Modelo de Boids de Craig Reynolds

Rodrigo Fritz

Importamos un montón de librerías de las cuales la más importante es `jax` y sus derivados, y definimos funciones para graficar, para la barra de progreso y para el rendering, el cual tiene tantos comandos que lo dejaremos oculto.

```
# Imports

!pip install -q git+https://www.github.com/google/jax-md

import numpy as onp

from jax.config import config ; config.update('jax_enable_x64', True)
import jax.numpy as np
from jax import random
from jax import jit
from jax import vmap
from jax import lax
vectorize = np.vectorize

from functools import partial

from collections import namedtuple
import base64

import IPython
from google.colab import output

import os

from jax_md import space, smap, energy, minimize, quantity, simulate, partition, util
from jax_md.util import f32
```

```
       Building wheel for jax-md (setup.py) ... done
```

```
# Plotting

import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style(style='white')

dark_color = [56 / 256] * 3
light_color = [213 / 256] * 3
axis_color = 'white'
```

```
def format_plot(x='', y='', grid=True):
  ax = plt.gca()

  ax.spines['bottom'].set_color(axis_color)
  ax.spines['top'].set_color(axis_color)
  ax.spines['right'].set_color(axis_color)
  ax.spines['left'].set_color(axis_color)

  ax.tick_params(axis='x', colors=axis_color)
  ax.tick_params(axis='y', colors=axis_color)
  ax.yaxis.label.set_color(axis_color)
  ax.xaxis.label.set_color(axis_color)
  ax.set_facecolor(dark_color)

  plt.grid(grid)
  plt.xlabel(x, fontsize=20)
  plt.ylabel(y, fontsize=20)

def finalize_plot(shape=(1, 1)):
  plt.gcf().patch.set_facecolor(dark_color)
  plt.gcf().set_size_inches(
    shape[0] * 1.5 * plt.gcf().get_size_inches()[1],
    shape[1] * 1.5 * plt.gcf().get_size_inches()[1])
  plt.tight_layout()


# Progress Bars

from IPython.display import HTML, display
import time

def ProgressIter(iter_fun, iter_len=0):
  if not iter_len:
    iter_len = len(iter_fun)
  out = display(progress(0, iter_len), display_id=True)
  for i, it in enumerate(iter_fun):
    yield it
    out.update(progress(i + 1, iter_len))

def progress(value, max):
    return HTML("""
        <progress
            value='{value}'
            max='{max}',
            style='width: 45%'
        >
            {value}
        </progress>
    """.format(value=value, max=max))

normalize = lambda v: v / np.linalg.norm(v, axis=1, keepdims=True)
```

