Boids Project

Colm McElwain, Naoise McManus

May, 2024

Figure 1: A photo by James Crombie of Starlings over Lough Ennell [Bro]

Abstract

The Boids algorithm, created by Craig Reynolds, uses very simple rules to create simulations of swarms of entities, giving results very similar to real life flocks of birds or swarms of insects [Rey]. The purpose of this project is to construct a model using this method, and then programme and observe the effects of different initial parameters. The original algorithm created by Craig Reynolds considers three accelerations, which are weighted, added together, and lastly that resulting vector is normalised. This is done simultaneously for each entity in the system, at each step in time.

After this first simulation, we plan on expanding the scope a little bit. We will see how initial conditions and different weighting affect the swarm's behaviour, observe how the addition of wind disperses the swarm and how the swarm then reforms, and lastly we will add a predator to the system and see how the swarm handles that.

Introduction

For this project we will be looking at the emergent flocking behaviour of sheps. A bunch of sheps flocking together may be written as a flock of sheep. These sheps are then free to walk in three dimensions, and will be told which direction to accelerate in, based on a few simple rules.

The original algorithm by Craig Reynolds uses three such rules, to calculate three "accelerations" on each shep in the flock. These accelerations are:

- Cohesion: The cohesion acceleration will tell each entity to move towards the average position of the other entities in its immediate vicinity.
- Alignment: The alignment acceleration will tell each entity to adjust its acceleration based of the velocities of the other entities in its immediate vicinity.
- Separation: We don't want the entities to hit each other, so we introduce an acceleration whenever they get too close.

These accelerations will be discussed later on. For now though, let's do up a sort of "flow-chart" for this project. We will be splitting our project into a couple of segments. The first will be to strictly programme the above accelerations, and plot it in matplotlib. After this, we will try and make the flock's movement a bit more realistic, and optimise the code a little bit. When we are happy the swarm looks and behaves as it should, we then plan on seeing how outside factors affect the flock; specifically some wind, and a predator.

Figure 2: An example of a shep that is topologically equivalent to a sphere [Bul]

Method

In this section we'll discuss the method by which we modelled the sheps. The script uses the semi-implicit Euler-Cromer method for creating the dynamics of the system. As we're using Euler-Cromer, this will take on an error of order $O(\Delta t^2)$.

The velocity and position are both given as functions of each other, i.e.,

$$\frac{dx}{dt} = f(v,t) \text{ and } \frac{dv}{dt} = g(t,x)$$

This gives us the relations that our system evolves by:

$$v_{n+1} = v_n + g(t_n, x_n)\Delta t$$
$$x_{n+1} = x_n + f(t_n, v_{n+1})\Delta t$$

where Δt is our time increment.

We also note that sheps are not omniscient and are limited by how much they can see. We've allowed our sheep to see 90 degrees to either side. To implement this we assume that the sheps are looking in the direction they are flying and take the dot product of the displacement between two sheps and the relative velocity between two sheps. If this yields a number less than 0, we don't consider the relation between these two birds.

Acceleration is where the system becomes interesting. Our sheps follow the algorithm created by James Reynolds. Reynolds gives the following three relations. For simplicity, c_i will denote a constant. Each constant is unique to each force.

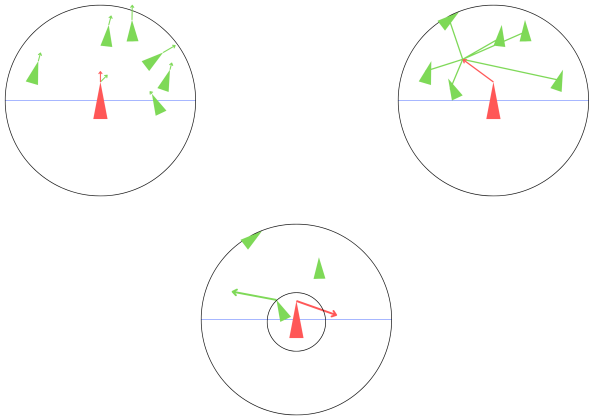

Figure 3: Effect of Alignment, Cohesion and Separation on a shep

Alignment

Our sheps can only care about other sheps in a nearby radius. Alignment scans sheps in a radius and follows the general direction of their movement. This is incremented using the following equation:

$$a_{n+1} = a_n + c_1 \cdot v_{rel}$$

Cohesion

To make our sheps remain together as a flock and not fly off into the void forever we use a cohesive force. This force also considers sheps only within a radius and moves the sheps towards the centre of mass of neighbouring sheps. We use Reynolds's following equation to simulate this behaviour:

$$a_{n+1} = a_n + c_2 \cdot r_{rel}$$

Separation

The sheps for separation scan a smaller radius around them for neighbours than for Alignment and Cohesion, if it detects any sheps within this smaller radius it applies a strong repellent force in the opposite direction of the neighbouring sheps stopping it from colliding and steering it away until they can no longer see those neighbours. This is implemented similarly to the other two but with a smaller area of influence.

