

Prefazione (bozza)

L'obiettivo di questa trattazione è di illustrare l'implementazione di un insieme di classi OpenModelica che abbiano la funzione 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

1	Il drone e le sue componenti	6
1.1	Ipotesi e semplificazioni	7
2	L'algoritmo di Lloyd	8
2.1	Descrizione	9
2.2	Esempio di applicazione dell'algoritmo	9
2.3	Convergenza dell'algoritmo	11
2.4	Esecuzione dell'algoritmo	11
3	Prototipazione	12
3.1	Codifica	12
3.2	Testing	12
4	Implementazione in <i>OpenModelica</i>	14
4.1	OpenModelica	14
4.2	Un approccio funzionale	15
4.3	Diagramma delle dipendenze	16
4.4	Scelte implementative	17
4.5	Rappresentazione del punto geometrico	18
4.6	Verifica della sovrapposizione di due punti	18
4.7	Rappresentazione del segmento geometrico	19
4.8	Verifica della sovrapposizione di due segmenti	19
4.9	Linea	20
4.10	Conversione da linea a segmento	20
4.11	Intersezione tra segmenti	21
4.12	Asse di un segmento	22
4.13	Centro di massa	23
4.14	Ricerca ed eliminazione dei segmenti in eccesso	24
4.15	La funzione VoronoiCell	26
4.15.1	Esempio grafico di applicazione dell'algoritmo di Voronoi	27
4.16	La funzione TargetPos	30
4.17	Testing del codice	31
5	Conclusioni	34

Capitolo 1

Il drone e le sue componenti



Possiamo schematizzare, ai fini di questa trattazione, un *drone* mediante 3 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 ipotesi 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'interno 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.

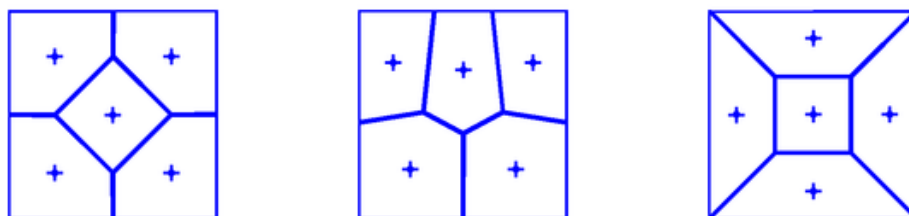


Figura 2.1: Tre esempi di CVT a 5 punti su un area quadrata

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.

¹Dall'inglese *Centroidal Voronoi Tessellation*



Figura 2.2: Scala del gigante, Irlanda del Nord

2.1 Descrizione

L'algoritmo esegue ripetutamente i seguenti *step*:

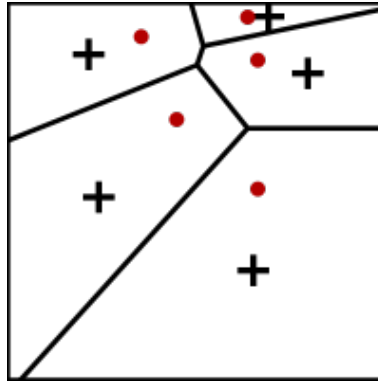
1. Viene generato il diagramma di Voronoi
2. Per ogni cella trovata, viene determinato il *baricentro*
3. Ogni punto viene spostato in corrispondenza del *baricentro* della propria *cella di Voronoi*

2.2 Esempio di applicazione dell'algoritmo

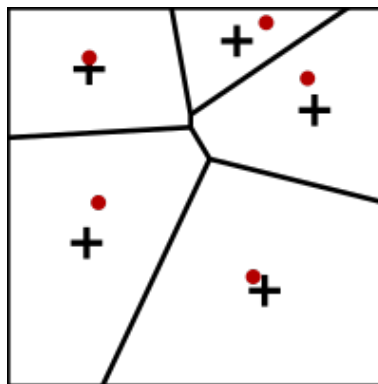
Viene qui presentata l'applicazione dell'algoritmo di Lloyd ad un'area quadrata nella quale sono presenti 5 partizioni.

