Prefazione (bozza)

L'obiettivo di questa trattazione è di illustrare l'implementazione di un insieme di classi OpenModelica con lo scopo di controllare un insieme di droni, per permettere ad esso di disporsi in modo equo su una data superficie.

In particolare, si vedrà l'implementazione dell'algoritmo di Lloyd (dopo aver fornito cenni teorici riguardo ad esso) ...

È prevista la realizzazione di un modulo nell'ambiente OpenModelica che permetta di simulare il suddetto algoritmo; più avanti verranno illustrate parti di codice e verranno fornite informazioni relative alla sua implementazione.

Indice

Capitolo 1

Il drone e le sue componenti

Possiamo schematizzare, ai fini di questa trattazione, un *drone* mediante tre componenti: il *centro fisico*, il *controllo di volo* e il *controllo della traiettoria*

Centro fisico Il centro fisico è un astrazione che rappresenta il rapporto del drone con il mondo esterno. Vengono quindi qui considerati parametri la massa, la velocità, l'accelerazione, la posizione, la rotazione, la velocità di rotazione dei motori e la potenza che viene fornita a questi.

Controllo di volo Il centro di volo considera il drone come un velivolo, ed utilizzando i dati forniti dal centro fisico e dal controllo della traiettoria permette il controllo della direzione, della quota e della velocità.

Controllo della traiettoria Componente centrale della trattazione, il controllo della traiettoria ha la funzione di comunicare al centro di volo la posizione da raggiungere.

Utilizza la funzione `TargetPos`, la cui ideazione ed implementazione viene illustrata in seguito.

1.1 Ipotesi e semplificazioni

La trattazione e l'implementazione richiedono che vengano effettuate alcune impotesi semplificative in relazione al mondo esterno nel quale l'algoritmo andrà ad operare. In particolare, si suppone che:

- l'area nella quale il drone opera sia un poligono convesso
- non siano presenti degli ostacoli all'iterno dell'area, in quanto questi non verrebbero considerati dal controllo di traiettoria portando il drone ad una collisione
- il mondo nel quale il drone opera sia bidimensionale, ovvero si suppone che non sia definita una componente z per le tuple rappresentanti la posizione. Nell'implementazione pratica, quest'ipotesi può essere gestita mediante l'utilizzo di un valore costante per la coordinata z (ad esempio ogni drone si trova a $650m$ sul livello del mare) oppure questa può essere determinata in funzione della distanza dal suolo.

Capitolo 2

L'algoritmo di Lloyd

L'algoritmo di Lloyd, conosciuto anche con il nome di *iterazione di Voronoi*, è un algoritmo che permette di suddividere un'area in celle convesse uniformemente dimensionate, denominata *tassellazione di Voronoi centroidale*, indicata con la sigla CVT[1].

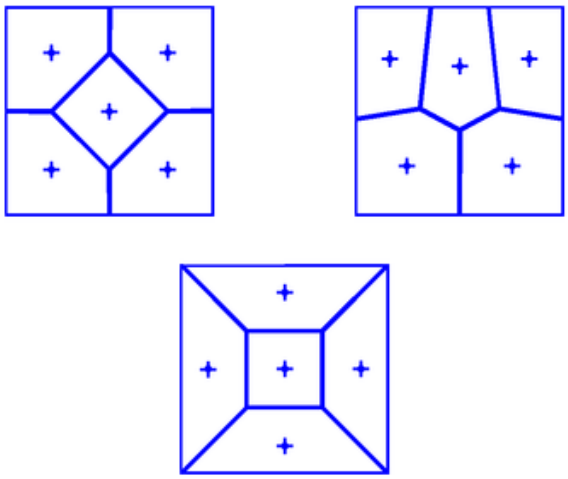


Figura 2.1: Tre esempi di CVT a 5 punti su un area quadrata

L'algoritmo, che prende il nome da Stuart P. Lloyd, il suo ideatore, è implementato mediante iterazioni continue dell'algoritmo di Voronoi.

In natura troviamo diversi esempi di CVT, tra cui il Selciato del gigante oppure le celle della cornea dell'occhio umano.

[1]Dall'inglese *Centroidal Voronoi Tesselation*

Figura 2.2: Scala del gigante, Irlanda del Nord

2.1 Descrizione

L'algoritmo esegue ripetutamente i seguenti *step*:

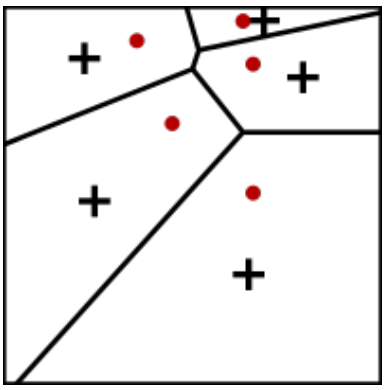
1. Viene generato il diagramma di Voronoi
2. Per ogni cella trovata, viene determinato il *baricentro*
3. Ogni punto viene spostato in corrispondenza del *baricentro* della propria *cella di Voronoi*

2.2 Esempio di applicazione dell'algoritmo

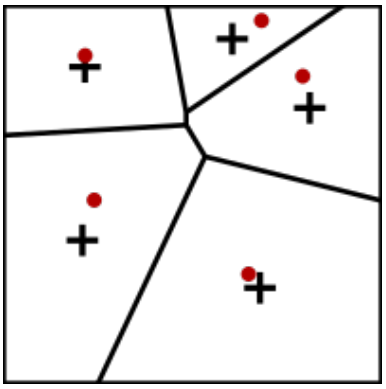
Viene qui presentata l'applicazione dell'algoritmo di Lloyd ad un'area quadrata nella quale sono presenti 5 partizioni.

Le croci rappresentano i *baricentri* delle varie partizioni; i punti rossi sono i droni, e le linee nere delimitano le varie celle dei singoli droni. Il quadrato che racchiude il tutto è l'area entro la quale i droni devono equidisporsi.

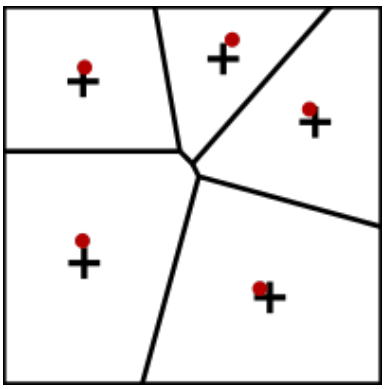
Iterazione 1 I droni sono mediamente distanti dal centro di massa della propria cella.



