```python
# Module computing complex numbers
# disclaimer : this class is not made to deal with less than 1e-10 values


from math import isclose
from typing import List, Union

from numpy import arctan2, cos, pi, sin, sqrt


class Complex:
    """Computing complex numbers"""

    def __init__(self, real=0.0, imaginary=0.0):
        self.re = real  # round(real, 15)
        self.im = imaginary  # round(imaginary,15)

    def __str__(self) -> str:
        if self.im == 0.0:
            string = f"{self.re}"
        elif self.re == 0:
            string = f"i({self.im})"
        else:
            string = f"{self.re} + i({self.im})"
        return string

    __repr__ = __str__

    def __eq__(self, other) -> bool:
        return bool(isclose(self.re, other.re) and isclose(self.im, other.im))

    def is_null(self):
        return isclose(self.re, 0) and isclose(self.im, 0)

    def is_real(self):
        return isclose(self.im, 0)

    def is_imaginary(self):
        return isclose(self.re, 0)

    def arg(self):
        """return the argument of the complex number
        return None if 0"""
        if self.is_null():
            arg = None
        elif isclose(self.re, 0) and self.im > 0:
            arg = pi / 2
        elif isclose(self.re, 0) and self.im < 0:
            arg = -pi / 2
        else:
            arg = round(arctan2(self.im, self.re), 15)
        return arg

    def module(self):
        """return the module of the complex number"""
        return round(sqrt(self.re**2 + self.im**2), 15)

    def conjuagate(self):
        return Complex(self.re, -self.im)

    # arithmetic
    def __add__(self, other):
        return Complex(self.re + other.re, self.im + other.im)

    def __sub__(self, other):
        return Complex(self.re - other.re, self.im - other.im)
```

```python
        return Complex(self.re - other.re, self.im - other.im)

    def __mul__(self, other):
        real = (self.re * other.re) - (self.im * other.im)
        imaginary = (self.re * other.im) + (self.im * other.re)
        return Complex(real, imaginary)

    def __truediv__(self, other):
        if other.is_null():
            raise ValueError("Error : dividing by 0")
        elif other.is_real():
            return Complex(self.re / other.re, self.im / other.re)
        else:
            denominator = (other.re**2) + (other.im**2)
            real = ((self.re * other.re) + (self.im * other.im)) / denominator
            imaginary = ((self.im * other.re) - (self.re * other.im)) / denominator
            return Complex(real, imaginary)


Num = Union[int, float]


def addition(
    *complexes: Complex,
) -> (
    Complex
):  # partially depreciated (can still be usefull for more iterable arguments)
    """calculate the sum of complex numbers

    parameters
    ----------
        - *complexes : iterable type of Complex

    return
    ------
        - sum of the complex numbers"""

    res = Complex(0)
    for number in complexes:
        res.re += number.re
        res.im += number.im
    return res


def difference(
    cpx1: Complex, cpx2: Complex = Complex(0)
):  # fully depreciated (replaced by __sub__ Complex methods)
    """calculate the difference of two complex numbers

    parameters
    ----------
        - cpx1 : Complex number
        - cpx2 : Complex number to subtract to cpx1 (=Complex(0) by default)

    return
    ------
        -   difference of the two complex numbers"""
    res = Complex()
    res.re = cpx1.re - cpx2.re
    res.im = cpx1.im - cpx2.im
    return res


def product(
    *complexes: Complex,
) -> (
    Complex
```

```python
):  # partially depreciated (can still be usefull for more iterable arguments)
    """calculate the product of complex numbers

    parameters
    ----------
        - *complexes : iterable type of Complex

    return
    ------
        - product of the complex numbers"""
    res = Complex(1)
    for number in complexes:
        re = res.re * number.re - res.im * number.im
        im = res.re * number.im + res.im * number.re
        res.re = re
        res.im = im
    return res


def exp_to_literal(arg: float, module: float = 1.0) -> Complex:
    """return the literal expression of a complex number defined by its argument and module

    parameters
    ----------
        - arg : type(float) (should be between 0 and 2pi)
        - module : type(float) (must have a positive value)(=1 by default)

    return
    ------
        - Complex number associated"""
    assert module >= 0, "second-argument(module) must have a positive value"
    return Complex(module * cos(arg), module * sin(arg))


def nth_root(n: int, cpx: Complex = Complex(1)) -> Complex:
    """calculate the nth root of a complex number

    parameters
    ----------
        - n : type(int)
        - complex : type(Complex) (=Complex(1) by default) (must not be Complex(0))

    return
    ------
        - list of the nth roots"""
    assert (
        cpx.re != 0 or cpx.im != 0
    ), "second argument must be a non-zero complex number"
