SAPIENZA
UNIVERSITÀ DI ROMA

Design e Verifica di un Centro di Controllo per UAV

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica
Corso di laurea in Informatica

Andrea Di Marco
Matricola 1835169

Responsabile
Prof. Enrico Tronci

A.A. 2021-2022

1. Abstract **3**

2. Il Modello **4**
2.1 Drone 4
2.1.1 Drone Flight System 5
2.1.2 Controller 5
2.1.3 Comunicazione con il CC 6
2.1.4 Drone 6
2.2 Centro di Controllo 7
2.2.1 Richieste e risposte 8
2.2.2 Controllo posizioni 8
2.2.3 Invio al server 9
2.2.4 Controllo aree 10
2.2.5 Controlli loop 10
2.2.6 Scelta del nuovo set point 11
2.2.7 Ricarica batteria 13
2.2.8 Invio nuovi set point 14
2.3 Monitor 15
2.3.1 Raccolta informazioni 16
2.3.2 Calcolo fattori 16
2.3.3 Termine analisi 18
2.4 Aree 18
2.5 Cloud 20
2.5.1 Client 22
2.5.1.1 Collegamento a Modelica 22
2.5.1.2 Funzione send2server 23
2.5.2 Server 25
2.5.2.1 API Database 25
2.5.3 Alternativa in locale 26
2.6 Database 27
2.6.1 Entity Relationship 27
2.6.1.1 Diagramma 27
2.6.1.2 Dizionario dei dati 28
2.6.1.3 Vincoli Esterni 28
2.6.2 SQL 29
2.6.2.1 Tabelle relazionali 29
2.6.2.2 Funzionalità 29
2.6.2.3 Trigger 32

3. Nevergrad **33**
3.1 Abstract 33
3.2 Algoritmi Base 33
3.2.1 NGOpt 33
3.2.2 Cobyla 34
3.2.3 Powell 34
3.3 Vincoli 35

3.4 Rumore 37

4. NOMAD **39**
4.1 Abstract 39
4.2 Modellazione 39
4.3 Rumore 41

5. Confronto tool di BBO **42**
5.1 Risultati 42
5.2 Scalabilità 44

6. Conclusione **47**

7. Bibliografia **52**

8. Note **53**

9. Ringraziamenti **54**

1. Abstract

Molti sistemi intelligenti odierni sono ***cyber-fisici*** (***CPS***) e ***cloud-based***, ovvero in grado di interagire in modo continuo con il mondo fisico su cui operano e affidano i calcoli più onerosi e complessi ad un server esterno.

Applicazioni tipiche di sistemi cyber-fisici ricadono sotto l'ombrello della comunicazione tra sensori e calcolatori, in quello che viene comunemente chiamato ***Machine-to-Machine Interaction*** (M2M), con lo scopo monitorare aspetti dell'ambiente fisico circostante e agire per cambiarlo.

I *CPS* sono estremamente diffusi negli ambienti critici, dalle reti elettriche alle serre, grazie alla necessità sempre maggiore di applicare tecniche *IT-driven* all'interno di sistemi di automazione tradizionali. Le Fabbriche del Futuro (*FoF*) faranno sempre più affidamento a grandi ecosistemi digitali in cooperazione fra loro, mediante pattern di *Software Engineering* avanzati e algoritmi di Intelligenza Artificiale. Questi algoritmi non potranno più preoccuparsi soltanto della matematica astratta, ma occorre definire metodi per gestire ed adattarsi efficacemente a tutta una serie di nuove problematiche quali ritardi nelle comunicazioni, lentezza delle componenti hardware, danneggiamento, rottura, indipendenza operativa di queste ultime e molte altre. Il sistema agile del futuro deve permettere monitoraggio, processamento e controllo del flusso di informazioni in modo altamente distribuito e stratificato. [SPR2014]

L'obiettivo di questa relazione è costruire il modello di un Centro di Controllo per UAV con *OpenModelica,* che oltre a gestire un numero variabile di droni e aree da pattugliare, sia in grado di comunicare il suo operato ad un database nel *cloud* e insieme ad esso valutare le prestazioni generali del sistema. [MOD2021]

Nella seconda fase della relazione mi occuperò di regolare gli iperparametri del sistema in modo da massimizzarne le performance utilizzando i diversi algoritmi di ottimizzazione *black-box* compresi nei *tool* di ***Nevergrad*** e ***NOMAD***.

Infine metterò a confronto le configurazioni proposte dai due *tool* con lo scopo di identificare i valori migliori per questo problema specifico.

2. Il Modello

Il modello in questione gestisce *n* droni all'interno di un'arena di dimensioni 200 m^3.
Gli *n* droni sono gestiti da un unico ***Centro di Controllo*** (***CC***) che si occupa di impostare le rotte dei droni e dirigerli verso le *aree* dell'arena con l'obiettivo di sorvegliare la zona.

2.1 Drone

Ogni drone parte dalla *docking bay* esterna all'arena e ha una batteria che si scarica nel tempo con un rateo ***batteryDischarge***. Appena raggiunge un livello di carica pericolosamente basso detto ***dangerousBatteryLevel***, questo ritornerà alla *docking bay* per ricaricare la batteria. Finché la batteria è sufficientemente carica il drone si dirigerà verso il *set point* fornito dal *CC*.

Andando più nel dettaglio, il drone è diviso nelle seguenti componenti.

2.1.1 Drone Flight System

Questa componente si occupa di far volare il drone, ovvero trasformare il *thrust T* del controller (vedremo più avanti) in velocità grazie alle seguenti equazioni differenziali:

- Velocità:

$$\dot{x} = v_x \qquad \dot{y} = v_y \qquad \dot{z} = v_z$$

- Accelerazione:

$$\dot{v}_x = \frac{T_x}{m} \qquad \dot{v}_y = \frac{T_y}{m} \qquad \dot{v}_z = \frac{T_z}{m}$$

Questo perché la velocità è la derivata della posizione e l'accelerazione è la derivata della velocità, ma l'accelerazione è forza/massa (del drone).

$$v_x = \dot{x} \quad \wedge \dot{a}_x = \dot{v}_x \quad \rightarrow a_x = \frac{T_x}{m}$$

Ad ogni battito ***Tdrone***, il drone consuma la batteria e controlla i dintorni. Dopo di che controlla la sua posizione e se il metodo ***returnedHome*** ritorna che il drone si trova nella *docking bay*, allora cambia il suo stato da ***flying*** a ***recharging*** e dal prossimo battito aumenterà la carica della batteria di ***rechargeRate*** mHa fino a che questa non sarà di nuovo carica al massimo. A quel punto cambierà nuovamente il suo stato in *flying* e ripartirà.

Se la batteria dovesse raggiungere lo zero, allora il drone sarà morto e la variabile ***droneDead*** verrà messa a *true*.

2.1.2 Controller

Il controllore decide la forza con cui variare l'accelerazione sulle diverse componenti, il *thrust*, e quindi fa sì che il drone si diriga verso il ***set point*** comunicatogli dal **Centro di Controllo** (*CC*) nelle variabili di *setx*, *sety* e *setz*.

Quando la batteria è sotto il ***dangerousBatteryLevel*** allora mette da parte il set point del CC e cambia le destinazione d_x , d_y e d_z con la posizione della docking bay, affinché il drone vada a ricaricare la batteria. Il *dangerousBatteryLevel* è ricavato moltiplicando il parametro normalizzato ***dangerourBatteryPercentage*** con la quantità di *mHa* della batteria.

Le equazioni differenziali che decidono la forza del *thrust* sono:

$$T_x = m\left(k_{x_1}\left(x - d_x\right) + k_{x_2} v_x\right)$$

$$T_y = m\left(k_{y_1}\left(y - d_y\right) + k_{y_2} v_y\right)$$

$$T_z = m\left(g + k_{z_1}\left(z - d_z\right) + k_{z_2} v_z\right)$$

Dove:

$$k_{x_1} = k_{y_1} = k_{z_1} = -p^2$$
$$k_{x_2} = k_{y_2} = k_{z_2} = 2p$$
$$p = -1$$

2.1.3 Comunicazione con il CC

I canali di comunicazione dal e per il centro di controllo sono modellati come due code *FIFO* dello stesso tipo, una per la scrittura e una per la lettura.

L'interfaccia della coda *FIFO* prevede che prima di leggere o scrivere, si invii un segnale, a quel punto la coda provvederà a scrivere i dati in *input*, o consegnare i dati in *output*. Per questo motivo è buona norma leggere i dati soltanto nel battito successivo al segnale, e mandare il segnale di scrittura soltanto nel battito successivo alla scrittura effettuata.

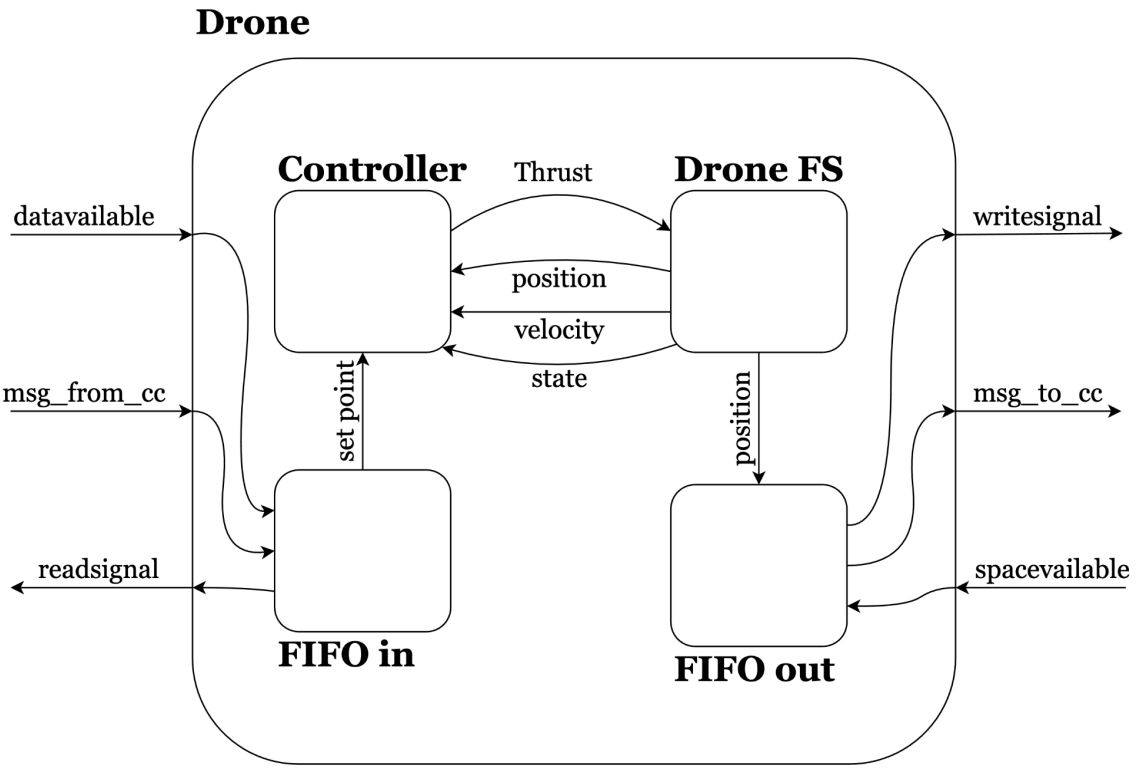
I messaggi sono definiti da *5* numeri di tipo *float*. Il primo numero indica la *flag* dell'operazione richiesta:

- ***flag* = *1***: Il *CC* chiede che il drone invii la sua posizione corrente
- ***flag* = *2***: Il *CC* ha inviato il nuovo *set point* per il drone

Quando il drone invia la posizione e la batteria al *CC* imposta *flag = 2*.