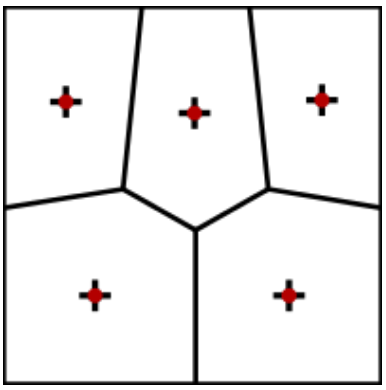
Iterazione 2 La distanza media tra i droni e il centro di massa delle rispettive celle è ridotta. Si noti come si è modificata la forma delle varie celle, in conseguenza del movimento dei vari droni.



Iterazione 3 La distanza media tra i droni ed il centro di massa è quasi nulla. L'algoritmo è prossimo alla convergenza.



Iterazione 4 Ogni drone si trova ora posizionato sopra il centro di massa della propria cella, di conseguenza l'algoritmo ha raggiunto la convergenza.



2.3 Convergenza dell'algoritmo

Intuitivamente, si può dire che l'algoritmo converga in quanto i punti che si trovano a minor distanza tra loro tendono a compiere un movimento più alto, mentre i punti che si trovano a distanze elevate tendono a muoversi meno.

2.4 Esecuzione dell'algoritmo

L'algoritmo, anziché essere eseguito centralmente, viene distribuito tra i centri di calcolo dei vari droni.

L'esecuzione, per ogni drone, diviene quindi:

1. Fino al segnale di *STOP*, ripeti:
   (a) Attendi le coordinate relative alla posizione degli altri droni, comunicate via radio
   (b) Ottieni dal *controllo di volo* le proprie coordinate
   (c) Esegui la funzione `TargetPos`, che restituisce le coordinate del punto da raggiungere (vedi la sezione 4.16 per i dettagli)
   (d) Comunica l'output al controllo di volo

Capitolo 3

Prototipazione

Al fine di realizzare l'algoritmo, si è reso necessario attraversare una prima fase di *prototipazione*.

Le specifiche del linguaggio di prototipazione erano le seguenti: un linguaggio dinamico che permettesse una concisa scrittura del codice e una rapida compilazione; strumenti di *testing* integrati per verificare l'effettiva correttezza delle funzioni implementate; la disponibilità di un framework grafico per effettuare simulazioni in modo rapido e dimanico.

La scelta è ricaduta quindi sul linguaggio *Python 3*[1] e sul framework *Pygame*[2]

3.1 Codifica

Il codice dell'implementazione *Python 3* non viene qui riportato, in quanto viene successivamente illustrato e commentato il porting dei prototipi nel linguaggio *OpenModelica*.

È comunque possibile prendere visione del codice di prototipazione presso il repository Github del progetto[3].

3.2 Testing

Al fine di implementare un meccanismo di testing rapido e completo si è optato per il framework *doctest*.

[1] https://www.python.org/
[2] https://www.pygame.org/
[3] https://github.com/zapateo/Tesi_GianlucaMondini/tree/master/implementazione_algoritmo/python

Esempio di testing tramite *doctest* Prendiamo, ad esempio, l'implementazione di una funzione triviale *somma* che, dati due numeri a e b, restituisca la loro somma.

La libreria *doctest* ci permette di definire, all'interno della documentazione della funzione, alcuni test che fungono inoltre da codice di esempio.

```
1  def somma(a, b):
2          """
3          Restituisce la somma di a e b
4
5          >>> somma(4, 5)
6          9
7
8          >>> somma(0, -3)
9          -3
10         """
11         return a + b
```

È adesso necessario richiamare all'interno *main* del codice sorgente l'esecuzione dei vari test, tramite

```
1  if __name__ == "__main__":
2      import doctest
3      doctest.testmod()
```

Eseguendo quindi il codice sorgente, verrano eseguiti i test presenti nella documentazione delle varie funzioni e l'output verrà confrontato con quello atteso. Nel caso in cui tutti i test vengano superati non verrà mostrato alcun output a schermo, in caso contrario un messaggio d'errore indicherà il test non superato, l'output atteso e l'output ottenuto.

Capitolo 4

Implementazione in *OpenModelica*

L'implementazione dell'algoritmo di Lloyd richiede una funzione che effettui la tassellazione di Voronoi, la quale a sua volta utilizza una serie di funzioni geometriche che operino nello spazio bidimensionale.

Nell'implementazione che verrà illustrata successivamente, si è scelto di implementare in primo luogo l'algoritmo utilizzando il linguaggio *Python* per avere a disposizione una maggior quantità di strumenti di debugging e testing; successivamente è stato effettuato il porting del codice in *OpenModelica*

4.1 OpenModelica

OpenModelica è un implementazione *open source* del linguaggio di modellazione *Modelica*, che permette la modellazione, simulazione, ottimizzazione e analisi di complessi sistemi dinamici. *OpenModelica* viene utilizzato sia in ambito accademico sia in ambito industriale.

L'ambiente *OpenModelica*, implementato nei linguaggi C e C++, fornisce una serie di strumenti di lavoro, tra cui:

omc compila un file sorgente *OpenModelica* generando la rispettiva *classe*

OMEdit software che permette, tra le varie cose, di "connettere" tramite un'interfaccia grafica le varie classi per comporre sistemi complessi

OMShell shell interattiva che esegue comandi impartiti singolarmente, utile per effettuare una rapida prototipazione

Per ulteriori informazioni si rimanda al sito ufficiale del progetto[1] e al documento di specifica Modelica[2]

4.2 Un approccio funzionale

Nell'implementazione del codice si è optato per un approccio funzionale. In pratica ogni "modulo" viene implementato tramite una funzione che abbia un determinato dominio di input e codominio di output e viene codificata in modo tale da evitare *effetti collaterali* al di fuori di se stessa. Questo garantisce che, data una funzione e un insieme di input, si possa determinare univocamente l'output, indipendentemente dal valore di parametri esterni alla funzione. Il tutto può essere chiarito con un esempio:

```
1 A = 0
2 B = 20
3
4 func F(C, D) {
5         A = A + 10
6         return C + D + A
7 }
8
9 output = F(30, 40)
```

Figura 4.1: Un implementazione non-funzionale di `F`

La funzione `F` restituisce la somma di `C`, `D` ed `A`. In questo caso il valore `output` sarà $30 + 40 + 10 = 80$.

La funzione presenta però un problema: eseguendo nuovamente la funzione con gli stessi argomenti, l'output sarà $30 + 40 + 20 = 90$ in quanto il valore di `A` è variato, ovvero ha subito un *effetto collaterale* dalla chiamata della funzione.

