SmallWorl2D (link to download)

"Specs":

- Small and simple (but rich/flexible enough)
- Manageable and didactic (student-oriented)
- Emphasis on robot algorithms (control-oriented)
- "Continuous" time: asynchronicity (but simple physics)

Choices:

- Python $\longrightarrow$ ROS (BTW, Python $\Rightarrow$ Some Fun Required)
- Time Scaling & Verification $\longrightarrow$ Distributed Simulation

Version: 2.2 (still in construction)

License:

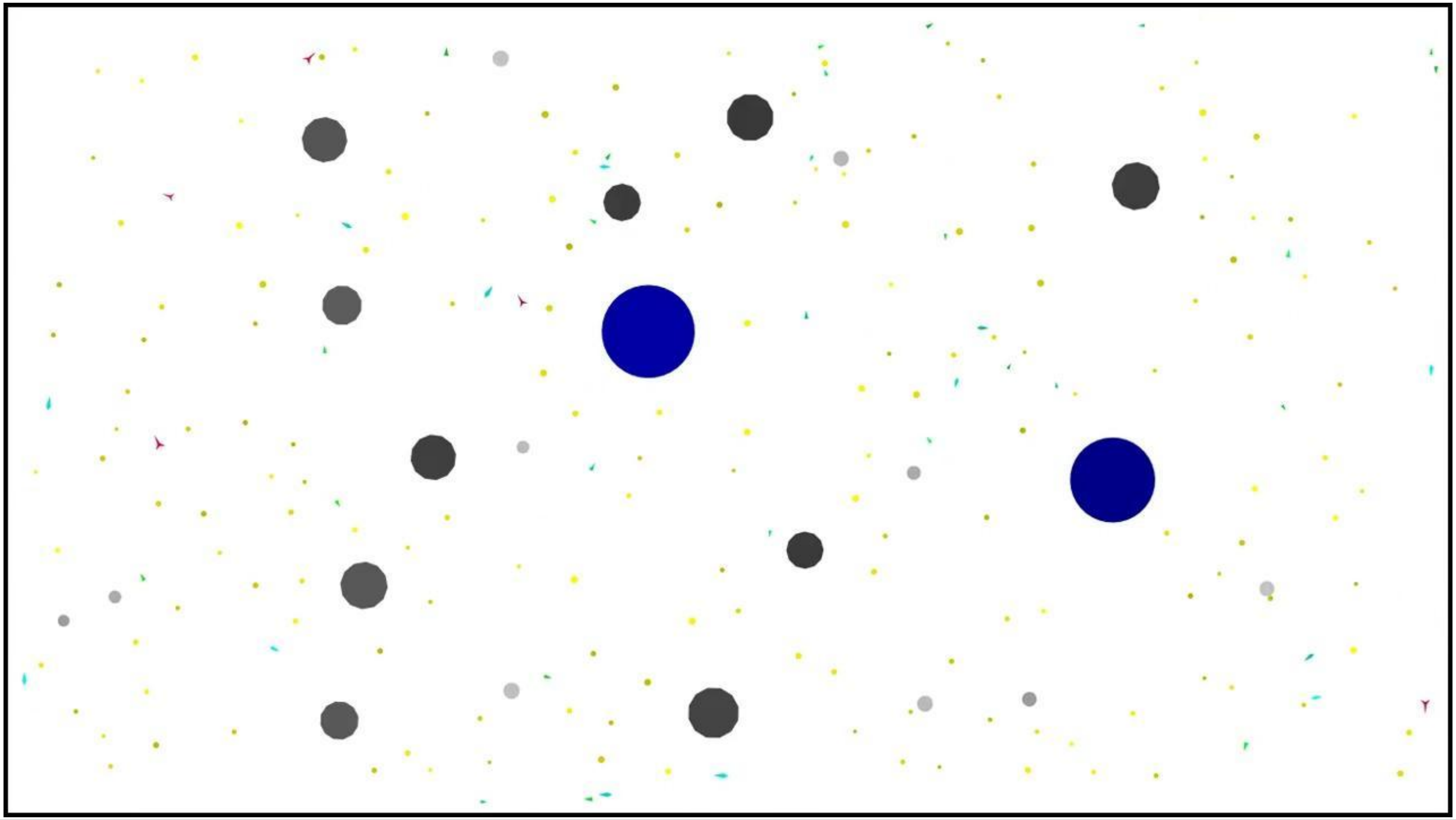
SmallWorl2D Creation (“Rationale”): In the Beginning there was Nothing…

*A **Space** is a rectangle populated by different kinds of **Body**'s, some of which are static (**Obstacle**’s, **Nest**’s, **Food**’s) and some of which are mobile, the **MoBody**’s.*

***MObstacles** are dumb MoBody’s, but there are also the more interesting, animated, **AniBody**’s (**Mobot**’s, **Killer**’s, **Shepherd**’s).*

*An AniBody typically has (one or more) **Soul**’s that animate it, and sometimes **Knows** something that influences its state and which it can tell to peers.*

A World Without (Control) Law



SmallWorl2D Depends

```python
## Front matters
# Not mine imports
import os
from pathlib import Path
from time import time, localtime, strftime
import numpy as np
from random import seed, random, uniform, choice
from shapely.geometry import Point, Polygon, LineString
from shapely.affinity import translate, rotate, scale
from shapely.ops import nearest_points
from point2d import Point2D
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib.animation import FFMpegWriter # requires having ffmpeg installed, from https://ffmpeg.org/
# Mine imports
from gadgETs import coordlistr, tuplestr, pipi, voronoi
# Personalities
working_dir = str(Path.home())+'whatever you like'
os.chdir(working_dir) # Working_dir for files
```