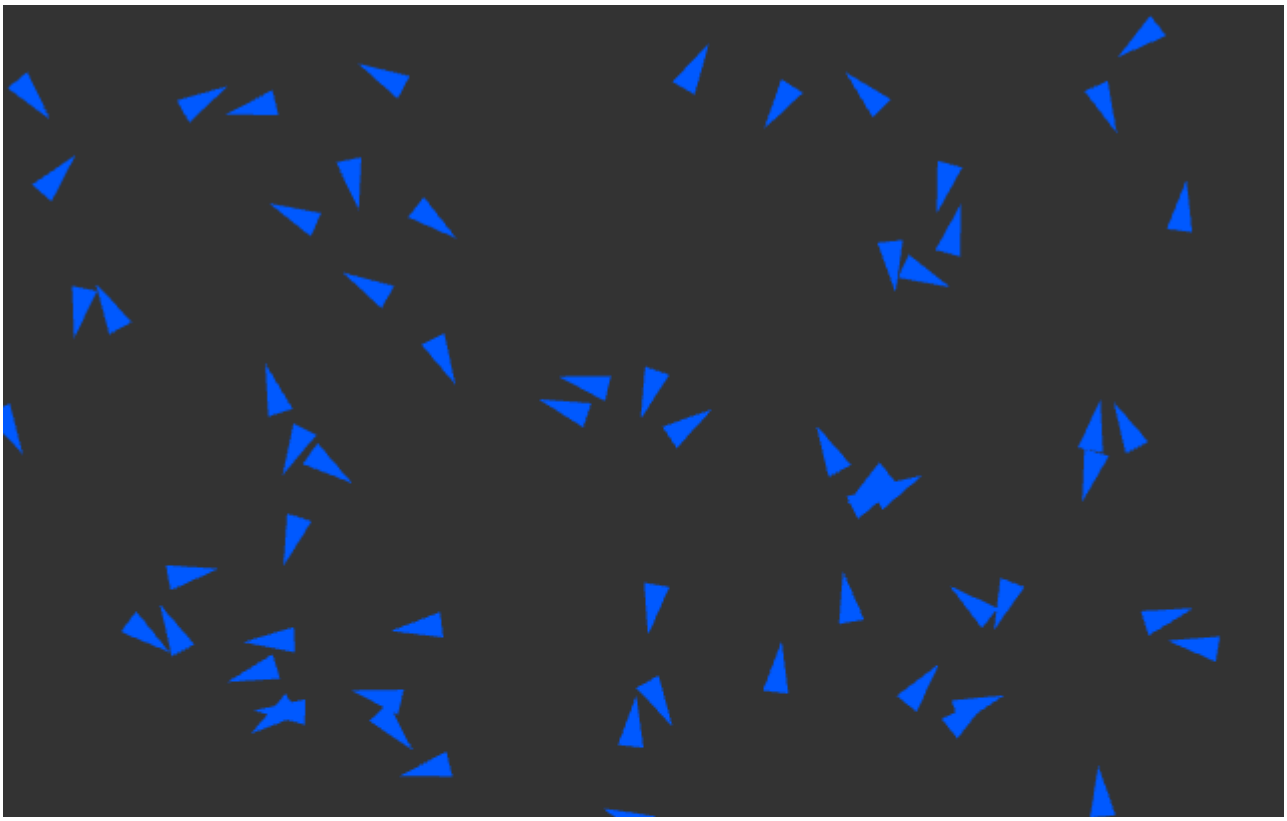
Rendering

Los bird-oids o *boids* están descritos por una posición R y un ángulo θ. `R` y `theta` son `ndarray`'s con shape `[boid_count, spatial_dimension]` y `[boid_count]`, respectivamente. Un boid individual es un índice dentro de estos arrrglos. El vector de orientación del boid es $N = (\cos\theta, \sin\theta)$.

```
Boids = namedtuple('Boids', ['R', 'theta'])
```

Primero veamos una colección de boids puestos aleatoriamente en un cuadrado de lado L con fronteras periódicas usando el comando `space.periodic` de jax-md.

```
# Simulation Parameters:
box_size = 400.  # A float specifying the side-length of the box.
boid_count = 100  # An integer specifying the number of boids.
dim = 2  # The spatial dimension in which we are simulating.

# Create RNG (random number generation) state to draw random numbers (see LINK).
rng = random.PRNGKey(2)

# Define periodic boundary conditions.
displacement, shift = space.periodic(box_size)

# Initialize the boids.
rng, R_rng, theta_rng = random.split(rng, 3)

boids = Boids(
  R = box_size * random.uniform(R_rng, (boid_count, dim)),
  theta = random.uniform(theta_rng, (boid_count,), maxval = 2*np.pi)
)

display(render(box_size, boids))
```



Dinámicas

Reynolds se dio cuenta de que las parvadas no tienen un tamaño máximo, sino que siempre se pueden unir más boids a la parvada. Cada boid no puede estar al tanto de toda la parvada, sino solo de su vecindad local. Así que Reynolds propone 3 reglas simples y locales que deben seguir los boids:

1. **Alineación:** Los boids se alinearán con sus vecinos.
2. **Separación:** Los boids evitarán chocar con sus vecinos.
3. **Cohesión:** Los boids se moverán hacia el centro de masa de sus vecinos.

Una forma de implementar estas reglas es mediante una función de "energía" $E(R, \theta)$, de tal forma que las configuraciones de mínima energía satisfagan las 3 reglas:

$$E(R, \theta) = E_{\text{Align}}(R, \theta) + E_{\text{Avoid}}(R, \theta) + E_{\text{Cohesion}}(R, \theta)$$

Veremos cada una de estas reglas por separado. Algunas configuraciones que se muevan a lo largo de trayectorias de mínima energía tendrán un aspecto agradable. Reynolds utiliza una dinámica sobreamortiguada, para lo cual podemos actualizar la posición de los boids de forma tal que minimicen su energía. Empezaremos con un modelo en el que los boids se mueven a velocidad constante v, a lo largo de cualquier dirección en la que apunten. Para ello se usa la integración de Euler hacia delante, cuya iteración es la siguiente:

$$R_{i+1} = R_i + \delta t(v\hat{N}_i - F_{R_i})$$

donde δt es el tamaño del paso, $\hat{N}_i$ es el vector normal y

$$F_{R_i} = -\nabla_{R_i} E(R, \theta)$$

es la fuerza dada por el gradiente negativo de la energía con respecto a la posición R_i del i-ésimo boid. Para orientar a los boids hacia las direcciones de mínima energía, usaremos de nuevo una integración de Euler hacia delante:

$$\theta_{i+1} = \theta_i - \delta t \nabla_{\theta_i} E(R, \theta)$$

Existen otras dinámicas que funcionan igual de bien, como usar un integrador más sofisticado, por ejemplo un Runge-Kutta 4.