Inoltre, variando `A` al di fuori della funzione il suo output cambierà di conseguenza

[1] https://openmodelica.org/

[2] https://www.modelica.org/documents/ModelicaSpec34.pdf

```
A = 0
B = 20

func F(C, D) {
        A = A + 10
        return C + D + A
}

output = F(30, 40)

A = 100

output = F(30, 40)
```

Figura 4.2: La stessa funzione `F`, chiamata con gli stessi argomenti in momenti diversi, restituisce output diversi

Non è quindi possibile stabilire, a priori, l'output della funzione dati gli argomenti se non si conosce anche, per intero, l'ambiente nel quale la funzione viene eseguita.

L'approccio funzionale prevede quindi di implementare la funzione `F` nel seguente modo:

```
A = 0
B = 20

func F(C, D, A) {
        A = A + 10
        return C + D + A, A
}

output, A = F(30, 40)
```

Figura 4.3: Un implementazione funzionale della funzione `F`

In questo caso la funzione non accetta più 2 argomenti, bensì 3, come era implicitamente codificato nella versione precedente. Si noti inoltre che la funzione non restituisce soltanto un valore di output ma 2, il risultato della somma (assegnato qui a `output`) e il nuovo valore di `A`, che ha subìto un incremento.

Questo nuovo modello semplifica notevolemente lo sviluppo, in quanto la funzione `F` è ora svincolata dall'ambiente che la circonda ed è liberamente eseguibile all'interno di altro codice.

Un altro aspetto da considerare è quello del *testing*; come si vedrà più avanti nella sezione 4.17, è indispensabile realizzare un meccanismo di testing che garantisca l'effettiva correttezza delle funzioni implementate. A questo riguardo l'approccio funzionale scelto semplifica notevolemente questa fase

di sviluppo, in quanto è sufficiente effettuare chiamate singole alle varie funzioni per ottenerne l'output da confrontare con un valore di riferimento.

Nel nostro esempio la funzione `F` come mostrata nella figura 4.2 chiamata con argomenti costanti restituirà sempre lo stesso valore di output.

4.3 Diagramma delle dipendenze

Viene qui illustrato un diagramma contenenti le varie funzioni dell'implementazione *OpenModelica*. Le frecce $\longrightarrow$ rappresentano una dipendenza; ad esempio $A \longrightarrow B$ indica che A effettua una chiamata a B durante la sua esecuzione, e di conseguenza A *dipende da* B.

Sono state omesse le funzioni di supporto al testing e al debugging in quanto non strettamente legate all'implementazione di per sè.

In alto è presente la funzione `TargetPos`, la quale dipende da `CenterOfMass`, `VoronoiCell` e `EdgesToVertices`.

In basso troviamo `CompareReal`, che dipende esclusivamente da funzioni integrate in *OpenModelica*.

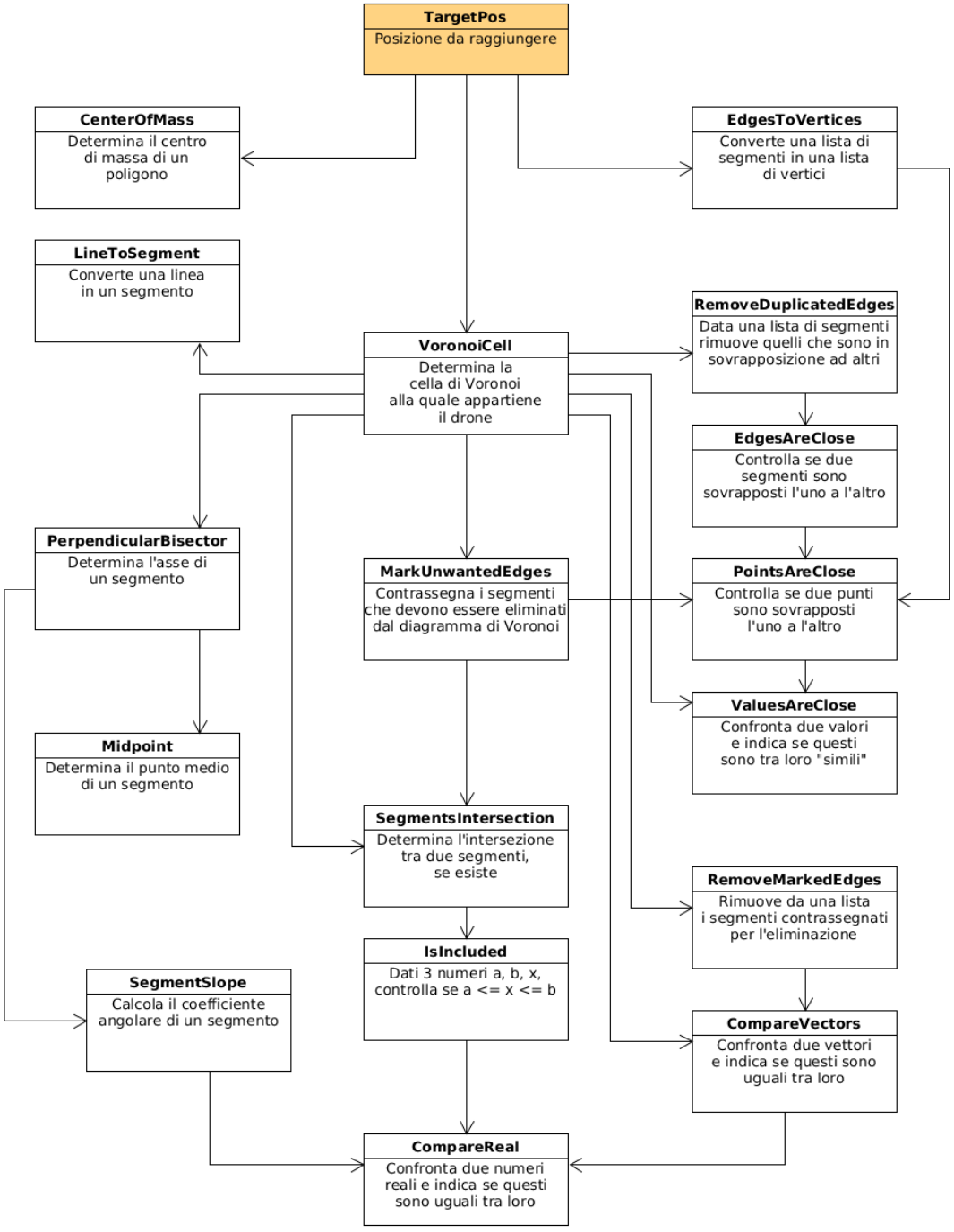


Figura 4.4: Diagramma delle dipendenze delle varie funzioni

4.4 Scelte implementative

Come si vedrà in seguito, anzichè definire *tipi* personalizzati si è deciso di ricorrere alle strutture dati già presenti in *OpenModelica*.