2.1.4 Drone

L'ultima componente è quella più esterna e incapsula le altre sopra, facendo sì che esse siano collegate e comunichino fra loro. Questa è quindi la componente che interagisce con il resto del mondo. La struttura è approssimabile alla seguente:



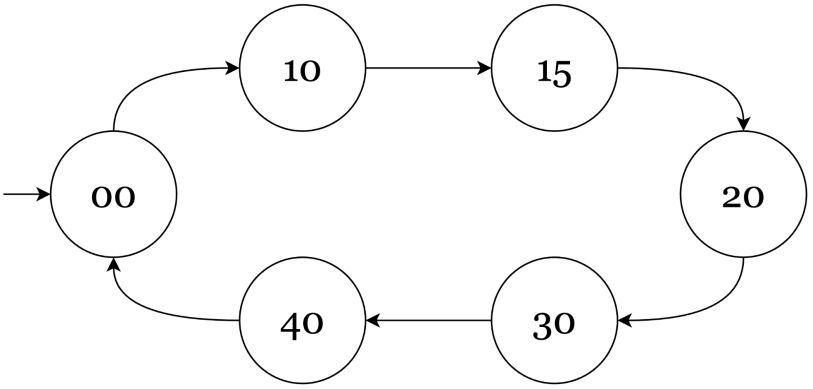
2.2 Centro di Controllo

Parte fondamentale del sistema, nonché la componente di cui vogliamo ottimizzare gli iperparametri.

Ogni ***Tcc*** battiti controlla la posizione di tutti i droni e quali aree sono state visitate, comunica poi con il database *SQL* nel *server*. Infine assegna ad ogni drone non ***busy*** (occupato) il nuovo *set point*. Un drone è da considerarsi *occupato* se non ha ancora raggiunto l'ultimo *set point* assegnato.

L'esecuzione del CC è divisa in stati:

- ***00***: Scrittura richieste per i droni
- ***10***: Invio richieste ai droni
- ***15***: Ricezione risposta dai droni
- ***20***: Elaborazione dati
- ***30***: Scrittura set point per i droni
- ***40***: Invio set point ai droni



2.2.1 Richieste e risposte

Per prima cosa il *CC* ha bisogno di sapere la posizione e il livello di carica della batteria di ogni drone. Quindi nello stato ***00*** scrive il messaggio contenente la richiesta nei canali di comunicazione di ogni drone, modellati come code *FIFO*. Questa richiesta avrà *flag = 1*.

La richiesta verrà inviata nello stato ***10*** e poi nello stato ***15*** si attenderanno le risposte dei droni. Il *CC* non abbandonerà questo stato finché non abbia ricevuto la risposta da ogni drone ancora in vita.

Nel caso un drone non dovesse rispondere alla richiesta del *CC* per più di ***comm_timeout*** secondi, verrà considerato morto.

Ricevuti tutti i messaggi si può passare allo stato ***20***.

2.2.2 Controllo posizioni

In questa fase il *CC* confronta le posizioni ricevute con i *set point* precedentemente assegnati. Se i droni si trovano entro un raggio di ***arrivalThreshold*** dal *set point*, sono considerati come *arrivati* e segnalati come *not busy*. L'area di arrivo è da segnare come *visited*, rimuovere dalle aree selezionate come destinazioni di droni e infine ne si aggiorna il tempo dell'ultima visita.

Se un drone, segnato come *busy*, non ha ancora raggiunto il *set point* da ***arrival_timeout*** secondi, allora il *CC* lo solleva dall'incarico e rende l'area di nuovo disponibile ad essere impostata come destinazione da un nuovo drone.

```
if ( pre(isBusy[drone]) // the drone was heading somewhere
     and ( sqrt((pre(pos[drone,1])-pre(setx[drone]))^2
               + (pre(pos[drone,2])-pre(sety[drone]))^2
               + (pre(pos[drone,3])-pre(setz[drone]))^2 )
                    < arrivalThreshold ) // drone has arrived
          ) then
     isBusy[drone] := false; // free the drone
     for area in 1:p.nAreas loop
          if ( hasBeenSelected[area] == drone ) then
               hasBeenSelected[area] : =0; // free the area
               hasBeenVisited[area] := true;
               lastVisited[area] := time;
               break;
elseif ( pre(isBusy[drone]) ) then // didn't reach its setpoint
     // if it has been too long
     if ( abs( time - pre(setpointChangeTime[drone]) )
               > p.arrival_timeout ) then
          isBusy[drone] := false; // free the drone
          for area in 1:p.nAreas loop // free the area
               if ( hasBeenSelected[area] == drone ) then
                    hasBeenSelected[area] := 0; break;
```

2.2.3 Invio al *server*

A questo punto il *CC* può comunicare al *server* la posizione dei droni ancora operativi.

L'invio del messaggio al *server* avviene grazie alla funzione ***update_drone_pos***, maggiori dettagli sul funzionamento della funzione nel capitolo ***2.5.1***.

```
if ( not(pre(isDead[drone])) ) then // send to DB
     err := update_drone_pos(drone, time, pos[drone]);
```

2.2.4 Controllo aree

Per ogni area, il *CC* chiede al *server* se nell'ultimo ***time_span*** l'area è stata visitata da qualcuno, se la risposta è positiva allora si aggiorna la variabile con il tempo dell'ultima visita e si etichetta l'area come *visited*.

La comunicazione con il *server* avviene grazie alla funzione ***how_many_drones_in_area***, maggiori dettagli sul funzionamento della funzione nel capitolo ***2.5.1***.

```
// has area been visited yet?
if ( how_many_drones_in_area(area, pre(time_span)) > 0 ) then
     // some drones got in the area since the last time_span
     hasBeenVisited[area] := true;
     lastVisited[area] := time;
```

2.2.5 Controlli *loop*

Per ultima cosa il *CC* effettua un controllo generale per vedere se ogni area è stata visitata e se ci sono ancora droni non morti. Se ogni area è stata visitata, si resettano le variabili e si ricomincia da capo. Se non ci sono più droni in vita, si termina la simulazione.

```
// arena check
err := 0.0;
for area in 1:nAreas loop
     // there is at least one area still left unchecked
     if ( not(pre(hasBeenVisited[area])) ) then
          err := 100.0; break;

if ( pre(err) < 1.0 ) then // every area has been visited
     time_passed_since_last_loop := time - pre(time_span);
     time_span := time; // new cycle starts now
     hasBeenVisited := fill(false, nAreas);
     hasBeenSelected := fill(0, nAreas);

// drones check
if ( aliveDrones <= 0 ) then // every drone is dead
     disconnect_from_db();
     endSimulation:= true;
```

2.2.6 Scelta del nuovo *set point*

Quale area assegnare a quale drone è deciso pesando due possibili aree con un parametro normalizzato ***cc_choice***.

A quale delle due scelte il *CC* dovrebbe dare più peso?

I. L'area non ancora visitata, **più vicina al drone**.
II. L'area non ancora visitata, **lasciata scoperta da più a lungo**.

L'algoritmo tiene conto delle aree già visitate e delle aree che sono già state assegnate ad altri droni, in modo da non ripetersi. C'è la possibilità che ogni area non ancora visitata sia già stata assegnata ad altri droni, in tal caso l'algoritmo non assegna nessun nuovo *set point* al drone. Una volta trovata la scelta migliore (***exploitation***), il drone tira un dado e con probabilità ***p_worst*** prende la scelta peggiore (***explotation***).

In caso l'algoritmo non abbia trovato alcuna area da assegnare al drone, lo dirigerà verso un punto randomico dell'arena.

```
// if the drone is not already heading somewhere
if ( not(isBusy[drone]) and not(isDead[drone]) ) then
      // find best two areas to choose from
      for area in 1:p.nAreas loop
            // area has not been visited AND no drone sent there
            if ( not(hasBeenVisited[area])
                  and (hasBeenSelected[area] == 0) ) then
                  // closest unchecked area
                  is_area_closer(area, min_d_area, min_distance);
                  // longest unchecked area
                  longest_unchecked(area, unckd_area, max_time);

      // if an area has been selected for this drone
      if ( not(min_d_area == 0) and not(unckd_area == 0) ) then
            // weight choices
            choice_value_1 := min_distance*cc_choice;
            choice_value_2 := max_time * (1 - cc_choice);
            if ( choice_value_1 > choice_value_2 ) then
                  if ( myrandom() > p_worst ) then
                        // closest unchecked
                        go_to(drone, areaCenter(min_d_area));
                        hasBeenSelected[min_d_area] := drone;
                  else // exploration chance
                        // longest unchecked
                        go_to(drone, areaCenter(p, unckd_area));
                        hasBeenSelected[unckd_area] := drone;
            else
                  if ( myrandom() > p_worst ) then
                        // longest unchecked
                        go_to(drone, areaCenter(unckd_area));
                        hasBeenSelected[unckd_area] := drone;
                  else // exploration chance
                        // closest unchecked
                        go_to(drone, areaCenter(min_d_area));
                        hasBeenSelected[min_d_area] := drone;

      else // choose a random point
            go_to(drone, random_coordinates);

      setpointChanged[drone] := true;
      setpointChangeTime[drone] := time;
      isBusy[drone] := true;
```

La funzione ***is_area_closer*** si occupa di verificare se l'area in input è più vicina dell'area più vicina trovata finora.

```
input Integer area, Integer min_d_area, Real min_distance;
output Integer min_d_area, Real min_distance;
     // find area center
     (area_x,area_y,area_z) := areaCenter(area);
     // compute distance from drone
     tmp_distance := sqrt((pos[drone,1]-area_x)^2
          + (pos[drone,2]-area_y)^2
          + (pos[drone,3]-area_z)^2 );
     // compare with current min
     if ( tmp_distance < min_distance ) then
          min_distance := tmp_distance; // update min distance
          min_d_area := area; // update chosen area
     end if;
```

La funzione ***longest_unchecked*** si occupa di verificare se l'area in input è stata lasciata scoperta da più tempo di quella lasciata scoperta più a lungo trovata finora.

```
input Integer area, Integer unckd_area, Real max_time;
output Integer unckd_area, Real max_time;
     tmp_time := time-lastVisited[area]; // get time
     // compare
     if ( tmp_time > max_time ) then
          max_time := tmp_time; // update time
          unckd_area := area; // update area
     end if;
```

Infine la funzione ***go_to*** si occupa di assegnare il nuovo *set point* al drone.

```
input Integer drone, Real area_coordinates;
     setx[drone] := area_coordinates[1];
     sety[drone] := area_coordinates[2];
     setz[drone] := area_coordinates[3];
```

2.2.7 Ricarica batteria

Quando un drone raggiunge il livello critico di carica ***dangerousBatteryLevel*** viene subito indirizzato verso la *docking bay* per ricaricarsi fino a che non raggiunga una carica accettabile ***rechargedThreshold***.

Se meno del *30%* dei droni è in sosta nella *docking bay*, allora il drone potrà andare preventivamente a ricaricarsi la batteria con probabilità inversamente proporzionale a quanta carica gli rimane in quel momento.

$$P(d,t) = p_r \cdot \left(1 - \frac{b_d(t)}{C_d}\right)$$

Dove

p_r: parametro ***p_recharge*** che indica l'*upper bound* di questa probabilità

$b_d(t)$: la batteria del drone d al tempo t

C_d: la capacità massima della batteria del drone d

Questa fase segue immediatamente la fase di scelta del *set point*, quindi se non si dovesse cambiare quest'ultimo verrà inviato il *set point* scelto nella fase ***2.2.6***, dopodiché si passa allo stato ***30***.

```
if ( pre(isRecharging[drone]) ) then // drone is recharging
     if ( battery[drone] > rechargedThreshold ) then
          isRecharging[drone] := false;
          rechargingDrones := pre(rechargingDrones) - 1;
     elseif ( battery[drone] < rechargedThreshold ) then
          isRecharging[drone] := true;
          go_to(drone, docking_bay);

else // drone is not recharging not it needs to
     if ( battery[drone] < dangerousBatteryLevel ) then
          rechargingDrones := pre(rechargingDrones) + 1;
          isRecharging[drone] := true; isBusy[drone] := true;
          go_to(drone, docking_bay);

     elseif ( pre(rechargingDrones) < (nDrones*0.30+1) ) then
          if ( myrandom() < p_recharge *
                    (1.0 - (battery[drone]/capacity)) ) then
               // go recharge anyway
               isRecharging[drone]:=true; isBusy[drone]:=true;
               rechargingDrones := pre(rechargingDrones) + 1;
               go_to(drone, docking_bay);
```

2.2.8 Invio nuovi *set point*

Scelti i nuovi *set point* per i droni, al CC resta che inviarli ai droni, in modo del tutto analogo a come vengono comunicate le richieste ai drone, nello stato ***30*** si scrive il messaggio che verrà poi inviato nello stato ***40***.