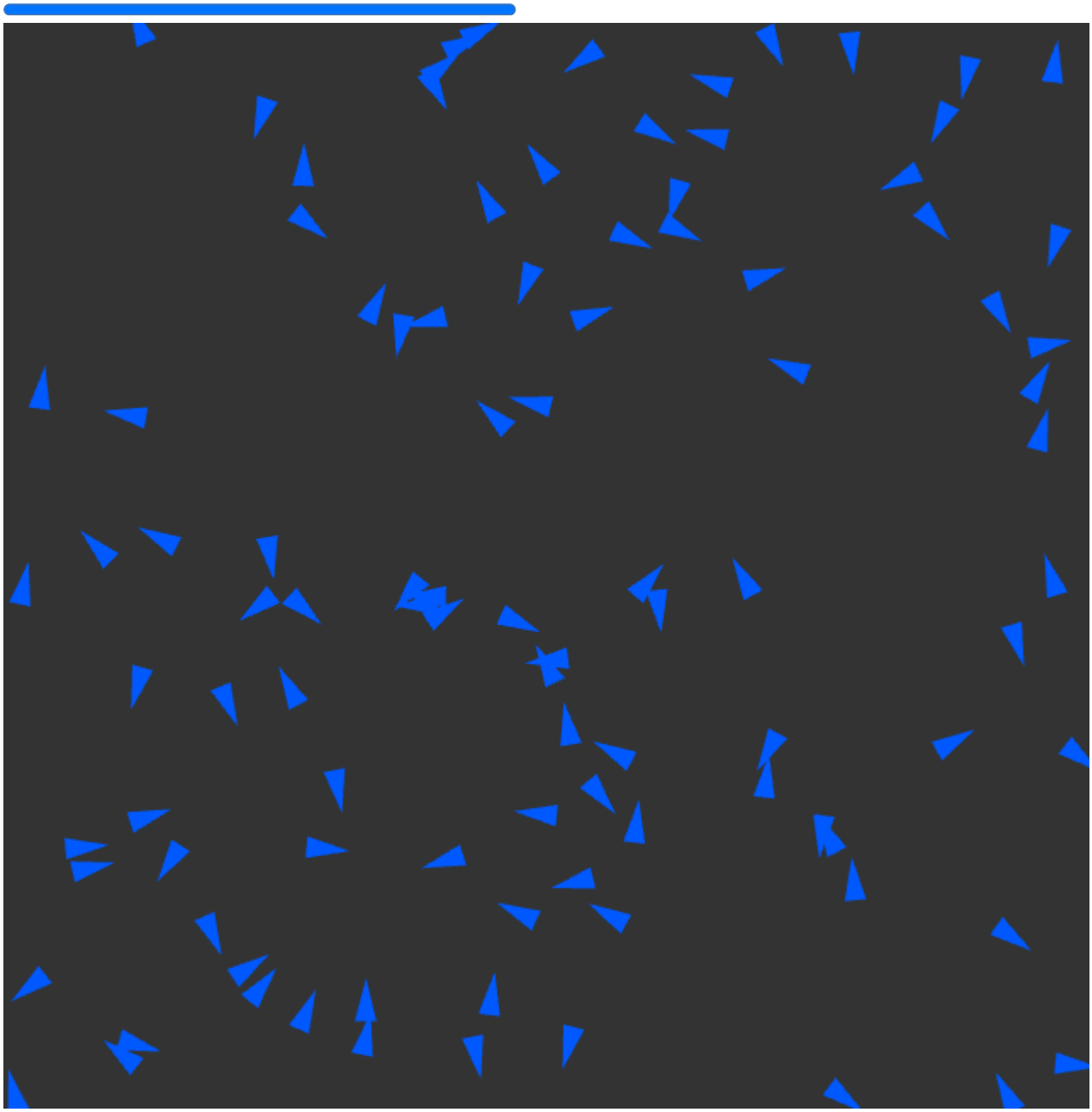
Antes de agregar las interacciones, correremos una simulación con $E(R, \theta) = 0$ definiendo una función de actualización `update` que toma el estado actual de los boids hacia un nuevo estado.

```
@vmap
def normal(theta):
  return np.array([np.cos(theta), np.sin(theta)])

def dynamics(energy_fn, dt, speed):
  @jit
  def update(_, state):
    R, theta = state['boids']

    dstate = quantity.force(energy_fn)(state)
    dR, dtheta = dstate['boids']
    n = normal(state['boids'].theta)

    state['boids'] = Boids(shift(R, dt * (speed * n + dR)),
                           theta + dt * dtheta)

    return state

  return update
```

Ahora podemos correr la simulación y guardar las posiciones de los boids en una lista llamada `boids_buffer`.

```
update = dynamics(energy_fn=lambda state: 0.0, dt=1e-1, speed=0.3)
#El primer argumento de dynamics será cada una de las funciones de energía

boids_buffer = []

state = {
    'boids': boids
```

```
}

for i in ProgressIter(range(500)):
  state = lax.fori_loop(0, 50, update, state)
  boids_buffer += [state['boids']]

display(render(box_size, boids_buffer))
```



Alineación

Para implentar la regla de alineación, lo haremos para solo un par de boids con la vectorización automática (auto-vectorization) de `jax` via `vmap` para extenderlos a toda la simulación.