Ad esempio, un punto dello spazio può essere rappresentato sia come *record* tramite

```
record Point
        Real x, y;
end Point;
```

sia come vettore

```
Real [2] point;
```

La scelta è ricade su quest'ultima implementazione in quanto è possibile utilizzare una serie di funzioni primitive (principalmente per lavorare su liste) già definite in *OpenModelica* che avrebbero richiesto altrimenti richiesto l'implementazione manuale tramite un linguaggio di più basso livello.

I punti a sfavore dell'utilizzo di un vettore al posto di una struttura dati personalizzata sono:

- L'impossibilità di accedere ai membri tramite il loro nome bensì tramite l'indice legato alla loro posizione. Ad esempio, per accedere alla coordinata y di un punto non è possibile utilizzare `point.y` ma `point[2]`, perdendo quindi di leggibilità
- L'impossibilità di aggiungere ulteriori campi contenenti informazioni. A questo riguardo si veda la sezione 4.14

4.5 Rappresentazione del punto geometrico

Per poter indicare un punto nello spazio bidimensionale, è necessario definire una struttura dati rappresentante una tupla x, y. A questo fine viene utilizzato un vettore bidimensionale definito tramite

```
Real [2] point;
```

Nel caso in cui debba essere dichiarato un vettore di punti, si dichiara una matrice tramite

```
Real [:, 2] some_points;
```

Il carattere `:` posto in prima posizione tra le parentesi quadre indica che la prima dimensione della matrice `some_points` non è conosciuta a priori.

4.6 Verifica della sovrapposizione di due punti

Può verificarsi che, in seguito a ..., ci si trovi nella situazione in cui due punti sovrapposti non superino il test di equalità. Ad esempio:

$$P_1 = (4.0, 2.0)$$

$$P_2 = (4.00001, 1.999999)$$

I punti P_1 e P_2, pur potendo essere considerati sovrapposti in relazione allo spazio metrico in cui si trovano, sono considerati diversi da loro per via di errori di approssimazione del calcolatore.

Si è quindi implementata una funzione che discrimini i punti diversi tra loro da quelli "apparentemente diversi", ovvero una funzione che verifichi se due punti sono tra loro *vicini*.

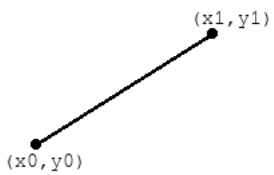
```
1  function PointsAreClose
2      input Real [2] p1;
3      input Real [2] p2;
4      output Boolean are_close;
5  algorithm
6      if ValuesAreClose(p1[1], p2[1]) then
7          if ValuesAreClose(p1[2], p2[2]) then
8              are_close := true;
9          else
10             are_close := false;
11         end if;
12     else
13         are_close := false;
14     end if;
15 end PointsAreClose;
```

4.7 Rappresentazione del segmento geometrico

Per rappresentare un segmento, si considera un vettore di 4 elementi, nella forma

$$(x_0, y_0, x_1, y_1)$$

dove (x_0, y_0) è il punto iniziale del segmento e (x_1, y_1) è il punto finale.



Questo si traduce, in ambiente *OpenModelica*, in:

```
Real [4] edge;

edge[1] := x_1;
edge[2] := y_1;
edge[3] := x_2;
edge[4] := y_2;
```

4.8 Verifica della sovrapposizione di due segmenti

La verifica della sovrapposizione di due segmenti viene effettuata in modo analogo a quella della sovrapposizione di due punti, e fa uso proprio di quest'ultima funzione.

Formalmente, due segmenti sono considerati sovrapposti se il punto iniziale del primo si sovrappone al punto iniziale del secondo e contemporaneamente il punto finale del primo si sovrappone al punto finale del secondo.

```
function EdgesAreClose

    input Real [4] e1, e2;

    output Boolean are_close;

algorithm

    are_close :=
        (PointsAreClose({e1[1], e1[2]}, {e2[1], e2[2]}) and
            PointsAreClose({e1[3], e1[4]}, {e2[3], e2[4]}))
            or
        (PointsAreClose({e1[1], e1[2]}, {e2[3], e2[4]}) and
            PointsAreClose({e1[3], e1[4]}, {e2[1], e2[2]}));

end EdgesAreClose;
```

4.9 Linea

Una *linea* è matematicamente identificata da una tupla di 3 elementi a, b e c tali che

$$ax + by + c = 0$$

In *OpenModelica*, si rappresenterà quindi come un vettore di 3 elementi

```
Real [3] line;

line[1] := a;
```

```
4  line[2] := b;
5  line[3] := c;
```

4.10 Conversione da linea a segmento

Può essere necessario convertire una linea, priva di un punto di inizio e fine, in un segmento, ben delimitato. Per fare ciò è stata implementata la funzione

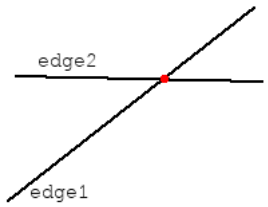
$$\text{LineToSegment} : R[3] \longrightarrow R[4]$$

```
1  function LineToSegment
2      input Real [3] line;
3      output Real [4] segment;
4  protected
5      parameter Real big = 10000;
6      Real a, b, c;
7      Real [2] p1, p2;
8      Real m, q;
9      Real y1, y2;
10 algorithm
11     a := line[1];
12     b := line[2];
13     c := line[3];
14
15     if b == 0 and (not (a == 0)) then // Retta verticale
16         p1 := {-c, big};
17         p2 := {-c, -big};
18         segment := {p1[1], p1[2], p2[1], p2[2]};
19         return;
20     else
21         m := -a/b;
22         q := -c/b;
23         y1 := m * (-big) + q;
24         y2 := m * (+big) + q;
25         segment := {-big, y1, big, y2};
26     end if;
27 end LineToSegment;
```

4.11 Intersezione tra segmenti

Una delle funzioni principalmente utilizzate dall'algoritmo di tassellazione di Voronoi è quella responsabile di determinare l'eventuale punto di intersezione tra due segmenti.

È necessario notare che, avendo a che fare con segmenti di lunghezza finita e non con delle rette, è possibile che due segmenti non si intersechino tra loro pur non essendo paralleli.