Lo stato verrà poi impostato a ***00***.

2.3 Monitor

Il monitor è una componente scollegata da tutte le altre che si occupa di controllare il rendimento del sistema. Prende le informazioni che gli servono dal *server* (quindi con *query* sul *DB*) e calcola i seguenti fattori:

I. ***AvgDrones***: Numero atteso di droni fra tutte le aree
II. ***PrDrones***: Probabilità di trovare un drone nell'area
III. ***StdDev***: Deviazione standard del numero di droni di ogni area
IV. ***AvgNoDrone***: La media del tempo per cui ogni area è lasciata scoperta
V. ***MaxNoDrone***: Il tempo massimo per cui un'area è stata lasciata scoperta

Questi fattori saranno l'*output* finale del sistema e rappresentano le diverse funzioni obiettivo che, con i tool di ottimizzazione, vorremmo minimizzare (o massimizzare).

2.3.1 Raccolta informazioni

Ogni ***Tm*** battiti, il monitor usa la funzione ***how_many_drones_in_area*** per chiedere quanti droni sono presenti in ogni area e aggiornare i parametri che gli occorrono per i calcoli successivi, ovvero:

- ***current_drones***: quanti droni ci sono al momento nell'area
- ***beenVisited***: se l'area è stata visitata
- ***no_drone_time***: da quanto tempo l'area è stata lasciata scoperta

```
z := how_many_drones_in_area(area, (time-p.Tm) );
if ( z > 0 ) then
     current_drones[area] := z;
     beenVisited[area] := 1;
     no_drone_time[area] := 0.0;
else
     no_drone_time[area] := no_drone_time[area] + p.Tm;
end if;
```

2.3.2 Calcolo fattori

A quel punto si può procedere area per area e calcolare i fattori elencati sopra. Per risparmiare sulla memoria, utilizzeremo tecniche di *dynamic programming* che ci permettono di calcolare media e deviazione standard in modo iterativo, senza dover immagazzinare ogni lettura passata, nel seguente modo:

- Media di droni nell'area a al t-esimo controllo

$$\mu_a(t) = \frac{\mu_a(t-1) \cdot (t-1)}{t} + \frac{c_a(t-1)}{t}$$

Dove

$c_a(t)$: Numero di droni nell'area a al t-esimo controllo

- Probabilità di trovare un drone nell'area a al t-esimo controllo

$$P_a(t) = \frac{P_a(t-1) \cdot (t-1)}{t} + \frac{b_a(t-1)}{t}$$

Dove

$b_a(t)$: funzione indicatrice del fatto che almeno un drone abbia visitato o meno l'area a al t-esimo controllo

- Media tempo in cui l'area è stata lasciata scoperta al t-esimo controllo

$$s_a(t) = \frac{s_a(t-1) \cdot (t-1)}{t} + \frac{o_a(t-1)}{t}$$

Dove

$o_a(t)$: da quanto tempo l'area a è stata lasciata scoperta, al t-esimo controllo

- Deviazione standard del numero di droni nell'area al t-esimo controllo

$$\sigma_a(t) = \sqrt[2]{\left| c_a(t) - \left(\mu_a(t)\right)^2 \right|}$$

I **valori di output** del sistema sono quindi calcolati, ad ogni iterazione, nei seguenti modi:

- Numero atteso di droni fra tutte le *n* aree al *t*-esimo controllo

$$\mu(t) = \frac{1}{n} \sum_{a=1}^{n} \mu_a(t)$$

- Probabilità di trovare un drone in una delle *n* aree al *t*-esimo controllo

$$P(t) = \frac{1}{n} \sum_{a=1}^{n} P_a(t)$$

- Tempo atteso in cui le *n* aree sono state lasciate scoperte

$$s(t) = \frac{1}{n} \sum_{a=1}^{n} s_a(t)$$

- Massimo tempo per cui un'area è stata lasciata scoperta, fino al *t*-esimo controllo

$$S(t) = MAX\left(S(t-1),\ o_1(t),\ \ldots,\ o_n(t)\right)$$

- Deviazione standard del numero atteso di droni nelle *n* aree

$$\sigma(t) = \sqrt[2]{\left|\mu(t) - \mu(t-1)^2\right|}$$

2.3.3 Termine analisi

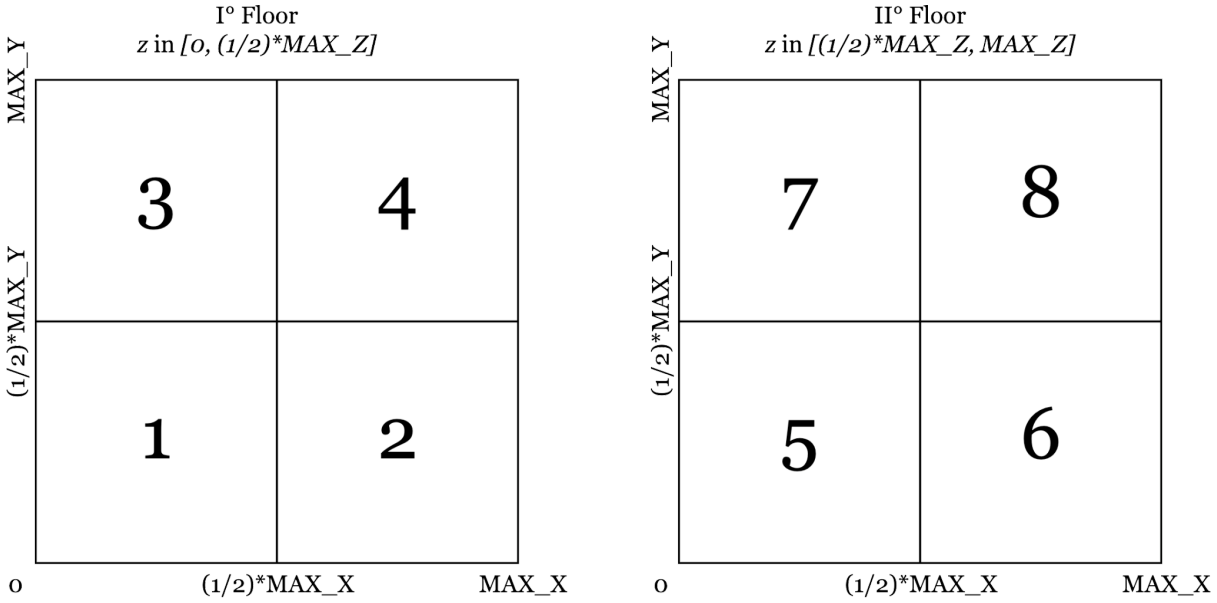
Raggiunto l'orizzonte di simulazione ***stop_simulation***, il *monitor* stampa i valori di output sul file ***outputs.txt***. Infine termina la simulazione.

2.4 Aree

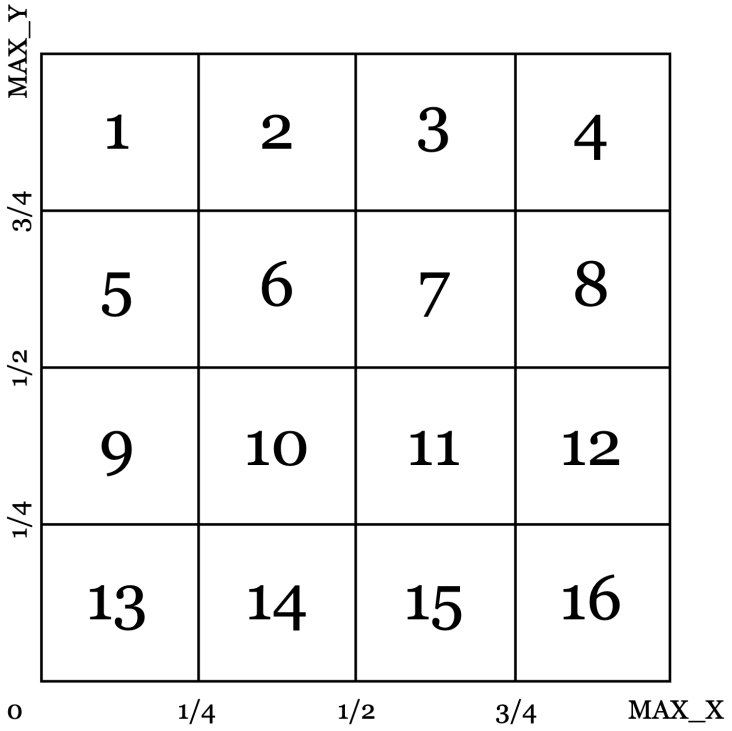
Ogni area è definita da *7* elementi:

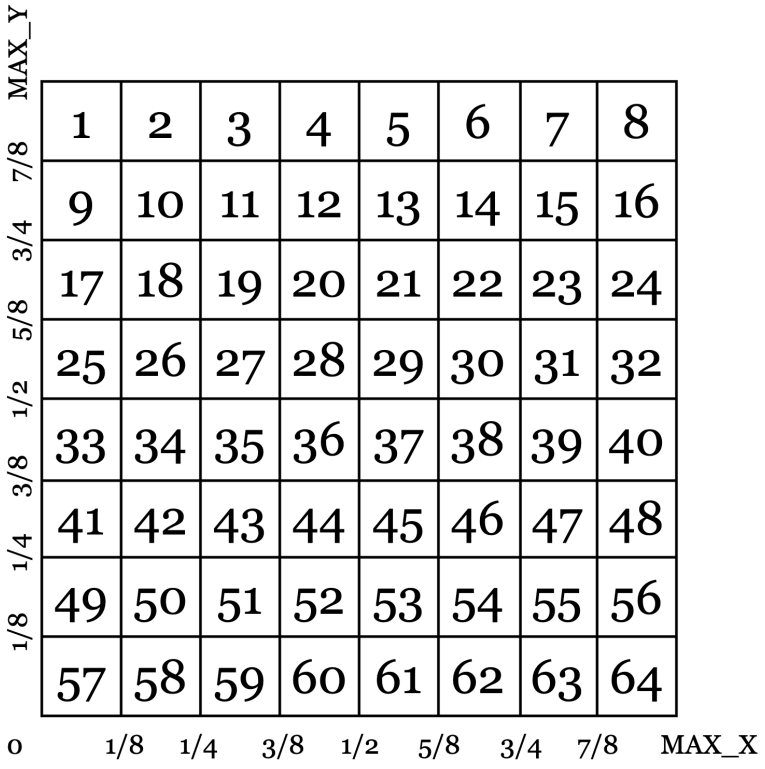
I. **ID**: un numero intero che identifica l'area
II. **Low$_x$**: il *lower bound* dell'area sulla coordinata *x*
III. **Up$_x$**: l'*upper bound* dell'area sulla coordinata *x*
IV. **Low$_y$**: il *lower bound* dell'area sulla coordinata *y*
V. **Up$_y$**: l'*upper bound* dell'area sulla coordinata *y*
VI. **Low$_z$**: il *lower bound* dell'area sulla coordinata *z*
VII. **Up$_z$**: l'*upper bound* dell'area sulla coordinata *z*

La configurazione di default per l'arena prevede *8* aree disposte nel seguente modo:



Gli iperparametri verranno calcolati basandosi sulla configurazione sopra, ma nel capitolo ***6*** ne verranno testate altre due rispettivamente con *16* e con *64* aree, disposte su un singolo piano nel seguente modo.





2.5 Cloud

Un *server* ed un *client* scritti in *C* che si occupano di fare da tramite fra il *CC* e il *Database*. Utilizzano le librerie di *C* ***unistd.h*** e ***arpa/inet.h***.

I messaggi sono una serie di *8 float* scritti su una stringa. Il primo valore rappresenta la *flag* dell'operazione, il resto sono argomenti. La flag di ogni operazione è:

- ***setup the database***: 00
- ***update the drone's position***: 01
- ***get the drone's position***: 02
- ***flush history***: 03
- ***create a new drone***: 04
- ***create a new area***: 05
- ***has the area been visited***: 06
- ***is the drone in the area***: 07
- ***turn off***: 08
- ***how many drones are in the area***: 09
- ***disconnect from DB***: 10

2.5.1 Client

Il *client* si connette al *server* all'inizio della simulazione e rimane connesso fino alla fine. In questo modo si velocizza l'esecuzione del modello dato che non c'è bisogno di perdere tempo a stabilire e a interrompere la connessione ad ogni chiamata.