Randomness and Boundary

We added two more forces to our system to aid with realism and to keep the renders somewhat centred. As sheps may not be rational creatures, they have a little bit of randomness in each of their decisions. We implemented this by rolling for a random number and applying that to the total acceleration of the shep.

To keep the sheps on screen to avoid them wandering off into the void forever, we added a boundary to the space; a 3 dimensional fence around a field if you will. If the sheps attempt to cross this boundary, they are pushed back with a substantial force to keep them in the field.

Total Acceleration

Our total acceleration then is given by the sum of each of the individual forces above. i.e.,

$$a = a_{align} + a_{separation} + a_{cohesion} + a_{random}$$

If the sheps have strayed outside of the field we can multiply this acceleration by our boundary force in the opposite direction of the shep's position vector. We can see this in the following pseudo-code:

```
if shep > boundary then
  calculate vector pointing opposite to boundary
  multiply vector by c_bound
else
  no boundary force

calculate total acceleration from sum of individual
                                  accelerations
```

Simulation

Having implemented the given forces, we were able to see emergent behaviour of our flock of sheep. The following figures show the evolution of the system after 0 steps, 50 steps and 100 steps.

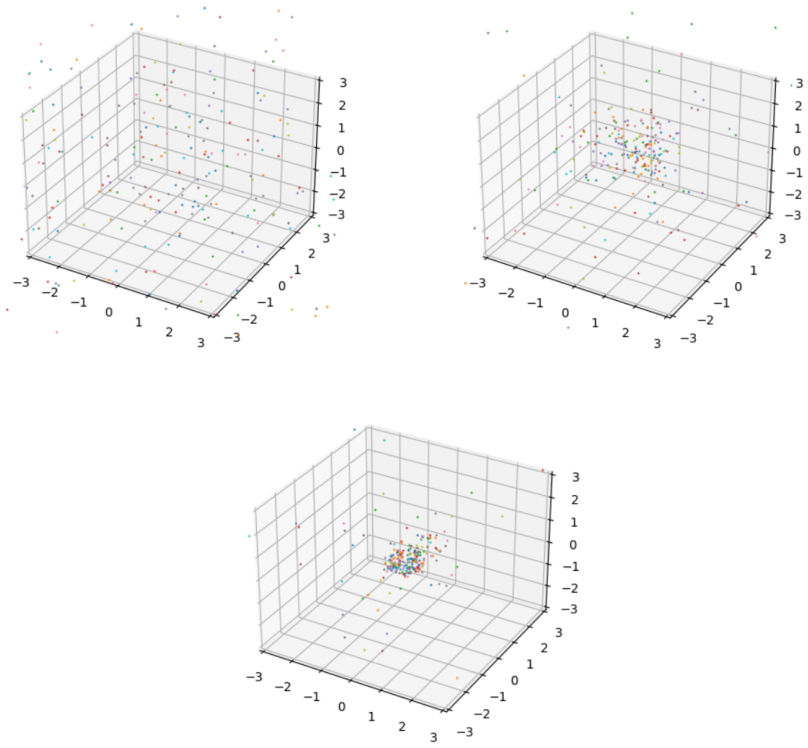

Figure 4: Simulation of 250 sheps after given time steps

We see after 100 frames, the sheps clump together. This behaviour remains from this point onwards. This is characteristic of a swarm of insects, such as mosquitoes.

To introduce more bird-like behaviour we added a constant acceleration to each of the sheps. This keeps the sheps from slowing down too much and clumping together, effectively adding a "bias" in a certain direction, much like birds flying to a particular location.

