1. Librerie di supporto

```
1.  from __future__ import division
2.  from pyqtgraph.Qt import QtCore, QtGui
3.  from PIL import Image
4.  import numpy as np
5.  import pyqtgraph as pg
6.  import sys
7.  import random
8.  import threading
9.
10. np.set_printoptions(threshold=sys.maxsize)
11. pg.setConfigOptions(imageAxisOrder='row-major')
12. pg.setConfigOption('background', 'w')
```

L'impostazione *imageAxisOrder='row-major'* è necessaria per evitare che l'immagine sia mostrata a video trasposta (comportamento di default della libreria grafica).

2. Attributi

La classe si compone dei seguenti attributi:

- **__bitmap : *str***
  Contiene il percorso della mappa bidimensionale (bitmap) in cui verrà effettuata la simulazione.

- **__batch_size : *int***
  Numero di simulazioni parallele da avviare.
  Il parametro deve assumere un valore maggiore o uguale a 1.

- **__drone_size : *int***
  Ampiezza di un drone: un drone è rappresentato come un quadrato di raggio pari a *drone_size*.
  Il parametro deve assumere un valore maggiore o uguale a 0.

- **__observation_range : *int***
  Ampiezza dello spazio di osservazione di un drone: un drone è capace di vedere lo spazio circostante, a partire da ogni sua estremità, per un raggio pari a *observation_range*.
  Il parametro deve assumere un valore maggiore o uguale a 0.

- **__amount_of_drones : *int***
  Numero di droni che partecipano a ciascuna simulazione.
  Il parametro deve assumere un valore maggiore o uguale a 1.

- **__stigmergy_evaporation_speed : *ndarray<int>***
  Array monodimensionale contenente le velocità di evaporazione di ciascun livello stigmergico; esse sono uguali per ogni simulazione parallela. Il numero dei livelli stigmergici – uno per ciascun tipo di feromone che è possibile rilasciare - è automaticamente considerato come pari alla dimensione dell'array, ovvero al numero di velocità fornite.
  Le velocità devono essere numeri interi maggiori o uguali a 0: in caso siano inseriti numeri reali, essi saranno approssimati all'intero più vicino. Per maggiori informazioni sul significato della velocità di evaporazione, si veda la funzione *__stigmergy_evaporation()*.

- **__stigmergy_colours : *ndarray<ndarray<int[3]>>***
  Array contenente i colori con cui saranno visualizzate le tracce stigmergiche: ogni colore è una terna RGB ed è associato ad un singolo feromone.
  Il numero di colori fornito devo corrispondere al numero di livelli stigmergici dichiarato tramite il parametro *__stigmergy_evaporation_speed*; ad esempio, per tre livelli stigmergici, si potrebbe avere:

```
1. stigmergy_evaporation_speed = numpy.array([20, 30, 40])
2. stigmergy_colours = numpy.array([[255,64,0],[255,128,0],[255,255,0]])
```

- **__inertia : *float***
  Rappresenta l'inerzia che caratterizza il movimento dei droni.

- **__collision_detection : *ndarray<bool>***
  Array monodimensionale che determina quali livelli della mappa, oltre al livello dei target, sono da considerare abilitati alle collisioni: un'entrata pari a True abilita le collisioni, un'entrata pari a False disabilita le collisioni. La dimensione dell'array deve essere pari al numero dei livelli presenti nella mappa, escluso il livello dei target.

- **__max_steps : *int***
  Indica il numero massimo di step che possono essere effettuati in ciascuna simulazione parallela: raggiunto tale numero, risulta necessario il reset del simulatore.
  Il parametro deve essere maggiore o uguale a 1.

- **__current_steps : *int***
  Tiene traccia del numero di step effettuati in ciascuna simulazione.

- **__rendering_allowed : *bool***
  Determina se il simulatore deve effettuare il rendering della simulazione, ovvero visualizzare a video l'andamento della stessa. Tale opzione è disponibile solamente se *batch_size* è pari a 1: in caso contrario, sarà generata un'eccezione.

- **__drone_colour : *ndarray<int[3]>***
  Array contenente il colore con cui saranno visualizzati i droni: il colore è una terna RGB ed è uguale per ogni drone.

```
1. drone_colour= numpy.array([255, 255, 255])
```

- **__targets : *ndarray (2D)***
  Array bidimensionale avente come dimensioni le dimensioni della mappa fornita. Un elemento pari a 1 indica la presenza di un target nella posizione; in caso contrario, l'elemento è pari a 0.

- **__collision : *ndarray (2D)***
  Array bidimensionale avente come dimensioni le dimensioni della mappa fornita. Un elemento pari a 1 indica la presenza di un elemento con cui il drone può collidere (un ostacolo); in caso contrario, l'elemento è pari a 0.

- **__no_collision : *ndarray (2D)***
  Array bidimensionale avente come dimensioni le dimensioni della mappa fornita. Un elemento pari a 1 indica la presenza di un elemento con cui il drone non può collidere (utile per la visualizzazione di informazioni aggiuntive); in caso contrario, l'elemento è pari a 0.

- **__environment_bitmap : *ndarray (3D)***
  Matrice bidimensionale, avente come dimensioni le dimensioni della mappa fornita, in cui ogni elemento è una terna RGB, la quale determina il colore dell'elemento contenuto nella posizione. Contiene la mappa iniziale e le sue informazioni fisse (terreno, target e ostacoli); gli elementi dinamici (droni e tracce stigmergiche) sono aggiunti in fase di rendering senza modificarne il contenuto.

- **__drones_position_float : *ndarray (3D)***
  Array tridimensionale che, per ogni simulazione, contiene le coordinate dei droni nella mappa, salvate come float. Sono utilizzate solamente nel calcolo dello spostamento con velocità.

- **__drones_position : *ndarray (3D)***
  Array tridimensionale che, per ogni simulazione, contiene le coordinate dei droni nella mappa, approssimate come interi. Sono utilizzate in tutti i calcoli e nel disegno della simulazione a video, ad eccezione del calcolo dello spostamento con velocità.

- **__drones_velocity : *ndarray (3D)***
  Array tridimensionale che, per ogni simulazione, contiene le componenti di velocità dei droni nelle due direzioni della mappa, salvate come float.