L'algoritmo implementato è il seguente [3]

```
function SegmentsIntersection
    input Real [4] edge1, edge2;
    output Boolean valid;
    output Real [2] intersection;
protected
    Real x1, x2, y1, y2, dx1, dy1, x, y, xB, yB, dx, dy, DET,
         DETinv, r, s, xi, yi;
    parameter Real DET_TOLERANCE = 0.00000001;
algorithm
    x1 := edge1[1];
    y1 := edge1[2];
    x2 := edge1[3];
    y2 := edge1[4];

    x := edge2[1];
    y := edge2[2];
    xB := edge2[3];
    yB := edge2[4];

    dx1 := x2 - x1;
    dy1 := y2 - y1;

    dx := xB - x;
    dy := yB - y;

    DET := ((-dx1 * dy) + (dy1 * dx));

    if abs(DET) < DET_TOLERANCE then
        valid := false;
        return;
    end if;

    DETinv := 1.0/DET;

    r := DETinv * (-dy  * (x-x1) +  dx * (y-y1));
    s := DETinv * (-dy1 * (x-x1) + dx1 * (y-y1));
    xi := (x1 + r*dx1 + x + s*dx)/2.0;
    yi := (y1 + r*dy1 + y + s*dy)/2.0;

    if (IsIncluded(0, 1, r)) and (IsIncluded(0, 1, s)) then
```

[3] Il codice è un adattamento di https://www.cs.hmc.edu/ACM/lectures/intersections.html

```
40          intersection := {xi, yi};
41          valid := true;
42          return;
43      else
44          valid := false;
45          return;
46      end if;
47  end SegmentsIntersection;
```

Oltre a restituire un vettore bidimensionale `intersection` viene anche restituito un booleano `valid`, che sta ad indicare se il valore contenuto in `intersection` è valido o meno.

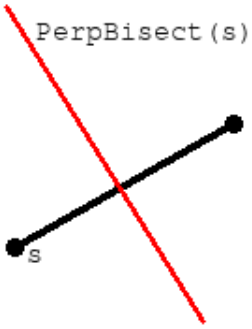
4.12 Asse di un segmento

La funzione `VoronoiCell` fa ampio uso della funzione `PerpendicularBisector`, che restituisce l'asse del segmento passato come argomento.

$$\text{PerpendicularBisector} : \text{Segmento} \longrightarrow \text{Linea}$$

Vengono discriminati i 3 casi in cui:

- Il segmento sia verticale; in tal caso l'asse sarà orizzontale;
- Il segmento sia orizzontale; in tal caso l'asse sarà verticale;
- Il segmento non sia nè verticale nè orizzontale: caso più frequente




```
1  function PerpendicularBisector
2      input Real [4] edge;
3      output Real [3] perp_bisect;
4  protected
5      Boolean vertical;
6      Real [2] p;
7      Real a, b, c, neg_c, m1, m2, q;
```

```
algorithm
    p := Midpoint(edge);
    (m1, vertical) := SegmentSlope(edge);
    if vertical then
        neg_c := (edge[2] + edge[4])/2;
        perp_bisect := {0, 1, -neg_c};
        return;
    elseif m1 == 0 then
        a := 1;
        b := 0;
        c := -(edge[1] + edge[3])/2;
        perp_bisect := {a, b, c};
        return;
    else
        m2 := -1/m1;
        q := - m2 * p[1] + p[2];
        perp_bisect := {-m2, 1, -q};
        return;
    end if;
end PerpendicularBisector;
```

4.13 Centro di massa

Al fine di implementare l'algoritmo di Lloyd è necessario definire una funzione che, presa in ingresso una lista di vertici, ne calcoli il centro di massa.

Dato un poligono di n vertici $(x_0, y_0), (x_1, y_1), ..(x_{n-1}, y_{n-1})$ il centro di massa ha coordinate (C_X, C_Y) definite come

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

dove A è l'area del poligono *con segno*

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

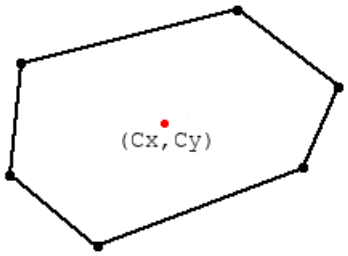


Figura 4.5: Il punto p viene restituito dalla chiamata a `CenterOfMass` che ha come argomento la lista delle coordinate dei vari vertici del poligono

Ne deriva quindi la seguente codifica:

```
function CenterOfMass

    input PointsArray points;

    output Real [2] out;

protected

    PointsArray vertices, v_local;
    Real [0, 2] empty_vertices;
    Real x_cent, y_cent, area, factor;

algorithm

    if points.len < 3 then
        out := {-1,-1};
    end if;

    // Adattato da https://stackoverflow.com/a/46937541

    vertices.len := 0;
    for i in 1:points.len loop
        vertices := PointsArrayAppend(vertices,
            points.elements[i]);
    end for;

    x_cent := 0;
    y_cent := 0;
    area := 0;

    v_local.len := 0;
    for i in 1:vertices.len loop
        v_local := PointsArrayAppend(v_local,
            vertices.elements[i]);
    end for;
    v_local := PointsArrayAppend(v_local, vertices.elements
        [1]);
```

```
35
36      for i in 1:(v_local.len - 1) loop
37          factor := v_local.elements[i,1] * v_local.elements[i
                +1, 2] - v_local.elements[i+1,1] *
                v_local.elements[i,2];
38          area := area + factor;
39          x_cent := x_cent + (v_local.elements[i, 1] +
                v_local.elements[i+1,1]) * factor;
40          y_cent := y_cent + (v_local.elements[i, 2] +
                v_local.elements[i+1, 2]) * factor;
41      end for;
42
43      area := area / 2.0;
44      x_cent := x_cent / (area * 6);
45      y_cent := y_cent / (area * 6);
46
47      out := {x_cent, y_cent};
48
49  end CenterOfMass;
```

4.14 Ricerca ed eliminazione dei segmenti in eccesso

Durante l'esecuzione dell'algoritmo di Voronoi si presenta la necessità di rimuovere dei segmenti che non fanno più parte del diagramma. Si presenta il problema di come effettuare questa operazione senza arrecare danno all'iterazione in corso proprio sulla lista dalla quale vanno rimossi gli elementi.

Inoltre, come precedentemente detto, non è possibile aggiungere un campo ad ogni segmento indicando l'effettiva necessità di eliminazione.