Il modello *OpenModelica* chiama una delle funzioni elencate sopra e passa gli argomenti all'omonima funzione in *C*. Questa scriverà gli argomenti e aggiungerà la *flag* corretta dell'operazione, infine chiamerà la funzione ***send2server*** che si occupa di inviare il messaggio e ricevere la risposta del *server*.

2.5.1.1 Collegamento a Modelica

Il modello chiama le funzioni *C* tramite l'API fornita da *OpenModelica*, passando in input gli argomenti e ottenendo in output il risultato. Purtroppo questa API non permette di passare array, né in input né in output, in questi casi occorre definire più funzioni, una per ogni argomento in output. Tutte le funzioni di messaggistica con il server si trovano nel file ***client-socket.c***.

```
function update_drone_pos
input Integer drone_id;
input Real position_time;
input Real pos[3];
output Real result; // 0.0 if ok, 1.0 if error
external "C" result = update_drone_pos(drone_id, position_time,
                                       pos[1], pos[2], pos[3]);
     annotation(Include = "#include \"client-socket.c\"");
end update_drone_pos;
```

2.5.1.2 Funzione *send2server*

Per prima cosa la funzione stabilisce una connessione con il *server* utilizzando variabili di tipo *static* in modo da non dover perdere tempo per farlo ad ogni chiamata.

```
if ( connection_online == 0 ) {
      // setup socket
      server_sock = socket(AF_INET, SOCK_STREAM, 0);
      if (server_sock < 0) {
            perror("[-]Socket error");
            exit(1); }
      // initialize
      memset(&server_addr, '\0', sizeof(server_addr));
      server_addr.sin_family = AF_INET;
      server_addr.sin_port = server_port;
      server_addr.sin_addr.s_addr = inet_addr(server_ip);
      // connection to server
      connect(server_sock, (struct sockaddr*)&server_addr,
            sizeof(server_addr));
      connection_online = 1; }
// write message and send
from_args_to_buffer(args, ARGS_SIZE, buffer, BUFFER_SIZE);
send(server_sock, buffer, BUFFER_SIZE, 0);
// clear the buffer and receive
clear_string(buffer, BUFFER_SIZE);
recv(server_sock, buffer, BUFFER_SIZE, 0);
// translate
from_buffer_to_args(args, ARGS_SIZE, buffer);
```

Tradizionalmente le comunicazioni avvengono tramite stringhe, per questo motivo la prima azione di questa funzione è chiamata ***from_args_to_buffer*** che si occupa di impacchettare gli *8 float* in un'unica stringa di numeri separati da spazi.

```
void from_args_to_buffer(float* args, int ARGS_SIZE, char* buffer,
int BUFFER_SIZE) {
      // variables
      int ARG_LEN = 10+1+1; // num_len + comma + \0
      char arg2str[ARG_LEN];
      // for each argument
      for(int arg = 0; arg < ARGS_SIZE-1; arg++) {
            ftoa(args[arg], arg2str, 0); // convert arg to string
            strcat(buffer, arg2str); // append arg to buffer
            strcat(buffer, " "); // insert separation character
    }
      ftoa(args[ARGS_SIZE-1], arg2str, 0); // last arg to string
      strcat(buffer, arg2str); // append last arg to buffer
      strcat(buffer, "\0"); // insert termination character
} /* from args to buffer */
```

A questo punto la stringa ottenuta può essere inviata al *server* con la funzione di ***send***, immediatamente seguita dalla chiamata ***recv*** per ricevere appunto la risposta del *server*.

La risposta del server sarà anch'essa una stringa, che verrà spacchettata con la funzione ***from_buffer_to_args***. Finalmente potremo consegnare al modello la risposta del *server*.

```
void from_buffer_to_args(float* args, int ARGS_SIZE, char* buffer)
{
      // variables
      int ARG_LEN = 10+1+1; char* arg_str; float arg_f = 0.0;
      int i = 0;
      // unpack buffer into different arguments
      arg_str = strtok(buffer, " ");
      while ( arg_str != NULL && i < ARGS_SIZE ) {
            arg_f = atof(arg_str); // cast to float
            args[i] = arg_f; // insert argument into array
            arg_str = strtok(NULL, " "); // get next word
            i++; }
} /* from buffer to args */
```

2.5.2 Server

Il *server* accetta la connessione del *client* all'inizio della simulazione e la chiude alla fine di essa. Nel mentre si occupa di ricevere la stringa con gli argomenti, interpretarla e chiamare la corretta funzione del *Database*, nella libreria ***DB-api.c***.

Non essendoci (*as of now*) una chiamata specifica per interrompere la connessione senza chiudere tutto, il server apprende che la simulazione è terminata dal comportamento del *client*.

La ricezione dei messaggi avviene in modo del tutto analogo a come è stato descritto nel capitolo ***2.5.1.2***.

2.5.2.1 API Database

Il collegamento tra il *server* e il *Database* avviene grazie alla libreria *C* fornita da *PostgreSQL* chiamata ***libpq-fe.h*** che per funzionare necessita che un *Database PostgreSQL* sia già esistente.

Allora il nostro API permette la connessione al *Database* con la funzione ***connect_to_DB***.

```
PGconn* connect_to_DB() {
      PGconn *conn = PQconnectdb("user=<user>
            dbname=<nome db> password=<password>"); // credentials
      if (PQstatus(conn) == CONNECTION_BAD) {
            fprintf(stderr, "Connection to database failed: %s\n",
                  PQerrorMessage(conn));
            do_exit(conn); }
      int ver = PQserverVersion(conn);
      return conn;
} /* connect */
```

A quel punto le diverse funzioni nella nostra libreria *DB-api.c* si limitano a riempire le stringhe delle *query* associate con i parametri dati in input, e chiamare la funzione ***PQexec*** della libreria *libpq-fe.h* che si occupa di eseguire le *query* sul *DB* e ritornare il risultato. Tale risultato sarà poi interpretato e passato al *server* che a sua volta lo interpreterà e invierà al client con la funzione di *send*.

2.5.3 Alternativa in locale

La comunicazione *client-server* richiede molto tempo, anche dopo aver ottimizzato il modello e ridotto le comunicazioni al minimo. Per questo motivo vi è un'alternativa in locale, che simula il *database* con i file testuali ***DB_AREA.txt*** e ***DB_HISTORY.txt***.

Per attivare questa opzione, basta sostituire il file *client-socket.c* con il file ***db-mockup-explainable.c*** oppure ***db-mockup-fast.c*** all'interno delle funzioni *OpenModelica* nel file ***db-api.mo***. Questi nuovi file presenta la stessa API di *client-socket.c* ma sostituisce la funzione *send2server* con la funzione ***db_mockup*** che, appunto, simula il *server* e le *query* su file locali.

La differenza tra la versione *explainable* e quella *fast* è che la prima scrive tutti su file di testo che possono essere quindi consultati. La seconda invece utilizza una matrice *static* interna al programma che non è consultabile in quanto si distrugge alla terminazione della simulazione, però il programma è più veloce.

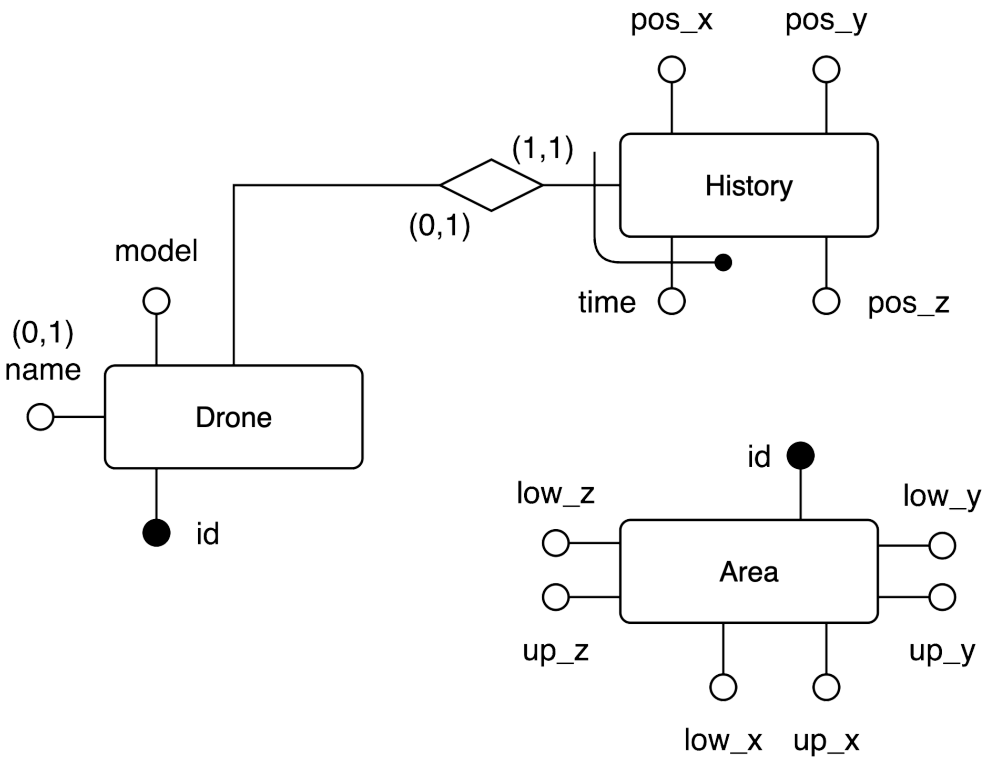
2.6 Database

Un database implementato con *PostgreSQL* che si occupa di mantenere lo storico delle posizioni dei droni e i limiti geografici delle aree.

Il *CC* dovrà fare quindi diverse query per inserire le posizioni dei droni del *DB* e per sapere se una certa area è stata visitata o meno.

2.6.1 Entity Relationship

2.6.1.1 Diagramma



2.6.1.2 Dizionario dei dati

Entità Drone

model / MidString
name / ShortString (0,1)
id / Integer

Entità Area

id / Integer	low_x / Real	low_y / Real	low_z / Real
	up_x / Real	up_y / Real	up_z / Real

Entità History

drone / Drone.id	time / Reale	
pos_x / Reale	pos_y / Reale	pos_z / Reale

2.6.1.3 Vincoli Esterni

[V.Area.Overlaps]

Le aree non devono sovrapporsi in nessun modo

$$\forall a_1, a_2, x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}, y_{1,1}, y_{1,2}, y_{2,1}, y_{2,2}, z_{1,1}, z_{1,2}, z_{2,1}, z_{2,2}$$
$$Area(a_1) \wedge Area(a_2)$$
$$\wedge\ low_x(a_1, x_{1,1}) \wedge low_x(a_2, x_{2,1}) \wedge up_x(a_1, x_{1,2}) \wedge up_x(a_2, x_{2,2})$$
$$\wedge\ low_y(a_1, y_{1,1}) \wedge low_y(a_2, y_{2,1}) \wedge up_y(a_1, y_{1,2}) \wedge up_y(a_2, y_{2,2})$$
$$\wedge\ low_z(a_1, z_{1,1}) \wedge low_z(a_2, z_{2,1}) \wedge up_z(a_1, z_{1,2}) \wedge up_z(a_2, z_{2,2})$$
$$\rightarrow$$
$$\left[\left(x_{2,2} < x_{1,1}\right) \vee \left(x_{2,1} > x_{1,2}\right)\right] \vee \left[\left(y_{2,2} < y_{1,1}\right) \vee \left(y_{2,1} > y_{1,2}\right)\right]$$
$$\vee \left[\left(x_{2,2} < x_{1,1}\right) \vee \left(x_{2,1} > x_{1,2}\right)\right]$$

[V.Area.Bounds]