- **__drawn_drones : *ndarray (4D)***
  Array che, per ogni simulazione, contiene il disegno di ciascun drone su una matrice bidimensionale di dimensioni pari alle dimensioni della mappa. Nello specifico, per ogni drone è mantenuta una matrice in cui gli elementi pari a 1 indicano la posizione del drone nella mappa, in accordo alle coordinate e alla dimensione dello stesso.

- **__targets_achieved : *ndarray (4D)***
  Array che mantiene, per ogni simulazione, i target raggiunti da ciascun drone durante la simulazione. In particolare, per ogni drone è mantenuta una matrice, di dimensioni pari alle dimensioni della mappa, in cui gli elementi pari a 1 indicano i target raggiunti.

- **__stigmergy_space : *ndarray (4D)***
  Array che mantiene, per ogni simulazione, lo stato dello spazio stigmergico. Nello specifico, lo spazio stigmergico di una simulazione è composto da un numero di matrici bidimensionali pari al numero di feromoni fornito: ogni matrice ha dimensione pari alla dimensione della mappa e presenta elementi maggiori di 0 in corrispondenza delle posizioni in cui i droni hanno rilasciato tracce stigmergiche. Il valore dell'elemento determina l'intensità della traccia stigmergica presente.

- **__image : *ndarray (3D)***
  Matrice bidimensionale, avente come dimensioni le dimensioni della mappa fornita, in cui ogni elemento è una terna RGB, la quale determina il colore dell'elemento contenuto nella posizione. Contiene la mappa iniziale, le sue informazioni fisse (terreno, target e ostacoli) e gli elementi dinamici (droni e tracce stigmergiche); viene utilizzata per la visualizzazione a video quando il rendering è abilitato.

- **__image_semaphore : *thread.lock***
  Lock utilizzato per l'accesso concorrente all'attributo *__image*.

3. Metodi

3.1 __init__()

```
def __init__(self, bitmap, batch_size, drone_size, observation_range, amount_of_drones,
             stigmergy_evaporation_speed, stigmergy_colours, inertia, collision_detection,
             max_steps, rendering_allowed=False, drone_colour=[255, 255, 255]):

    self.__init_simulator_parameters(bitmap, batch_size, drone_size, observation_range,
                                     amount_of_drones, stigmergy_evaporation_speed,
                                     stigmergy_colours, inertia, collision_detection,
                                     max_steps, rendering_allowed, drone_colour)

    self.__init_environment_parameters()
    self.__parse_bitmap()

    self.__init_drones_parameters()
    self.__init_drones()

    self.__init_stigmergy_space()
    self.__init_rendering_parameters()
```

Inizializza un oggetto *DroneSimulator*, a partire dai parametri forniti dall'utente: per maggiori informazioni sui parametri in ingresso al metodo, si veda l'elenco degli attributi della classe.
Se non specificato, il rendering è disabilitato (*rendering_allowed = False*).
Se non specificato, il colore dei droni è bianco (*drone_colour=[255, 255, 255]*).

3.2 __init_simulator_parameters()

```
def __init_simulator_parameters(self, bitmap, batch_size, drone_size, observation_range,
                                amount_of_drones, stigmergy_evaporation_speed,
                                stigmergy_colours, inertia, collision_detection,
                                max_steps, rendering_allowed, drone_colour):

    self.__bitmap = bitmap
    self.__batch_size = batch_size
    self.__observation_range = observation_range
    self.__drone_size = drone_size
    self.__amount_of_drones = amount_of_drones
    self.__stigmergy_evaporation_speed = stigmergy_evaporation_speed.astype(int)
    self.__stigmergy_colours = stigmergy_colours
    self.__inertia = inertia
    self.__collision_detection = collision_detection
    self.__max_steps = max_steps
    self.__rendering_allowed = rendering_allowed
    self.__drone_colour = drone_colour
```

Inizializza i parametri del simulatore: per maggiori informazioni sui parametri in ingresso al metodo, si veda l'elenco degli attributi della classe.

3.3 __ init_environment_parameters()

```
def __init_environment_parameters(self):
    self.__environment_bitmap = None
    self.__targets = np.array([])
    self.__collision = np.array([])
    self.__no_collision = np.array([])
```

Definisce i parametri dell'ambiente; essi saranno inizializzati nel metodo *__parse_bitmap()*.