    module = cpx.module()
    arg = cpx.arg()
    if arg is not None:
        return exp_to_literal((arg / n), module ** (1 / n))
    else:
        return Complex(
            1
        )  # Not used case but just here to ensure nth_root cannot return None


def nth_roots_unity(n: int) -> list:
    """calculate the n roots of unity

    parameter
    ---------
        - n : type(int) : must be a positive integer

    return
    ------
```

```python
        - a list of Complex containing the n roots of unity"""
    roots = [Complex(1) for i in range(n)]
    for k in range(0, n):
        roots[k] = exp_to_literal((2 * k * pi / n), 1.0)
    return roots


def inverse_nth_roots_unity(n: int) -> list:
    """calculate the inversed n roots of unity

    parameter
    ---------
    - n : type(int) : must be a positive integer

    return
    ------
    - a list of Complex containing the inversed n roots of unity"""
    roots = [Complex(1) for i in range(n)]
    for k in range(0, n):
        roots[k] = exp_to_literal((-2 * k * pi / n), 1.0)
    return roots


def make_complex(values: List[Num]) -> List[Complex]:
    res = []
    for value in values:
        res.extend([Complex(value)])
    return res


if __name__ == "__main__":
    pass
```

```python
# fast-fourier transforms

from cmath import cos, exp, pi

import complex as cpx
from numpy import log2


def FFT(vector: list) -> list:
    """calculate the fast fourier tranform of a vector

    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - 1-D fast fourier transform of the vector"""
    n = len(vector)
    assert log2(
        n
    ).is_integer(), "make sure that the length of the arguement is a power of 2"
    if n == 1:
        return vector
    poly_even, poly_odd = vector[::2], vector[1::2]
    res_even, res_odd = FFT(poly_even), FFT(poly_odd)
    res = [cpx.Complex(0)] * n
    for j in range(n // 2):
        w_j = cpx.exp_to_literal(-2 * pi * j / n)
        product = w_j * res_odd[j]
        res[j] = res_even[j] + product
        res[j + n // 2] = res_even[j] - product
    return res


def IFFT_aux(vector: list) -> list:
    """auxiliary function that makes the recursive steps of the IFFT algorithm
    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - partial inverse of the 1-D fast fourier transform of the vector (lack the division by n)
    """
    n = len(vector)
    assert log2(
        n
    ).is_integer(), "make sure that the length of the arguement is a power of 2"
    if n == 1:
        return vector
    poly_even, poly_odd = vector[::2], vector[1::2]
    res_even, res_odd = IFFT_aux(poly_even), IFFT_aux(poly_odd)
    res = [cpx.Complex(0)] * n
    for j in range(n // 2):
        w_j = cpx.exp_to_literal((2 * pi * j) / n)
        product = w_j * res_odd[j]
        res[j] = res_even[j] + product
        res[j + n // 2] = res_even[j] - product
    return res


def IFFT(vector: list) -> list:
    """caclulate the inverse of the fast fourier tranform of a vector (in order to have ifft(fft(poly)) == poly)

    parameters
    ----------
        -vector : list of Complex object

    return
    ------
        - inverse of the 1-D fast fourier transform of the vector"""
    n = len(vector)
    res = IFFT_aux(vector)
    for i in range(n):
        res[i] = res[i] / cpx.Complex(n)
    return res
```

```python
import time as t

import matplotlib.animation as anim
import matplotlib.pyplot as plt
import numpy as np

DPI = 100


def create_movie(
    X, evolve, path, steps=100, cmap=None, interpolation="bicubic", interval=50
):

    print(f"Rendering {path}")
    time = t.time()
    if len(X.shape) == 2 and cmap is None:
        cmap = "gray_r"

    fig = plt.figure(figsize=(16, 12))
    im = plt.imshow(X, cmap=cmap, interpolation=interpolation, vmin=0, vmax=1)
    plt.axis("off")

    def update(i):

        if i % (steps // 10) == 0:
            print(f"Step {i}/{steps}")

        if i == 0:
            return (im,)
        nonlocal X
        X = evolve(X)
        im.set_array(X)
        return (im,)

    ani = anim.FuncAnimation(fig, update, steps, interval=interval, blit=True)
    ani.save(path, fps=25, dpi=DPI)
    time = t.time() - time
    print(f"Done in {time//60}min{time%60}s")


def create_movie_multi(Xs, evolve, path, steps=100, interpolation="bicubic"):

    fig = plt.figure(figsize=(16, 9))
    im = plt.imshow(np.dstack(Xs), interpolation=interpolation)
    plt.axis("off")

    def update(i):

        if i % (steps // 10) == 0:
            print(f"Step {i}/{steps}")

        if i == 0:
            return (im,)
        nonlocal Xs
        Xs = evolve(Xs)
        im.set_array(np.dstack(Xs))
        return (im,)

    ani = anim.FuncAnimation(fig, update, steps, interval=50, blit=True)
    ani.save(path, fps=25, dpi=DPI)
```

```python
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
from movie import *

import species

# Utils

path_simul = (
    "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/simul/"  # absolute path ! careful !
