Coding Example: Implement the behavior of Boids (NumPy approach)

In this lesson we will try to implement all three rules that Boids follow using the NumPy approach.

We'll cover the following

NumPy Implementation

As you might expect, the NumPy implementation takes a different approach and we'll gather all our boids into a position array and a velocity array:

```
n = 500
velocity = np.zeros((n, 2), dtype=np.float32)
position = np.zeros((n, 2), dtype=np.float32)
```

The first step is to compute the local neighborhood for all boids, and for this, we need to compute all paired distances:

```
#np.subtract.outer  apply the ufunc op to all pairs (a, b) with a in A and b in B.
dx = np.subtract.outer(position[:, 0], position[:, 0])
dy = np.subtract.outer(position[:, 1], position[:, 1])
distance = np.hypot(dx, dy)
```

We could have used the `scipy` cdist but we'll need the `dx` and `dy` arrays later. Once those have been computed, it is faster to use the hypot method. Note that distance shape is `(n, n)` and each line relates to one boid, i.e. each line gives the distance to all other boids (including self).

From these distances, we can now compute the local neighborhood for each of the three rules, taking advantage of the fact that we can mix them together. We can actually compute a mask for distances that are strictly positive (i.e. have no self-interaction) and multiply it with other distance masks.

> **Note:** If we suppose that boids cannot occupy the same position, how can you compute `mask_0` more efficiently?

```
mask_0 = (distance > 0)
mask_1 = (distance < 25)
mask_2 = (distance < 50)
mask_1 *= mask_0
mask_2 *= mask_0
mask_3 = mask_2
```

Then, we compute the number of neighbors within the given radius and we ensure it is at least 1 to avoid division by zero.

```
mask_1_count = np.maximum(mask_1.sum(axis=1), 1)
mask_2_count = np.maximum(mask_2.sum(axis=1), 1)
mask_3_count = mask_2_count
```

We're ready to write our three rules:

Alignment

```
# Compute the average velocity of local neighbours
target = np.dot(mask, velocity)/count.reshape(n, 1)

# Normalize the result
norm = np.sqrt((target*target).sum(axis=1)).reshape(n, 1)
target *= np.divide(target, norm, out=target, where=norm != 0)

# Alignment at constant speed
target *= max_velocity

# Compute the resulting steering
alignment = target - velocity
```

Cohesion

```
# Compute the gravity center of local neighbours
center = np.dot(mask, position)/count.reshape(n, 1)

# Compute direction toward the center
target = center - position

# Normalize the result
norm = np.sqrt((target*target).sum(axis=1)).reshape(n, 1)
target *= np.divide(target, norm, out=target, where=norm != 0)

# Cohesion at constant speed (max_velocity)
target *= max_velocity

# Compute the resulting steering
cohesion = target - velocity
```

Separation

```
# Compute the repulsion force from local neighbours
repulsion = np.dstack((dx, dy))

# Force is inversely proportional to the distance
repulsion = np.divide(repulsion, distance.reshape(n, n, 1)**2, out=repulsion,
                      where=distance.reshape(n, n, 1) != 0)

# Compute direction away from others
target = (repulsion*mask.reshape(n, n, 1)).sum(axis=1)/count.reshape(n, 1)

# Normalize the result
norm = np.sqrt((target*target).sum(axis=1)).reshape(n, 1)
target *= np.divide(target, norm, out=target, where=norm != 0)

# Separation at constant speed (max_velocity)
target *= max_velocity

# Compute the resulting steering
separation = target - velocity
```

All three resulting steerings (separation, alignment & cohesion) need to be limited in magnitude. We leave this as an exercise for the reader. Combination of these rules is straightforward as well as the resulting update of velocity and position:

```
acceleration = 1.5 * separation + alignment + cohesion
velocity += acceleration
position += velocity
```

We are now ready to visualize our boids!