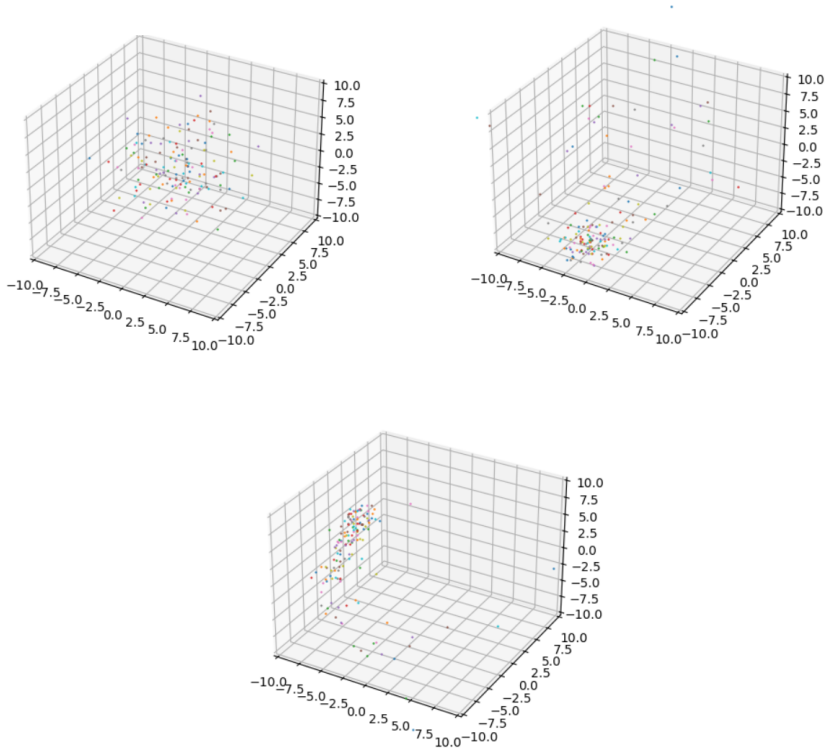

Figure 5: Simulation of 250 sheps after given time steps with constant acceleration

Environmental Factors

Wind

Currently, our sheps are unaffected by external factors: they are free particles. Given that we'd like to realistically mimic the ways flocks form, our first environmental factor to consider will be wind. We want to see how much the flock gets disrupted, and how it recovers (or attempts to recover) from this disruption. Adding wind into the system is done by defining a new force on the sheps, a vector field, and adding that to the total acceleration. The force will be given as:

$$F = (\frac{-y}{\sqrt{(x^2+y^2)}}, \frac{x}{\sqrt{(x^2+y^2)}}, 0)$$

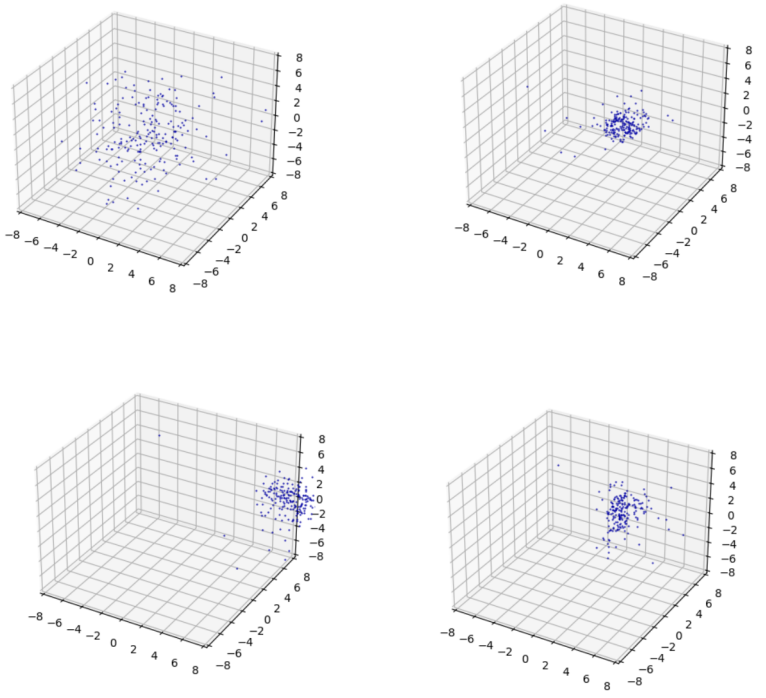

Figure 6: Simulation of 200 sheps in windy conditions at time steps 0, 25, 50 and 75

As we can see, the emergent behaviour evolves differently in this simulation, The flock quickly forms as usual, however after 50 frames the entire flock has been blown to the side. Although it fights against the wind after 75 frames, the entire swarm has been spread out flying in almost a disk against the wind.

Predator

This one is a slightly less natural environmental factor, but animals that flock do so as a form of protection against predators. So let's look at what happens when we get some predator to chase around our sheps.

Programming in the predator involved creating a new class, which we call "EvilShep". This then has roughly the same cohesion and alignment rules as normal sheps (albeit scaled differently), but has no seperation force (for obvious reasons). Then the regular sheps are told to avoid the predator with a very strong coefficient, which overrides all other forces when normalised. We have also taken the liberty to say that the sheps can sense the predator even without looking at them (albeit at a smaller range).