La scelta cade quindi sull'assegnare il valore -1 ad ogni componente del segmento, considerandolo quindi un segmento nullo, da eliminare.

```
1  edge := {-1, -1, -1, -1};
```

Successivamente, utilizzando la funzione `CompareVector` si controlla se il segmento è contrassegnato per l'eliminazione

```
1  if CompareVector(edge, {-1, -1, -1, -1}) then
2          // Considera 'edge' come un segmento da eliminare
3  else
4          // Considera 'edge' come un segmento valido
5  end if;
```

La funzione `MarkUnwantedEdges` contrassegna tutti i segmenti che devono essere rimossi dalla lista di segmenti. In particolare, se un segmento si

trova dietro ad un altro quello "nascosto" deve essere eliminato in quanto non può più far parte dell'insieme.

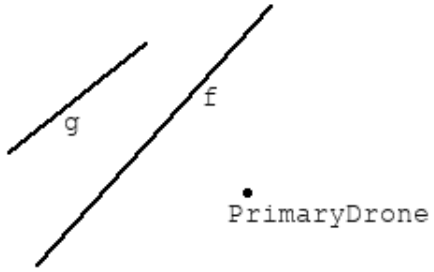


Figura 4.6: Esempio nel quale il segmento g deve essere rimosso, in quanto il segmento f si pone tra esso e il *PrimaryDrone*. La funzione `MarkUnwantedEdges` contrassegnerà g per l'eliminazione, mentre `RemoveMarkedEdges` lo cancellerà definitivamente.

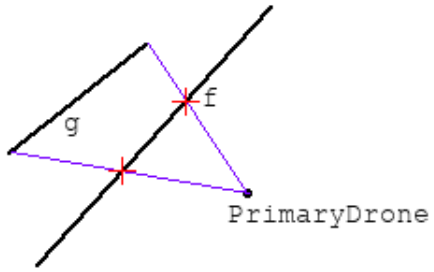


Figura 4.7: I segmenti di unione tra `PrimaryDrone` e gli estremi di g collidono con il segmento f; questo fa sì che g venga contrassegnato per l'eliminazione.

```
1  function MarkUnwantedEdges
2
3      input  EdgesArray edges;
4      input  Real [2]   primary_drone;
5      output EdgesArray marked_edges;
6
7  protected
8
9      Real [2] point, intersection;
10     Real [4] inner_edge, join_edge;
11     Boolean valid;
```

```
12
13 algorithm
14
15     marked_edges.len := 0;
16     for i in 1:edges.len loop
17         marked_edges := EdgesArrayAppend(marked_edges,
               edges.elements[i]);
18     end for;
19
20     // Scorro tra tutti i bordi
21     for outer_i in 1:edges.len loop
22
23         if not CompareVectors(edges.elements[outer_i],
               {-101010, -101010, -101010, -101010}) then
24             // Per ogni bordo, seleziono prima un estremo e
                   poi l'altro
25             for point_index in 0:1 loop
26                 if point_index == 1 then
27                     point := {edges.elements[outer_i,1],
                           edges.elements[outer_i,2]};
28                 else // point_index == 2
29                     point := {edges.elements[outer_i,3],
                           edges.elements[outer_i,4]};
30                 end if;
31
32                 // Scorro tutti i bordi e vedo se la
                       congiunzione
33                 // tra l'estremo trovato e il primary_drone
34                 // crea una collisione, in tal caso
                       contrassegno il bordo esterno
35                 // per l'eliminazione
36                 for inner_edge_index in 1:edges.len loop
37
38                     inner_edge := edges.elements[
                           inner_edge_index];
39                     join_edge := {primary_drone[1],
                           primary_drone[2], point[1], point[2]};
40
41                     (valid, intersection) :=
                           SegmentsIntersection(inner_edge,
                           join_edge);
42
43                     if valid and (not PointsAreClose(
                           intersection, point)) then
44                         marked_edges.elements[outer_i] :=
                               {-1, -1, -1, -1};
45                     end if;
46                 end for;
47             end for;
48         end if;
49     end for;
50
51 end MarkUnwantedEdges;
```

La funzione `RemoveMarkedEdges` restituisce una lista di segmenti dalla quale sono stati rimossi tutti i segmenti contrassegnati per l'eliminazione.

```
1  function RemoveMarkedEdges
2
3      input EdgesArray edges;
4      output EdgesArray clean_edges;
5
6  algorithm
7
8      clean_edges.len := 0;
9
10     for i in 1:edges.len loop
11         if not CompareVectors(edges.elements[i], {-1, -1, -1,
               -1}) then
12             clean_edges := EdgesArrayAppend(clean_edges,
                   edges.elements[i]);
13         end if;
14     end for;
15
16 end RemoveMarkedEdges;
```

4.15 La funzione `VoronoiCell`

Come precedentemente detto, l'algoritmo di Lloyd è implementato tramite iterazioni successive dell'algoritmo di Voronoi.

La funzione `VoronoiCell` restituisce esclusivamente la cella di appartenenza di un drone specificato, anziché la tassellazione dell'intera area; considerato che l'algoritmo è distribuito ogni drone necessita di conoscere esclusivamente la propria posizione *target*, e soltanto la posizione attuale dei restanti droni.

L'idea è quindi quella di implementare la seguente funzione:

$$\text{VoronoiCell} : (\text{drone stesso}, \text{altri droni}, \text{bordi}) \longrightarrow \text{cella drone stesso}$$

I passaggi fondamentali dell'algoritmo implementato sono i seguenti:

1. Per ogni `drone` contenuto in `other drones` (ovvero la lista degli altri droni)
   (a) viene creato un segmento `union edge` che unisce il drone stesso (`primary drone`) con `drone`
   (b) viene determinato l'asse di `union edge`, che prende il nome di `perp bisect`
   (c) viene inizializzata una lista vuota `intersections` che andrà a contenere i punti di intersezione trovati

(d) Per ogni bordo `edge` contenuto in `edges`

i. viene cercato l'eventuale punto di intersezione tra `perp bisect` ed `edge`

ii. se il punto di intersezione esiste

A. contrassegno `edge` per la cancellazione, in quanto verrà sostituito da un nuovo bordo

B. determino quale estremo di `edge` conservare per la creazione del nuovo bordo

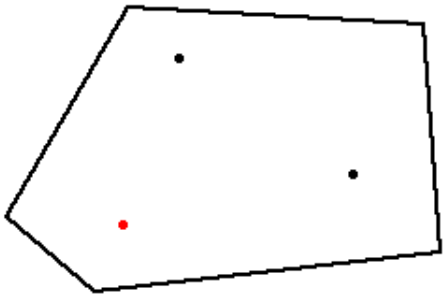
C. aggiunto il punto di intersezione trovato alla lista `intersections`