I limiti delle aree devono essere coerenti (low_i < up_i)

$$\forall a_1, x_1, x_2, y_1, y_2, z_1, z_2$$
$$Area(a_1)$$
$$\wedge\ low_x(a_1, x_1) \wedge up_x(a_1, x_2) \wedge low_y(a_1, y_1) \wedge up_y(a_1, y_2)$$
$$\wedge\ low_z(a_1, z_1) \wedge up_z(a_1, z_2)$$
$$\rightarrow$$
$$\left(x_1 < x_2\right) \wedge \left(y_1 < y_2\right) \wedge \left(z_1 < z_2\right)$$

2.6.2 SQL

2.6.2.1 Tabelle relazionali

Drone (id: PosInt, name*: StringS, model*: StringM)
id è generato automaticamente dal DBMS

Area (id: Integer, low_x: Real, up_x: Real, low_y: Real, up_y: Real, low_z: Real, up_z: Real)

History (drone: Integer, time: Real, pos_x: Real, pos_y: Real pos_z: Real)
foreign key drone references Drone(id)
unique(pos_x, pos_y, pos_z)

2.6.2.2 Funzionalità

create_drone

descrizione: Aggiunge un drone nel DB.
input: drone_id: Integer
output: NULL
algoritmo:

```
1. INSERT INTO Drone (id)
   VALUES (:drone_id);
```

update_drone_pos

descrizione: Aggiorna lo storico del drone con la posizione corrente.
input: drone_id: Integer, modelica_time: Real, new_x: Real, new_y: Real, new_z: Real
output: NULL
algoritmo:

```
2. INSERT INTO History (drone_id, time, pos_x, pos_y, pos_z)
   VALUES (:drone_id, :modelica_time, :new_x, :new_y, :new_z);
```

get_drone_pos

descrizione: Chiede la posizione più recente del drone presente nel DB.
input: drone_id: Integer
output: drone_x: Real, drone_y: Real, drone_z: Real;
algoritmo:

1. answer ←
```
SELECT pos_x, pos_y, pos_z
FROM History
WHERE drone_id = :drone_id
      AND time = (
            SELECT max(time) FROM History );
```
2. drone_x = answer.pos_x
3. drone_y = answer.pos_y
4. drone_z = answer.pos_z

flush_history

descrizione: Cancella lo storico di ogni drone, e mette come posizioni iniziali, l'ultima posizione di ogni drone.
input: NULL
output: NULL
algoritmo:

1. `DELETE FROM History;`

create_area

descrizione: Crea una nuova area
input: id: Integer, low_x: Real, up_x: Real, low_y: Real, up_y: Real, low_z: Real, up_z: Real
output: NULL
algoritmo:

3.
```
INSERT INTO Area ( id, low_x,up_x, low_y,up_y, low_z,up_z )
VALUES ( :id, :low_x,:up_x, :low_y,:up_y, :low_z,:up_z );
```

has_area_been_visited

descrizione: Dato un lasso di tempo, dice se l'area è stata visitata (da un qualsiasi drone) nell'ultimo lasso di tempo specificato.
input: area_id: Integer, time_span: Real
output: BeenVisited: Boolean
algoritmo:

1. BeenVisited ←

```
EXISTS (
      SELECT *
      FROM History h, Area a
      WHERE  a.id = :area_id
            AND h.time >= :time_span
            AND h.pos_x >= a.low_x AND h.pos_x <= a.up_x
            AND h.pos_y >= a.low_y AND h.pos_y <= a.up_y
            AND h.pos_z >= a.low_z AND h.pos_z <= a.up_z
            AND h.drone_id > 0 );
```

has_drone_been_in_area

descrizione: Dato un lasso di tempo, dice se l'area è stata visitata dal drone.
input: drone_id: Integer, area_id: Integer, time_span: Real
output: BeenVisited: Boolean
algoritmo:

1. BeenVisited ←

```
EXISTS (
      SELECT *
      FROM History h, Area a
      WHERE  a.id = :area_id
            AND h.drone_id = :drone_id
            AND h.time >= :time_span
            AND h.pos_x >= a.low_x AND h.pos_x <= a.up_x
            AND h.pos_y >= a.low_y AND h.pos_y <= a.up_y
            AND h.pos_z >= a.low_z AND h.pos_z <= a.up_z
);
```

how_many_drones_in_area

descrizione: Dato un lasso di tempo, ritorna il numero di droni passati per l'area.
input: drone_id: Integer, area_id: Integer, time_span: Real
output: droneAmount: Integer
algoritmo:

2. droneAmount ←

```
SELECT COUNT(*)
FROM (
      SELECT *
      FROM History h, Area a
      WHERE  a.id = :area_id
            AND h.drone_id = :drone_id
            AND h.time >= :time_span
            AND h.pos_x >= a.low_x AND h.pos_x <= a.up_x
            AND h.pos_y >= a.low_y AND h.pos_y <= a.up_y
            AND h.pos_z >= a.low_z AND h.pos_z <= a.up_z
            GROUP BY h.drone_id );
```

2.6.2.3 Trigger

Area_Bounds

Operazione: inserimento ennupla in Area
Istante: prima dell'inserimento
Funzione:

1. `new` ← ennupla che si vuole inserire
2.
```
isError ←
      ( ( new.low_x >= new.up_x )
            OR ( new.low_y >= new.up_y )
            OR ( new.low_z >= new.up_z ) );
```
3. `if ( isError )` then *blocca operazione*
4. `else` *permetti operazione*

3. Nevergrad

3.1 Abstract

Nevergrad è una libreria di *Python*, sviluppata da *Meta*, che utilizza algoritmi di *IA* (simili a quelli visti sopra) per ottimizzazioni *black-box* senza gradienti, da qui il nome.

L'obiettivo di *Nevergrad* è trovare la combinazione migliore di iperparametri per minimizzare l'errore (***loss***) di un modello, o massimizzare le performance (***gain***) offrendo una vasta gamma di algoritmi diversi di ottimizzazione tra cui scegliere.

3.2 Algoritmi Base

I molteplici algoritmi di *Nevergrad* hanno diversi pro e contro, per questo ho modellato il problema più volte.

La nostra funzione da ottimizzare si chiama ***`model`***, si occupa di chiamare il modello passandogli i valori degli iperparametri tramite un file testuale, in modo che Modelica possa leggerlo. A quel punto esegue la simulazione, ne legge i risultati e infine ritorna il valore della variabile del *monitor* che ci interessa massimizzare, ovvero la funzione obiettivo ***obj***.

3.2.1 NGOpt

NGOpt sta per *Nevergrad Optimizer* ed è un *wizard* che seleziona l'algoritmo migliore al posto nostro, dato quello che il programma sa della funzione da ottimizzare.
Tiene conto di:

- Numero di variabili da ottimizzare
- Tipo di variabili da ottimizzare
- Numero di trial (*budget*)
- Possibilità di parallelizzare (*n_worker*)

NGOpt non ha informazioni a priori sulla presenza o meno di rumore nella funzione.
[NEV2018]

Con *budget* di *10·000*:

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.0033	0.964	6	185	0.49	018
p_worst	0.001					
p_rec	0.00218					

3.2.2 Cobyla

Cobyla sta per *Constrained Optimization BY Linear Approximation*.
É un algoritmo *model-based* che approssima iterativamente il problema utilizzando tecniche di *linear programming* (LP) e alla fine di ogni *trial* estrae la prossima configurazione da provare, basandosi sulle approssimazioni fatte fino a quel punto. [COB2007]

Con *budget* di *10·000*.

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.0137	0,985	7,5	225	0,55	0,30
p_worst	0.005					
p_rec	0.00311					

Cobyla suggerisce quindi di dare più peso alla scelta 2 e di aumentare il tempo di aggiornamento del *CC* risultando quindi in meno richieste inviate.

All'inizio di questo progetto il modello non aveva alcun rumore, rendendo *Cobyla* un'ottima scelta per l'ottimizzazione dei parametri. Nel corso dello sviluppo il modello ha acquisito nuovi parametri che generano rumore, rendendo questo algoritmo non più adatto alla scelta degli iperparametri. Ho comunque deciso di continuare ad utilizzare *Cobyla* in ogni versione del modello in modo da poter poi confrontare i risultati con altri algoritmi.

Un'alternativa a *Cobyla*, fra quelle presenti su *Nevergrad*, è l'algoritmo di *Michael J. D. Powell*.

3.2.3 Powell

Questo algoritmo cerca la configurazione migliore definendo due vettori h_1, h_2 con direzioni randomiche, nello spazio delle configurazioni ed effettuando la ricerca lungo le direzioni di questi vettori, fino a convergere in un minimo locale. [POW1964]
Dato un budget abbastanza alto e alcuni restart, *Powell* ha riportato ottimi risultati.

Con *budget* di *10·000*.

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.0060	0.958	6	200	0.55	0.19
p_worst	0.1165					
p_rec	0.00186					

3.3 Vincoli

Si può notare sperimentalmente come Nevergrad ritorni una configurazione ottima per minimizzare/massimizzare la funzione obiettivo ma che performi molto male sugli altri output del sistema. Tool di black-box optimization come NOMAD (che vedremo nel prossimo capitolo) permettono di definire vincoli da dover soddisfare, in modo da ottenere una configurazione che sia in grado di ottenere buoni risultati su più fronti.

L'API di Nevergrad fa sì che la funzione da minimizzare possa ritornare un unico valore Reale, per questo motivo per introdurre i vincoli su questo tool ci occorre definire la funzione obiettivo finale come combinazione lineare dei diversi fattori, ciascuno pesato con un adeguato moltiplicatore di Lagrange l_i. [LAG2009]

I vincoli del modello sono:

- $\boldsymbol{c_1}$*: AvgDrones* deve essere maggiore di 0,8
- $\boldsymbol{c_2}$*: AvgNoDrone* deve essere minore di 30
- $\boldsymbol{c_3}$*: MaxNoDrone* deve essere minore di 180
- $\boldsymbol{c_4}$*: PrDrones* deve essere maggiore di 0,2
- $\boldsymbol{c_5}$*: StdDev* minore di 0,5
- $\boldsymbol{c_6}$*: cc_choice* deve essere compreso tra il 10% e il 90% del suo dominio
- $\boldsymbol{c_7}$*: p_worst* deve essere compreso tra il 20% e il 80% del suo dominio

La modellazione scelta è stata la seguente:

$$gain = o \cdot l_0 + c_1 \cdot l_1 + c_2 \cdot l_2 + c_3 \cdot l_3 + c_4 \cdot l_4 + c_5 \cdot l_5 + c_6 \cdot l_6 + c_7 \cdot l_7$$

Dove

o: la funzione obiettivo, *AvgDrones*
μ: l'output *AvgDrones*
s: l'output *AvgNoDrone*
S: l'output *MaxNoDrones*
P: l'output *PrDrones*
σ: l'output *AvgDrones*
x_i: l'i-esimo argomento
L_i: il *Lower Bound* dell'i-esimo argomento
U_i: l'*Upper Bound* dell'i-esimo argomento

$$c_1 = 0.8 - \mu \qquad c_2 = (s - 30)\frac{1}{10} \qquad c_3 = (S - 180)\frac{1}{10}$$

$$c_4 = 0.20 - P \qquad c_5 = \sigma - 0.5$$

$$c_6 = \left|\left(\frac{x_1 - L_1}{U_1 - L_1} \cdot 100\right) - 50\right| + 40 \qquad c_7 = \left|\left(\frac{x_2 - L_2}{U_2 - L_2} \cdot 100\right) - 50\right| + 40$$

$$l_0 = 0,6 \qquad l_1 = 0,6 \qquad l_2 = 0,1 \qquad l_3 = 0,025$$

$$l_4 = 0,2 \qquad l_5 = 0,2 \qquad l_6 = 0,025 \qquad l_7 = 0,025$$

L'obiettivo è quello di ritornare un valore finale il cui 50% sia costituito dalla funzione obiettivo, il 40% sia costituito dai vincoli c_1,..., c_5 ed il restante 10% dai vincoli c_6, c_7.
A tale scopo ho adattato i moltiplicatori di Lagrange in modo da pesare adeguatamente i valori che non essendo normalizzati hanno range molto differenti. Ad esempio il vincolo c_3 può oscillare da 0 a anche 1000 a seconda dell'esecuzione.