)
path_graphs = "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/doc/slideshow/img/"


def gauss(x, mu: float, sigma: float):
    """Return non-normalized gaussian function of expected value mu and
    variance sigma ** 2"""
    return np.exp(-0.5 * ((x - mu) / sigma) ** 2)


def polynomial(x, alpha: int):
    return (4 * x * (1 - x)) ** alpha


# Game of life (GoL)


def evolution_gol(grid):
    # count the neighbor considering a periodic grid (wrappred around its border)
    neighbor_count = sum(
        np.roll(np.roll(grid, i, 0), j, 1)
        for i in (-1, 0, 1)
        for j in (-1, 0, 1)
        if (i != 0 or j != 0)
    )
    return (neighbor_count == 3) | (grid & (neighbor_count == 2))


# simulation test
grid = np.random.randint(0, 2, (64, 64))
# create_movie(
#     grid,
#     evolution_gol,
#     path_simul + "gol_simul.mp4",
#     200,
#     cmap="plasma",
#     interpolation="none",
#     interval=200,
# )

# GoL with continuous kernel and growth

kernel_gol = np.array([[1, 1, 1], [1, 0, 1], [1, 1, 1]])


def growth_gol(neighbor_val):
    cond1 = (neighbor_val >= 1) & (neighbor_val <= 3)
    cond2 = (neighbor_val > 3) & (neighbor_val <= 4)
    return -1 + (neighbor_val - 1) * cond1 + 8 * (1 - neighbor_val / 4) * cond2


def evolution_continuous_gol(grid):
    neighbor_count = sp.signal.convolve2d(
        grid, kernel_gol, mode="same", boundary="wrap"
    )
    grid = grid + growth_gol(neighbor_count)
```

```
    grid = np.clip(grid, 0, 1)
    return grid


# simulation test
grid = np.random.randint(0, 2, (64, 64))
# create_movie(
#     grid,
#     evolution_continuous_gol,
#     path_simul + "gol_continuous_simul.mp4",
#     200,
#     cmap="plasma",
#     interpolation="none",
#     interval=200,
# )

# Lenia

scale = 1  # scaling factor to speed up rendering when testing

# Ring filter

R = 13  # radius of kernel
x, y = np.ogrid[-R:R, -R:R]  # grid
dist_norm = (((1 + x) ** 2 + (1 + y) ** 2) ** 0.5) / R  # normalized so that dist(R) = 1

gamma = 0.5
delta = 0.15
kernel_shell = (dist_norm <= 1) * gauss(
    dist_norm, gamma, delta
)  # we don't consider neighbor having dist > 1
kernel_shell = kernel_shell / np.sum(kernel_shell)  # normalizing values

# show ring
plt.figure(figsize=(20, 10))
plt.subplot(121)
plt.imshow(dist_norm, interpolation="none", cmap="plasma")
plt.subplot(122)
plt.imshow(kernel_shell, interpolation="none", cmap="plasma")
plt.savefig(path_simul + "ring_kernel.png")

# Growth function


def growth_lenia(region):
    mu = 0.15
    sigma = 0.015
    return -1 + 2 * gauss(region, mu, sigma)


# Evolve function

dt = 0.1  # set the time step


def evolution_lenia(grid):
    neighbor = sp.signal.convolve2d(grid, kernel_shell, mode="same", boundary="wrap")
    grid = grid + dt * growth_lenia(neighbor)
    grid = np.clip(grid, 0, 1)
    return grid


# simulation test
size = int(256 * scale)
mid = size // 2
grid = np.ones((size, size))

# gaussian spot initialization
```

```python
# gaussian spot initialization
radius = int(36 * scale)
y, x = np.ogrid[-mid:mid, -mid:mid]
grid = np.exp(-0.5 * (x**2 + y**2) / radius**2)

# create_movie(grid, evolution_lenia, path_simul + "lenia_spot.mp4", 700, cmap="plasma")


# Graphs

# basic example

# random initialization
grid = np.random.rand(size, size)


def plot_basic_lenia():
    global grid
    fig, ax = plt.subplots(3, 4)
    step = 500
    plotted_steps = [0, 1, 3, 5, 9, 15, 30, 60, 90, 125, 200, 500]
    k = 0
    i, j = 0, 0
    while k <= step:
        if k in plotted_steps:
            ax[i, j].imshow(grid, cmap="plasma")
            ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
            ax[i, j].axis("off")
            if j == 3:
                i += 1
                j = 0
            else:
                j += 1
        grid = evolution_lenia(grid)
        k += 1

    fig.tight_layout()
    plt.savefig(path_graphs + "evolution_lenia_random_init.png", transparent=True)


# plot_basic_lenia()


# example with perturbation

# random initialization
grid = np.random.rand(size, size)


def plot_basic_lenia_with_perturbation():
    global grid
    fig, ax = plt.subplots(3, 4)
    step = 500
    plotted_steps = [0, 15, 30, 60, 125, 199, 200, 205, 210, 220, 250, 500]
    step_perturbation = 200
    k = 0
    i, j = 0, 0
    while k <= step:
        if k == step_perturbation:
            for x in range(1 * len(grid) // 3, 2 * len(grid) // 3):
                for y in range(3 * len(grid[i]) // 6, 5 * len(grid[i]) // 6):
                    grid[x, y] = 0
        if k in plotted_steps:
            ax[i, j].imshow(grid, cmap="plasma")
            if k == step_perturbation:
                ax[i, j].set_title(
                    f"t = {k} : Perturbation", color="r", fontweight="bold", fontsize=7
                )
            else:
```

```
            else:
                ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
            ax[i, j].axis("off")
            if j == 3:
                i += 1
                j = 0
            else:
                j += 1