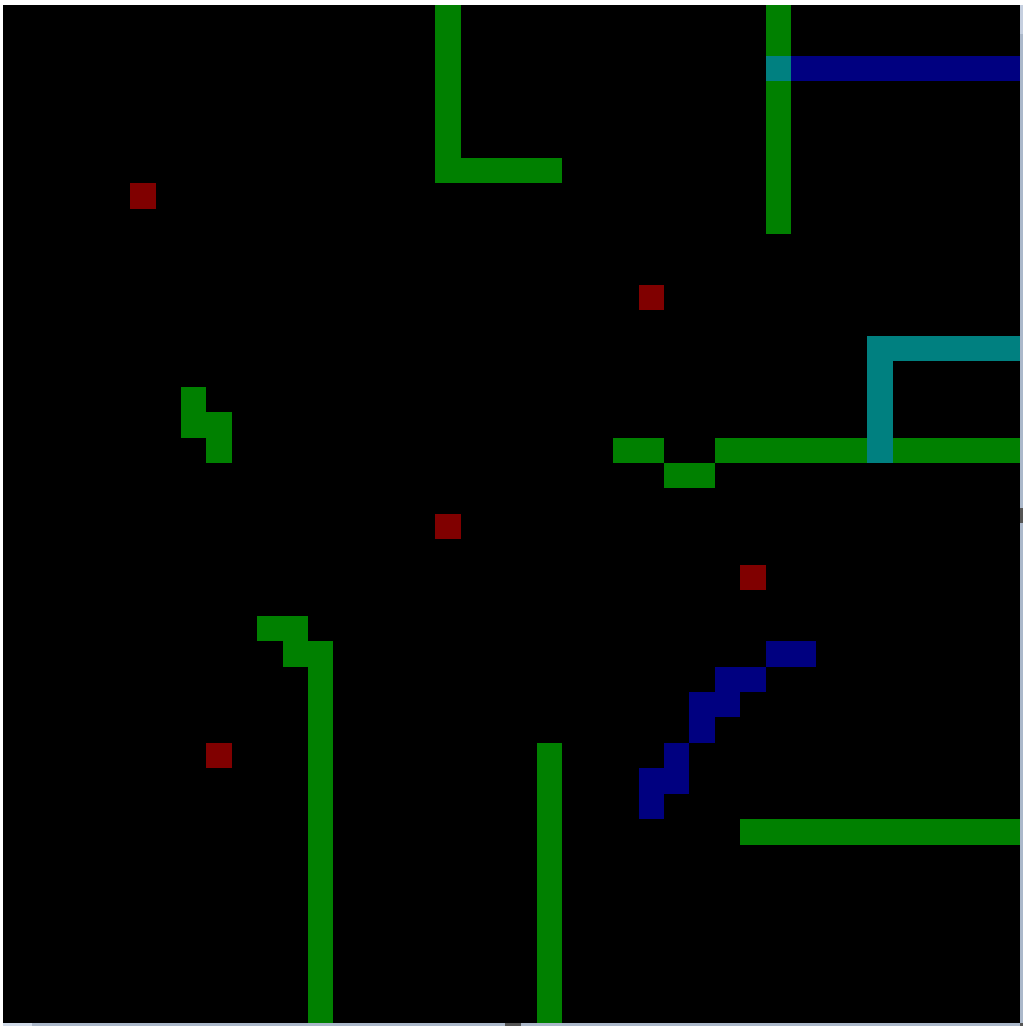
3.4 __parse_bitmap()

```
def __parse_bitmap(self):
    input_array = np.asarray(Image.open(self.__bitmap))
    rgb_bit_array = np.unpackbits(input_array, axis=2)
    # rgb_bit_array is a matrix of pixels, where each cell (each pixel) is
    # a 24-bit array

    collision = []
    no_collision = []
    level_founded = 0

    for i in range(0, 24):
        level = rgb_bit_array[:, :, i]
        # Only levels with at least 1 item are inserted in the environment
        if np.any(level):
            if level_founded == 0:
                # First level is composed of targets
                self.__targets = np.asarray(level)
                self.__environment_bitmap = np.full((self.__targets.shape[0],
                                                     self.__targets.shape[1], 3), 0)
            else:
                if self.__collision_detection[level_founded - 1]:
                    collision.append(level)
                else:
                    no_collision.append(level)

            self.__environment_bitmap[level == 1, :] = (
                    self.__environment_bitmap[level == 1, :] + self.__get_colour(i))
            level_founded += 1

    if not collision:
        collision = np.zeros((1, self.__targets.shape[0], self.__targets.shape[1]))
    else:
        collision = np.asarray(collision)

    if not no_collision:
        no_collision = np.zeros((1, self.__targets.shape[0], self.__targets.shape[1]))
    else:
        no_collision = np.asarray(no_collision)

    self.__collision = np.sum(collision, axis=0)
    self.__collision[self.__collision > 0] = 1

    self.__no_collision = np.sum(no_collision, axis=0)
    self.__no_collision[self.__no_collision > 0] = 1
```

Preleva la mappa dal percorso specificato in fase di inizializzazione e la trasforma in una matrice di pixel manipolabile: per ogni livello individuato, vengono aggiornate le mappe dei target, delle collisioni e delle non-collisioni; inoltre, è costruita la mappa dell'ambiente con le informazioni fisse ed i colori associati in formato RGB.
I colori sono individuati da 24 bit, dove ogni bit rappresenta un livello: con quest'ultimo termine individuiamo un insieme di informazioni denotate dallo stesso colore (la mappa sarà sempre in due dimensioni, a prescindere dal numero di livelli presenti); i livelli sono esaminati a partire dal bit più significativo. La scelta dei colori deve rispettare la seguente politica:

- il terreno è sempre nero, ovvero è individuato da soli 0;
- il primo livello è sempre quello dei target ed un singolo target occupa sempre una ed una sola casella (in caso siano occupate più caselle contigue, saranno riconosciuti target multipli);
- ogni colore utilizzato deve avere un solo bit a 1;
- l'unica eccezione alla regola precedente è data dal caso in cui due o più oggetti appartenenti a livelli differenti siano sovrapposti, ovvero debbano essere disegnati nella stessa casella. In questo caso, il colore della casella deve presentare più bit ad 1: nello specifico, deve presentare un bit 1 in corrispondenza di ogni livello sovrapposto.

Un esempio di utilizzo dei colori è il seguente:



Terreno: colore nero (0000 0000 0000 0000 0000 0000)
Target: colore rosso (1000 0000 0000 0000 0000 0000)
Ostacolo: colore verde (0000 0000 1000 0000 0000 0000)
Ostacolo: colore blu (0000 0000 0000 0000 1000 0000)
Sovrapposizione ostacoli verdi e blu: colore azzurro (0000 0000 1000 0000 1000 0000)

3.5 __get_colour()

```
1.  def __get_colour(self, i):
2.      colour = np.zeros(shape=3)
3.      i = 24 - i - 1
4.
5.      colour[2 - i // 8] = 2 ** (i % 8)
6.      return colour
```

Funzione di utilità che restituisce un colore in formato RGB a partire dalla posizione del bit 1 nella rappresentazione binaria.

Parametri

- **i : *int***
  Posizione del bit 1 nella rappresentazione binaria del colore.

Restituisce

- **colour : *ndarray<int[3]>***
  Array monodimensionale che contiene la rappresentazione RGB del colore.

3.6 __init_drones_parameters()

```
1.  def __init_drones_parameters(self):
2.      self.__drones_position_float = None
3.      self.__drones_position = np.full((self.__batch_size,
4.                                        self.__amount_of_drones, 2), -1)
5.
6.      self.__drones_velocity = np.zeros((self.__batch_size, self.__amount_of_drones, 2))
7.
8.      self.__drawn_drones = np.zeros((self.__batch_size, self.__amount_of_drones,
9.                                      self.__targets.shape[0], self.__targets.shape[1]))
10.
11.     self.__targets_achieved = np.zeros((self.__batch_size, self.__amount_of_drones,
12.                                         self.__targets.shape[0], self.__targets.shape[1]))
13.     self.__current_steps = 0
```

Definisce i parametri relativi ai droni; essi saranno inizializzati nel metodo *__init_drones()*.