Dopo diverse configurazioni, questa si è rivelata essere la più bilanciata e in grado di pesare adeguatamente i diversi vincoli in modo che l'ottimizzatore non trascuri la funzione obiettivo per soddisfare i vincoli con valori più influenti o viceversa.

I nuovi test sono stati effettuati sempre con budget di *10·000*.

NGOpt

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.5342	0.875	4.5	134	0.60	0.33
p_worst	0.1485					
p_rec	0.00147					

Cobyla

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.1824	0.912	6	167	0.55	0.28
p_worst	0.099					
p_rec	0.00298					

Powell

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.4999	0.875	1.6	133	0.59	0.33
p_worst	0.1886					
p_rec	0.00079					

3.4 Rumore

Il modello presenta però il grosso problema del rumore. Dati i numerosi parametri casuali, e anche la probabilità di malfunzionamento dei droni, è possibile che una stessa configurazione possa portare a risultati contrastanti.

Il valore più sensibile al rumore è ***MaxNoDrone***, la variabile che monitora il tempo massimo (in secondi) in cui un'area è stata lasciata scoperta. Questo a causa dell'elevato tempo di ricarica di ogni singolo drone (2 ore), infatti se tutti i droni dovessero andare a ricaricarsi nello stesso momento il valore di *MaxNoDrone* può arrivare fino a *6000s* mentre nei migliori casi registrati si ferma a *130s*.

Per mitigare questo problema ho inserito un modulo nella funzione python da dover ottimizzare che si occupa di eseguire la simulazione ***n_samples*** volte e di ritornare il valore medio di ogni variabile in uscita dal monitor. In questo modo si andrà a ridurre l'impatto negativo di esecuzioni particolarmente sfortunate e anche l'impatto positivo di esecuzioni particolarmente fortunate. Di default *n_samples* ha valore 10.

Il budget è stato dimezzato a *5'000* a causa degli elevati tempi di esecuzione che portavano a crash della macchina. Maggiori dettagli al capitolo *7*.
I nuovi valori suggeriti sono stati i seguenti:

NGOpt

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.50114	0.85730	4.1	139.6	0.59871	0.34981
p_worst	0.15120					
p_rec	0.00484					

Cobyla

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.48956	0.85116	4.5	148	0.59794	0.35596
p_worst	0.14902					
p_rec	0.01356					

Powell

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.48557	0.84278	3.225	148.4	0.58784	0.36414
p_worst	0.11837					
p_rec	0.01224					

4. NOMAD

4.1 Abstract

NOMAD è un implementazione in C/*C++*/*Python* degli algoritmi di *Mash Adaptive Direct Search* (MADS), ovvero algoritmi ideati per l'ottimizzazione Black-Box di funzioni particolarmente costose con possibilità di rumore.

Il nome *NOMAD* infatti è l'acronimo di *Nonlinear Optimization by Mesh Adaptive Direct Search*. Esistono diverse versioni di NOMAD, in questa relazione utilizzerò NOMAD 4. La più recente. [GER2020]

4.2 Modellazione

Per prima occorre la funzione ***obj*** da ottimizzare. Avendo già collegato il sistema a *Python* per *Nevergrad*, per *NOMAD* ho utilizzato lo stesso linguaggio.
A differenza di *Nevergrad*, su *NOMAD* la funzione da ottimizzare deve prendere in input un oggetto di *PyNomad* e deve dare in output una stringa del formato corretto.

In compenso però NOMAD permette di definire ulteriori vincoli sulla funzione oltre a quelli di *upper bound* e *lower bound*. Questi vincoli devono essere valutati all'interno della funzione e scritti nella stringa dell'output. Se il vincolo ha un valore > *0* allora è considerato violato.

Per sfruttare al meglio *NOMAD* ho deciso di scrivere un vincolo c_i per ogni variabile di output del *monitor*. I vincoli sono:

- $\boldsymbol{c_1}$*: AvgDrones* deve essere maggiore di 0,8
- $\boldsymbol{c_2}$*: AvgNoDrone* deve essere minore di 30
- $\boldsymbol{c_3}$*: MaxNoDrone* deve essere minore di 180
- $\boldsymbol{c_4}$*: PrDrones* deve essere maggiore di 0,2
- $\boldsymbol{c_5}$*: StdDev* minore di 0,5
- $\boldsymbol{c_6}$*: cc_choice* deve essere compreso tra il 10% e il 90% del suo dominio
- $\boldsymbol{c_7}$*: p_worst* deve essere compreso tra il 20% e il 80% del suo dominio

NOMAD cerca la configurazione migliore che rispetti tutti i vincoli ma comunica nell'output anche la configurazione migliore che però non ha rispettato i vincoli, e di quanto.

I vincoli scelti sono di tipo *Progressive Barrier* (*PB*), ovvero rappresentano la violazione (o non) progressivamente e non con booleani. Per una visione binaria dei vincoli si utilizza *Extreme Barrier* (*EB*).

Con un budget di *10·000* e la funzione come obiettivo *obj* da massimizzare, ***AvgDrones***.

Best Feasible

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.383	0.942	1.5	137	0.61	0.36
p_worst	0.177					
p_rec	0.00335					

Best Infeasible

Con $h = 0.04$

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.383	0.847	4.5	164	0.61	0.34
p_worst	0.177					
p_rec	0.00335					

Si può notare come la seconda configurazione, seppur identica a quella feasible, abbia violato i vincoli. Questo è chiaramente dovuto al rumore del modello.

Per nostra fortuna gli algoritmi *MADs* sono resistenti al rumore e l'ottimizzatore ha pensato bene di ritentare la stessa configurazione più volte, evidentemente perché portava a risultati in generale migliori rispetto alle altre configurazioni.

Anche per *NOMAD* occorre però mitigare il rumore del modello, come è stato fatto per i test con *Nevergrad*.

4.3 Rumore

Come per *Nevergrad*, anche con *NOMAD*, per mitigare il rumore del modello ho inserito un modulo nella funzione python da dover ottimizzare che si occupa di eseguire la simulazione ***n_samples*** volte e di ritornare il valore medio di ogni variabile in uscita dal monitor. In questo modo si andrà a ridurre l'impatto negativo di esecuzioni particolarmente sfortunate e anche l'impatto positivo di esecuzioni particolarmente fortunate. Di default *n_samples* ha valore 10.

Il budget è stato dimezzato a *5'000* a causa degli elevati tempi di esecuzione che portavano a crash della macchina. Maggiori dettagli al capitolo *7*. I nuovi valori suggeriti sono stati i seguenti.

Best Feasible

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.101	0.90125	12.475	179.3	0.53745	0.29578
p_worst	0.09					
p_rec	0.0004					

Best Infeasible

Con $h = 102.31$

Param	Value	(obj) Avg Drones	Avg No Drone	Max No Drone	Pr Drones	Std Dev
cc_choice	0.00072	0.89562	7.4	195.2	0.4907	0.30448
p_worst	0.08655					
p_rec	0.00016					

5. Confronto tool di BBO

5.1 Risultati

Possiamo ora confrontare i risultati ottenuti con i diversi tool e algoritmi di black-box optimization. Le soluzioni proposte per ogni algoritmo sono quelle che:

I. Hanno vincoli.
II. Rispettano tutti i vincoli.
III. Hanno provato la stessa configurazione ***n_samples*** (5) ad ogni trial

Con un *budget* di 5·000 trial ed un orizzonte di simulazione di *14* ore, i risultati sono stati:

Optimizer		**Hyperparameters**		
		cc_choice	***p_worst***	***p_rec***
Nevergrad	***NGOpt***	0.50114	0.1512	0.00484
	Cobyla	0.48956	0.14902	0.01356
	Powell	0.48557	0.11837	0.01224
NOMAD	***MAD***	0.101	0.09	0.0004

Optimizer		***Starting Points***	***Trial Time***	***CPU Time***	***RAM (MB)***
Nevergrad	***NGOpt***	11	30.68	42 : 36 : 39	115.96
	Cobyla	13	29.09	40 : 24 : 09	309.23
	Powell	16	28.38	38 : 27 : 12	102.48
NOMAD	***MAD***	15	23.53	32 : 40 : 38	62.18

Dai test si evince come *NOMAD* abbia suggerito una configurazione degli iperparametri in meno tempo e utilizzando meno memoria rispetto agli altri algoritmi.

Ricordo il significato di ogni vincolo del modello:

- c_1: *AvgDrones* deve essere maggiore di 0,8
- c_2: *AvgNoDrone* deve essere minore di 30
- c_3: *MaxNoDrone* deve essere minore di 180
- c_4: *PrDrones* deve essere maggiore di 0,2
- c_5: *StdDev* minore di 0,5
- c_6: *cc_choice* deve essere compreso tra il 10% e il 90% del suo dominio
- c_7: *p_worst* deve essere compreso tra il 20% e il 80% del suo dominio
- ***obj***: massimizzare l'output associato ad *AvgDrones*

Un valore negativo indica il soddisfacimento del vincolo.

Opt	***obj***	**Constraints**						
		c1	***c2***	***c3***	***c4***	***c5***	***c6***	***c7***
NGOpt	0.8573	-0.0445	-2.59	-4.04	-0.3987	-0.1501	-39.935	-44.764
Cobyla	0.8511	-0.0011	-2.55	-3.2	-0.3979	-0.1440	-38.905	-44.506
Powell	0.8427	-0.0427	-2.6774	-3.1599	-0.3878	-0.1258	-38.505	-34.255
MAD	0.9012	-0.1012	-1.7525	-0.07	-0.3374	-0.204	-0.1	-22.586

Per quanto riguarda il rispetto dei vincoli, l'algoritmo *Powell* ha riportato i risultati migliori, suggerendo la configurazione che rispetta maggiormente i vincoli definiti.

5.2 Scalabilità

Infine per testare quanto le configurazioni proposte siano scalabili, sono state effettuate più simulazioni con numero crescente di aree da coprire e con diverso numero di droni.

La percentuale di droni fa riferimento al numero di aree nell'arena.

10% di Droni

N° Aree	N° Droni	Opt	Model Output				
			Avg Drones	***Avg No Drone***	***Max No Drone***	***Pr Drones***	***Std Dev***
8	**1**	***NGOpt***	0.21963	41.925	2029.4	0.21963	0.41384
		Cobyla	0.21048	11.4	2223.3	0.21048	0.40766
		Powell	0.20737	9.75	3299.7	0.20737	0.40533
		MAD	0.21013	8.55	2135.8	0.21013	0.40736
16	**2**	***NGOpt***	0.24749	10.2125	1791.2	0.1995	0.4315
		Cobyla	0.24505	10.075	1032.3	0.19923	0.43009
		Powell	0.24834	9.962	822.3	0.20099	0.43202
		MAD	0.25459	9.375	1096.6	0.18764	0.43558
64	**7**	***NGOpt***	0.26552	54.7531	779.2	0.10528	0.44162
		Cobyla	0.26311	70.4124	730.1	0.10555	0.44029
		Powell	0.29824	12.8281	665.8	0.11423	0.45746
		MAD	0.29789	12.0343	635.4	0.11435	0.45732

20% di Droni

N° Aree	N° Droni	Opt	Model Output				
			Avg Drones	***Avg No Drone***	***Max No Drone***	***Pr Drones***	***Std Dev***
8	**2**	***NGOpt***	0.43888	7.6375	488.9	0.39217	0.49606
		Cobyla	0.42784	16.8375	1024.8	0.37870	0.49471
		Powell	0.4316	24.3	527.2	0.37829	0.49534
		MAD	0.43377	31.8625	1689.7	0.33697	0.49548
16	**4**	***NGOpt***	0.50039	8.625	454	0.32835	0.49998
		Cobyla	0.49964	70.5126	297.4	0.32465	0.49987
		Powell	0.49804	22.4126	305.7	0.31620	0.49996
		MAD	0.51765	8.725	1336.1	0.27888	0.49982
64	**13**	***NGOpt***	0.49145	9.6	365	0.33746	0.47889
		Cobyla	0.49964	20.4	305.7	0.32354	0.48456
		Powell	0.52713	18.3	297	0.32731	0.49885
		MAD	0.50652	19.2	822.4	0.28787	0.49561

40% di Droni

N° Aree	N° Droni	Opt	Model Output				
			Avg Drones	***Avg No Drone***	***Max No Drone***	***Pr Drones***	***Std Dev***
8	**4**	***NGOpt***	0.8573	4.1	139.6	0.59871	0.34981
		Cobyla	0.85116	4.5	148	0.59794	0.35596
		Powell	0.84278	3.225	148.4	0.58784	0.3641
		MAD	0.90125	12.475	179.3	0.53745	0.29578
16	**7**	***NGOpt***	0.88422	6.3375	237.4	0.49323	0.31953
		Cobyla	0.88033	6.79375	252.8	0.48749	0.32388
		Powell	0.88269	20.7687	242.5	0.47275	0.32178
		MAD	0.90128	19.51875	278	0.39039	0.29863
64	**26**	***NGOpt***	0.80821	20.3	145.3	0.56962	0.33565
		Cobyla	0.83207	6.4	158	0.58885	0.34605
		Powell	0.90389	4.3	176	0.57893	0.3552
		MAD	0.89034	17.8	158.2	0.54836	0.30489

Si può notare come dato il rapporto droni-aree, il modello riesca a garantire risultati stabili e costanti, nonostante l'incremento del numero di aree da coprire sulla stessa superficie.