Dado un par de boids i, j queremos escoger una función de energía que se minimiza cuando apuntan en la misa dirección. La localidad se presenta cuando los boids solo interactúan con los boids más cercanos. Para hacer esto, introducimos una cota D_{Align} para ignorar los pares de boids tales que $\|\Delta R_{ij}\| \geq D_{\text{Align}}$, donde $\Delta R_{ij} = R_i - R_j$. Para que los boids reaccionen suavemente, la energía partirá de cero cuando $\|R_i - R_j\| = D_{\text{Align}}$ e incrementará suavemente conforme se acercan los boids:

$$\mathcal{E}_{\text{Align}}(\Delta R_{ij}, \hat{N}_i, \hat{N}_j) = \begin{cases} \frac{J_{\text{Align}}}{\alpha}\left(1 - \frac{\|\Delta R_{ij}\|}{D_{\text{Align}}}\right)^{\alpha}(1 - \hat{N}_i \cdot \hat{N}_j)^2, & \text{si } \|\Delta R_{ij}\| < D_{\text{Align}} \\ 0 & \text{en otro caso} \end{cases}$$

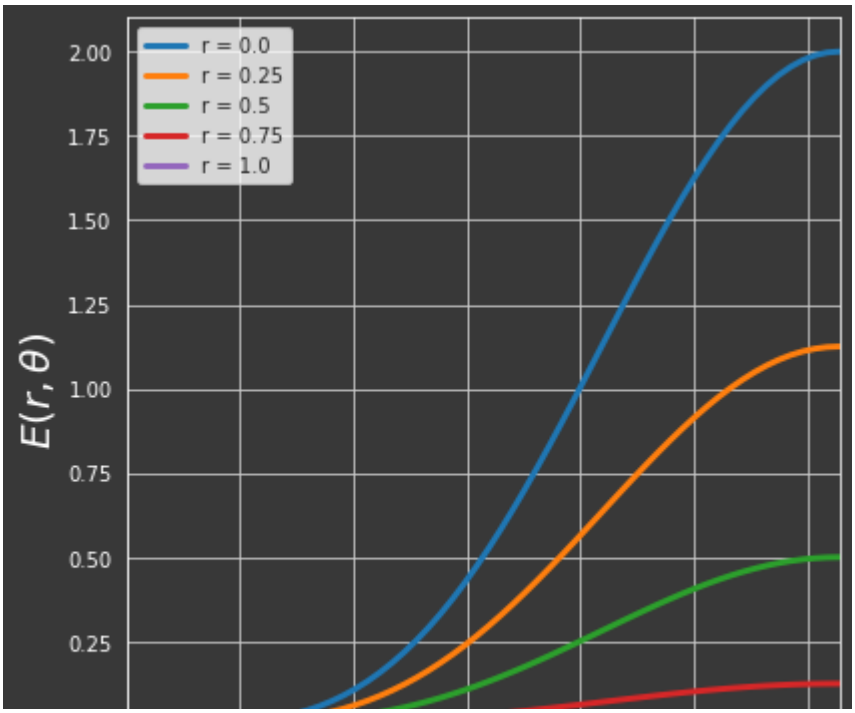
La energía será máxima cuando θ_i y θ_j apunten en direcciones opuestas, y mínima cuando $\theta_i = \theta_j$. Como queremos que los boids se alineen con sus vecinos sin alejarse, agregaremos un `stop-gradient` en el desplazamiento:

```
def align_fn(dR, N_1, N_2, J_align, D_align, alpha):
  dR = lax.stop_gradient(dR)
  dr = space.distance(dR) / D_align
  energy = J_align / alpha * (1. - dr)**alpha * (1 - np.dot(N_1, N_2))**2
  return np.where(dr < 1.0, energy, 0.)
```

Al graficar la energía para distantas alineaciones θ y para distintas separaciones r entre los boids, vemos que la energía disminuye para separaciones grandes y tiende a cero cuando los boids están alineados, i.e. $\theta = 0$.