3.7 __init_drones()

```
def __init_drones(self):
    for batch_index in range(self.__batch_size):
        drone_index = 0
        while drone_index < len(self.__drones_position[batch_index]):
            self.__drones_position[batch_index, drone_index] = np.asarray([
                random.randint(0, self.__targets.shape[0] - 1),
                random.randint(0, self.__targets.shape[1] - 1)
            ])

            # A drone is correctly positioned if it's rendered completely inside the map
            # and it doesn't collide with environment or other drones
            if not self.__out_of_map(batch_index, drone_index, self.__drone_size):
                self.__draw_drone(batch_index, drone_index)
                if not self.__detect_collision(batch_index):
                    drone_index += 1

    self.__drones_position_float = np.copy(self.__drones_position).astype(float)
```

Inizializza i droni di tutte le simulazioni parallele, assegnando a ciascuno di essi una posizione casuale nella mappa: la posizione è valida solamente se il drone risulta completamente dentro la mappa e non collide con elementi dell'ambiente (tra cui eventuali altri droni).

3.8 __draw_drone()

```
def __draw_drone(self, batch_index, drone_index):
    radius = self.__drone_size
    position_axis_0 = self.__drones_position[batch_index, drone_index, 0]
    position_axis_1 = self.__drones_position[batch_index, drone_index, 1]
    interval_axis_0, interval_axis_1 = self.__drawing_boundaries(position_axis_0,
                                                                 position_axis_1, radius)

    drone_level = np.zeros((self.__targets.shape[0], self.__targets.shape[1]))
    drone_level[interval_axis_0, interval_axis_1] = 1
    self.__drawn_drones[batch_index, drone_index] = drone_level
```

Disegna un drone appartenente ad una simulazione: nel caso in cui il drone sia parzialmente o completamente fuori dalla mappa, sarà disegnata solamente la parte interna a quest'ultima.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone da disegnare.

3.9 __drawing_boundaries()

```
1.  def __drawing_boundaries(self, position_axis_0, position_axis_1, radius):
2.      start_point_axis_0 = position_axis_0 - radius
3.      end_point_axis_0 = position_axis_0 + radius + 1
4.      start_point_axis_1 = position_axis_1 - radius
5.      end_point_axis_1 = position_axis_1 + radius + 1
6.
7.      if position_axis_0 - radius < 0:
8.          start_point_axis_0 = 0
9.      elif position_axis_0 - radius >= self.__targets.shape[0]:
10.         start_point_axis_0 = self.__targets.shape[0]
11.
12.     if position_axis_0 + radius < 0:
13.         end_point_axis_0 = 0
14.     elif position_axis_0 + radius >= self.__targets.shape[0]:
15.         end_point_axis_0 = self.__targets.shape[0]
16.
17.     if position_axis_1 - radius < 0:
18.         start_point_axis_1 = 0
19.     elif position_axis_1 - radius >= self.__targets.shape[1]:
20.         start_point_axis_1 = self.__targets.shape[1]
21.
22.     if position_axis_1 + radius < 0:
23.         end_point_axis_1 = 0
24.     elif position_axis_1 + radius >= self.__targets.shape[1]:
25.         end_point_axis_1 = self.__targets.shape[1]
26.
27.     return slice(start_point_axis_0, end_point_axis_0),
28.            slice(start_point_axis_1, end_point_axis_1)
```

Per ogni asse, calcola gli intervalli in cui un elemento rettangolare è interno alla mappa.

Parametri

- **position_axis_0 : *int***
  Posizione dell'elemento sull'asse 0 (righe della matrice).

- **position_axis_1: *int***
  Posizione dell'elemento sull'asse 1 (colonne della matrice).

- **radius : *int***
  Raggio dell'elemento.

Restituisce

- **interval_axis_0 : *slice object***
  Intervallo sull'asse 0 in cui l'elemento è interno alla mappa, rappresentato nel formato *slice(start, stop)*.

- **interval_axis_1 : *slice object***
  Intervallo sull'asse 1 in cui l'elemento è interno alla mappa, rappresentato nel formato *slice(start, stop)*.

3.10 __detect_collision()

```
1.  def __detect_collision(self, batch_index):
2.      collision_level = self.__collision[np.newaxis, ...]
3.      collision_detection = np.append(self.__drawn_drones[batch_index],
4.                                      collision_level, axis=0)
5.      collision_detection = np.sum(collision_detection, axis=0)
6.
7.      if np.any(collision_detection > 1):
8.          return True
9.
10.     return False
```

Verifica se è presente una qualsiasi collisione tra droni oppure tra droni e livelli della mappa abilitati alle collisioni. Il metodo viene usato dopo lo spostamento di un singolo drone: se la situazione di partenza è priva di collisioni, è univocamente determinabile quale drone ha colliso senza l'utilizzo di indici identificativi.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

Restituisce

- **collision_detected : *bool***
  Determina se è stata individuata una collisione (True) o meno (False).

3.11 __out_of_map()

```
1.  def __out_of_map(self, batch_index, drone_index, radius):
2.      position_axis_0 = self.__drones_position[batch_index, drone_index, 0]
3.      position_axis_1 = self.__drones_position[batch_index, drone_index, 1]
4.
5.      if (position_axis_0 - radius < 0 or
6.            position_axis_0 -radius >= self.__targets.shape[0] or
7.              position_axis_0 + radius < 0 or
8.                position_axis_0 + radius >= self.__targets.shape[0]):
9.          return True
10.
11.     if (position_axis_1 - radius < 0 or
12.           position_axis_1 - radius >= self.__targets.shape[1] or
13.             position_axis_1 + radius < 0 or
14.               position_axis_1 + radius >= self.__targets.shape[1]):
15.         return True
16.
17.     return False
```

Verifica se un drone si trova al di fuori dei confini della mappa (parzialmente o completamente).

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare la verifica.

- **radius : *int***
  Raggio del drone.