This gives us a very realistic looking animation, with the evil shep being faster than the sheps, but the swarm "confuses" the predator, as it hesitates when the swarm splits.

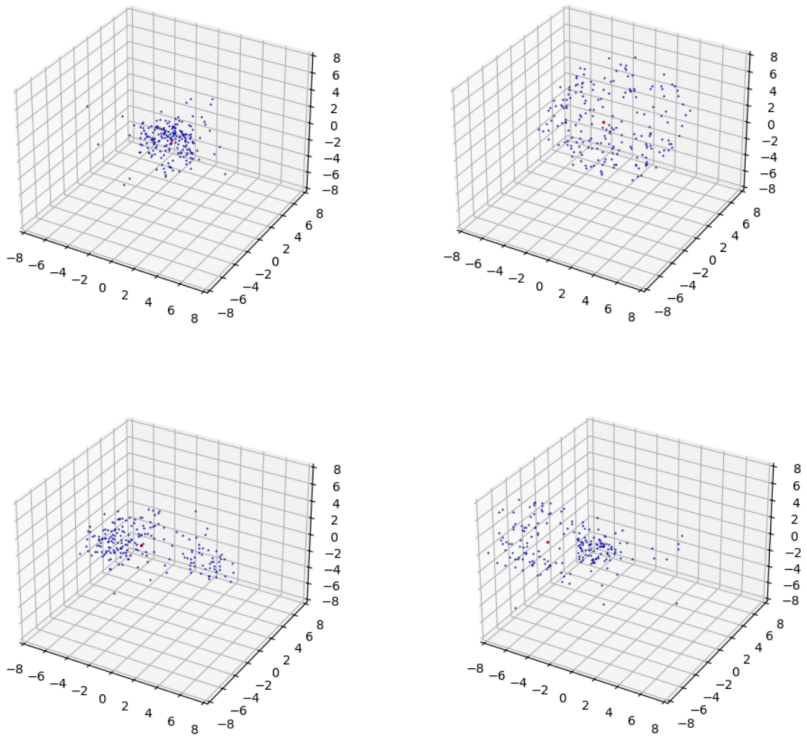

Figure 7: Simulation of 200 sheps being chased after by an evil shep at time steps 25, 50, 75, 100

Conclusions

For the first part of the project, we replicated the boids algorithm, and verified it worked. We produced swarm-like behaviour from very simple instructions.

After doing this we immediately noticed some flaws in our code and areas that the system could be quickly improved upon to create more realistic renders (most notably with the introduction of a constant acceleration bias). This resulted in the second iteration of our code.

After the flock was up and running, it was time to introduce external factors into the system, namely a wind blowing the sheps around the place and a predator to chase the sheps. This tested our system to see if it could react to environmental factors in a realistic manner.

For future developments of this algorithm, the implementation of a gravity field and a ground could create some interesting emergent behaviour as the flock sweeps down near it, or divebomb if heading towards the ground. This is something that future researchers could look at.

Overall, the boids algorithm is a very nice example of how some simple rules can create very complex systems. In this case, these simple rules give us incredibly true to life simulations of flocks. This is really highlighted at its best in the simulation with the predator. The predator has a hard time picking out any particular shep in the flock, similarly to how a school of fish avoids a shark. [Sny]

Bibliography & Figures

[Bro] Philip Bromwell. *The murmuration 'chasers' of Lough Ennell on a high again*. URL: https://www.rte.ie/news/2022/0301/1283498-murmuration-ennell-westmeath/. (accessed: 14.09.24).

[Bul] Bulbapedia. *Wooloo (Pokemon)*. URL: https://bulbapedia.bulbagarden.net/wiki/Wooloo_(Pok%C3%A9mon). (accessed: 14.09.24).

[Rey] Craig Reynolds. *Boids*. URL: http://www.red3d.com/cwr/boids/. (accessed: 14.09.24).

[Sny] Marty Snyderman. *Defense Mechanisms: How Marine Creatures Avoid Predation*. URL: https://dtmag.com/thelibrary/defense-mechanisms-how-marine-creatures-avoid-predation. (accessed: 14.09.24).