Visualization

We finally visualize the result using a custom oriented scatter plot! The easiest way is to use the `matplotlib` animation function and a `scatter plot`. Unfortunately, scatters cannot be individually oriented and we need to make our own objects using a `matplotlib PathCollection`. A simple triangle path can be defined as:

```
v= np.array([(-0.25, -0.25),
             ( 0.00,  0.50),
             ( 0.25, -0.25),
             ( 0.00,  0.00)])
c = np.array([Path.MOVETO,
              Path.LINETO,
              Path.LINETO,
              Path.CLOSEPOLY])
```

This path can be repeated several times inside an array and each triangle can be made independent.

```
n = 500
vertices = np.tile(v.reshape(-1), n).reshape(n, len(v), 2)
codes = np.tile(c.reshape(-1), n)
```

We now have a `(n,4,2)` array for vertices and a `(n,4)` array for codes representing `n` boids. We are interested in manipulating the vertices array to reflect the translation, scaling and rotation of each of the n boids.

> **Note:** Rotate is really tricky.

How would you write the `translate`, `scale` and `rotate` functions?

Complete Solution

Now we will merge all this logic into one code to form a complete solution.

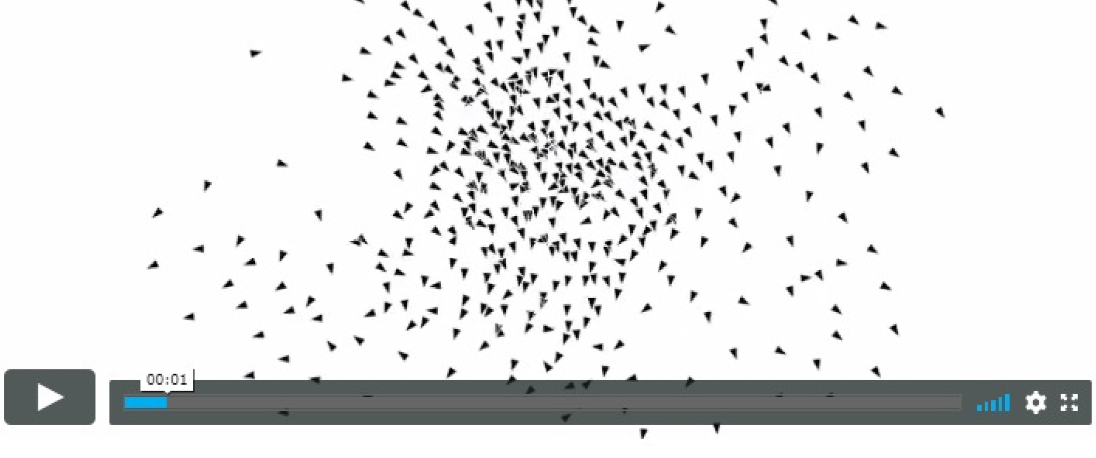
```
# -----------------------------------------------------------------------------
# From Python to Numpy
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
from matplotlib.collections import PathCollection
import matplotlib.animation as animation



class MarkerCollection:
    """
    Marker collection
    """

    def __init__(self, n=100):
        v = np.array([(-0.25, -0.25), (+0.0, +0.5), (+0.25, -0.25), (0, 0)])
        c = np.array([Path.MOVETO, Path.LINETO, Path.LINETO, Path.CLOSEPOLY])
        self._base_vertices = np.tile(v.reshape(-1), n).reshape(n, len(v), 2)
        self._vertices = np.tile(v.reshape(-1), n).reshape(n, len(v), 2)
        self._codes = np.tile(c.reshape(-1), n)

        self._scale = np.ones(n)
        self._translate = np.zeros((n, 2))
        self._rotate = np.zeros(n)
```

RUN SAVE RESET

output.mp4

Output

The output of above code would look exactly like this if you run it and download the file that it generates.



Solve this Quiz!

1 What is the purpose of `numpy.dstack`?

A) Stack arrays in sequence breadth wise (along third axis).

B) Stack arrays in sequence depth wise (along third axis).

COMPLETED 0% 1 of 2

Further Readings

- Flocking, Daniel Shiffman, 2010.
- Flocks, herds and schools: A distributed behavioral model, Craig Reynolds, SIGGRAPH, 1987