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.

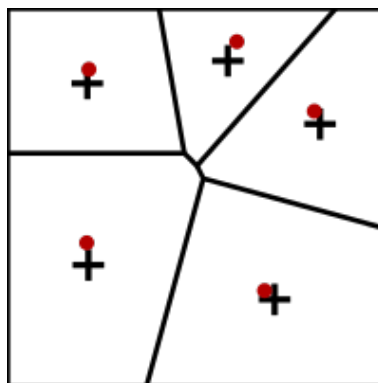
Iterazione 1 I droni sono mediamente distanti dal centro di massa della propria cella.



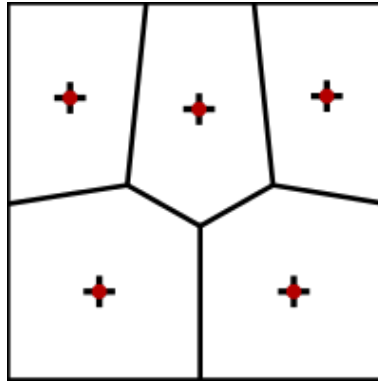
Iterazione 2 La distanza media tra i droni e il centro di massa delle rispettive celle è ridotta. Si noti come si è modificata la forma delle varie celle, in conseguenza del movimento dei vari droni.



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 dinamico.

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 progetto³.

3.2 Testing

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

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<https://www.python.org/>

²<https://www.pygame.org/>

³https://github.com/zapateo/Tesi_GianlucaMondini/tree/master/implementazione_algoritmo/python

```
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, verranno 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 progetto¹ e al documento di specifica Modelica²

¹<https://openmodelica.org/>

²<https://www.modelica.org/documents/ModelicaSpec34.pdf>

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 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)
10
11 A = 100
12
13 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:

```

1 A = 0
2 B = 20
3
4 func F(C, D, A) {
5     A = A + 10
6     return C + D + A, A
7 }
8
9 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 subito un incremento.

Questo nuovo modello semplifica notevolmente 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 notevolmente 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*.