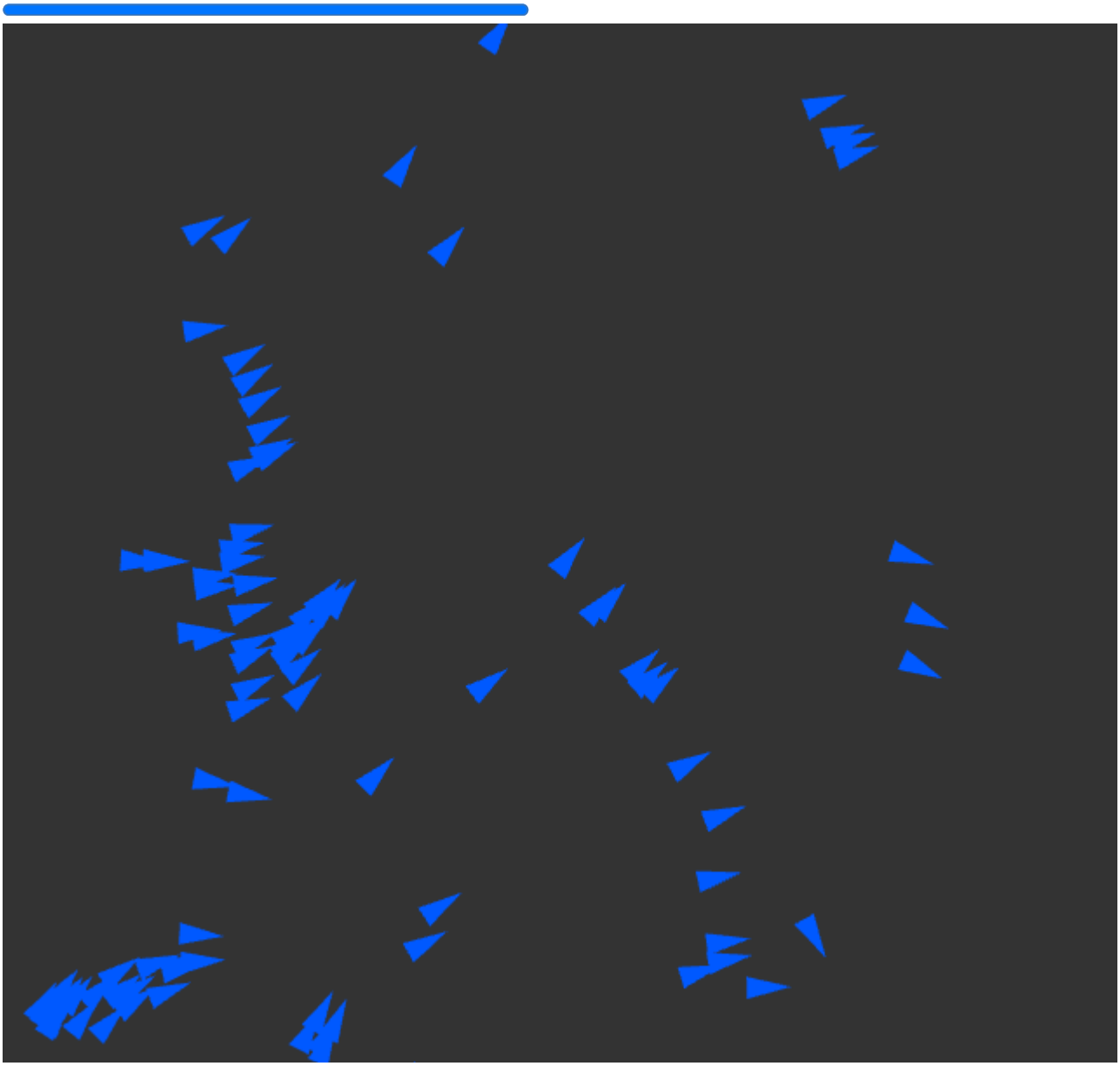
Energía de Alineación con alpha = 2

```
#@title Energía de Alineación con alpha = 2

N_1 = np.array([1.0, 0.0])
angles = np.linspace(0, np.pi, 60)
N_2 = vmap(lambda theta: np.array([np.cos(theta), np.sin(theta)]))(angl
distances = np.linspace(0, 1, 5)
dRs = vmap(lambda r: np.array([r, 0.]))(distances)

fn = partial(align_fn, J_align=1., D_align=1., alpha=2.)
energy = vmap(vmap(fn, (None, None, 0)), (0, None, None))(dRs, N_1, N_2

for d, e in zip(distances, energy):
  plt.plot(angles, e, label='r = {}'.format(d), linewidth=3)

plt.xlim([0, np.pi])
format_plot('$\\theta$', '$E(r, \\theta)$')
plt.legend()
finalize_plot()
```



Veamos la simulación con energía de alineación:

```
def energy_fn(state):
  boids = state['boids']
  E_align = partial(align_fn, J_align=2, D_align=30., alpha=3.)
  # Map the align energy over all pairs of boids. While both applications
  # of vmap map over the displacement matrix, each acts on only one normal.
  E_align = vmap(vmap(E_align, (0, None, 0)), (0, 0, None))

  dR = space.map_product(displacement)(boids.R, boids.R)
  N = normal(boids.theta)

  return 0.5 * np.sum(E_align(dR, N, N))

update = dynamics(energy_fn=energy_fn, dt=1e-1, speed=0.5)

boids_buffer = []

state = {
    'boids': boids
}

for i in ProgressIter(range(700)):
  state = lax.fori_loop(0, 50, update, state)
  boids_buffer += [state['boids']]

display(render(box_size, boids_buffer))
```



Los boids se alinean pero se enciman, así que agregaremos la separación.