        grid = evolution_lenia(grid)
        k += 1

    fig.tight_layout()
    plt.savefig(
        path_graphs + "evolution_lenia_random_init_perturbation.png", transparent=True
    )


# plot_basic_lenia_with_perturbation()


# random initialization
grid = np.random.rand(size, size)

# create_movie(
#     grid,
#     evolution_lenia,
#     path_simul + "lenia_random.mp4",
#     300,
#     cmap="plasma",
#     interpolation="none",)

# Orbium (gol's glider "equivalent")

orbium = species.orbium

plt.imshow(orbium.T, cmap="plasma")
plt.savefig(path_simul + "orbium.png")
plt.savefig(path_graphs + "orbium.png")

size = 128
grid = np.zeros((size, size))
pos = size // 6
grid[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T


def plot_orbium():

    size = 64
    grid_basic = np.zeros((size, size))
    pos = size // 6
    grid_basic[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T
    grid_perturbation = grid_basic.copy()

    fig, ax = plt.subplots(2, 6, figsize=(12, 5))
    fontsize = 10

    step = 100
    plotted_steps = [0, 25, 50, 75, 100]
    step_perturbation = 50

    ax[0, 0].imshow(orbium, cmap="plasma", interpolation="bicubic", vmin=0, vmax=1)
    ax[0, 0].set_title("Forme", fontsize="x-large")
    ax[0, 0].axis("off")
    k = 0
    j1, j2 = 1, 0
    while k <= step:
        if k == step_perturbation:
```

```
            for x in range(len(grid_perturbation)):
                for y in range(len(grid_perturbation[x])):
                    if grid_perturbation[x, y] > 0:
                        v = np.random.choice(
                            [0, grid_perturbation[x, y]], p=[1 / 3, 1 - 1 / 3]
                        )
                        grid_perturbation[x, y] = v
        if k in plotted_steps:
            ax[0, j1].imshow(grid_basic, cmap="plasma")
            ax[1, j2].imshow(grid_perturbation, cmap="plasma")
            ax[0, j1].axis("off")
            ax[1, j2].axis("off")
            ax[0, j1].set_title(f"t = {k}", fontweight="bold", fontsize=fontsize)
            if k == step_perturbation:
                ax[1, j2].set_title(
                    f"t = {k} : Perturbation",
                    color="r",
                    fontweight="bold",
                    fontsize=fontsize,
                )
            else:
                ax[1, j2].set_title(f"t = {k}", fontweight="bold", fontsize=fontsize)
            j1 += 1
            j2 += 1

        if k == step_perturbation - 1:
            ax[1, j2].imshow(grid_perturbation, cmap="plasma")
            ax[1, j2].axis("off")
            ax[1, j2].set_title(f"t = {k}", fontweight="bold", fontsize=7)
            j2 += 1

        grid_basic = evolution_lenia(grid_basic)
        if k < step_perturbation:
            grid_perturbation = grid_basic.copy()
        else:
            grid_perturbation = evolution_lenia(grid_perturbation)
        k += 1

    fig.tight_layout()
    plt.savefig(path_graphs + "evolution_orbium.png", transparent=True)
    plt.clf()


# plot_orbium()


# create_movie(
#     grid, evolution_lenia, path_simul + "lenia_orbium.mp4", 100, cmap="plasma", interval=50
# )

# Lenia optimization with fft

size = 128
mid = size // 2
grid = np.zeros((size, size))
pos = size // 6
grid[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T

# redefine kernel to meet fft's requirements

R = 13
x, y = np.ogrid[-mid:mid, -mid:mid]  # grid
dist_norm = (((x**2 + y**2) ** 0.5)) / R  # normalized so that dist(R) = 1
kernel_shell = (dist_norm <= 1) * gauss(dist_norm, 0.5, 0.15)
kernel_shell = kernel_shell / np.sum(kernel_shell)
f_kernel = sp.fft.fft2(sp.fft.fftshift(kernel_shell))  # fft of kernel
# show ring fft
plt.figure(figsize=(20, 10))
```

```
plt.subplot(121)
plt.imshow(dist_norm, interpolation="none", cmap="plasma")
plt.subplot(122)
plt.imshow(kernel_shell, interpolation="none", cmap="plasma")
plt.savefig(path_simul + "ring_kernel_fft.png")
plt.clf()


def evolution_lenia_fft(grid):
    neighbor = np.real(sp.fft.ifft2(f_kernel * sp.fft.fft2(grid)))
    grid = np.clip(grid + dt * growth_lenia(neighbor), 0, 1)
    return grid


# create_movie(
#     grid,
#     evolution_lenia_fft,
#     path_simul + "lenia_orbium_fft.mp4",
#     500,
#     cmap="plasma",
#     interval=50,
# )


# Multi Kernel

size = 128
mid = size // 2
x, y = np.ogrid[-mid:mid, -mid:mid]
R = 36
amplitude = [1, 0.667, 0.333, 0.667]
dist_norm = (x**2 + y**2) ** 0.5 / R * len(amplitude)

kernel_multi_quadrium = np.zeros_like(dist_norm)
alpha = 4
for i in range(len(amplitude)):
    kernel_multi_quadrium += (
        (dist_norm.astype(int) == i) * amplitude[i] * polynomial(dist_norm % 1, alpha)
    )
kernel_multi_quadrium /= np.sum(kernel_multi_quadrium)
f_kernel = sp.fft.fft2(sp.fft.fftshift(kernel_multi_quadrium))  # fft of kernel


def growth_quadrium_lenia(region):
    mu = 0.16
    sigma = 0.01
    return 2 * gauss(region, mu, sigma) - 1


def evolution_quadrium_fft(grid):
    neighbor = np.real(sp.fft.ifft2(f_kernel * sp.fft.fft2(grid)))
    grid = np.clip(grid + dt * growth_quadrium_lenia(neighbor), 0, 1)