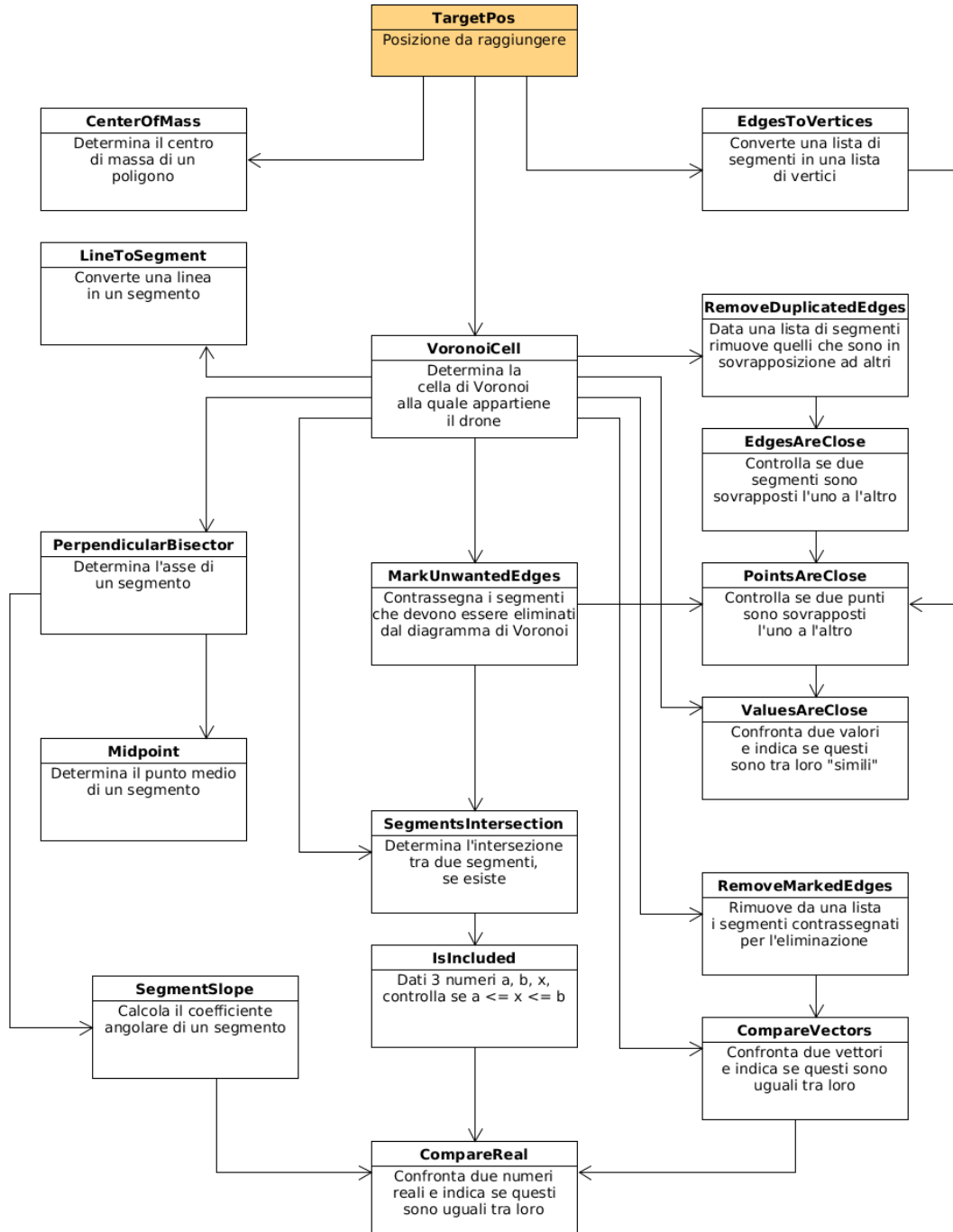


Figura 4.4: Diagramma delle dipendenze delle varie funzioni

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

```

1 record Point
2     Real x, y;
3 end Point;

```

sia come vettore

```

1 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 richiedo 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

```

1 Real [2] point;

```

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

```

1 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 uguaglianza. 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;

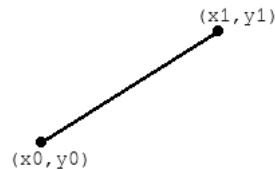
```

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:

```

1  Real [4] edge;
2
3  edge[1] := x_1;
4  edge[2] := y_1;
5  edge[3] := x_2;
6  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.

```

1 function EdgesAreClose
2
3     input Real [4] e1, e2;
4
5     output Boolean are_close;
6
7 algorithm
8
9     are_close := PointsAreClose({e1[1], e1[2]}, {e2[1], e2[2]}) and
        PointsAreClose({e1[3], e1[4]}, {e2[3], e2[4]});
10
11 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

```

1 Real [3] line;
2
3 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;

```

```

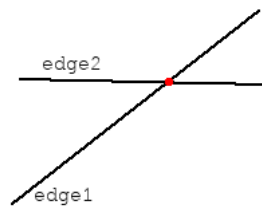
23         y1 := m * (-big) + q;
24         y2 := m * (+big) + q;
25         segment := {-big, y1, big, y2};
26     end if;
27 end LineToSegment;

```

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