Restituisce

- **out_of_map : *bool***
  Determina se il drone si trova parzialmente o completamente fuori dalla mappa (True) oppure se si trova completamente dentro la mappa (False).

3.12 __init_stigmergy_space()

```
def __init_stigmergy_space(self):
    self.__stigmergy_space = np.zeros((self.__batch_size,
                                       self.__stigmergy_evaporation_speed.shape[0],
                                       self.__targets.shape[0],
                                       self.__targets.shape[1]),
                                       int)
```

Inizializza lo spazio stigmergico: gli elementi che compongono le mappe stigmergiche sono interi, in modo tale da evitare problemi di visualizzazione dovuti all'imprecisione della rappresentazione dei float (in particolare, nel meccanismo di evaporazione).

3.13 __init_rendering_parameters()

```
def __init_rendering_parameters(self):
    self.__image = np.full((self.__targets.shape[0], self.__targets.shape[1], 3), 0)
    self.__image_semaphore = None

    if self.__rendering_allowed:
        if self.__batch_size > 1:
            raise ValueError("Rendering is allowed only when batch_size is equal to 1")

        self.__image_semaphore = threading.Lock()
        rendering = threading.Thread(target=self.__init_rendering)
        rendering.start()
```

Inizializza i parametri necessari per il rendering e genera un thread che disegna a video.

Lancia

- **ValueError**
  L'eccezione è lanciata nel caso in cui il numero delle simulazioni parallele sia maggiore di 1 e il rendering sia abilitato.

3.14 __init_rendering()

```
1.  def __init_rendering(self):
2.      app = QtGui.QApplication([])
3.
4.      w = pg.GraphicsView()
5.      w.show()
6.      w.showMaximized()
7.      w.setWindowTitle('Drone simulator')
8.
9.      view = pg.ViewBox()
10.     view.invertY()
11.     view.setAspectLocked(True)
12.     w.setCentralItem(view)
13.
14.     self.__image_semaphore.acquire()
15.     img = pg.ImageItem(self.__image)
16.     view.addItem(img)
17.     self.__image_semaphore.release()
18.
19.     timer = QtCore.QTimer()
20.     timer.timeout.connect(lambda: self.__update_rendering(view))
21.     timer.start(100)
22.
23.     app.instance().exec()
```

Inizializza la finestra in cui avverrà il rendering e predispone un timer di aggiornamento dello stesso.

3.15 __update_rendering()

```
1.  def __update_rendering(self, view):
2.      self.__image_semaphore.acquire()
3.
4.      view.clear()
5.      img = pg.ImageItem(self.__image)
6.      view.addItem(img)
7.
8.      self.__image_semaphore.release()
```

Aggiorna l'immagine mostrata a video: l'immagine cambia effettivamente se, in precedenza, è stata chiamata la funzione *render().*

Parametri

- **view : *ViewBox***
  Elemento dell'interfaccia grafica su cui effettuare l'aggiornamento dell'immagine.

3.16 step()

```
1.  def step(self, drones_actions, stigmergy_actions):
2.      self.__current_steps += 1
3.      if self.__current_steps > self.__max_steps:
4.          return None, None, True, "Maximum number of steps reached"
5.
6.      observation_dimension = 2*(self.__drone_size + self.__observation_range) + 1
7.      observations_table = np.zeros((self.__batch_size, self.__amount_of_drones,
8.                                     observation_dimension, observation_dimension))
9.
10.     rewards_table = np.zeros((self.__batch_size, self.__amount_of_drones, 1))
11.
12.     self.__stigmergy_evaporation()
13.     self.__update_stigmergy_space(stigmergy_actions)
14.     self.__update_drones(drones_actions, rewards_table, observations_table)
15.
16.     return observations_table, rewards_table, False, self.__environment_info()
```

Il metodo compie un passo della simulazione, ovvero compie nell'ordine:

- l'evaporazione delle tracce stigmergiche presenti;
- il rilascio di nuove tracce stigmergiche (nella posizione non aggiornata dei droni, ovvero quella risultante dallo step precedente);
- lo spostamento dei droni, con il calcolo della ricompensa e dello spazio di osservazione per ciascun drone.

Parametri

- **drones_actions : *ndarray (3D)***
  Array contenente gli spostamenti che i droni devono effettuare in ciascuna simulazione parallela.
  L'array fornito deve essere di dimensioni pari a *shape = (batch_size, amount_of_drones, 2)*, dove il 2 è dovuto al fatto che il comando da impartire esprime uno spostamento bidimensionale: il primo elemento si riferisce all'asse 0 (righe), il secondo si riferisce all'asse 1 (colonne).
  Ad esempio, si potrebbe avere:

```
1.  # batch_size = 1, amount_of_drones = 3
2.  shape = (1, 3, 2)
3.  actions = np.array([[[1,0],[0,1],[-3,5]]])
```

```
1.  # batch_size = 4, amount_of_drones = 10
2.  shape = (4, 10, 2)
3.  actions = np.random.randint(7, size=shape)
```

- **stigmergy_actions: *ndarray (3D)***
  Array contenente le azioni che i droni devono effettuare all'interno dello spazio stigmergico di ciascuna simulazione parallela.
  L'array fornito deve essere di dimensioni pari a *shape = (batch_size, amount_of_drones, 2)*, dove il 2 è dovuto al fatto che il comando da impartire deve specificare:
  - in prima posizione, il livello su cui effettuare l'azione (quale feromone rilasciare), a partire dal livello zero fino al livello massimo diminuito di 1 (dipende dal numero di feromoni impostato in fase di inizializzazione).
    Se non si vuole rilasciare un feromone, si imposta il livello pari a -1;
  - in seconda posizione, il raggio che determina l'ampiezza della traccia da rilasciare (deve essere maggiore o uguale a 0).

Restituisce

- **observation :** ***ndarray (4D)***
  Array contenente, per ogni simulazione parallela, lo spazio di osservazione di ciascun drone.
  Tale spazio è composto da elementi pari a:
  - 0 se la casella della mappa è occupata dal solo terreno;
  - 1 se la casella della mappa è occupata da un drone, da un ostacolo o da una traccia stigmergica;
  - -1 se la casella contiene un elemento fuori dalla mappa.