Appendix A: Boids Algorithm

```
"""
The first 'working' model using Boids.
This code:
-> creates a flock of sheep that walk in 3 dimensions,
-> applies the boids algorithm
-> plots each step of the process and compiles that into an
                                    animation, that is then saved as
                                    a gif

"""

import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from matplotlib.animation import FuncAnimation
from IPython.display import HTML


'''
this is done so we can see the effects of changing the initial
                                    conditions
in a more controlled manner
'''
rng = np.random.default_rng(seed=1234)

'''class creation. this class is to create the 'flock' of sheep
we give each 'shep' in the flock a random position and velocity'''
@dataclass
class Sheep:
    N: int
    X: float
    V: float

    def flock(self):
        temp = []
        for _ in range(self.N):
            x = self.X * rng.normal(size=3)
            v = self.V * rng.normal(size=3)
            temp.append(Shep(x, v))
        return np.array(temp)


'''class creation again. this class is to create the sheps in
                                    question.
'shep' is of course the singular of the word 'sheep', which we can
                                    then
make plural by adding an 's'.
these sheps are not constrained to walking on the ground and are
                                    free to walk
in 3 (spacial) dimensions'''
@dataclass
class Shep:

    x: list[float]
    v: list[float]
```

```
    '''I like to think its pretty obvious what this function will
                                        do given its name'''
    def update_acceleration(self, herd, c_alg, c_coh, c_sep,
                                        c_bound, c_rand, theta):
        '''initialising our acceleration components'''
        a_align = np.zeros(3)
        a_cohesion = np.zeros(3)
        a_sep = np.zeros(3)
        a_rand = c_rand * rng.normal(size=3)

        for shep in herd:
            '''ignore comparisons between itself and... itself'''
            if shep is self:
                continue
            r = shep.x - self.x
            vr = shep.v - self.v
            rmag_sq = r.dot(r)

            '''r_mag_sq > 9 => magnitude of the distance is sqrt9 =
                                            3.
            we dont' want the sheps to consider every single shep
                                            in the system
            so we give it a sight distance, and say that it can
                                            only see
            'in front' of it'''
            if rmag_sq > 9 or r.dot(vr) < theta:
                continue
            '''actual meat of the function'''
            a_align += c_alg * vr # dependant on velocities
            a_cohesion += c_coh * r # dependant on position vector
            a_sep -= c_sep * r / rmag_sq # dont hit each other

        '''we then add the acceleration components together, and
                                        normalise'''
        a_total = a_align + a_cohesion + a_sep + a_rand
        a_total += (np.abs(self.x) > boundary) * -np.sign(self.x) *
                                        c_bound
        self.a = a_total
        try:
            self.a = a_total * (1 / float(np.sqrt(a_total.dot(
                                            a_total))))
        except:
            self.a = np.zeros(3)

    def dynamics(self, dt):
        self.v += self.a * dt
        '''instead of normalising the accleration, we can set a max
                                        speed.
        this means though that the actual values for the constants
                                        matter,
        as with the normalisation, only the ratios between them
                                        matter'''

        '''if np.abs(np.sqrt(self.v.dot(self.v))) > np.abs(max_vel)
                                        :
            self.v = self.v*(max_vel/np.sqrt(self.v.dot(self.v)))
        '''
```

```
        self.x += self.v * dt


N = 150
X = 3
V = 1

'''a bunch of constants'''
c_alg = 0.01
c_coh = 0.05
c_sep = 0.02
c_rand = 0.01
c_bound = 50
boundary = 4
theta = 0
max_vel = 2


herd = Sheep(N, X, V).flock()


dt = 0.2
end_time = 20
'''creating and rendering the animation'''
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

def update(_):
    ax.clear()
    ax.set_xlim(-3, 3)
    ax.set_ylim(-3, 3)
    ax.set_zlim(-3, 3)

    for shep in herd:
        ax.scatter(*shep.x, s=0.5)
        shep.update_acceleration(herd, c_alg, c_coh, c_sep, c_bound
                                        , c_rand, theta)

    for shep in herd:
        shep.dynamics(dt)
        # print("This is an update to say the code is running: {}".
                                        format(shep))


animation = FuncAnimation(fig, update, frames=int(end_time/dt),
                                  interval=100)
animation.save('sheep_animation_1_0.gif', writer='pillow')

HTML(animation.to_jshtml())
```