```

1  function SegmentsIntersection
2      input Real [4] edge1, edge2;
3      output Boolean valid;
4      output Real [2] intersection;
5  protected
6      Real x1, x2, y1, y2, dx1, dy1, x, y, xB, yB, dx, dy, DET, DETinv,
          r, s, xi, yi;
7      parameter Real DET_TOLERANCE = 0.00000001;
8  algorithm
9      x1 := edge1[1];
10     y1 := edge1[2];
11     x2 := edge1[3];
12     y2 := edge1[4];
13
14     x := edge2[1];
15     y := edge2[2];
16     xB := edge2[3];
17     yB := edge2[4];
18
19     dx1 := x2 - x1;
20     dy1 := y2 - y1;
21
22     dx := xB - x;
23     dy := yB - y;
24
25     DET := ((-dx1 * dy) + (dy1 * dx));
26
27     if abs(DET) < DET_TOLERANCE then
28         valid := false;
29         return;

```

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

```

30     end if;
31
32     DETinv := 1.0/DET;
33
34     r := DETinv * (-dy * (x-x1) + dx * (y-y1));
35     s := DETinv * (-dy1 * (x-x1) + dx1 * (y-y1));
36     xi := (x1 + r*dx1 + x + s*dx)/2.0;
37     yi := (y1 + r*dy1 + y + s*dy)/2.0;
38
39     if (IsIncluded(0, 1, r)) and (IsIncluded(0, 1, s)) then
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.

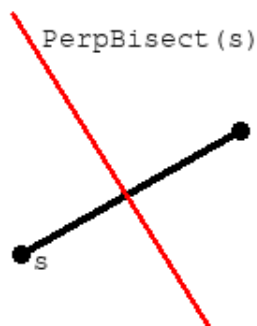
4.12 Asse di un segmento

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;
8   algorithm
9     p := Midpoint(edge);
10    (m1, vertical) := SegmentSlope(edge);
11    if vertical then
12      neg_c := (edge[2] + edge[4])/2;
13      perp_bisect := {0, 1, -neg_c};
14      return;
15    elseif m1 == 0 then
16      a := 1;
17      b := 0;
18      c := -(edge[1] + edge[3])/2;
19      perp_bisect := {a, b, c};
20      return;
21    else
22      m2 := -1/m1;
23      q := - m2 * p[1] + p[2];
24      perp_bisect := {-m2, 1, -q};
25      return;
26    end if;
27 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 $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$ 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*

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

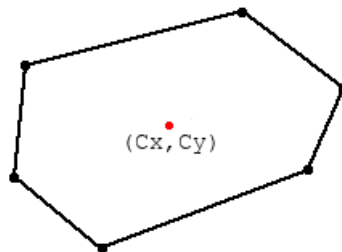


Figura 4.5: Il punto p viene restituito dalla chiamata a `CenterOfMass` che ha come argomento la lista delle coordinate dei vari vertici del poligono

Ne deriva quindi la seguente codifica:

```
1 function CenterOfMass
2
3     input Real[:, 2] points;
4
5     output Real[2] out;
6
7     protected
8
9     Real[:, 2] vertices, v_local;
10    Real x_cent, y_cent, area, factor;
11
12    algorithm
13
14        // Adattato da https://stackoverflow.com/a/46937541
15        vertices := points;
16
17        x_cent := 0;
18        y_cent := 0;
19        area := 0;
20
21        v_local := cat(1, vertices, {vertices[1]});
22
23        for i in 1:(size(v_local, 1) - 1) loop
24            factor := v_local[i, 1] * v_local[i+1, 2] - v_local[i+1, 1] *
25                    v_local[i, 2];
26            area := area + factor;
27            x_cent := x_cent + (v_local[i, 1] + v_local[i+1, 1]) * factor;
28            y_cent := y_cent + (v_local[i, 2] + v_local[i+1, 2]) * factor;
29        end for;
30
31        area := area / 2.0;
32        x_cent := x_cent / (area * 6);
33        y_cent := y_cent / (area * 6);
34
35        out := {x_cent, y_cent};
36    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.

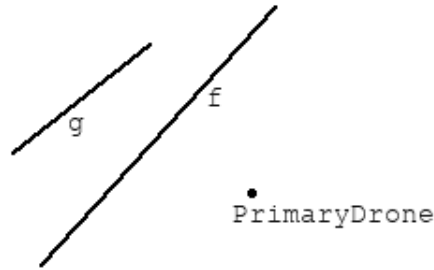


Figura 4.6: Esempio nel quale il segmento g deve essere rimosso, in quanto il segmento f si pone tra esso e il *PrimaryDrone*. La funzione **MarkUnwantedEdges** contrassegnerà g per l'eliminazione, mentre **RemoveMarkedEdges** lo cancellerà definitivamente.

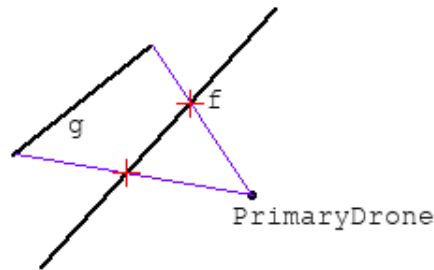


Figura 4.7: I segmenti di unione tra *PrimaryDrone* e gli estremi di g collidono con il segmento f ; questo fa sì che g venga contrassegnato per l'eliminazione.