    return grid


grid = np.zeros((size, size))
quadrium = species.quadrium
pos = size // 10
grid[size - quadrium.shape[0] - pos : size - pos, 0 : quadrium.shape[1]] = quadrium

# create_movie(
#     grid,
#     evolution_quadrium_fft,
#     path_simul + "quadrium_simul.mp4",
#     200,
#     cmap="plasma",
#     interpolation="none",
#     interval=200,
```

```python
# )


def plot_quadrium():
    global grid
    fig, ax = plt.subplots(2, 4)
    step = 500
    plotted_steps = [0, 50, 100, 200, 300, 400, 500]
    k = 0
    i, j = 0, 1
    ax[0, 0].imshow(quadrium, cmap="plasma")
    ax[0, 0].axis("off")
    ax[0, 0].set_title("Forme", fontsize="x-large")
    while k <= step:
        if k in plotted_steps:
            ax[i, j].imshow(grid, cmap="plasma")
            ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
            ax[i, j].axis("off")
            if j == 3:
                i += 1
                j = 0
            else:
                j += 1
        grid = evolution_quadrium_fft(grid)
        k += 1

    fig.tight_layout()
    plt.savefig(path_graphs + "evolution_quadrium.png", transparent=True)


# plot_quadrium()

# Multi-channel Lenia

kernels_table = species.aquarium["kernels"]

betas = [k["b"] for k in kernels_table]
mus = [k["m"] for k in kernels_table]
sigmas = [k["s"] for k in kernels_table]
heights = [k["h"] for k in kernels_table]
radii = [k["r"] for k in kernels_table]
sources = [k["c0"] for k in kernels_table]
destinations = [k["c1"] for k in kernels_table]

gamma = 0.5
delta = 0.15

dt = 0.5
R = 12
size = 128
mid = size // 2
x, y = np.ogrid[-mid:mid, -mid:mid]


kernel_shells = []
kernels_fft = []

fig, ax = plt.subplots(1)

for b, r in zip(betas, radii):
    r *= R
    dist_norm = (x**2 + y**2) ** 0.5 / r * len(b)
    kernel_shell = np.zeros_like(dist_norm)
    for i in range(len(b)):
        mask_norm = dist_norm.astype(int) == i
        kernel_shell += mask_norm * b[i] * gauss(dist_norm % 1, gamma, delta)
    kernel_shell /= kernel_shell.sum()
    kernel_shells.append(kernel_shell)
```

```python
    kernels_fft.append(sp.fft.fft2(sp.fft.fftshift(kernel_shell)))

colors = {0: "r", 1: "g", 2: "b"}
for i, k in enumerate(kernel_shells):
    ax.plot(k[size // 2, :], color=colors[sources[i]], label=f"canal {sources[i]}")
    ax.xaxis.set_visible(False)

fig.tight_layout()
# plt.savefig(path_graphs + "plot_kernel_multi_channel.png", transparent=True)

grids = [np.zeros((size, size)) for _ in range(3)]


def evolution_multi_channel(grids):
    grids_fft = [sp.fft.fft2(grid) for grid in grids]
    potentials = [
        np.real(np.fft.ifft2(kernel_fft * grids_fft[source]))
        for kernel_fft, source in zip(kernels_fft, sources)
    ]
    growths_potential = [
        2 * gauss(potential, mus[i], sigmas[i]) - 1
        for i, potential in enumerate(potentials)
    ]
    growths = np.zeros_like(grids)
    for destination, height, growth in zip(destinations, heights, growths_potential):
        growths[destination] += height * growth
    grids = [np.clip(grid + dt * growth, 0, 1) for grid, growth in zip(grids, growths)]
    return grids


aquarium = [np.array(species.aquarium["cells"][c]) for c in range(3)]

for c in range(3):
    grids[c][mid : mid + aquarium[c].shape[0], mid : mid + aquarium[c].shape[1]] = (
        aquarium[c]
    )


def plot_aquarium():
    global grids
    fig, ax = plt.subplots(3, 4)
    step = 600
    plotted_steps = [0, 10, 20, 50, 100, 150, 200, 300, 400, 500, 600]
    k = 0
    i, j = 0, 1
    ax[0, 0].imshow(np.dstack(aquarium))
    ax[0, 0].axis("off")
    ax[0, 0].set_title("Forme", fontsize="x-large")
    while k <= step:
        if k in plotted_steps:
            ax[i, j].imshow(np.dstack(grids))
            ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
            ax[i, j].axis("off")
            if j == 3:
                i += 1
                j = 0
            else:
                j += 1
        grids = evolution_multi_channel(grids)
        k += 1

    fig.tight_layout()
    plt.savefig(path_graphs + "evolution_aquarium.png", transparent=True)


# plot_aquarium()
```

```python
import torch
from gymnasium import Dict

# This code is part of bigger system, that I haven't code myself,
# here is only the imgep algorithm run, that I did code myself (inspired ofc
# from the work made by INRIA bordeaux)


def execute_imgep_exploration(self, exploration_runs, resume_existing_run=False):
    retry = True

    while retry:
        print("STARTING NEW INITIALIZATION")
        print("Exploration: ")

        if resume_existing_run:
            current_run = len(self.policy_archive)
        else:
            self.policy_archive = []
            self.goal_archive = torch.empty((0,) + self.goal_space.shape)
            current_run = 0

        alive_randoms = 0

        while current_run < exploration_runs:
            policy_params = Dict.fromkeys(["init_state", "update_strategy"])

            if len(self.policy_archive) < self.config.initial_random_runs:
                target = None
                selected_policy = None
                goal_achieved = torch.ones(19)

                policy_params["init_state"] = self.system.init_space.sample()
                policy_params["update_strategy"] = self.system.strategy_space.sample()
                policy_params["update_strategy"].h /= 3

                self.system.reset(
                    initialization_parameters=policy_params["init_state"],
                    update_rule_parameters=policy_params["update_strategy"],
                )

                with torch.no_grad():
                    self.system.random_obstacle(8)
                    self.system.generate_init_state()
                    results = self.system.run()
                    goal_achieved = self.goal_space.map(results)

                is_failed = goal_achieved[0] > 0.9 or goal_achieved[1] < -0.5
                if not is_failed:
                    alive_randoms += 1

                optimization_steps = 0
                distance_to_goal = None

            else:
                if len(self.policy_archive) - self.config.initial_random_runs < 8:
                    target = torch.ones(3) * -10
                    target[0] = 0.065
                    target[2] = (
                        0.19
                        - (len(self.policy_archive) - self.config.initial_random_runs)
                        * 0.06
                    )
                    target[1] = 0
                else:
```

```python
    target = self.sample_interesting_goal()

if len(self.policy_archive) - self.config.initial_random_runs >= 2:
    print(f"Run {current_run}, optimizing towards goal: ")
    print("TARGET =", str(target))

selected_policy_idx = self.find_source_policy(target)
selected_policy = self.policy_archive[selected_policy_idx]

if (
    len(self.policy_archive) - self.config.initial_random_runs < 8
    or len(self.policy_archive) % 5 == 0
):
    policy_params["init_state"] = deepcopy(
        selected_policy["init_state"]
    )
    policy_params["update_strategy"] = deepcopy(
        selected_policy["update_strategy"]
    )
    self.system.reset(
        initialization_parameters=policy_params["init_state"],
        update_rule_parameters=policy_params["update_strategy"],
    )
    iterations = self.config.goal_optimizer.steps
else:
    iterations = 15
    mutation_failed = True
    while mutation_failed:
        policy_params["init_state"] = self.system.init_space.mutate(
            selected_policy["init_state"]
        )
        policy_params["update_strategy"] = (
            self.system.strategy_space.mutate(
                selected_policy["update_strategy"]
            )
        )
        self.system.reset(
            initialization_parameters=policy_params["init_state"],
            update_rule_parameters=policy_params["update_strategy"],
        )
        with torch.no_grad():
            self.system.generate_init_state()
            results = self.system.run()
            goal_achieved = self.goal_space.map(results)

        if (
            results.states[-1, :, :, 0].sum() > 10
            or goal_achieved[0] > 0.11
        ):
            mutation_failed = False

if (
    isinstance(self.system, torch.nn.Module)
    and self.config.goal_optimizer.steps > 0
):
    optimizer_class = eval(
        f"torch.optim.{self.config.goal_optimizer.name}"
    )
    self.goal_optimizer = optimizer_class(
        [
            {
                "params": self.system.init_state.parameters(),
                **self.config.goal_optimizer.init_cppn.parameters,
            },
            {
                "params": self.system.step.parameters(),
```

```
                    **self.config.goal_optimizer.step.parameters,
                },
            ],
            **self.config.goal_optimizer.parameters,
        )

        last_failed = False
        for optimization_steps in range(1, iterations):
            self.system.random_obstacle(8)
            self.system.generate_init_state()
            results = self.system.run()
            goal_achieved = self.goal_space.map(results)

            x = torch.arange(self.system.config.SX)
            y = torch.arange(self.system.config.SY)
            xx = x.view(-1, 1).repeat(1, self.system.config.SY)
            yy = y.repeat(self.system.config.SX, 1)
            X = (
                xx - (target[1] + 0.5) * self.system.config.SX
            ).float() / 35
            Y = (
                yy - (target[2] + 0.5) * self.system.config.SY
            ).float() / 35
            D = torch.sqrt(X**2 + Y**2)
            mask = 0.85 * (D < 0.5).float() + 0.15 * (D < 1).float()

            loss = (
                (0.9 * mask - results.states[-1, :, :, 0])
                .pow(2)
                .sum()
                .sqrt()
            )

            self.goal_optimizer.zero_grad()
            loss.backward()
            self.goal_optimizer.step()

            self.system.step.compute_kernel()

            failed = results.states[-1, :, :, 0].sum() < 10
