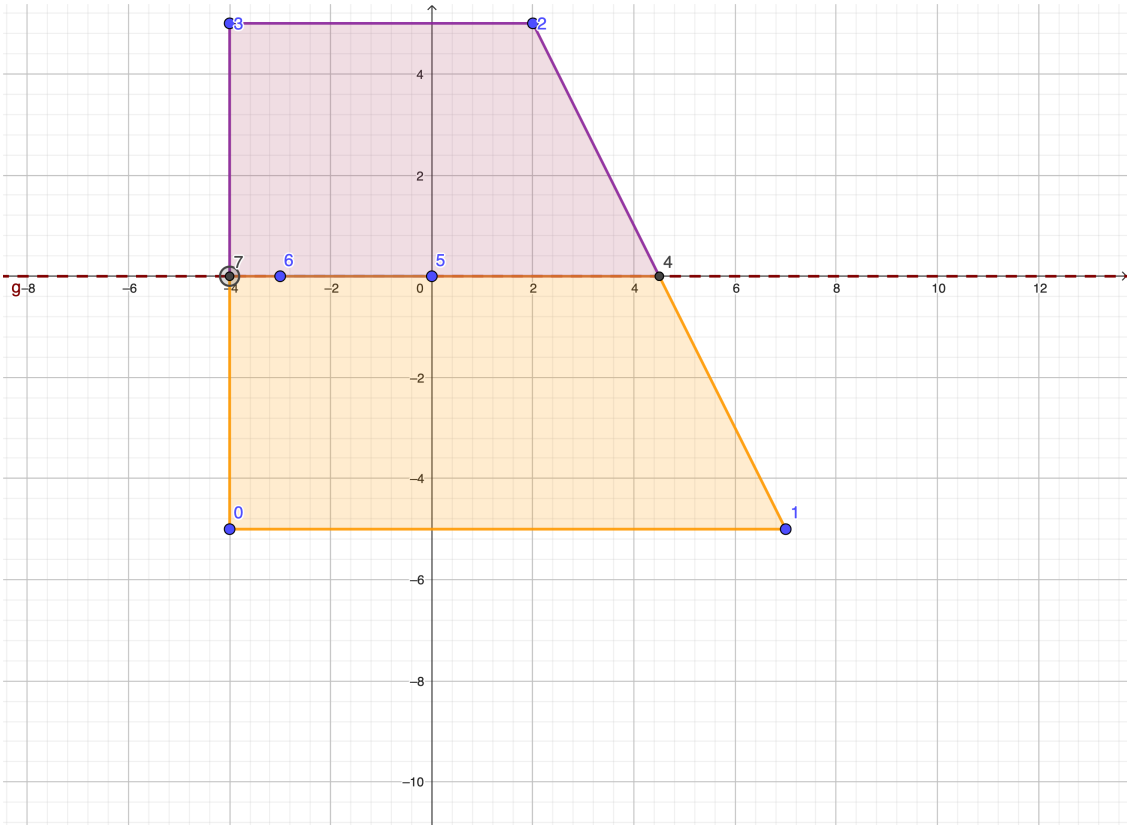



```
//creo segmento
Vector2d A(-3, 0);
    Vector2d B(0, 0);
    Segment AB(A,B);

//creo poligono
  vector<Vector2d> Vertici{{-4,-5},{7,-5},{2,5},{-4,5}};
    Poligon P(Vertici);

//creo oggetto della classe Intersector_segment_polygon
  Intersector_segment_polygon Int(P,AB);

//creo poligoni e vertici attesi
   vector<Vector2d> Vertici_1_{{-4,-5},{7,-5},{4.5,0},{0,0},{-3,0},{-4,0}};
   vector<Vector2d> Vertici_2_{{4.5,0.0},{2.0,5.0},{-4,5},{-4,0},{-3,0},{0,0}};

    vector<Poligon> Poligoni_aspettati {{Poligon (Vertici_1_)},{Poligon (Vertici_2_)}};
    vector<vector<int>> Vertici_predetti {{0, 1, 4, 5, 6, 7}, {4, 2, 3, 7,6,5}};

//verifico la corrispondenza tra risultato atteso e risultato ottenuto
   EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));
```

```
EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[9]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test4.1.png")
```

[9]:

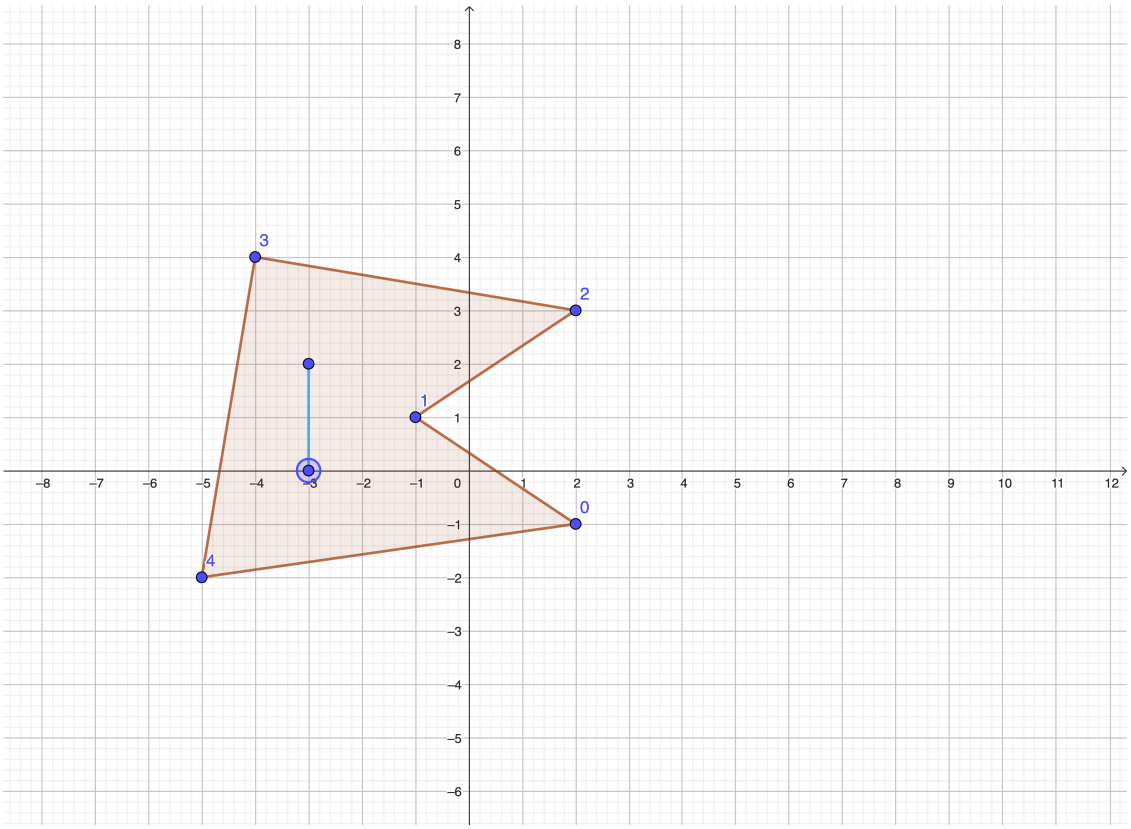


In questo esempio, per via del taglio con il segmento AB, il trapezio è viene diviso correttamente nei due trapezi rettangolo

5. **Poligono concavo 2**:

[10]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test5.png")
```

[10]:

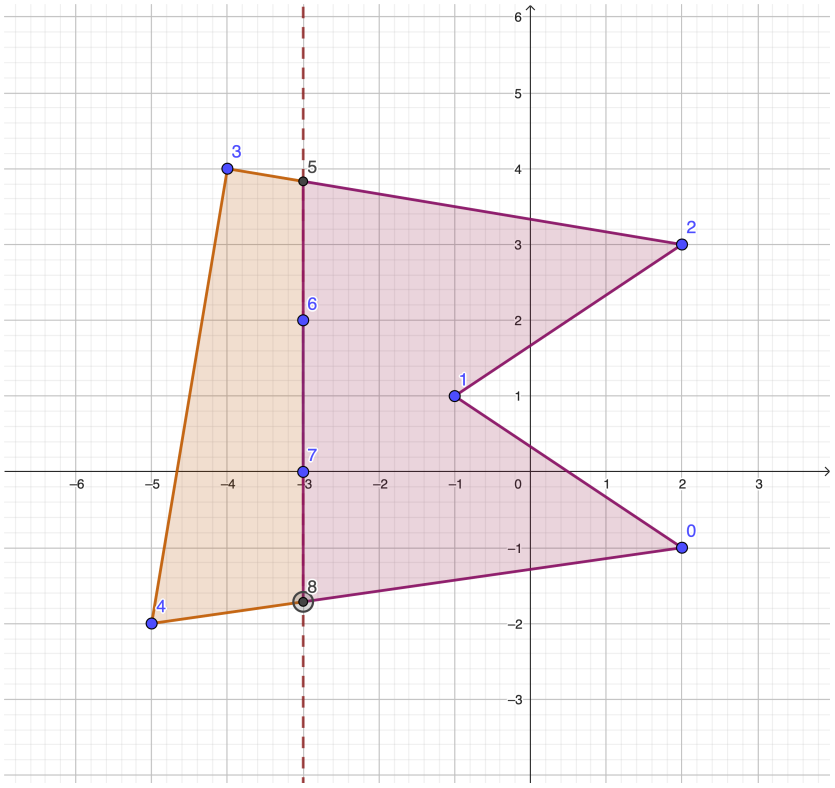