```

1 function MarkUnwantedEdges
2     input Real[:,4] edges;
3     input Real[2] primary_drone;
4     output Real[:,4] marked_edges;
5 protected
6     Real[0,4] empty_marked_edges;
7     Real[2] point, intersection;
8     Real[4] inner_edge, join_edge;
9     Boolean valid;
10 algorithm
11     marked_edges := edges;
12     for outer_i in 1:size(edges, 1) loop

```

```

13     for point_index in 1:2 loop
14         if point_index == 1 then
15             point := {edges[outer_i,1], edges[outer_i,2]};
16         elseif point_index == 2 then
17             point := {edges[outer_i,3], edges[outer_i,4]};
18         end if;
19         for inner_edge_index in 1:size(edges, 1) loop
20             inner_edge := edges[inner_edge_index];
21             join_edge := {primary_drone[1], primary_drone[2], point
22                           [1], point[2]};
23             (valid, intersection) := SegmentsIntersection(
24                 inner_edge, join_edge);
25             if valid and (not PointsAreClose(intersection, point))
26                 then
27                 marked_edges[outer_i] := {-1, -1, -1, -1};
28             else
29                 end if;
30             end for;
31         end for;
32     end for;
33 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     input Real[:,4] edges;
3     output Real[:,4] clean_edges;
4     protected
5         Real [0,4] empty_clean_edges;
6     algorithm
7         clean_edges := empty_clean_edges;
8         for i in 1:size(edges, 1) loop
9             if not CompareVectors(edges[i], {-1, -1, -1, -1}) then
10                 clean_edges := cat(1, clean_edges, {edges[i]});
11             end if;
12         end for;
13 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:

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

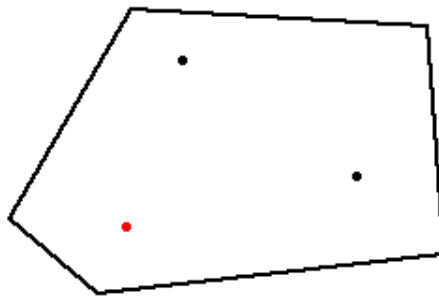
I passaggi fondamentali dell'algoritmo implementato sono i seguenti:

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`

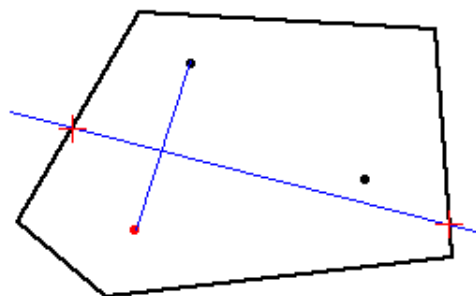
- (b) viene determinato l'asse di **union edge**, che prende il nome di **perp bisect**
 - (c) viene inizializzata una lista vuota **intersections** che andrà a contenere i punti di intersezione trovati
 - (d) Per ogni bordo **edge** contenuto in **edges**
 - i. viene cercato l'eventuale punto di intersezione 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.