  Nel caso in cui siano stati effettuati un numero di step pari a *max_steps*, assume il valore None.

- **reward :** ***ndarray (3D)***
  Array che, per ogni simulazione parallela, contiene la ricompensa ottenuta da ciascun drone in seguito allo spostamento effettuato con il comando corrente; la ricompensa è fornita sotto forma di float.
  Nel caso in cui siano stati effettuati un numero di step pari *a max_steps*, assume il valore None.

- **done :** ***bool***
  Determina se la simulazione è finita e l'ambiente necessita di un reset (True) o meno (False).
  La simulazione termina quando è stato raggiunto il massimo numero di step previsti.

- **info :** ***dict***
  Raccolta di informazioni sui droni e sull'ambiente di ciascuna simulazione. Nello specifico, sono presenti le seguenti chiavi: *"Drones position - float", "Drones position", "Drones velocity", "Targets achieved", "Stigmergy Space"*.
  Nel caso in cui siano stati effettuati un numero di step pari a *max_steps*, viene restituita una stringa *"Maximum number of steps reached".*

3.17 __stigmergy_evaporation()

```
def __stigmergy_evaporation(self):
    evaporation_levels = np.zeros((self.__stigmergy_evaporation_speed.shape[0],
                                   self.__targets.shape[0],
                                   self.__targets.shape[1]),
                                   int)

    for index in range(self.__stigmergy_evaporation_speed.shape[0]):
        evaporation_levels[index] = np.full((self.__targets.shape[0],
                                             self.__targets.shape[1]),
                                             self.__stigmergy_evaporation_speed[index])

    for batch_index in range(self.__batch_size):
        self.__stigmergy_space[batch_index] -= evaporation_levels

    self.__stigmergy_space[self.__stigmergy_space < 0] = 0
```

Gestisce il meccanismo di evaporazione dello spazio stigmergico, effettuando un aggiornamento su tutte le simulazioni parallele. Un drone, mediante il rilascio di un tipo di feromone, genera una traccia stigmergica: ad ogni step della simulazione, tale valore viene decrementato in base alla velocità di evaporazione definita per il livello in fase di inizializzazione. L'intensità di rilascio è sempre pari a 100 e diminuirà di un valore pari alla velocità di evaporazione ad ogni step. Ad esempio:

	Velocità di evaporazione: 20	**Velocità di evaporazione: 30**	**Velocità di evaporazione: 50**
Intensità iniziale	100	100	100
Intensità dopo 1 step	80	70	50
Intensità dopo 2 step	60	40	0
Intensità dopo 3 step	40	10	0

La diminuzione dell'intensità non si traduce in una maggiore o minore intensità del colore, ma influisce solamente sul tempo di permanenza della traccia.
Nel caso siano effettuati più rilasci di feromone nello stesso punto, l'intensità aumenta ad incrementi di 100, ma l'evaporazione causa sempre una diminuzione pari alla velocità impostata (il meccanismo di evaporazione rimane inalterato).

	Velocità di evaporazione: 20	**Velocità di evaporazione: 20**	**Velocità di evaporazione: 20**
Intensità iniziale	100	200	300
Intensità dopo 1 step	80	180	280
Intensità dopo 2 step	60	160	260
Intensità dopo 3 step	40	140	240

L'utilizzo di numeri interi è necessario per evitare problemi di precisione dovuti ai float: ad esempio, nell'effettuare una sottrazione, si potrebbe ottenere un numero molto piccolo, ma non pari a 0, causando un permanere non voluto della traccia stigmergica e una inaffidabilità del meccanismo di evaporazione.

3.18 __update_stigmergy_space()

```
def __update_stigmergy_space(self, stigmergy_actions):
    for batch_index in range(self.__batch_size):
        for drone_index in range(self.__amount_of_drones):
            position_axis_0 = self.__drones_position[batch_index, drone_index, 0]
            position_axis_1 = self.__drones_position[batch_index, drone_index, 1]
            stig_level = int(stigmergy_actions[batch_index, drone_index, 0])
            stig_radius = int(stigmergy_actions[batch_index, drone_index, 1])

            if stig_level == -1:  # No pheromone release
                continue

            interval_axis_0, interval_axis_1 = self.__drawing_boundaries(position_axis_0,
                                                                         position_axis_1,
                                                                         stig_radius)

            # 100 is the intensity of the pheromone released.
            # It is the same for all the drones and for every pheromone type.
            self.__stigmergy_space[batch_index][stig_level][interval_axis_0,
                                                            interval_axis_1] += 100
```

Aggiorna lo spazio stigmergico di tutte le simulazioni implementando il rilascio del feromone; ciascun feromone è rilasciato sempre con un'intensità pari a 100. Anche in questo caso, l'utilizzo di numeri interi è necessario per evitare problemi di precisione dovuti ai float.

Parametri

- **stigmergy_actions:** ***ndarray (3D)***
  Array contenente le azioni che i droni devono effettuare all'interno dello spazio stigmergico di ciascuna simulazione parallela.

3.19 __update_drones()

```
1.  def __update_drones(self, drones_actions, rewards_table, observations_table):
2.      for batch_index in range(self.__batch_size):
3.          for drone_index in range(self.__amount_of_drones):
4.              self.__update_velocity(batch_index, drone_index, drones_actions)
5.              self.__update_position(batch_index, drone_index)
6.              self.__draw_drone(batch_index, drone_index)
7.              self.__target_achieved(batch_index, drone_index)
8.              rewards_table[batch_index, drone_index] = self.__reward(batch_index,
9.                                                                      drone_index)
10.             observations_table[batch_index,
11.                                drone_index] = self.__get_observation(batch_index,
12.                                                                       drone_index)
```

Per ogni simulazione, aggiorna lo stato dei droni calcolando le nuove velocità e posizioni; inoltre, esegue il calcolo dei punteggi e degli spazi di osservazione.

Parametri

- **drones_actions : *ndarray (3D)***
  Array contenente gli spostamenti che i droni devono effettuare in ciascuna simulazione parallela.

- **rewards_table : *ndarray (3D)***
  Array contenente, per ogni simulazione parallela, la ricompensa ottenuta da ciascun drone in seguito allo spostamento effettuato mediante il comando corrente. La ricompensa è fornita sotto forma di float.