Appendix B: Constant Acceleration

```
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from IPython.display import HTML
from matplotlib.animation import FuncAnimation
'''
Changes:
Added a constant acceleration to the sheps
(consistent with real life as when sheps fly
they need to maintain speed so as to not fall
from the sky)
'''
rng = np.random.default_rng(seed=1234)


@dataclass
class Sheep:
    N: int
    X: float
    V: float

    def flock(self):
        temp = []
        for _ in range(self.N):
            x = self.X * rng.normal(size=3)
            v = self.V * rng.normal(size=3)
            temp.append(Shep(x, v))
        return np.array(temp)


@dataclass
class Shep:

    x: list[float]
    v: list[float]

    def update_acceleration(self, herd, c_alg, c_coh, c_sep,
                                    c_const, c_bound, c_rand,
                                    theta):

        a_align = np.zeros(3)
        a_cohesion = np.zeros(3)
        a_sep = np.zeros(3)
        a_const = np.zeros(3)
        a_rand = c_rand * rng.normal(size=3)

        for shep in herd:
            if shep is self:
                continue
            r = shep.x - self.x
            vr = shep.v - self.v
            rmag_sq = r.dot(r)

            if rmag_sq > 25 or r.dot(vr) < theta:
                continue
```

```
            a_align += c_alg * vr
            a_cohesion += c_coh * r
            a_sep -= c_sep * r / rmag_sq
            a_const += c_const*self.v/np.sqrt(self.v.dot(self.v))

        a_total = a_align + a_cohesion + a_sep + a_const + a_rand
        a_total += (np.abs(self.x) > boundary) * -np.sign(self.x) *
                                            c_bound
        self.a = a_total
        try:
            self.a = a_total * (1 / float(np.sqrt(a_total.dot(
                                                a_total))))
        except:
            self.a = np.zeros(3)

    def dynamics(self, dt):
        self.v += self.a * dt
        '''if np.abs(np.sqrt(self.v.dot(self.v))) > np.abs(max_vel)
                                                :
            self.v = self.v*(max_vel/np.sqrt(self.v.dot(self.v)))
        '''
        self.x += self.v * dt


N = 150
X = 3
V = 2
n = 300

c_alg = 2/n
c_coh = 2/n
c_sep = 10/n
c_const = 1.5/n
c_rand = 1/n
c_bound = 500
boundary = 8
theta = 0
max_vel = 2


herd = Sheep(N, X, V).flock()


dt = 0.2
end_time = 20

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

def update(_):
    ax.clear()
    ax.set_xlim(-10, 10)
    ax.set_ylim(-10, 10)
    ax.set_zlim(-10, 10)

    for shep in herd:
```

```
        ax.scatter(*shep.x, s=0.5)
        shep.update_acceleration(herd, c_alg, c_coh, c_sep, c_const
                                          , c_bound, c_rand, theta)

    for shep in herd:
        shep.dynamics(dt)
        print(shep.x)



animation = FuncAnimation(fig, update, frames=int(end_time/dt),
                                  interval=100)
animation.save('sheep_animation1_1.gif', writer='pillow')

HTML(animation.to_jshtml())
```

Appendix C: Wind

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-


import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from IPython.display import HTML
from matplotlib.animation import FuncAnimation

'''For this version, we add wind:
-> created a wind acceleration that acts on each shep
-> this is a cylindrical vector field centred at (1,1,z)
(cylinder is pointing upward)
-> this vector field only extends 5 units of length.
it is a perfectly cylindrical tornado I decided'''

rng = np.random.default_rng(seed=12)


@dataclass
class Sheep:
    N: int
    X: float
    V: float

    def flock(self):
        temp = []
        for _ in range(self.N):
            x = self.X * rng.normal(size=3)
            v = self.V * rng.normal(size=3)
            temp.append(Shep(x, v))
        return np.array(temp)


@dataclass
class Shep:

    x: list[float]
    v: list[float]

    def update_acceleration(self, herd, c_alg, c_coh, c_sep,
                                    c_const, c_bound, c_rand,
                                    theta, wind):

        a_align = np.zeros(3)
        a_cohesion = np.zeros(3)
        a_sep = np.zeros(3)
        a_const = np.zeros(3)
        a_rand = c_rand * rng.normal(size=3)

        for shep in herd:
            if shep is self:
                continue
            r = shep.x - self.x
```

```
            vr = shep.v - self.v
            rmag_sq = r.dot(r)
            a_wind = np.zeros(3)
            wind_dist = self.x-wind_centre
            wind_dist_mag_sq = wind_dist.dot(wind_dist)


            if rmag_sq > 25 or r.dot(vr) < theta:
                continue
            a_wind = [wind*self.x[1]/(np.sqrt(self.x[0]**2+self.x[1
                                              ]**2)*
                                              wind_dist_mag_sq)+
                                              wind_centre[0],wind*
                                              self.x[0]/(np.sqrt(
                                              self.x[0]**2+self.x[1
                                              ]**2)*
                                              wind_dist_mag_sq)+
                                              wind_centre[1],wind*0
                                              ]
            a_align += c_alg * vr
            a_cohesion += c_coh * r
            a_const += c_const*self.v/np.sqrt(self.v.dot(self.v))