Separación

La regla de separación hace que la parvada tenga algo de volumen en lugar de estar toda colapsada. Para ello queremos que los boids se separen cuando su distancia de separación es menor a una distancia límite D_{Avoid}, pero que se mantengan unidos si están a una distancia mayor. La energía de separación no depende del ángulo de orientación:

$$\mathcal{E}_{\text{Avoid}}(\Delta R_{ij}) = \begin{cases} \frac{J_{\text{Avoid}}}{\alpha}\left(1 - \frac{||\Delta R_{ij}||}{D_{\text{Avoid}}}\right)^{\alpha}, & \text{si } ||\Delta R_{ij}|| < D_{\text{Avoid}} \\ 0 & \text{en otro caso} \end{cases}$$

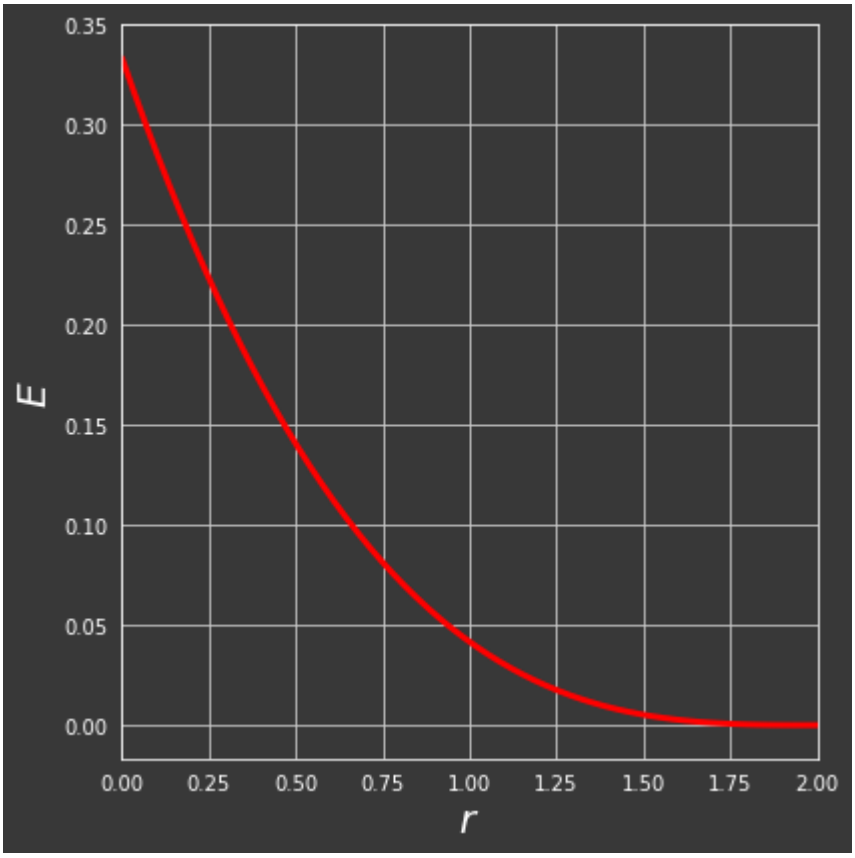
Esto lo implementamos en la siguiente función. A diferencia de la alineación, como queremos que los boids se alejen los unos de los otros, aquí no se requiere el `stop-gradient` para ΔR. Sin

embargo, para evitar que se desarme el grupo tendremos que agregar la cohesión más adelante.

```
def avoid_fn(dR, J_avoid, D_avoid, alpha):
  dr = space.distance(dR) / D_avoid
  return np.where(dr < 1.,
                  J_avoid / alpha * (1 - dr) ** alpha,
                  0.)
```

Al graficar la energía de separación, vemos que es mayor cuando los boids están encimados y luego tiende a cero suavemente hasta que $||\Delta R|| = D_{\text{Align}}$.

```
#@title Energía de Separación

dr = np.linspace(0, 2., 60)
dR = vmap(lambda r: np.array([0., r]))(dr)
Es = vmap(partial(avoid_fn, J_avoid=1., D_avoid=2., alpha=3.))(dR)
plt.plot(dr, Es, 'r', linewidth=3)

plt.xlim([0, 2])

format_plot('$r$', '$E$')
finalize_plot()
```