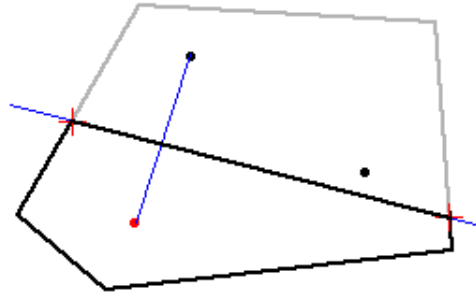


Si determina quindi il segmento che unisce **PrimaryDrone** con uno degli altri droni, in questo caso quello più in alto; si traccia l'asse del segmento trovato e si determinano i punti di intersezione con i bordi dell'area, che nella figura sono contrassegnati da croci rosse.



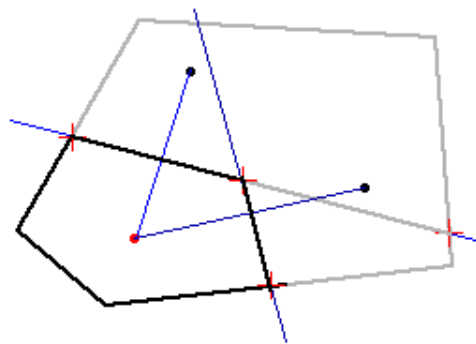
Si crea un nuovo bordo che unisce i due punti di intersezione. Tutti i segmenti che si trovino "dietro" (si veda la sezione 4.14) al nuovo segmento vengono contrassegnati per l'eliminazione.

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:

```

1  function VoronoiCell
2
3      input  Real    [ :, 4 ]  input_edges;
4      input  Real    [ 2 ]     primary_drone;
5      input  Real    [ :, 2 ]  other_drones;
6
7      output Real    [ :, 4 ]  output_edges;
8
9  protected
10
11      Real    [ :, 4 ]  edges;
12      Real    [ 2 ]     drone, point, intersect, p1, p2, int1, int2, keep;
13      Real    [ 4 ]     union_edge, edge_p1_primary_drone,
14                          edge_p2_primary_drone, new_edge;
15      Real    [ 3 ]     perp_bisect;
16      Real    [ :, 2 ]  intersections;      Real [ 0,2 ] empty_intersections;
17      Real    [ :, 4 ]  new_edges;          Real [ 0,4 ] empty_new_edges;
18      Boolean have_intersection, int1_valid, int2_valid,
19                          add_to_intersections;

```

```

19 algorithm
20
21     edges := input_edges;
22     for other_drones_index in 1:size(other_drones, 1) loop
23         drone := other_drones[other_drones_index];
24
25         union_edge := {primary_drone[1], primary_drone[2], drone[1],
26                         drone[2]};
27         perp_bisect := PerpendicularBisector(union_edge);
28         intersections := empty_intersections;
29         new_edges := empty_new_edges;
30         edges := MarkUnwantedEdges(edges, primary_drone);
31
32         for i in 1:size(edges, 1) loop
33             // Controllo che il segmento non sia gia' stato
34             // contrassegnato per essere eliminato
35             if not CompareVectors(edges[i], {-1, -1, -1, -1}) then
36
37                 (have_intersection, intersect) := SegmentsIntersection(
38                     LineToSegment(perp_bisect),
39                     edges[i]
40                 );
41
42                 if have_intersection then
43
44                     // Determino quale estremo del segmento mantenere
45                     p1 := {edges[i, 1], edges[i, 2]};
46                     p2 := {edges[i, 3], edges[i, 4]};
47                     edges[i] := {-1, -1, -1, -1};
48                     edge_p1_primary_drone := {
49                         p1[1], p1[2],
50                         primary_drone[1], primary_drone[2]
51                     };
52                     edge_p2_primary_drone := {
53                         p2[1], p2[2],
54                         primary_drone[1], primary_drone[2]
55                     };
56                     (int1_valid, int1) := SegmentsIntersection(
57                         LineToSegment(perp_bisect),
58                         edge_p1_primary_drone
59                     );
60                     (int2_valid, int2) := SegmentsIntersection(
61                         LineToSegment(perp_bisect),
62                         edge_p2_primary_drone
63                     );
64                     assert(int1_valid or int2_valid, "non viene creata
65                         un intersezione ne con p1 ne con p2. p1 = " +
66                         VectorToString(p1) + ", p2 = " + VectorToString(
67                         p2) + ", primary_drone = " + VectorToString(
68                         primary_drone));
69                     if int1_valid and not int2_valid then
70                         keep := p2;
71                     elseif not int1_valid and int2_valid then
72                         keep := p1;
73                     elseif int1_valid and int2_valid then
74                         else
75                             assert(false, "errore logico");
76                         end if;
77
78                     // Costruisco il nuovo bordo
79                     new_edge := {intersect[1], intersect[2], keep[1],