“Global” variables

They’re set in Space.init()

```
randomseed=0 # Random seed for reproducibility
visual=False # to see the world (and selected KPI's) on the run, and record a movie
shoul=True # to show Soul's or not
showconn=False # to show the graph of connections or not (only in visual mode)
# Info detail
loginfo=False # dump info to a log file
movietxt=False # dump the data to make a movie afterwards
# Canvas sizing; default aspect ratio W:H; coords (-W:W,-H:H)
SS=15 # Screen Size SS in ches
W=16
H=9
room=[(-W,-H),(W,-H),(W,H),(-W,H)]
# Time Scale (check or set OK!)
TS=1 # simulation-time seconds per real-time second (adjust to case)
fps=10 # frames per second in the movies (10 or 25 are usual choices)
# Max vel's should be a fraction of the following vN and wN (intended for relatively small
displacements per redraw)
vN=4/TS # traverse 4 units distance in 1 sec real-time (32 canvas-width in 8 secs)
wN=8*np.pi/TS # 4 full revolutions in 1 secs real-time (quite fast)
```

class Space

A rectangle populated by Body's

Data:

```
name
fig,ax,limits (W, H)
  # the space is a white rectangle 2Wx2H, limited (or not) ‘v’ or ‘h’
  # x, y coords of everything range from -W to W, -H to H
bodies=[] # list of Body's in this Space
dist,R,conn,conngraph # distances and connections
time,TS,t0,T,updates,avgT (fps, T=TS/fps)
   # timing, and statistics
```

class Space

Functions:

```
__init__(self,name,TS,R=1,limits='',...“globals”...)
""" Also creates and inits files and graphics
resetime(self,t0)
""" Resets the t0 and time of Space and its Body's and Soul's
update(self)
""" Redraws whatever newer than self.time and advances its self.time
has_been_closed(self)
""" Returns True when the figure where self is drawn is not active
close(self)
logprint(self,string) """ Log printing from outside
```

class Space

```
## Body management functions
findbody(self,name):
""" Returns the index in self.bodies of the Body named so
typindices(self,type)
""" Returns the indices in self.bodies of the Body's of the type
update_dist(self)
""" Updates the matrix of dist between all Body’s
update_conn(self)
""" Updates the list of conn pairs between AniBody's of the same type
def graph(self,type)
""" Returns the graph formed by the AniBody's of type
```

class Space

```
## Body management functions (cont.)
fits(self,new,where,noverlap=True,safe=0)
""" Returns True if the new Body fits in where
remobodies(self,ko,why='unspecified reason')
""" Removes (offs) all the self.bodies in list ko

## Perception functions
nearby(self,i,type,r,rng=np.pi)
""" Returns a list with the Body's of type "visible" from Body i
RnB(self,i,type,r,rng=np.pi,fast=True)
""" Simulates a range&bearing sensor (*)
```

class Space

```
## Perception functions (cont.)
clearway(self,i,type,r,rng=np.pi,N=30,th0=0)
""" Returns the closest to th0 angle with clear straight way (*)
nearest(self,i,type,r,rng=np.pi)
""" Returns the nearest to Body i of the nearby Body's of type
nearestpoint(self,i,b):
""" Returns the coords of the nearest to Body i point of b
incontact(self,i,type):
""" Returns a list with the Body's of type in contact with Body i
```

class KPIdata

The key performance indices

Data:

```
name,fig,ax # x from 0 to now, y’s from 0 to 1
time,t0,T,updates,avgT
KPI,KPIplot # [(i,'{.-:o+*}{rgbcmyk}'),...]
```

Functions:

```
__init__(self,name,TS,T,KPI0,KPIplot=[],...)
resetime(self,t0)
update(self,KPI) """ Writes/Plots the last KPI values
has_been_closed(self), close(self)
```

class Body

Something (depicted by a polygon) in a Space

Data:

```
space,name,time,t0,updates,avgT,on
pos,th,area,vertices,r_encl,fc,pp
```

Functions:

```
__init__(self,space,name,vertices,pos,th,area,fc)
__repr__(self)
switch(self,on) """ Turn on or off, i.e., "remove"
update(self)
index(self) """ Returns the index in self.space.bodies
```

Body subclasses

- Obstacle, Nest, Food, ***MoBody***
- MoBody: MObstacle, ***AniBody***

  (Extra) Data: `v, w, vth, v_max, w_max`

  (Extra) Functions:

  ```
  teleport(self,pos=None,th=None)
  set_vel_max(self,v_max=0,w_max=0)
  cmd_vel(self,v='=',w='=',vth='=')
  ```

- AniBody: Mobot, Killer, Shepherd

  (Extra) Data: `souls=[], knows=None`

class Soul

Something *within* a (Ani)Body that controls *it*: manipulates *its* behavior in response to *its* environment, possibly using what *it* knows (*)

Data:

```
body,T,time,t0,updates,avgT
fc,vertices,pp
```

Functions:

```
__init__(self,body,T,fc=None)
update(self)
```

Soul Subclasses

GoTo

""" Low-level Soul to go somewhere by several step-by-step cmd_vel's. When no where to go keepgoing, stop, wander or zigzag. Avoids obstacles

Voronoid

""" Soul of a Voronoi's-based mobile sensor in a network covering a static region, Cortes2004 like