6. Conclusione

Nel corso di questa relazione è stato ideato il modello di un Centro di Controllo per UAV con *OpenModelica,* che oltre a gestire un numero variabile di droni e aree da pattugliare, sia in grado di comunicare il suo operato ad un database nel *cloud* e insieme ad esso valutare le prestazioni generali del sistema.

Nella seconda fase della relazione mi sono occupato di regolare gli iperparametri del sistema in modo da massimizzarne le performance utilizzando i diversi algoritmi di ottimizzazione *black-box* compresi nei *tool* di ***Nevergrad*** e ***NOMAD***.

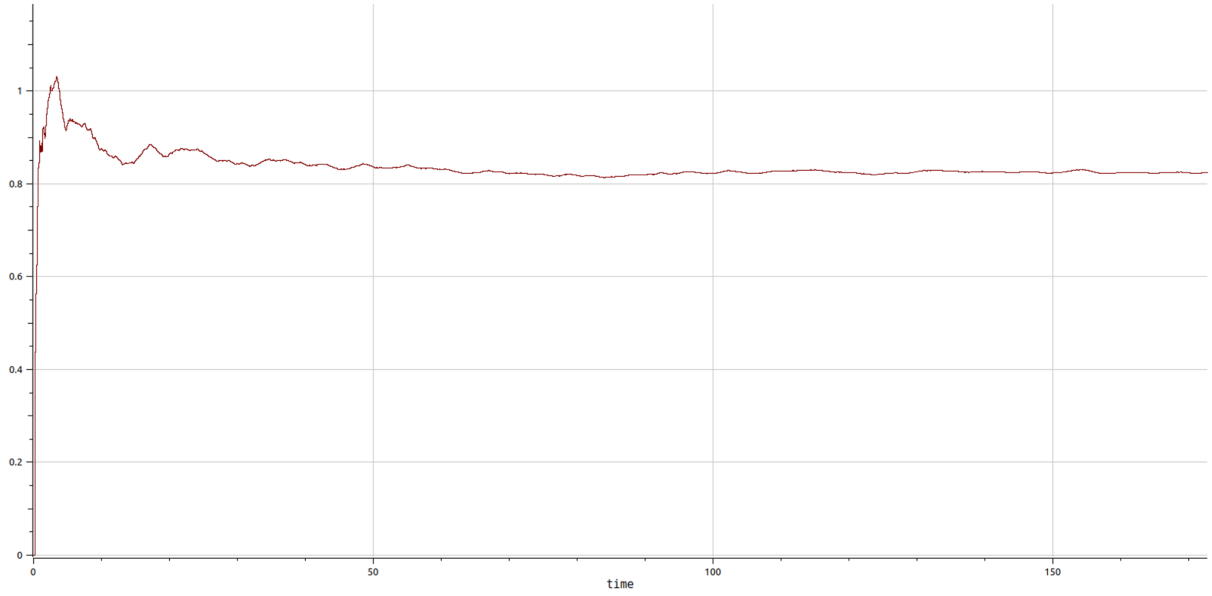
Le tecniche di *black-box optimization* descritte nei capitoli precedenti hanno permesso di regolare gli iperparametri del sistema in modo da ottenere risultati ottimi e che rispettino consistentemente i vincoli imposti del sistema.

Dati i numerosi esperimenti, l'algoritmo la cui configurazione si è rivelata essere la migliore per questo problema specifico è stato *Powell*. Questa configurazione si è infatti dimostrata quella complessivamente migliore e in grado di portare ai risultati desiderati per i vari output del sistema.

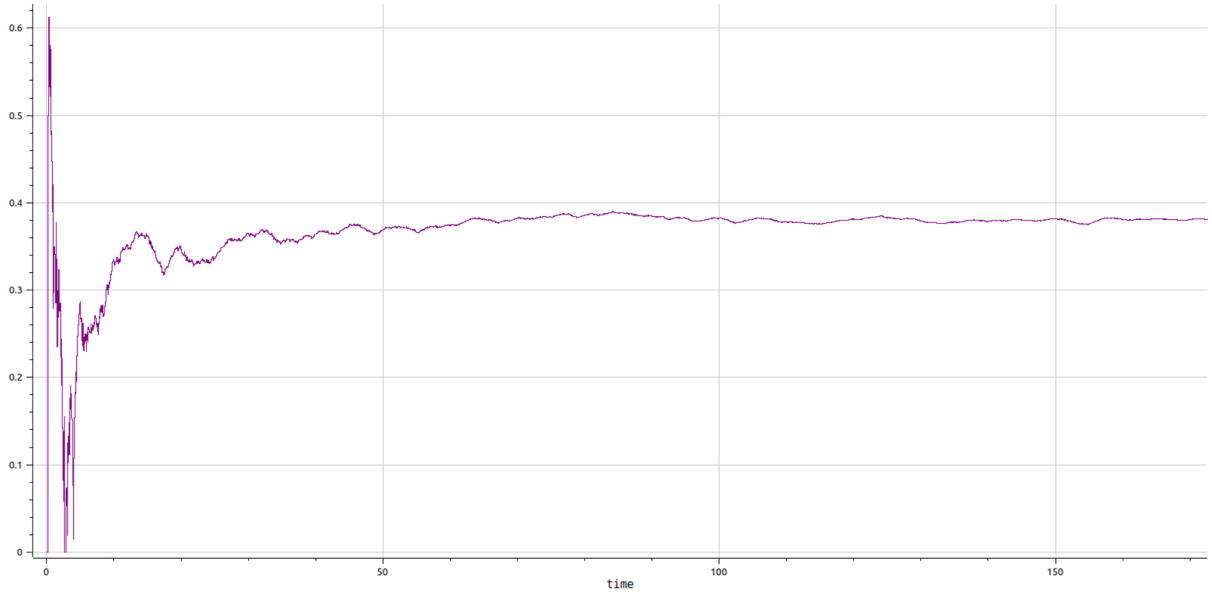
cc_choice	***p_worst***	***p_rec***
0.48557	0.11837	0.01224

I risultati ottenuti con questi parametri, simulando il modello per una durata di ben 2 giorni, con l'ambiente di *default* di 8 aree e 4 droni, sono i seguenti.