            if failed and last_failed:
                self.goal_optimizer.zero_grad()
                break
            last_failed = failed

        if (
            len(self.policy_archive) >= self.config.initial_random_runs
            and len(self.policy_archive) - self.config.initial_random_runs
            < 2
        ):
            if loss > 19.5:
                break
            elif (
                len(self.policy_archive) - self.config.initial_random_runs
                == 2
            ):
                retry = False

        self.system.update_initialization_parameters()
        self.system.update_update_rule_parameters()
        policy_params["init_state"] = self.system.initialization_parameters
        policy_params["update_strategy"] = (
            self.system.update_rule_parameters
        )
        distance_to_goal = loss.item()

```

```
                goal_achieved = torch.zeros(3).cpu()
                with torch.no_grad():
                    for _ in range(20):
                        self.system.random_obstacle(8)
                        self.system.generate_init_state()
                        results = self.system.run()
                        if results.states[-1, :, :, 0].sum() < 10:
                            goal_achieved[0] = 10
                            break
                        goal_achieved = (
                            goal_achieved + self.goal_space.map(results).cpu() / 20
                        )

                if len(self.policy_archive) - self.config.initial_random_runs >= 2:
                    print("reached=", str(goal_achieved))

            goal_achieved = goal_achieved.cpu()
            self.db.add_run_data(
                id=current_run,
                policy_parameters=policy_params,
                observations=results,
                source_policy_idx=selected_policy_idx,
                target_goal=target,
                reached_goal=goal_achieved,
                n_optim_steps_to_reach_goal=optimization_steps,
                dist_to_target=distance_to_goal,
            )

            self.policy_archive.append(policy_params)
            self.goal_archive = torch.cat(
                [
                    self.goal_archive,
                    goal_achieved.reshape(1, -1).to(self.goal_archive.device).detach(),
                ]
            )

            if len(self.policy_archive) >= self.config.initial_random_runs:
                plt.imshow(self.system.init_wall.cpu())
                plt.scatter(
                    (
                        (self.goal_archive[:, 0] < 0.11).float()
                        * (self.goal_archive[:, 2] > -0.5).float()
                        * (self.goal_archive[:, 2] + 0.5)
                        * self.system.config.SY
                    ).cpu(),
                    (
                        (self.goal_archive[:, 0] < 0.11).float()
                        * (self.goal_archive[:, 1] > -0.5).float()
                        * (self.goal_archive[:, 1] + 0.5)
                        * self.system.config.SX
                    ).cpu(),
                )
                plt.show()

            current_run += 1

            if len(self.policy_archive) == self.config.initial_random_runs:
                if alive_randoms < 2:
                    break
                print(current_run)

            if len(self.policy_archive) == exploration_runs - 1:
                retry = False
```

```python
# Rendering the different graphs used for the project
import os

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import colors
from matplotlib.gridspec import GridSpec, GridSpecFromSubplotSpec
from mpl_toolkits.axes_grid1.inset_locator import mark_inset, zoomed_inset_axes
from PIL import Image

path = "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/doc/slideshow/img/"


## Game of Life

# evolution step

gol_cmap = colors.ListedColormap(["#960c6b", "#000066", "#cdd300"])
bounds = [-1.5, -0.5, 0.5, 1.5]
norm = colors.BoundaryNorm(bounds, gol_cmap.N)


def evolution_gol(grid, step=1):
    assert step >= 0, "step argument must be >= 0"
    evolved = grid.copy()
    evolved = evolved.astype(int)
    for _ in range(step):
        # count the neighbor considering a periodic grid (wrappred around its border)
        neighbor_count = sum(
            np.roll(np.roll(evolved, i, 0), j, 1)
            for i in (-1, 0, 1)
            for j in (-1, 0, 1)
            if (i != 0 or j != 0)
        )
        evolved = (neighbor_count == 3) | (evolved & (neighbor_count == 2))
    return evolved


# simulation test
np.random.seed(69)
grid = np.random.choice([0, 1], (64, 64), True, p=[0.8, 0.2])
evolved = evolution_gol(grid)
grid[42, 5] = -1
evolved[42, 5] = -1

fig, axs = plt.subplots(1, 2, figsize=(10, 5))

axs[0].imshow(grid, interpolation="none", cmap=gol_cmap, norm=norm)
axs[0].tick_params(which="minor", bottom=False, left=False)
axs[0].invert_yaxis()
axs[0].set_title("Situation Initiale", fontsize=20)


axs[1].imshow(evolved, interpolation="none", cmap=gol_cmap, norm=norm)