... Your work

class Knowledge

What a (Ani)Body knows

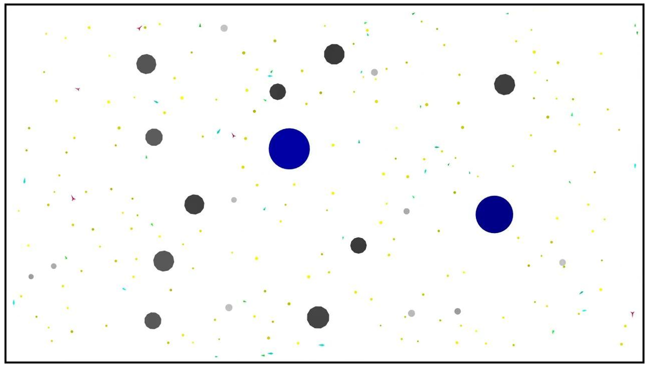
Data: `body, state`

Functions:

```
    __init__(self,body,state='idle')
    set_state(self,state)
    tell_state(self)
```

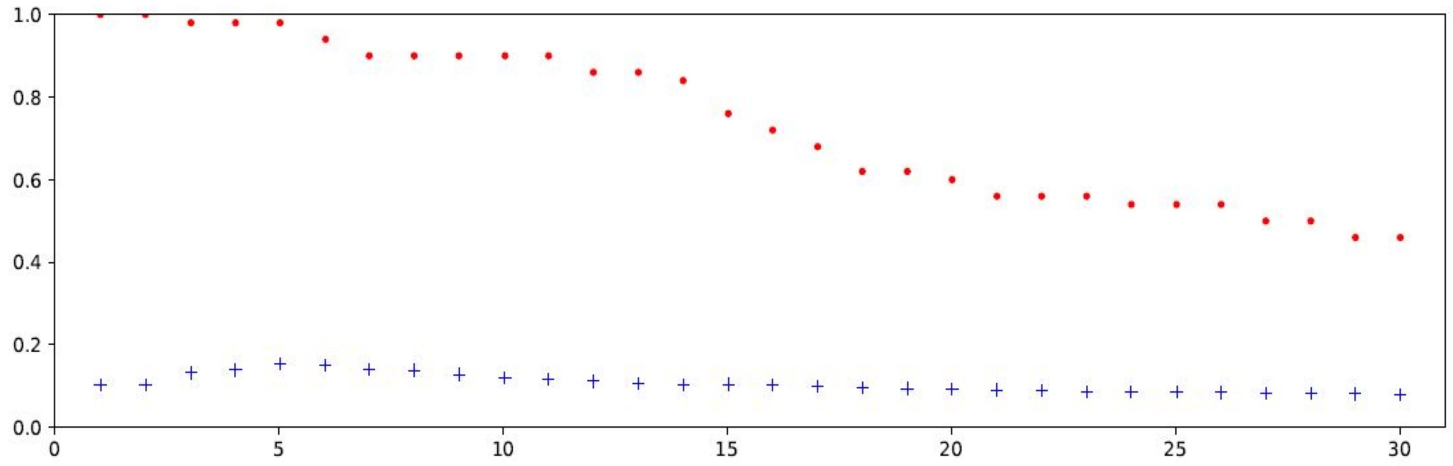
Knowledge Subclasses: Your work

small_worl2d MAIN



A meaningless demo of almost everything

(and a pattern for the main code in Your work)



KPIs:

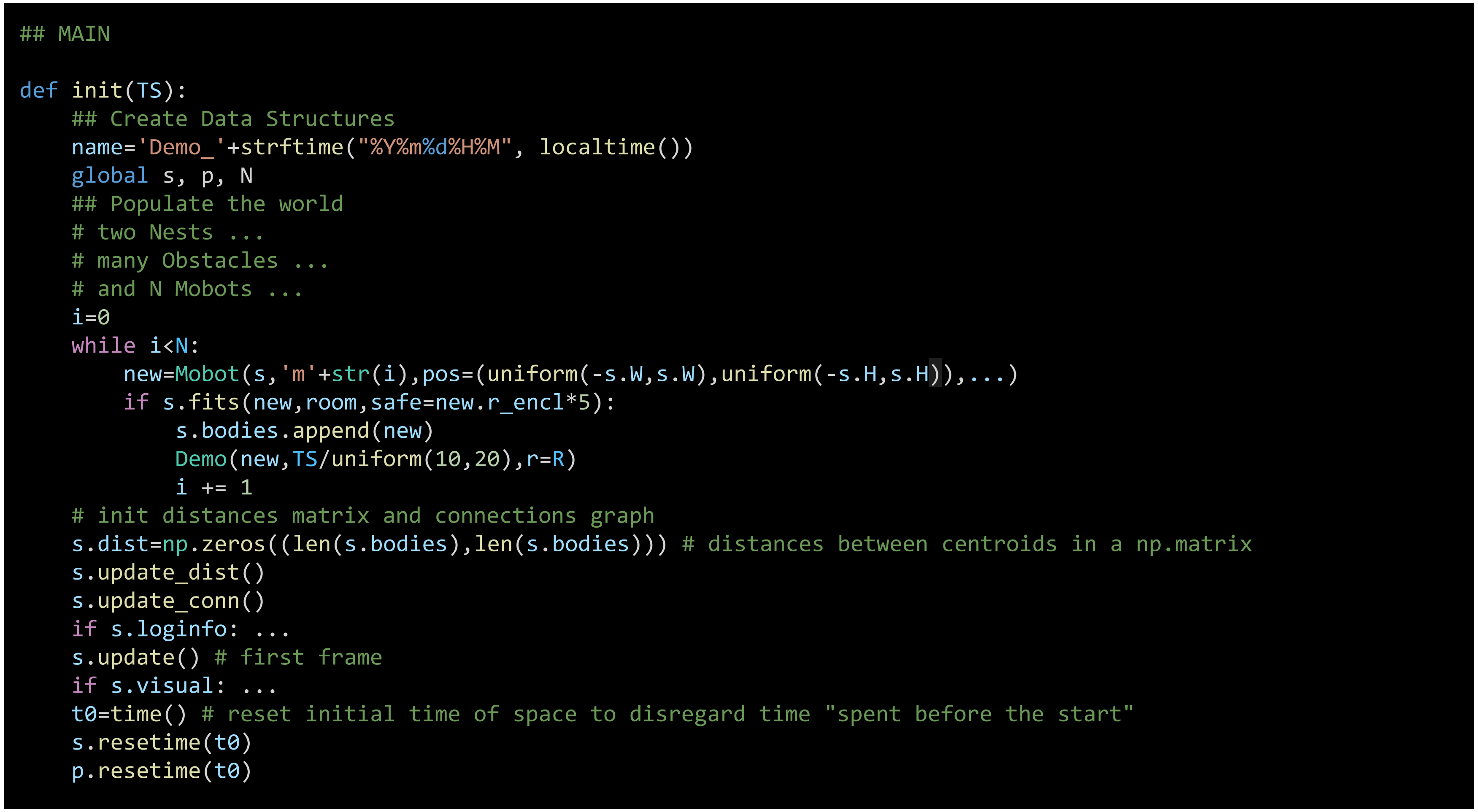
- Fraction of surviving Mobots
- Average Mobot update time (s) / Space T