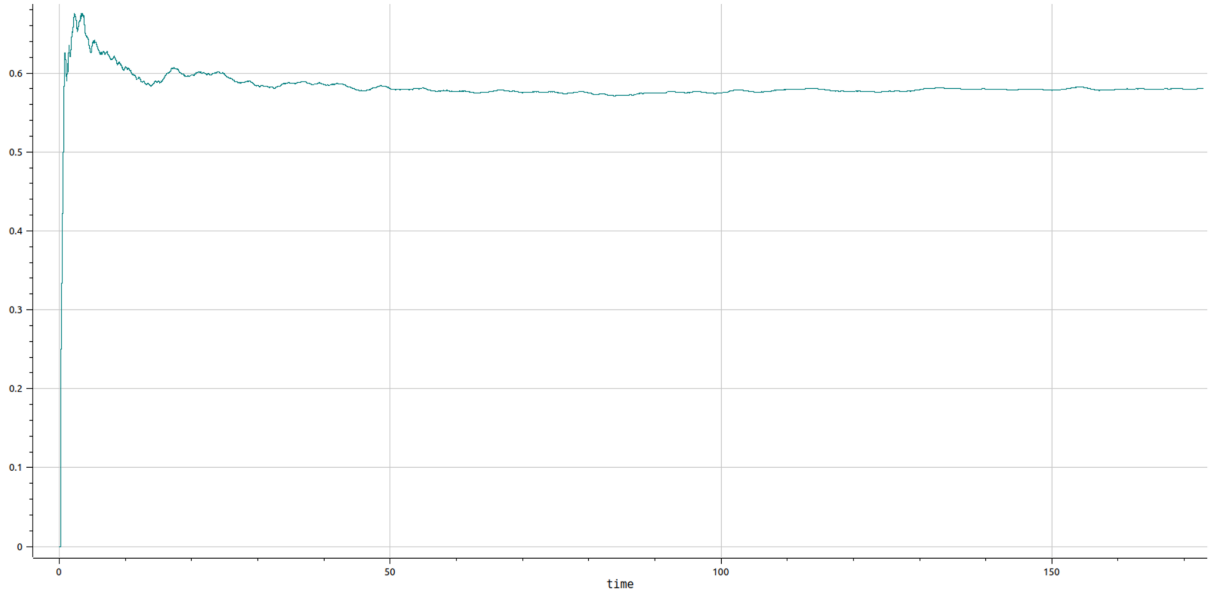
AvgDrones



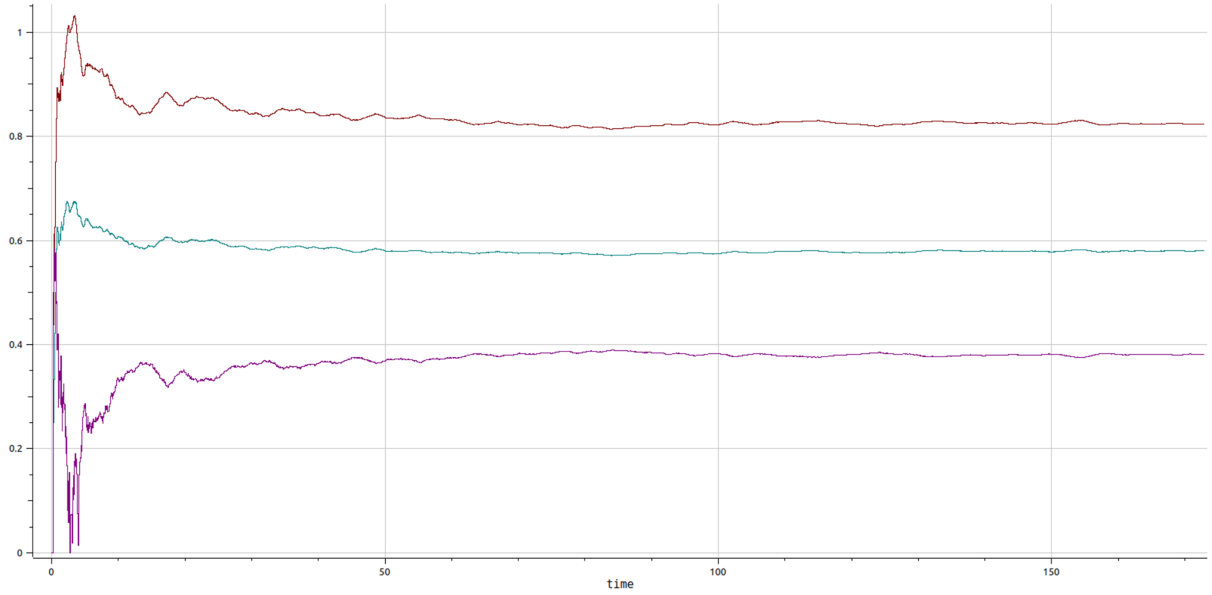
StdDev



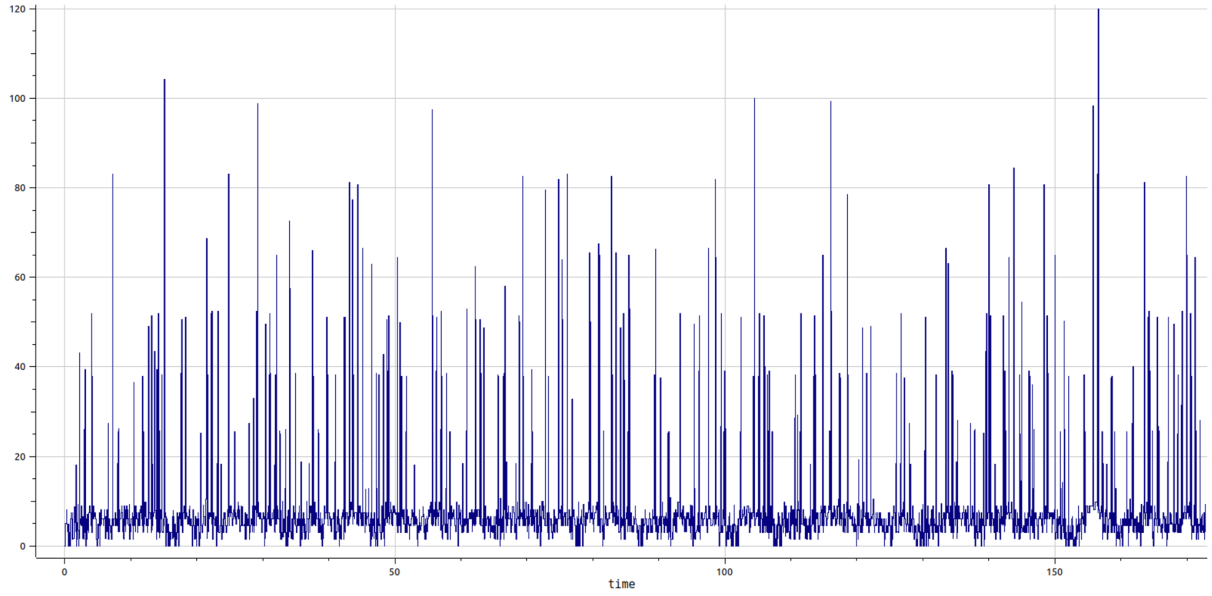
PrDrones



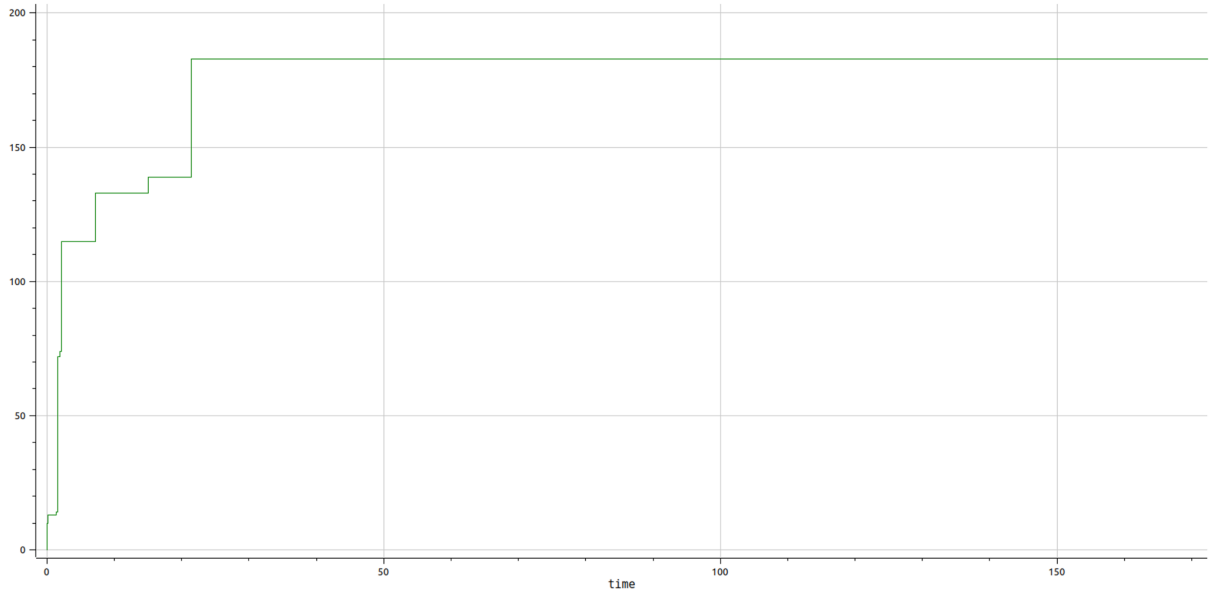
AvgDrones + StdDev + PrDrones



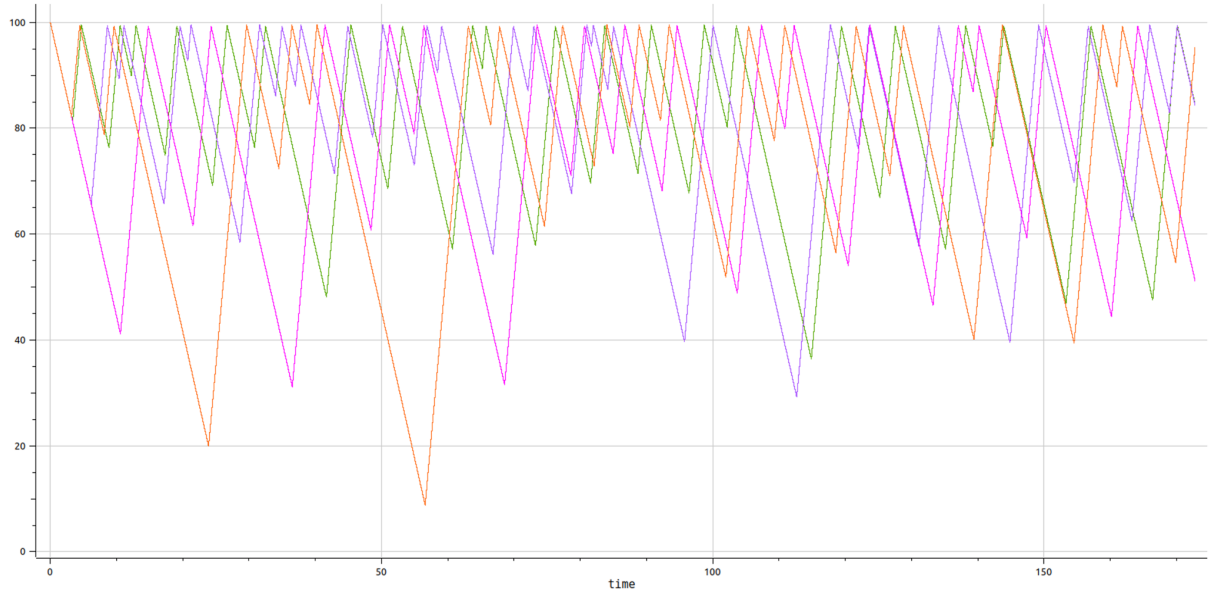
AvgNoDrone



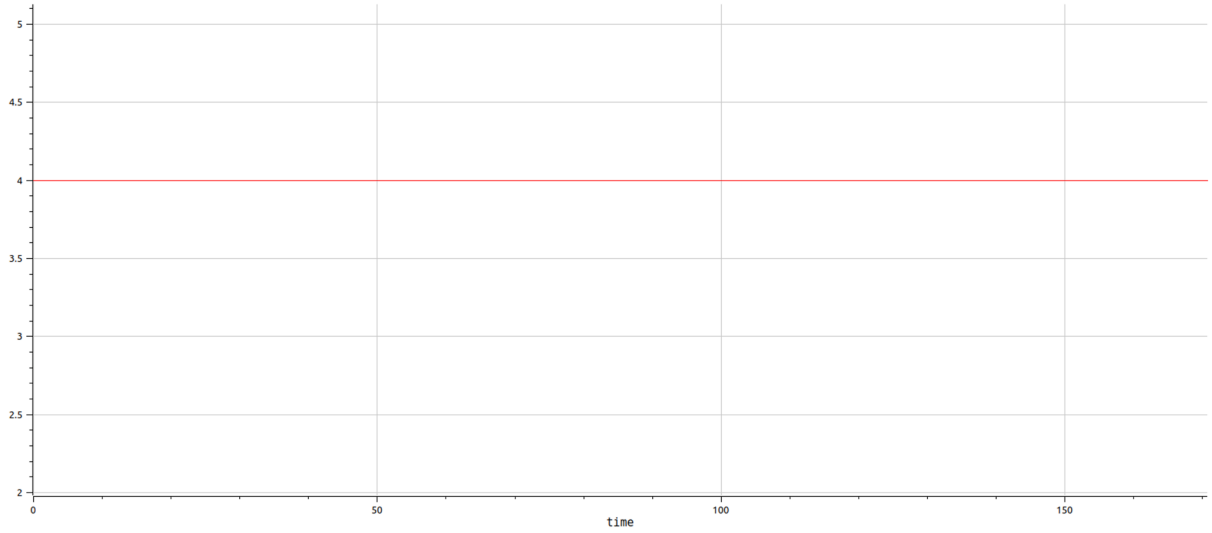
MaxNoDrone



Battery Percentages dei 4 droni



Numero di droni ***Alive***



Nessun drone ha perso la vita nel corso di queste simulazioni.

7. Bibliografia

[COB2007]
M. J. D. Powell (2007). A view of algorithms for optimization without derivatives. Cambridge University Technical Report DAMTP 2007.

[GER2020]
GERAD (2020). “NOMAD: A blackbox optimization software”.

[LAG2009]
Kalman, Dan (2009). "Leveling with Lagrange: An Alternate View of Constrained Optimization". Mathematics Magazine. 82 (3): 186–196. doi:10.1080/0025570X.2009.11953617. JSTOR 27765899. S2CID 121070192.

[MOD2021]
Modelica Association (2021). “Modelica - A Unified Object-Oriented Language food Systems Modeling”.

[NEV2018]
Meta (2018). "Nevergrad: An open source tool for derivative-free optimization".

[POW1964]
Powell, M. J. D. (1964). "An efficient method for finding the minimum of a function of several variables without calculating derivatives". Computer Journal. 7 (2): 155–162.

[SPR2014]
A. W. Colombo, T. Bangemann, S. Karnouskos, J. Delsing, P. Stluka, R. Harrison, F. Jammes, and J. Lastra: Industrial Cloud-based Cyber- Physical Systems: The IMC-AESOP Approach. Springer Verlag, 2014,

8. Note

Tutti i test sono stati eseguiti su una macchina virtuale Parallels (versione 17.1.4) con Ubuntu su un MacBook Pro (2018) con processore 2,2 GHz Intel Core i7 6 core e memoria 16GB 2400 MHz DDR4.

9. Ringraziamenti

La mia più sincera stima e gratitudine al Prof. Enrico Tronci per avermi accompagnato in questo viaggio ed aver svegliato in me la passione verso questo ramo dell'informatica, tanto difficile quanto gratificante.
Ringrazio di cuore il Prof. Toni Mancini per avermi seguito e sopportato nel corso della stesura di questo documento, cosa che non sarei mai riuscito a fare da solo e senza la giusta motivazione.
Ringrazio i miei genitori Maddalena e Roberto, senza i loro innumerevoli sforzi non sarei mai potuto arrivare così lontano, questa laurea è anche un po' vostra e a parole non riesco ad esprimere il bene che vi voglio e quanto vi sono grato.
Ringrazio tutta la mia famiglia per il sostegno e in particolare mia zia Silvia per essersi assicurata che rimanessi sulla strada giusta.
Un grazie speciale alla mia ragazza Denise, la persona che più di tutte è stata capace di capirmi, sostenermi nei momenti difficili e tirare fuori il meglio di me.
Infine voglio ringraziare tutti gli amici e i colleghi che mi sono stati vicini e che mi hanno accompagnato in questi anni di Università e nel corso della stesura della tesi di laurea, Tommaso, Enrico, Luca, Martina, Livio, Pasquale e Lavinia. Grazie per essere le spalle su cui posso sempre contare.