(e) se `intersections` contiene 2 punti, creo un nuovo bordo che abbia come estremi i 2 punti

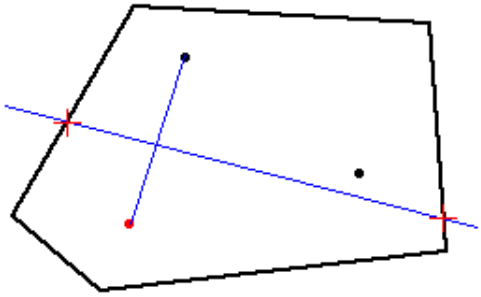
2. tolgo da `edges` i bordi duplicati e quelli contrassegnati per l'eliminazione

4.15.1 Esempio grafico di applicazione dell'algoritmo di Voronoi

Vediamo adesso l'applicazione dell'algoritmo di Voronoi alla seguente situazione: l'area è delimitata da un poligono a 5 lati; in rosso è visibile quello che viene chiamato `PrimaryDrone`, ovvero il drone che effettua la chiamata a `VoronoiCell` e per il quale si intende calcolare la cella associata; i restanti 2 puntini neri rappresentano gli altri due droni.

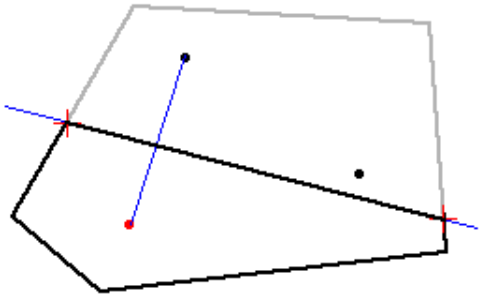


Si determina quindi il segmento che unisce `PrimaryDrone` con uno degli altri droni, in questo caso quello più in alto; si traccia l'asse del segmento trovato e si determinano i punti di intersezione con i bordi dell'area, che nella figura sono contrassegnati da croci rosse.



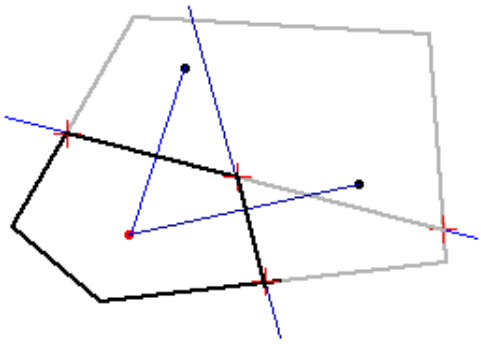
Si crea un nuovo bordo che unisce i due punti di intersezione. Tutti i segmenti che si trovino "dietro" (si veda la sezione 4.14) al nuovo segmento vengono contrassegnati per l'eliminazione.

Si noti inoltre che i due bordi con i quali l'asse intersecava sono stati sostituiti da due nuovi bordi, di lunghezza inferiore.



Si ripete il procedimento anche per l'altro drone, ed essendo in questo caso i droni totali in numero pari a 3 si conclude l'algoritmo.

Il poligono contornato in nero è quindi la cella di Voronoi associata al `PrimaryDrone`



Il codice che ne deriva è il seguente:

```
function VoronoiCell

    input  Real [:, 4]  input_edges;
    input  Real [2]     primary_drone;
    input  Real [:, 2]  other_drones;
    output EdgesArray   edges;

protected

    Real [2]     drone, point, intersect, p1, p2, int1, int2,
        keep;
    Real [4]     union_edge, edge_p1_primary_drone,
       edge_p2_primary_drone;
    Real [3]     perp_bisect;
    PointsArray  intersections;
    EdgesArray   new_edges;
    Boolean      have_intersection, int1_valid, int2_valid,
       add_to_intersections;

algorithm

    edges.len := 0;

    // Copio in edges i segmenti contenuti in input_edges
    for iei in 1:size(input_edges, 1) loop
        edges := EdgesArrayAppend(edges, input_edges[iei]);
    end for;

    for other_drones_index in 1:size(other_drones, 1) loop
        drone := other_drones[other_drones_index];

        // Determino il segmento che unisce primary_drone e
            drone
        union_edge := {primary_drone[1], primary_drone[2],
            drone[1], drone[2]};

        // Determino la bisettrice di union_edge
        perp_bisect := PerpendicularBisector(union_edge);

        // Svuoto intersections
        intersections.len := 0;

        // Svuoto new_edges
        new_edges.len := 0;

        // Contrassegno i bordi di edges da cancellare
        edges := MarkUnwantedEdges(edges, primary_drone);

        for i in 1:edges.len loop

            // Controllo che il segmento non sia gia' stato
                contrassegnato per essere eliminato
```

```
47              if not CompareVectors(edges.elements[i], {-1, -1,
                    -1, -1}) then
48
49                  (have_intersection, intersect) :=
                        SegmentsIntersection(
50                      LineToSegment(perp_bisect),
51                      edges.elements[i]
52                  );
53
54                  if have_intersection then
55
56                      // Determino quale estremo del segmento
                            mantenere
57                      p1 := {edges.elements[i, 1],
                            edges.elements[i, 2]};
58                      p2 := {edges.elements[i, 3],
                            edges.elements[i, 4]};
59                      edges.elements[i] := {-1, -1, -1, -1};
60                      edge_p1_primary_drone := {
61                          p1[1], p1[2],
62                          primary_drone[1], primary_drone[2]
63                      };
64                      edge_p2_primary_drone := {
65                          p2[1], p2[2],
66                          primary_drone[1], primary_drone[2]
67                      };
68                      (int1_valid, int1) :=
                            SegmentsIntersection(
69                          LineToSegment(perp_bisect),
70                          edge_p1_primary_drone
71                      );
72                      (int2_valid, int2) :=
                            SegmentsIntersection(
73                          LineToSegment(perp_bisect),
74                          edge_p2_primary_drone
75                      );
76                      assert(int1_valid or int2_valid, "non
                            viene creata un intersezione ne con p1
                             ne con p2. p1 = " + VectorToString(p1
                            ) + ", p2 = " + VectorToString(p2) + "
                            , primary_drone = " + VectorToString(
                            primary_drone));
77                      if int1_valid and not int2_valid then
78                          keep := p2;
79                      elseif not int1_valid and int2_valid then
80                          keep := p1;
81                      elseif int1_valid and int2_valid then
82                      else
83                          assert(false, "errore logico");
84                      end if;
85
86                      // Costruisco il nuovo bordo
87                      new_edges := EdgesArrayAppend(new_edges,
                            {intersect[1], intersect[2], keep[1],
```

```
                        keep[2]});
88
89                  // Valuto se aggiungere il nuovo bordo
                        alla lista
90                  // e nel caso lo aggiungo
91                  add_to_intersections := true;
92                  for intersections_index in 1:
                        intersections.len loop
93                      if ValuesAreClose(
                            intersections.elements[
                            intersections_index, 1], intersect
                            [1])
94                          and ValuesAreClose(
                                intersections.elements[
                                intersections_index, 2],
                                intersect[2]) then
95                          add_to_intersections := false;
96                      end if;
97                  end for;
98                  if add_to_intersections then
99                      intersections := PointsArrayAppend(
                            intersections, intersect);
100                 end if;
101
102             end if;
103
104             // Contrassegno tutti i bordi contenuti in
                    edges
105             // che devono essere cancellati in quanto "
                    oscurati"
106             // da altri bordi
107             edges := MarkUnwantedEdges(edges,
                    primary_drone);
108
109         end if;
110     end for;
111
112     // Aggiungo ad edges tutti i bordi contenuti in
            new_edges
113     for j in 1:new_edges.len loop
114         edges := EdgesArrayAppend(edges,
                new_edges.elements[j]);
115     end for;
116
117     // Svuoto new_edges
118     new_edges.len := 0;
119
120     // Se ho due punti di intersezione, creo un segmento
            che li unisce
121     // e lo aggiungo a edges
122     if intersections.len == 2 then
123         edges := EdgesArrayAppend(edges, {
                intersections.elements[1,1],
                intersections.elements[1,2],
```

```
                    intersections.elements[2,1],
                    intersections.elements[2,2]});
124         end if;
125
126     end for;
127
128     assert(edges.len > 1, "edges deve contenere almeno un
            elemento");
129
130     edges := MarkUnwantedEdges(edges, primary_drone);
131     edges := RemoveMarkedEdges(edges);
132     edges := RemoveDuplicatedEdges(edges);
133
134 end VoronoiCell;
```