Your Work

Create exercise modules like following Demo.py, with your Soul's, Knowledge's, etc, and a ## MAIN, and analyse behavior (KPI's vs parameters)

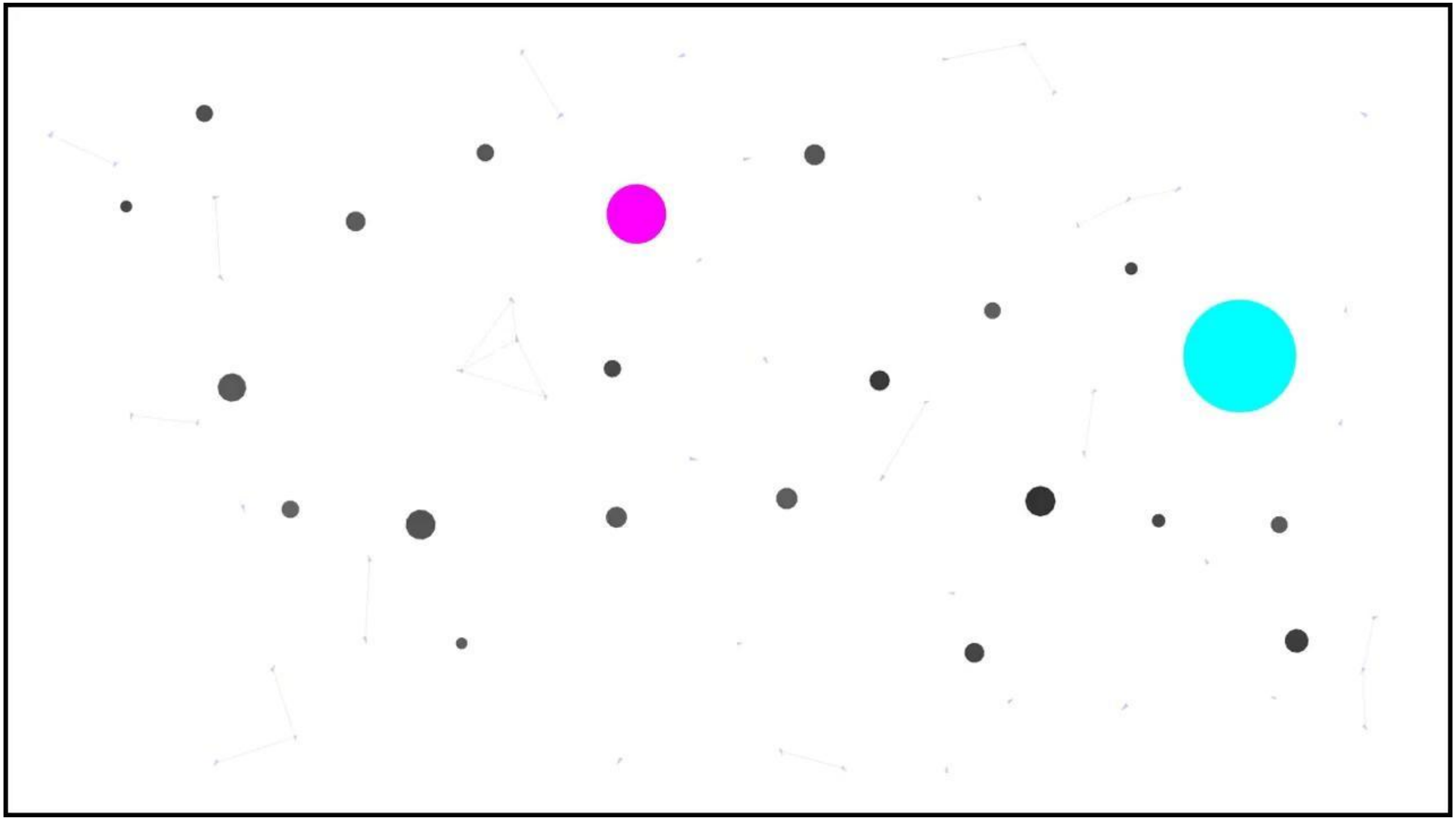
```
## A Demo
...
from small_worl2d import Space, KPIdata, Obstacle, MoBody,Nest, Mobot, Soul, GoTo, Knowledge

## A couple of functions related to quadrants
def qdrnt(body): ...
def cmykdrn(qdrnt): ...

class GoToQdrnd(GoTo):
    """ A specialization of the GoTo Soul that zigzags randomly towards a quadrant destination """
    def __init__(self,body,T,destination=0): ...
    def set_dest(self,destination): ...
    def update(self): ...
```