- **observations_table : *ndarray (3D)***
  Array contenente, per ogni simulazione parallela, lo spazio di osservazione di ciascun drone.

3.20 __update_velocity()

```
def __update_velocity(self, batch_index, drone_index, drones_actions):
    drone_velocity = self.__drones_velocity[batch_index, drone_index]
    drone_command = drones_actions[batch_index, drone_index]

    self.__drones_velocity[batch_index, drone_index] =
                (drone_velocity * self.__inertia + drone_command * (1 - self.__inertia))
```

Dato un drone in una simulazione, calcola la nuova velocità a partire dal comando impartito.
La velocità è calcolata secondo la formula:

$$Velocità = velocità \cdot inerzia + comando \cdot (1 - inerzia)$$

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare il calcolo.

- **drones_actions : *ndarray (3D)***
  Array contenente gli spostamenti che i droni devono effettuare in ciascuna simulazione parallela.

3.21 __update_position()

```
def __update_position(self, batch_index, drone_index):
    drone_position = self.__drones_position_float[batch_index, drone_index]
    drone_velocity = self.__drones_velocity[batch_index, drone_index]
    t_constant = 1

    self.__drones_position_float[batch_index, drone_index] = drone_position +
                                                             drone_velocity * t_constant
    self.__drones_position = np.copy(self.__drones_position_float).astype(int)
```

Dato un drone in una simulazione, calcola la nuova posizione a partire dalla velocità corrente.
La posizione è calcolata secondo la formula:

$$Posizione = posizione + velocità \cdot t$$

dove t è una costante di tempo, per semplicità considerata pari a 1.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare il calcolo.

3.22 __target_achieved()

```
1.  def __target_achieved(self, batch_index, drone_index):
2.      target_collision = self.__drawn_drones[batch_index, drone_index] + self.__targets
3.      self.__targets_achieved[batch_index][drone_index][target_collision > 1] = 1
```

Dato un drone in una simulazione, verifica se è stato raggiunto un target: in caso affermativo, lo aggiunge alla matrice dei target trovati da quel particolare drone.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare la verifica.

3.23 __reward()

```
1.  def __reward(self, batch_index, drone_index):
2.      average_distance = self.__average_distance_to_targets(batch_index, drone_index)
3.      alpha = 0.5
4.      num_target_achieved = np.count_nonzero(self.__targets_achieved[batch_index,
5.                                                                     drone_index])
6.
7.      reward_score = 1 / average_distance + alpha * num_target_achieved
8.
9.      if self.__detect_collision(batch_index):
10.         return reward_score * (-1)
11.
12.     if self.__out_of_map(batch_index, drone_index, self.__drone_size):
13.         return reward_score * (-2)
14.
15.     return reward_score
```

Dato un drone in una simulazione, calcola la ricompensa ottenuta dal drone stesso.
Definiamo il punteggio base come:

$$Punteggio = \frac{1}{distanza\ media} + \alpha \cdot target\ trovati$$

dove *distanza media* è la distanza media del drone da tutti i target non ancora raggiunti, *target trovati* è il numero di target trovati fino a quel momento e α è un coefficiente di proporzionalità pari a 0.5.
Distribuiamo la ricompensa nel seguente modo:

1. se il drone non collide e non esce dalla mappa, riceve una ricompensa pari a $Punteggio$;
2. se il drone collide, riceve una ricompensa pari a $-Punteggio$;
3. se il drone esce dalla mappa (parzialmente o completamente), riceve una ricompensa pari a $-2 \cdot Punteggio$.

Si assume che ogni target occupi una sola casella: il numero di target trovati e la distanza media rispecchieranno tale supposizione.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare il calcolo.

Restituisce

- **reward_score : *float***
  Ricompensa ottenuta dal drone.

3.24 __average_distance_to_targets()

```
1.  def __average_distance_to_targets(self, batch_index, drone_index):
2.      targets_not_achieved = self.__targets + self.__targets_achieved[batch_index,
3.                                                                       drone_index]
4.      targets_not_achieved[targets_not_achieved > 1] = 0
5.      average_distance = 1
6.
7.      if np.any(targets_not_achieved):
8.          targets_positions = np.array(list(zip(*np.nonzero(targets_not_achieved))))
9.          drone_position = np.repeat(self.__drones_position[batch_index,
10.                                                            drone_index].reshape(1, 2),
11.                                     targets_positions.shape[0], 0)
12.
13.         average_distance = np.average(
14.           np.sqrt(np.sum(np.power(np.subtract(drone_position, targets_positions), 2), 1)))
15.
16.     return average_distance
```

Dato un drone in una simulazione, calcola la distanza media tra il drone e i target non ancora raggiunti.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare il calcolo.

Restituisce

- **average_distance : *float***
  Distanza media del drone dai target non raggiunti. Nel caso in cui il drone abbia raggiunto tutti gli obiettivi, è pari a 1.

3.25 __get_observation()

```
1.  def __get_observation(self, batch_index, drone_index):
2.      position_axis_0 = self.__drones_position[batch_index, drone_index, 0]
3.      position_axis_1 = self.__drones_position[batch_index, drone_index, 1]
4.      observation_radius = self.__drone_size + self.__observation_range
5.
6.      drones = np.sum(self.__drawn_drones[batch_index], axis=0)
7.      stigmergy_space = np.sum(self.__stigmergy_space[batch_index], axis=0)
8.      environment = self.__targets + self.__collision + drones + stigmergy_space
9.      environment[environment > 1] = 1
10.
11.     if self.__out_of_map(batch_index, drone_index, observation_radius):
12.         # Map view enlargement: the drone will see -1 if a space is outside the map
13.         # The position of the drone is reevaluated according to the new map dimensions
14.         environment, position_axis_0, position_axis_1 =
15.                                                 self.__enlarge_map_view(position_axis_0,
16.                                                                         position_axis_1,
17.                                                                         observation_radius,
18.                                                                         environment)
19.
20.     observation = environment[position_axis_0 - observation_radius:
21.                               position_axis_0 + observation_radius + 1,
22.                               position_axis_1 - observation_radius:
23.                               position_axis_1 + observation_radius + 1]
24.
25.     return observation
```

Dato un drone in una simulazione, calcola il suo spazio di osservazione. Nel caso in cui lo spazio di osservazione sia composto da spazi fuori dalla mappa, esegue l'allargamento della mappa stessa.