```

```

76         keep[2]};
77     new_edges := cat(1, new_edges, {new_edge});
78     // Valuto se aggiungere il nuovo bordo alla lista
79     add_to_intersections := true;
80     for intersections_index in 1:size(intersections, 1)
81         loop
82             if ValuesAreClose(intersections[
83                 intersections_index, 1], intersect[1])
84                 and ValuesAreClose(intersections[
85                     intersections_index, 2], intersect[2])
86                 then
87                     add_to_intersections := false;
88                 end if;
89             end for;
90         end if;
91
92         edges := MarkUnwantedEdges(edges, primary_drone);
93     end if;
94 end for;
95 edges := cat(1, edges, new_edges);
96
97 if size(intersections, 1) == 2 then
98     edges := cat(1, edges, {{intersections[1,1], intersections
99         [1,2], intersections[2,1], intersections[2,2]}});
100 end if;
101
102 end for;
103
104 assert(size(edges, 1) > 1, "edges deve contenere almeno un elemento
105     ");
106
107 edges := MarkUnwantedEdges(edges, primary_drone);
108 edges := RemoveMarkedEdges(edges);
109 edges := RemoveDuplicatedEdges(edges);
110
111 output_edges := edges;
112 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    [2]    self_position;
4      input    Real    [2]    other_drones_positions;
5      input    Real    [2]    area_boundaries;
6
7      output   Real    [2]    target_position;
8
9  algorithm
10     target_position := CenterOfMass(
11         EdgesToVertices(
12             VoronoiCell(
13                 area_boundaries,
14                 self_position,
15                 other_drones_positions
16             )
17         )
18     );
19 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

```

1  #!/bin/env python3
2
3  import re
4  import subprocess
5  import os
6  import sys
7
8  CLASSES_FILENAMES = list(
9      filter(
10         lambda name: name.endswith(".mo"),
11         os.listdir("classes")
12     )
13 )
14 CLASSES_FILENAMES = list(
15     map(
16         lambda name: "classes/" + name,
17         CLASSES_FILENAMES
18     )
19 )

```

```

19 )
20
21 TEST_FILENAMES = list(
22     filter(
23         lambda name: name.endswith(".mo"),
24         os.listdir("test")
25     )
26 )
27 TEST_FILENAMES = list(
28     map(
29         lambda name: "test/" + name,
30         TEST_FILENAMES
31     )
32 )
33
34 FILENAMES = CLASSES_FILENAMES + TEST_FILENAMES
35
36 class bcolors:
37     HEADER = '\033[95m'
38     OKBLUE = '\033[94m'
39     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
71                     ("-----")
72                     ;\n')
73             else:
74                 for i in identifiers:
75                     f.write(f"simulate({i}); getErrorString();\n")
76                     f.write('print
77                         ("-----")
78                         ;\n')
79
80 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.",
91         bcolors.OKGREEN + "The simulation finished successfully!" +
92         bcolors.ENDC)
91     result = result.replace("assert", bcolors.FAIL + "assert" + bcolors
92         .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>