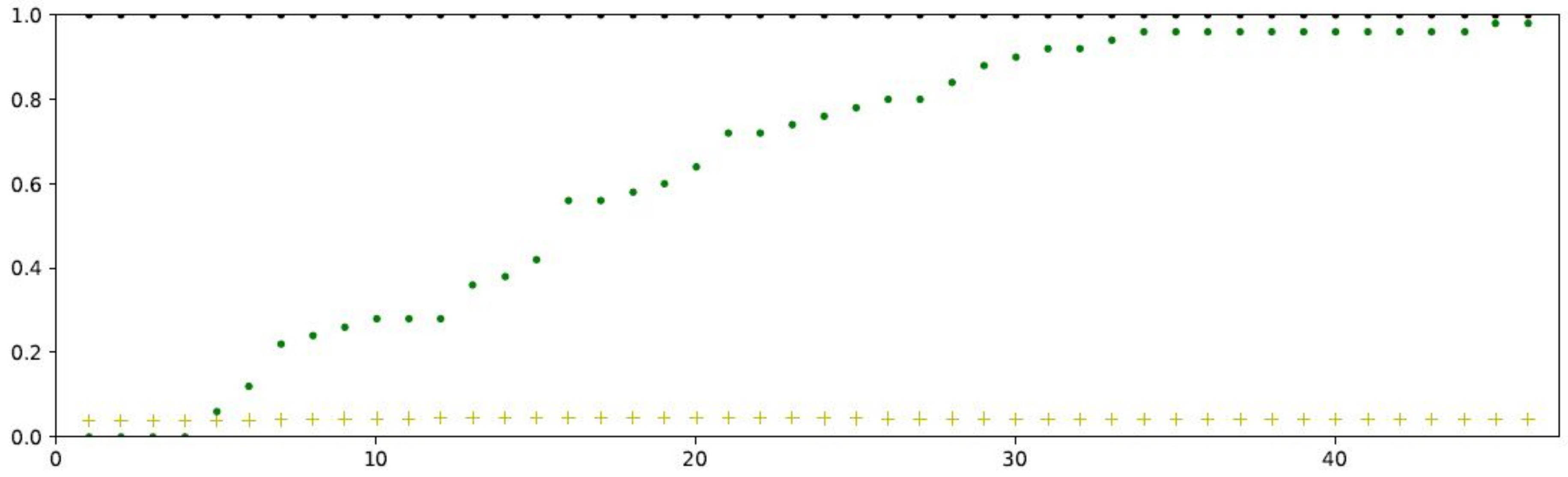
```python
class Nestxists(Knowledge):
    """ A trivial specialization of Knowledge, actually it is just the basic Knowledge
        with an integer state that informs of the integer quadrant were there is a Nest (1:4),
        0 when no Nest known
    """
    def __init__(self,body,state):
        super().__init__(body,state)

class Demo(Soul):
    """ A Soul that makes stay by the Nest known by my own contact or through my contacts """
    def __init__(self,body,T,r=1,rng=np.pi):
        self.r=r
        self.rng=rng # might be smaller than pi, but not too much
        GoToQdrnd(body,T) # requires a GoToQdrnd soul in the same body
        self.GoToQdrnd=body.souls[-1] # this way it knows how to call it
        Nestxists(body) # this Soul needs a Knowledge in its Body to work
        super().__init__(body,T)
    def update(self):
        current=self.body.knows.tell_state()
        if current==0:
            i=self.body.index()
            if self.body.space.incontact(i,Nest):
                current=qdrnt(self.body)
                self.body.knows.set_state(current) # I've been in one!
            else:
                neigh=self.body.space.nearby(i,type(self.body),self.r,self.rng)
                for n in neigh:
                    current=n.knows.tell_state()
                    if current>0:
                        self.body.knows.set_state(current) # If some neigh is aware of Nests, then so I am
                        break
            if current>0: # changes color and set destination to quadrant
                self.body.fc=cmykdrn(current) # this is NOT the usual way to show a soul
                self.GoToQdrnd.set_dest(current)
        super().update()
```



```python
## MAIN

def init(TS):
    ## Create Data Structures
    name='Demo_'+strftime("%Y%m%d%H%M", localtime())
    global s, p, N
    ## Populate the world
    # two Nests ...
    # many Obstacles ...
    # and N Mobots ...
    i=0
    while i<N:
        new=Mobot(s,'m'+str(i),pos=(uniform(-s.W,s.W),uniform(-s.H,s.H)),...)
        if s.fits(new,room,safe=new.r_encl*5):
            s.bodies.append(new)
            Demo(new,TS/uniform(10,20),r=R)
            i += 1
    # init distances matrix and connections graph
    s.dist=np.zeros((len(s.bodies),len(s.bodies))) # distances between centroids in a np.matrix
    s.update_dist()
    s.update_conn()
    if s.loginfo: ...
    s.update() # first frame
    if s.visual: ...
    t0=time() # reset initial time of space to disregard time "spent before the start"
    s.resetime(t0)
    p.resetime(t0)
```

```python
TS=10 # adjust so that the first KPI below is OK
init(TS)
end=False
while not end: # THE loop
    ko=[]
    avgT=0
    count=0
    for b in s.bodies: # movement update
        if isinstance(b,MoBody):
            b.update()
            avgT+=b.avgT
            count+=1
    avgT/=count
    s.update_dist()
    s.update_conn()
    for b in s.bodies: # collision management
        i=b.index()
        if i>=0 and isinstance(b,(Obstacle,Mobot)):
            ko+=s.incontact(i,Mobot)
    s.remobodies(ko,'collision')
    if time()>s.t0+(s.updates+1)*s.T: # graphics refresh (new frame)
        s.update()
    if time()>p.t0+(p.updates+1)*p.T: # KPI's computation
        KPI=[avgT/(s.TS/s.fps),...]
        ...
        p.update(KPI)
    end=ts.has_been_closed() or p.has_been_closed() or ...
else:
    s.close()
    p.close()
```