4.16 La funzione `TargetPos`

La funzione `TargetPos` effettua una singola iterazione dell'algoritmo di Lloyd, ovvero:

1. Determina la cella di Voronoi relativa al drone stesso
2. Calcola il centro di massa della cella determinata al punto 1

```
1  function TargetPos
2
3      input   Real   [:,4]  area_boundaries;
4      input   Real   [2]    self_position;
5      input   Real   [:,2]  other_drones_positions;
6      output  Real   [2]    target_position;
7
8  algorithm
9      target_position := CenterOfMass(
10         EdgesToVertices(
11             VoronoiCell(
12                 area_boundaries,
13                 self_position,
14                 other_drones_positions
15             )
16         )
17     );
18 end TargetPos;
```

4.17 Testing del codice

Attualmente *OpenModelica* non fornisce un meccanismo di testing solido e standardizzato. Per questo motivo è stato necessario realizzare un sistema che automatizzasse la verifica della correttezza delle funzioni implementate.

A questo proposito si è deciso di ricorrere ad uno script *Python* che:

1. Ottenga la lista delle classi implementate
2. Ottenga la lista dei test da effettuare
3. Compili le classi
4. Crei un file contenente le istruzioni per eseguire tutti i test
5. Esegua, uno dopo l'altro, i vari test
6. Restituisca a schermo un output indicante l'esito dei vari test

```
#!/bin/env python3

import re
import subprocess
import os
import sys

CLASSES_FILENAMES = list(
    filter(
        lambda name: name.endswith(".mo"),
        os.listdir("classes")
    )
)
CLASSES_FILENAMES = list(
    map(
        lambda name: "classes/" + name,
        CLASSES_FILENAMES
    )
)

TEST_FILENAMES = list(
    filter(
        lambda name: name.endswith(".mo"),
        os.listdir("test")
    )
)
TEST_FILENAMES = list(
    map(
        lambda name: "test/" + name,
        TEST_FILENAMES
    )
)

FILENAMES = CLASSES_FILENAMES + TEST_FILENAMES

class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
```

```
40      WARNING = '\033[93m'
41      FAIL = '\033[91m'
42      ENDC = '\033[0m'
43      BOLD = '\033[1m'
44      UNDERLINE = '\033[4m'
45
46  os.system("clear")
47
48  forced_tests = list(
49      filter(
50          lambda s: s.startswith("test_"),
51          sys.argv
52      )
53  )
54
55  content = ""
56  for filename in FILENAMES:
57      with open(filename) as f:
58          content += f.read()
59
60  identifiers = re.findall(r"test_\w+", content)
61  identifiers = list(set(identifiers))
62
63  with open("tmp_simulate.mos", mode="w+") as f:
64      for filename in FILENAMES:
65          f.write(f'loadFile("{filename}"); getErrorString();\n
                ')
66      f.write('cd("/tmp");\n')
67      if forced_tests:
68          for t in forced_tests:
69              f.write(f"simulate({t}); getErrorString();\n")
70              f.write('print
                    ("----------------------------------------------------------------------
                    ;\n')
71      else:
72          for i in identifiers:
73              f.write(f"simulate({i}); getErrorString();\n")
74              f.write('print
                    ("----------------------------------------------------------------------
                    ;\n')
75
76  result = subprocess.run(['omc', './tmp_simulate.mos'], stdout
        =subprocess.PIPE)
77  result = result.stdout.decode()
78
79  if "--raw" in sys.argv:
80      print(result)
81  else:
82      for func in [
83          lambda line: not line.startswith("    time"),
84          lambda line: "record SimulationResult" not in line,
85          lambda line: "end SimulationResult" not in line,
86          lambda line: line != '""',
87          lambda line: line != 'true',
```

```
88      ]:
89          result = "\n".join(filter(func, result.split("\n")))
90      result = result.replace("The simulation finished
            successfully.", bcolors.OKGREEN + "The simulation
            finished successfully!" + bcolors.ENDC)
91      result = result.replace("assert", bcolors.FAIL + "assert"
             + bcolors.ENDC)
92      for i in identifiers:
93          result = result.replace(i, bcolors.BOLD + i + bcolors
                .ENDC)
94      print(result)
```

Capitolo 5

Conclusioni

Bibliografia

- https://en.wikipedia.org/wiki/Lloyd%27s_algorithm
- github.com/zapateocallisto
- https://en.wikipedia.org/wiki/Centroid