Parametri

- **batch_index : *int***
  Indice della simulazione parallela su cui effettuare l'operazione.

- **drone_index : *int***
  Indice del drone su cui effettuare il calcolo.

Restituisce

- **observation : *ndarray (2D)***
  Array bidimensionale contenente lo spazio di osservazione di un drone.
  Tale spazio è composto da elementi pari a:
  - 0 se la casella della mappa è occupata dal solo terreno;
  - 1 se la casella della mappa è occupata da un drone, da un ostacolo o da una traccia stigmergica;
  - -1 se la casella contiene un elemento fuori dalla mappa.

3.26 __enlarge_map_view()

```
def __enlarge_map_view(self, position_axis_0, position_axis_1,
                               observation_radius, environment):

    enlargement_axis_before_0 = 0
    enlargement_axis_after_0 = 0
    enlargement_axis_before_1 = 0
    enlargement_axis_after_1 = 0

    if position_axis_0 - observation_radius < 0:
        enlargement_axis_before_0 = abs(position_axis_0 - observation_radius)

    if position_axis_0 + observation_radius >= self.__targets.shape[0]:
        enlargement_axis_after_0 = position_axis_0 + observation_radius -
                                                    (self.__targets.shape[0] - 1)

    if position_axis_1 - observation_radius < 0:
        enlargement_axis_before_1 = abs(position_axis_1 - observation_radius)

    if position_axis_1 + observation_radius >= self.__targets.shape[1]:
        enlargement_axis_after_1 = position_axis_1 + observation_radius -
                                                      (self.__targets.shape[1] - 1)

    enlarged_map = np.pad(environment, [(enlargement_axis_before_0,
                                         enlargement_axis_after_0),
                                        (enlargement_axis_before_1,
                                         enlargement_axis_after_1)],
                          constant_values=(-1))

    return enlarged_map, position_axis_0 + enlargement_axis_before_0,
                          position_axis_1 + enlargement_axis_before_1
```

Ingrandisce le dimensioni della mappa fornita: sono aggiunti spazi esterni inizializzati a -1, ma solamente dove necessario (lo sforamento della mappa è considerato in base alle coordinate del punto e al raggio di osservazione forniti).

Parametri

- **position_axis_0 : *int***
  Posizione del drone sull'asse 0 (righe).

- **position_axis_1 : *int***
  Posizione del drone sull'asse 1 (colonne).

- **observation_radius : *int***
  Raggio necessario per stabilire lo sforamento della mappa. Si osservi che il raggio è applicato a partire dalle coordinate fornite e deve comprendere la dimensione del drone se si vuole che sia considerata.

- **environment : *ndarray (2D)***
  Array bidimensionale che rappresenta la mappa dell'ambiente da ingrandire.

Restituisce

- **enlarged_map : *ndarray (2D)***
  Array bidimensionale che rappresenta la mappa dell'ambiente espansa.

- **new_position_axis_0 : *int***
  Posizione del drone sull'asse 0 (righe) nel sistema di riferimento della nuova mappa.

- **new_position_axis_1 : *int***
  Posizione del drone sull'asse 1 (colonne) nel sistema di riferimento della nuova mappa.

3.27 __environment_info()

```
def __environment_info(self):
    info = {
        "Drones position - float": self.__drones_position_float,
        "Drones position": self.__drones_position,
        "Drones velocity": self.__drones_velocity,
        "Targets achieved": self.__targets_achieved,
        "Stigmergy Space":  self.__stigmergy_space
    }

    return info
```

Costruisce un dizionario contenente informazioni aggiuntive sull' ambiente e sui droni di ciascuna simulazione.

Restituisce

- **info : *dict***
  Raccolta di informazioni sui droni e sull'ambiente di ciascuna simulazione

3.28 render()

```
def render(self):
    if self.__rendering_allowed:
        environment = np.copy(self.__environment_bitmap)
        drones = np.sum(self.__drawn_drones[0], axis=0)
        stigmergy_space = self.__stigmergy_space[0]

        environment[drones == 1, :] = environment[drones == 1, :] + self.__drone_colour
        environment[environment > 255] = environment[environment > 255]//2

        for index in range(stigmergy_space.shape[0]):
            environment[stigmergy_space[index] > 0, :] += self.__stigmergy_colours[index]
            environment[environment > 255] = environment[environment > 255]//2

        self.__image_semaphore.acquire()
        np.copyto(self.__image, environment)
        self.__image_semaphore.release()
```

Se il rendering è abilitato, aggiorna il contenuto dell'immagine utilizzata per il disegno a video, aggiungendo le informazioni dinamiche (droni e tracce stigmergiche) all'ambiente "fisso". La chiamata

a questo metodo non modifica direttamente il disegno a video: questo è effettuato da un timer interno allo strumento grafico.

3.29 reset()

```
def reset(self):
    observation_dimension = 2 * (self.__drone_size + self.__observation_range) + 1
    observations_table = np.zeros((self.__batch_size, self.__amount_of_drones,
                                   observation_dimension, observation_dimension))

    self.__init_drones_parameters()
    self.__init_drones()
    self.__init_stigmergy_space()

    # Initial observation
    for batch_index in range(self.__batch_size):
        for drone_index in range(self.__amount_of_drones):
            observations_table[batch_index, drone_index] =
                                        self.__get_observation(batch_index, drone_index)

    return observations_table
```

Reimposta l'ambiente tramite la reinizializzazione dei droni e dello spazio stigmergico, in modo tale che possa essere avviata una nuova simulazione.

Restituisce

- **observation :** ***ndarray (4D)***
  Array contenente, per ogni simulazione parallela, lo spazio di osservazione iniziale di ciascun drone. Tale spazio è composto da elementi pari a:
  - 0 se la casella della mappa è occupata dal solo terreno;
  - 1 se la casella della mappa è occupata da un drone, da un ostacolo o da una traccia stigmergica;
  - -1 se la casella contiene un elemento fuori dalla mappa.