KPIs:

- ● Fraction of surviving Mobots
- ● Fraction of Mobots who know a Nest
- ● Average Mobot update time (s) / Space T

Exercise 1: PFSM

Create a Space with two Nest's in different (random) quadrants, with different areas (a=1 A=1.1, 2, 5), and N (25, 50, or 100) Mobot's in random initial positions.

Design a PFSM micro-behavior (Souls and Knowledge) for the Mobot's such that they end up aggregating in the large Nest.

The perception range of a Mobot is r=1, rng=π (omnidirecional). The communication range between Mobot's is R, equal to r, 2r, or 4r (omnidirectional).

Besides perceiving other Body's nearby or incontact (but not their position or size), these Mobot's know the quadrant (1 to 4) where they are, and they can remember, increase, compare or replace (only) a few integer numbers. They can tell them to peers.

The KPI's must include the fraction of Mobot's aggregated in the large Nest, over time, and the fraction of surviving Mobot's.

Analyse (using simulation replications) the behavior depending on the parameters of the case (A, N, R) and the tuning of your parameters (probabilities, etc).

(Proper aggregation - be careful with collisions! - within the nest is an extension.)

Exercise 1: Hints/Suggestions

A Mobot (PFSM, call it "you") may have the following states:

1. Knows nothing
2. Knows one Nest
3. Knows the two Nest's (but is not yet convinced about which is the largest)
4. Is convinced about which Nest is the largest
5. Is resting in the (supposed to be) largest Nest, aggregated with others
6. ...

In each of the states, "you" do different things, particularly moving, often in mode "wander" or "zigzag", in the appropriate fashion - probably making use of the GoToQdrnd Soul from Demo.py. For instance, it does make little sense to wander in a quadrant where there are no Nest's when both are already known. Of course, in whichever state, you talk to peers nearby to share information (in both directions).

The state transitions are often affected by (state dependent) probabilities. Even a Mobot in state 5 should have some (small) probability to change to state 3, and go re-visiting the two Nests.

Besides the fraction of Mobot's aggregated in the large Nest , the number of Mobot's in some states can be useful (insightful) KPI's, in order to tune the parameters (probabilities, etc).

Exercise 2: Boids (basic)

Program, adjust, and describe a Boid Soul for Mobot's.

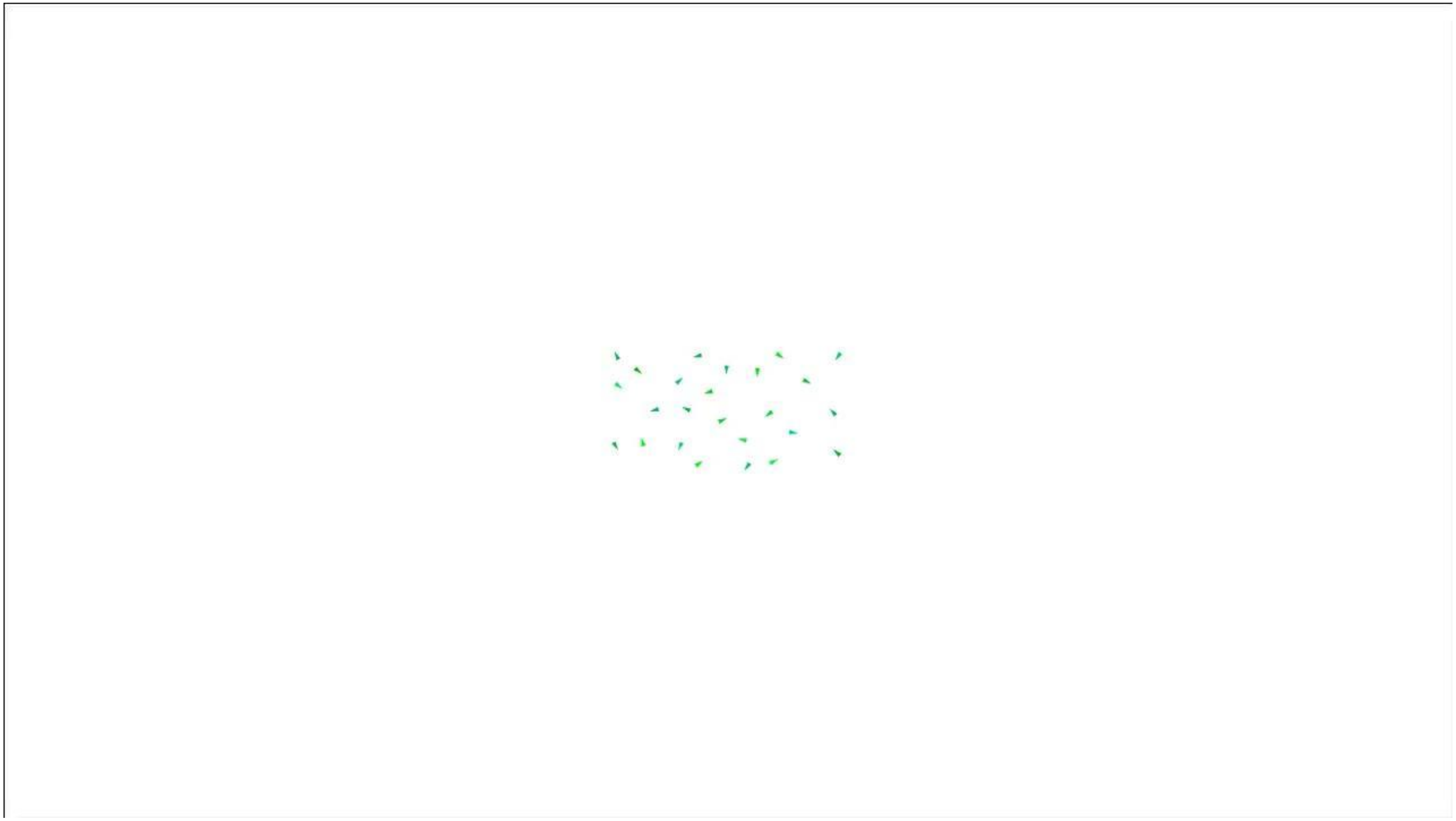
You can (should) adapt the literature proposals, e.g., use scaling factors for the flocking (steering) vector components, use alternative functions for the cohesion/repulsion or potential gradient, use or not heading perception, take or not into account own heading in alignment, make the motion control magnitude dependent or not, etc, and of course adjust the parameters.