            if rmag_sq < 4:
                a_sep -= c_sep * r / rmag_sq

        a_total = a_align + a_cohesion + a_sep + a_const + a_rand +
                                                 a_wind
        a_total += (np.abs(self.x) > boundary) * -np.sign(self.x) *
                                                 c_bound
        self.a = a_total
        # normalisation of the acceleration vector
        try:
            self.a = a_total * (1 / float(np.sqrt(a_total.dot(
                                                  a_total))))
        except:
            self.a = np.zeros(3)

    def dynamics(self, dt):
        self.v += self.a * dt

        # some code to set a maximum speed.
        # this requires tweaking to the parameters to work properly
                                                 .
        # currently, the forces are too high for this to work

        '''if np.abs(np.sqrt(self.v.dot(self.v))) > np.abs(max_vel)
                                                 :
            self.v = self.v*(max_vel/np.sqrt(self.v.dot(self.v)))
        elif abs(np.abs(np.sqrt(self.v.dot(self.v))) - np.abs(
                                                 max_vel))>0.1:
            print(abs(np.abs(np.sqrt(self.v.dot(self.v))) - np.abs(
                                                  max_vel)))'''

        self.x += self.v * dt

dt = 0.15
```

```
N = 200
X = 3
V = 5*dt
n = 300

c_alg = 1/n
c_coh = 4/n
c_sep = 15/n
c_const = 0.7/n
c_rand = 1/n
c_bound = 500
wind_centre = [1,1,1]
wind = 2
boundary = 6.5
theta = 0
max_vel = 2


herd = Sheep(N, X, V).flock()



end_time = 200

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

def update(_):
    ax.clear()
    ax.set_xlim(-8, 8)
    ax.set_ylim(-8, 8)
    ax.set_zlim(-8, 8)

    for shep in herd:
        ax.scatter(*shep.x, s=0.3, c='#0000A4')
        shep.update_acceleration(herd, c_alg, c_coh, c_sep, c_const
                                        , c_bound, c_rand, theta,
                                         wind)

    for shep in herd:
        shep.dynamics(dt)
        print(shep.x)



animation = FuncAnimation(fig, update, frames=int(end_time/dt),
                                  interval=80)
animation.save('sheep_animation_temp.gif', writer='pillow')

HTML(animation.to_jshtml())
```

Appendix D: Predator

```
    #!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Addition of a predator to the flock:
-> So what if there was an evil shep added into the flock
-> The evil shep acts in a pretty similar way to the others
but it chases them down. as such, there is no seperation
                                    acceleration
for the evil shep. the regular sheep have been updated to avoid the
                                     predator at as a priority"""

import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from IPython.display import HTML
from matplotlib.animation import FuncAnimation

rng = np.random.default_rng(seed=12)


@dataclass
class Sheep:
    N: int
    X: float
    V: float

    def flock(self):
        temp = []
        for _ in range(self.N):
            x = self.X * rng.normal(size=3)
            v = self.V * rng.normal(size=3)
            temp.append(Shep(x, v))
        return np.array(temp)


@dataclass
class Shep:

    x: list[float]
    v: list[float]

    def update_acceleration(self, herd, c_alg, c_coh, c_sep,
                                       c_const, c_bound, c_rand,
                                       c_pred, theta, wind):

        a_align = np.zeros(3)
        a_cohesion = np.zeros(3)
        a_sep = np.zeros(3)
        a_const = np.zeros(3)
        a_rand = c_rand * rng.normal(size=3)
        a_predator = np.zeros(3)

        for shep in herd:
            if shep is self:
                continue
            r = shep.x - self.x
```

```
            vr = shep.v - self.v
            rmag_sq = r.dot(r)
            '''a_wind = np.zeros(3)
            wind_dist = self.x-wind_centre
            wind_dist_mag_sq = wind_dist.dot(wind_dist)
            '''

            if rmag_sq > 25 or r.dot(vr) < theta:
                continue
            '''a_wind = [wind*self.x[1]/(np.sqrt(self.x[0]**2+self.
                                            x[1]**2)*
                                            wind_dist_mag_sq)+
                                            wind_centre[0],wind*
                                            self.x[0]/(np.sqrt(
                                            self.x[0]**2+self.x[1
                                            ]**2)*
                                            wind_dist_mag_sq)+
                                            wind_centre[1],wind*0
                                            ]'''
            a_align += c_alg * vr
            a_cohesion += c_coh * r
            a_const += c_const*self.v/np.sqrt(self.v.dot(self.v))

            if rmag_sq < 4:
                a_sep -= c_sep * r / rmag_sq

        r_evil = predator.x - self.x
        rmag_sq_evil = r_evil.dot(r_evil)