```
//creo segmento
Vector2d A(-3, 2);
    Vector2d B(-3, 0);
    Segment AB(A,B);

//creo poligono
  vector<Vector2d> Vertici{{2,-1},{-1,1},{2,3},{-4,4},{-5,-2}};
    Poligon P(Vertici);

//creo oggetto della classe Intersector_segment_polygon
  Intersector_segment_polygon Int(P,AB);

//creo poligoni e vertici attesi
    vector<Vector2d> Vertici_1_{{2,-1},{-1,1},{2,3},{-3,3.833333333333333},{-3,2},{-3,0},{-3,-1
  vector<Vector2d> Vertici_2_{{-3,3.833333333333333},{-4,4},{-5,-2},{-3,-1.7142857142857144},{-

    vector<Poligon> Poligoni_aspettati {{Poligon (Vertici_1_)},{Poligon (Vertici_2_)}};
    vector<vector<int>> Vertici_predetti {{0, 1, 2, 5, 6, 7, 8}, {5,3,4,8,7,6}};

//verifico la corrispondenza tra risultato atteso e risultato ottenuto
   EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));
```

```
EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[11]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\test5.1.png")
```

[11]:

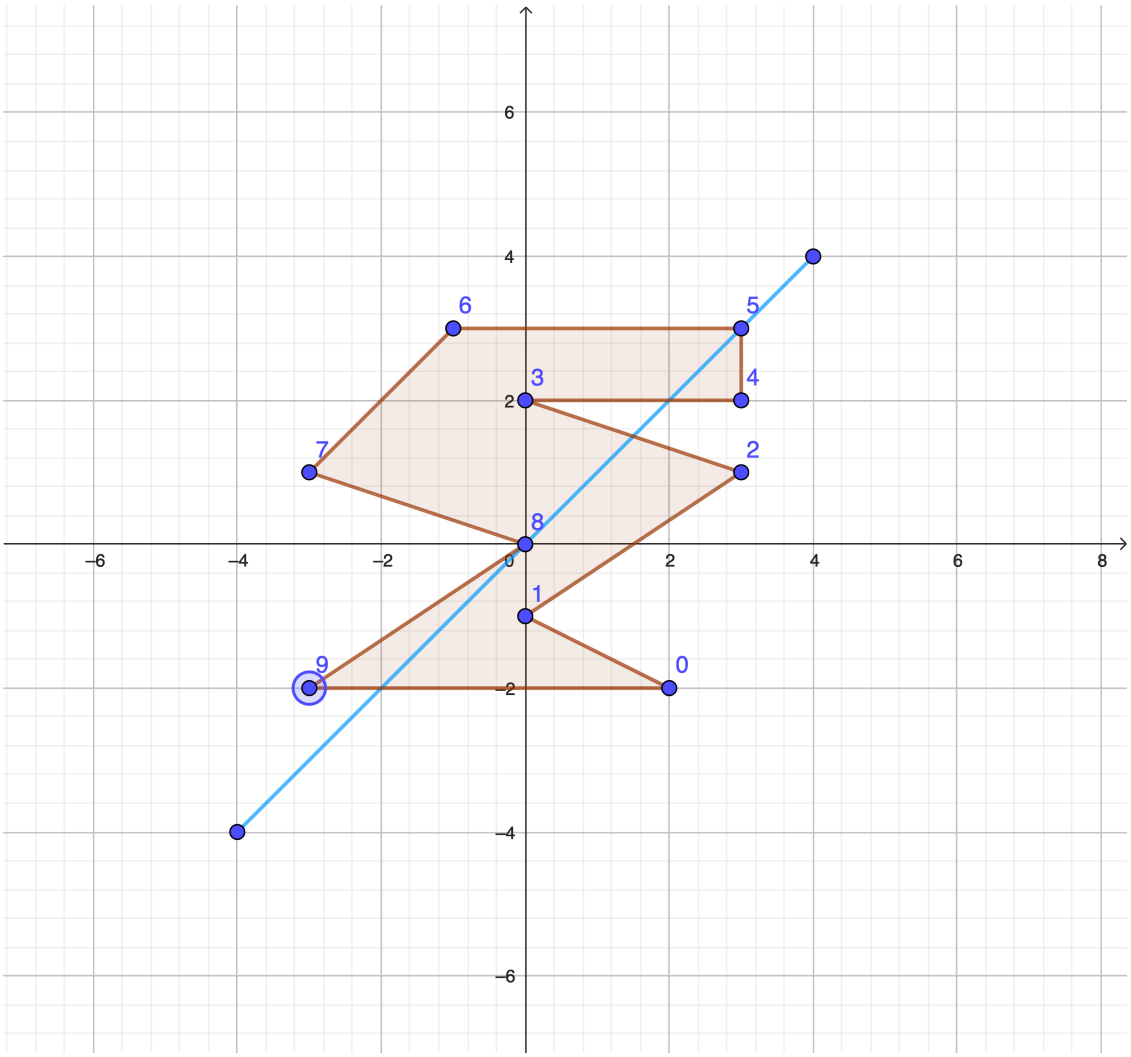


In questo quinto esempio, il taglio del poligono concavo con il segmento AB ha prodotto le figure attese

6. **Test aggiuntivo**:

[12]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\testaggiuntivo.
↪png")
```