In your report, state complete and clearly your solution, and in case you've tried alternative solutions justify briefly your final choice.

Demonstrate it with a video with N=25 Mobot's (*).

Evaluate KPI's (*) wrt. some parameter(s) of your interest (e.g., different adjustments of critical control settings, N, etc).

Basic Boids



Exercise 2: Boids (extensions)

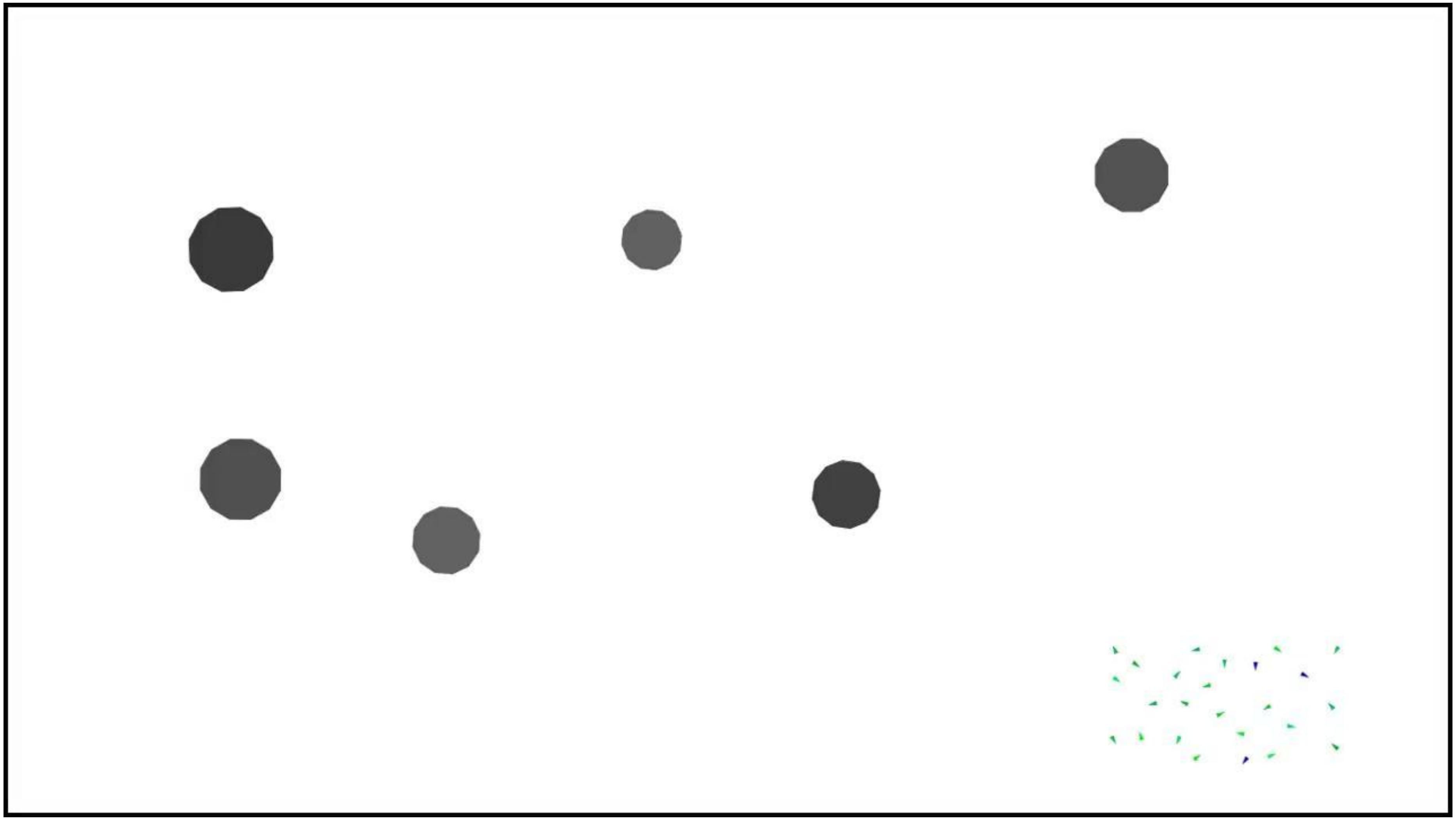
Extensions:

- Influence of a fraction of informed or Mobot's who have a favorite direction (and include accuracy as a KPI)
- Avoidance of Obstacle's or/and evasion from Killer's

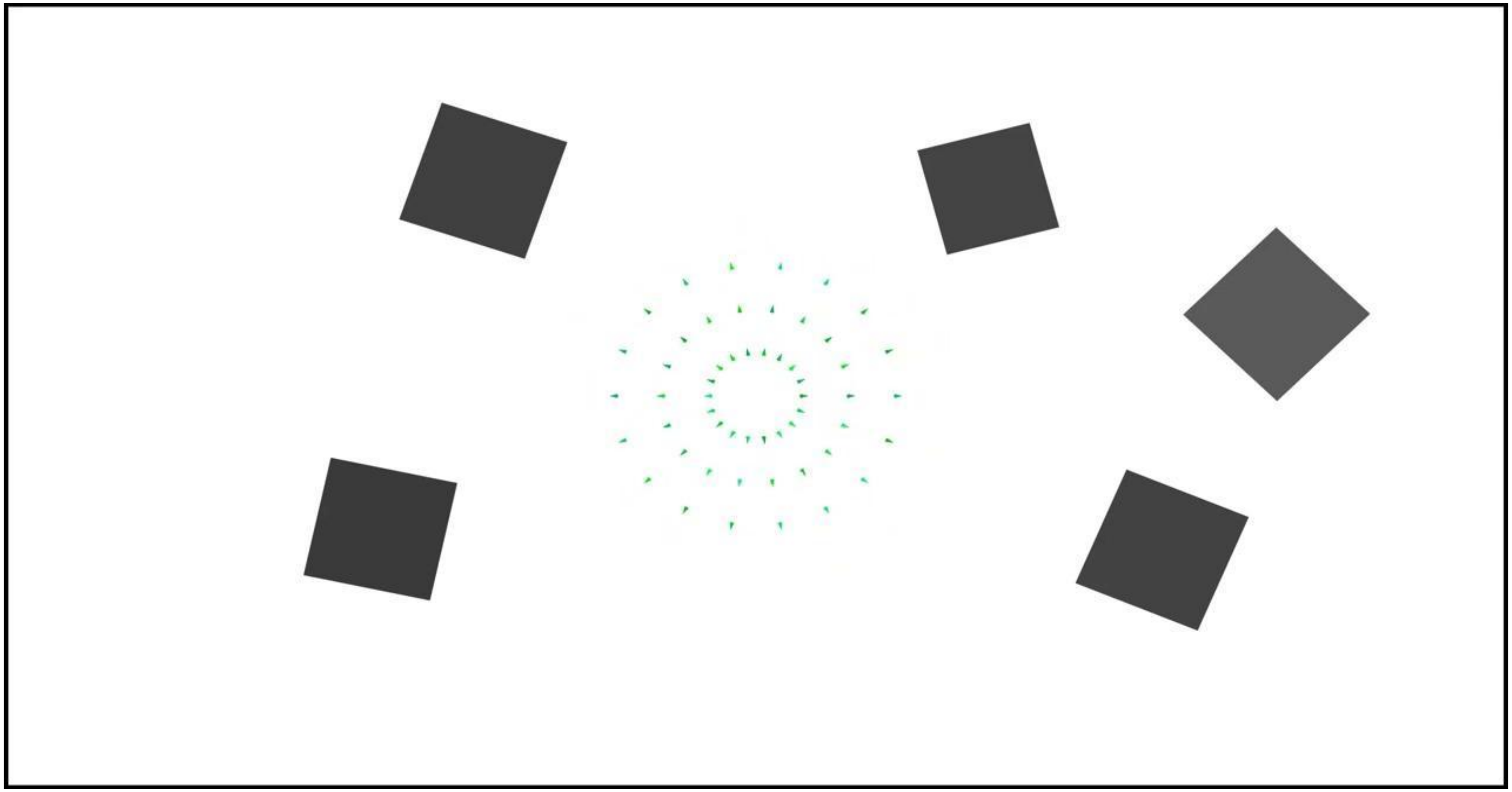
Off-topic Exercise 2x:

Adapt your algorithm to get something like the behavior in the following video (I call them Wowid's, and it requires a bit of out-of-the-box thinking).

Extended Boids



Wowids



Exercise 3: Voronoids

Program a VoronoiDynamic Soul for Mobot's with 360° perception of radius r and high level control period TS, based on the basic Voronoid Soul

- Compute current target region interpolated between the previous and next waypoints (*), using a constant velocity, a fraction of r in one TS
- Estimate next movement (assume equal to last one)
- Compute current Voronoi cell (this is already done)
- GoTo (*) its centroid plus estimated movement
- Update your KPI's

Exercise 3: Voronoids

Define an “interesting” mission, several waypoints in a 16x9 SmallWorl2D, and:

- Choose an appropriate r “just enough” for your waypoints areas and the number of Mobot’s
- Simulate 50 Mobot’s covering the first waypoint, and then the subsequent ones. Record a movie.
- Evaluate KPI’s (*) from 5 runs with different values of N (25, 50, 100) and η (0.2, 0.4, 0.6).