Veamos la simulación con alineación y separación:

```
def energy_fn(state):
  boids = state['boids']

  # Align Energy
  E_align = partial(align_fn, J_align=2., D_align=30., alpha=3.)
  E_align = vmap(vmap(E_align, (0, None, 0)), (0, 0, None))

  # + Avoidance Energy
  E_avoid = partial(avoid_fn, J_avoid=2., D_avoid=30., alpha=3.)
  E_avoid = vmap(vmap(E_avoid))

  dR = space.map_product(displacement)(boids.R, boids.R)
  N = normal(boids.theta)

  return 0.5 * np.sum(E_align(dR, N, N) + E_avoid(dR))

update = dynamics(energy_fn=energy_fn, dt=1e-1, speed=0.5)

boids_buffer = []

state = {
    'boids': boids
}

for i in ProgressIter(range(700)):
  state = lax.fori_loop(0, 50, update, state)
  boids_buffer += [state['boids']]

display(render(box_size,  boids_buffer))
```

Cohesión

Con solo alineación y separación, los boids tienden a moverse en la misma dirección pero también a menudo se separan. Así que para mantener un arrreglo más compacto, agregamos un término de cohesión en la energía.

El objetivo del término de cohesión es alinear a los boids con el centro de masa de sus veciones. Dado un boid i, la posición del centro de masa de sus vecinos está dada por

$$\Delta R_i = \frac{1}{|\mathcal{N}|} \sum_{j \in \mathcal{N}} \Delta R_{ij}$$

donde $\mathcal{N}$ es el conjunto de boids para los cuales $||\Delta R_{ij}|| < D_{\text{Cohesion}}$. Dados los desplazamientos del centro de masa, podemos deifinir una energía de cohesión razonable como

$$\mathcal{E}_{\text{Cohesion}}\left(\widehat{\Delta R_i}, N_i\right) = \frac{1}{2} J_{\text{Cohesion}} \left(1 - \widehat{\Delta R_i} \cdot N\right)^2$$

donde $\widehat{\Delta R_i} = \Delta R_i / ||\Delta R_i||$ es el vector normalizado que apunta en la dirección del centro de masa. Esta energía es mínima cuando el boid apunta en la dirección del centro de masa.

En la siguiente función de cohesión se ha vuelto a añadir el `stop-gradient` sobre el vector de desplazamiento para controlar la orientación de los boids.