[12]:

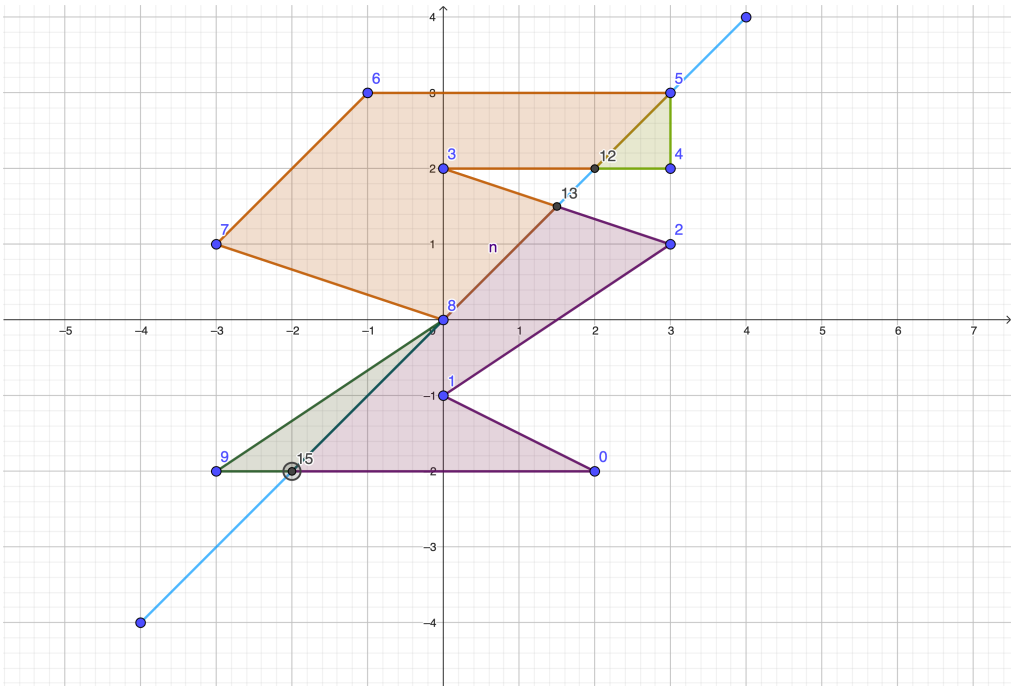



```
//creo segmento
Vector2d A(-4, -4);
    Vector2d B(4, 4);
    Segment AB(A,B);

//creo poligono
  vector<Vector2d> Vertici{{2,-2},{0,-1},{3,1},{0,2},{3,2},{3,3},{-1,3},{-3,1},{0,0},{-3,-2}};
    Poligon P(Vertici);

//creo oggetto della classe Intersector_segment_polygon
  Intersector_segment_polygon Int(P,AB);

//creo poligoni e vertici attesi
    vector<Vector2d> Vertici_1_{{2,-2},{0,-1},{3,1},{1.5,1.5},{0,0},{-2,-2}};
    vector<Vector2d> Vertici_2_{{1.5,1.5},{0,2},{2,2},{3,3},{-1,3},{-3,1},{0,0}};
```

```
    vector<Vector2d> Vertici_3_{{2,2},{3,2},{3,3}};
    vector<Vector2d> Vertici_4_{{0,0},{-3,-2},{-2,-2}};
    vector<Poligon> Poligoni_aspettati{ {Poligon (Vertici_1_)},{Poligon (Vertici_2_)},{Poligon
   vector<vector<int>> Vertici_predetti {{0, 1, 2, 13, 8, 15}, {13, 3, 12, 5,6,7,8}, {12, 4,5}


//verifico la corrispondenza tra risultato atteso e risultato ottenuto
   EXPECT_EQ(Vertici_predetti, Int.CutPoly(P,AB));

   EXPECT_EQ(Poligoni_aspettati, Int.listapolygon);
```

[13]:
```
display.Image("C:
↪\\Users\\aless\\Desktop\\Esercitazione\\mioambiente\\Images\\testaggiuntivo.
↪1.png")
```

[13]:



Anche in questo test aggiuntivo, il taglio viene eseguito correttamente