        '''for a small radius around them, the sheps can just "
                                        detect"
        the predator regardless of whether they're looking at it or
                                        not'''
        if rmag_sq_evil < 10:
            a_predator -= c_pred*r_evil/rmag_sq_evil
        a_total = a_align + a_cohesion + a_sep + a_const + a_rand +
                                        a_predator # a_wind
        a_total += (np.abs(self.x) > boundary) * -np.sign(self.x) *
                                        c_bound
        self.a = a_total
        # normalisation of the acceleration vector
        try:
            self.a = a_total * (1 / float(np.sqrt(a_total.dot(
                                            a_total))))
        except:
            self.a = np.zeros(3)

    def dynamics(self, dt):
        self.v += self.a * dt

        # some code to set a maximum speed.
        # this requires tweaking to the parameters to work properly
                                        .
        # currently, the forces are too high for this to work

        '''if np.abs(np.sqrt(self.v.dot(self.v))) > np.abs(max_vel)
                                        :
```

```
            self.v = self.v*(max_vel/np.sqrt(self.v.dot(self.v)))
        '''

        self.x += self.v * dt

"""here is our evil shep in question. I'm using an init function
for this one. this is not done with any particular reason in mind
                                    """
@dataclass
class EvilShep:

    def __init__(self, x, v, a_total, evil_c_pos, evil_c_dir,
                                        evil_c_rand):
        self.x = [np.random.uniform(-6,6),np.random.uniform(-6,6),
                                            np.random.uniform(-6,6)]
        self.v = np.zeros(3)
        self.a_total = np.zeros(3)
        self.evil_c_pos = evil_c_pos
        self.evil_c_dir = evil_c_dir
        self.evil_c_rand = evil_c_rand

    def evil_acceleration(self):
        a_pos = np.zeros(3)
        a_dir = np.zeros(3)
        a_rand = evil_c_rand * rng.normal(size=3)
        for shep in herd:
            evil_r = shep.x - self.x
            evil_vel_r = shep.v - self.v
            evil_r_mag = np.sqrt(evil_r.dot(evil_r))
            if evil_r_mag > 4 or evil_r.dot(evil_vel_r) < theta:
                continue
            a_pos += evil_c_pos * evil_r
            a_dir += evil_c_dir * evil_vel_r

            a_total = a_pos + a_dir
            a_total += 1.5*(np.abs(self.x) > boundary) * -np.sign(
                                                self.x) * c_bound
            try:
                self.a_total = a_total * (1 / float(np.sqrt(a_total
                                                    .dot(a_total))))
            except:
                self.a_total = np.zeros(3)

    def evil_dynamics(self):
        self.x += self.v * dt
        self.v += self.a_total * dt


predator = EvilShep(np.zeros(3), np.zeros(3), np.zeros(3), np.zeros
                                    (3), np.zeros(3), np.zeros(3))
print(predator.x)


dt = 0.15
N = 200
X = 3
V = 5*dt
```

```
n = 300

c_alg = 1/n
c_coh = 4/n
c_sep = 15/n
c_const = 0.7/n
c_rand = 1/n
c_bound = 500
c_pred = 100
'''wind_centre = [1,1,1]
wind = 2'''
boundary = 6.5
theta = 0
max_vel = 2

evil_c_dir = 2
evil_c_pos = 1
evil_c_rand = 0.5


herd = Sheep(N, X, V).flock()


end_time = 40

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")


def update(_):
    ax.clear()
    ax.set_xlim(-8, 8)
    ax.set_ylim(-8, 8)
    ax.set_zlim(-8, 8)

    for shep in herd:
        ax.scatter(*shep.x, s=0.3, c='#0000A4')
        shep.update_acceleration(herd, c_alg, c_coh, c_sep, c_const
                                        , c_bound, c_rand, c_pred
                                        , theta, '''wind''')

    for shep in herd:
        shep.dynamics(dt)
        #print(shep.x)
    ax.scatter(*predator.x, s=2, c='#DD0000')
    predator.evil_acceleration()
    predator.evil_dynamics()
    print("Acceleration:", predator.a_total,
          "\nVelocity:", predator.v,
          "\nPosition:", predator.x)


animation = FuncAnimation(fig, update, frames=int(end_time/dt),
                                    interval=80)
animation.save('sheep_animation_1_3.gif', writer='pillow')

HTML(animation.to_jshtml())
```