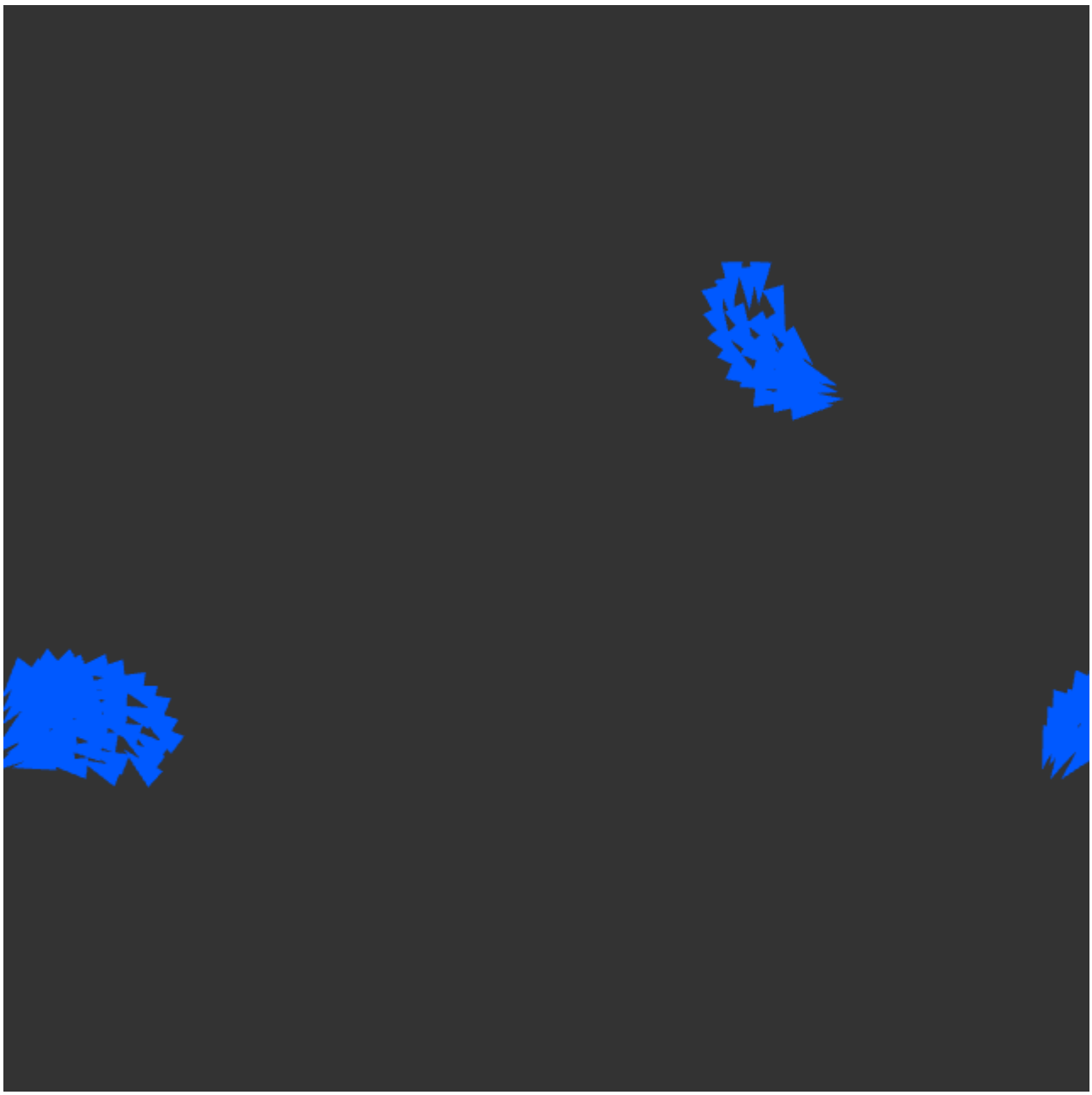
```
def cohesion_fn(dR, N, J_cohesion, D_cohesion, eps=1e-7):
  dR = lax.stop_gradient(dR)
  dr = np.linalg.norm(dR, axis=-1, keepdims=True)

  mask = dr < D_cohesion

  N_com = np.where(mask, 1.0, 0)
  dR_com = np.where(mask, dR, 0)
  dR_com = np.sum(dR_com, axis=1) / (np.sum(N_com, axis=1) + eps)
```

```
  dR_com = dR_com / np.linalg.norm(dR_com + eps, axis=1, keepdims=True)
  return f32(0.5) * J_cohesion * (1 - np.sum(dR_com * N, axis=1)) ** 2


def energy_fn(state):
  boids = state['boids']

  # Align Energy
  E_align = partial(align_fn, J_align=2., D_align=30., alpha=3.)
  E_align = vmap(vmap(E_align, (0, None, 0)), (0, 0, None))

  # + Avoidance Energy
  E_avoid = partial(avoid_fn, J_avoid=2., D_avoid=30., alpha=3.)
  E_avoid = vmap(vmap(E_avoid))

  # + Cohesion Energy
  E_cohesion = partial(cohesion_fn, J_cohesion=0.001, D_cohesion=100.)

  dR = space.map_product(displacement)(boids.R, boids.R)
  N = normal(boids.theta)

  return ( np.sum( E_align(dR, N, N) + E_avoid(dR) ) + np.sum(E_cohesion(dR, N)) )

update = dynamics(energy_fn=energy_fn, dt=1e-1, speed=1.)

boids_buffer = []

state = {
    'boids': boids
}

for i in ProgressIter(range(700)):
  state = lax.fori_loop(0, 50, update, state)
  boids_buffer += [state['boids']]

display(render(box_size,  boids_buffer))
```

Ahora los boids vuelan en grupos más adheridos. Uno puede ajustar qué tanto se mantienen unidos los boids al cambiar el radio de cohesión y su intensidad, pero si aumentamos demasiado la intensidad de cohesión se obtienen comportamientos no deseados:

```
def energy_fn(state):
  boids = state['boids']

  # Align Energy
  E_align = partial(align_fn, J_align=2., D_align=30., alpha=3.)
  E_align = vmap(vmap(E_align, (0, None, 0)), (0, 0, None))

  # + Avoidance Energy
  E_avoid = partial(avoid_fn, J_avoid=2., D_avoid=30., alpha=3.)
  E_avoid = vmap(vmap(E_avoid))

  # + Cohesion Energy
  E_cohesion = partial(cohesion_fn, J_cohesion=0.1, D_cohesion=100.)
  # J_cohesion raised from 0.001 to 0.5

  dR = space.map_product(displacement)(boids.R, boids.R)
  N = normal(boids.theta)

  return (0.5 * np.sum(E_align(dR, N, N) + E_avoid(dR)) +
          np.sum(E_cohesion(dR, N)))

update = dynamics(energy_fn=energy_fn, dt=1e-1, speed=1.)

boids_buffer = []
```

```
state = {
    'boids': boids
}

for i in ProgressIter(range(700)):
  state = lax.fori_loop(0, 50, update, state)
  boids_buffer += [state['boids']]

display(render(box_size,  boids_buffer))
```



✓ 0s completed at 5:11 PM