axs[1].tick_params(which="minor", bottom=False, left=False)
axs[1].invert_yaxis()
axs[1].set_title("AprĂ¨s une Ăšvolution", fontsize=20)

ax_zoom = zoomed_inset_axes(axs[0], zoom=8, loc="upper right")
ax_zoom.imshow(grid, cmap=gol_cmap, norm=norm)

# subregion of the original image
x1, x2, y1, y2 = 3.5, 6.5, 40.5, 43.5
ax_zoom.set_xlim(x1, x2)
ax_zoom.set_ylim(y1, y2)  # fix the number of ticks on the inset Axes
```

```
ax_zoom.yaxis.get_major_locator().set_params(nbins=4)
ax_zoom.xaxis.get_major_locator().set_params(nbins=4)
ax_zoom.set_xticks(np.arange(3.5, 7, 1), minor=True)
ax_zoom.set_yticks(np.arange(40.5, 44, 1), minor=True)

ax_zoom.tick_params(labelleft=False, labelbottom=False)
ax_zoom.grid(which="minor", color="black", linewidth=2)

# draw a bbox of the region of the inset Axes in the parent Axes and
# connecting lines between the bbox and the inset Axes area
mark_inset(
    axs[0], ax_zoom, loc1=2, loc2=4, fc="none", ec="0.5", color="red", linewidth=3
)

plt.savefig(path + "plot_evolution_gol.png", transparent=True)
plt.clf()


# species

grid_block = np.zeros((6, 6))
coord_block = [(2, 2), (2, 3), (3, 2), (3, 3)]
for coord in coord_block:
    grid_block[coord] = 1

grid_blinker = np.zeros((5, 5))
coord_blinker = [(2, 1), (2, 2), (2, 3)]
for coord in coord_blinker:
    grid_blinker[coord] = 1

grid_glider = np.zeros((16, 16))
coord_glider = [(1, 1), (2, 2), (2, 3), (3, 1), (3, 2)]
for coord in coord_glider:
    grid_glider[coord] = 1


fig = plt.figure(figsize=(10, 6))
outer_gs = GridSpec(
    2,
    2,
    figure=fig,
    height_ratios=[1, 1],
    width_ratios=[1, 1.5],
    hspace=0.1,
    wspace=0.2,
)


def add_centered_pcolor(sub_gs, data_list, plot_titles, line_title):
    num_plots = len(data_list)
    ax_line = fig.add_subplot(sub_gs)
    ax_line.text(
        0.5,
        1,
        line_title,
        ha="center",
        va="center",
        fontsize=20,
        fontweight="bold",
        transform=ax_line.transAxes,
    )
    ax_line.axis("off")

    inner_gs = GridSpecFromSubplotSpec(1, num_plots, subplot_spec=sub_gs, wspace=0.1)
    for i, (data, plot_title) in enumerate(zip(data_list, plot_titles)):
        ax = fig.add_subplot(inner_gs[i])
```

```python
        ax.pcolor(data, cmap="plasma", edgecolor="grey", linewidth=0.5)
        ax.set_aspect("equal")
        ax.invert_yaxis()
        ax.set_title(plot_title, fontsize=12)
        ax.tick_params(left=False, bottom=False, labelleft=False, labelbottom=False)


data_block = [grid_block, evolution_gol(grid_block)]
data_blinker = [evolution_gol(grid_blinker, i) for i in range(3)]
data_glider = [evolution_gol(grid_glider, 11 * i) for i in range(5)]

titles_block = ["t = 0", "t = 1"]
titles_blinker = ["t = 0", "t = 1", "t = 2"]
titles_glider = [f"t = {11 * i}" for i in range(5)]

plot_block = outer_gs[0, 0]
add_centered_pcolor(plot_block, data_block, titles_block, "Block")

plot_blinker = outer_gs[0, 1]
add_centered_pcolor(plot_blinker, data_blinker, titles_blinker, "Blinker")

plot_glider = outer_gs[1, :]
add_centered_pcolor(plot_glider, data_glider, titles_glider, "Glider")

plt.savefig(path + "plot_species_gol.png", transparent=True)
plt.clf()


# Kernels
def indicator(arr, lower_bound: float = 0, upper_bound: float = 1):
    if type(arr) in ["float", "int"]:
        return int(lower_bound <= arr <= upper_bound)
    else:
        arr = np.copy(arr)
        mask = (arr >= lower_bound) & (arr <= upper_bound)
        arr[mask] = 1
        arr[~mask] = 0
        return arr


def gauss(x, gamma: float = 0.5, delta: float = 0.15):
    return np.exp(-0.5 * ((x - gamma) / delta) ** 2)


def polynomial(x, alpha: int = 4):
    return (4 * x * (1 - x)) ** alpha


fig = plt.figure(figsize=(10, 13))

subfigs = fig.subfigures(1, 2)

dist_1d = np.arange(0, 1, 0.001)
step = 100j
x, y = np.ogrid[-1 : 1 : 2 * step, -1 : 1 : 2 * step]  # grid
dist_norm = ((x) ** 2 + (y) ** 2) ** 0.5

kernel_gauss = (dist_norm <= 1) * (gauss(dist_norm))
kernel_polynomial = (dist_norm <= 1) * (polynomial(dist_norm))
kernel_rectangle = (dist_norm <= 1) * (indicator(dist_norm, 1 / 3, 2 / 3))

ax1 = subfigs[0].subplots(2, 1)
ax1[0].plot(dist_1d, gauss(dist_1d))
ax1[0].text(
    0.5,
    0,
```

```
    r"$\gamma = 0.5, \delta = 0.15$",
    fontsize=20,
    horizontalalignment="center",
)
ax1[0].set_xlabel("Distance", fontsize="x-large")
im = ax1[1].imshow(kernel_gauss, interpolation="none", cmap="plasma")
ax1[1].axis("off")
subfigs[0].suptitle("Exponentiel ", fontsize=30, fontweight="bold")

ax2 = subfigs[1].subplots(2, 1)
ax2[0].plot(
    dist_1d,
    indicator(dist_1d, 2 / 4, 3 / 4),
    label=r"$[a, b] = [\frac{2}{4}, \frac{3}{4}]$",
)
ax2[0].text(
    0,
    0.5,
    r"$[a, b] = [\frac{2}{4}, \frac{3}{4}]$",
    fontsize=20,
)
ax2[0].set_xlabel("Distance", fontsize="x-large")
ax2[1].imshow(kernel_rectangle, interpolation="none", cmap="plasma")
ax2[1].axis("off")
subfigs[1].suptitle("Rectangle", fontsize=30, fontweight="bold")

fig.tight_layout(pad=4, h_pad=1, w_pad=4)
fig.subplots_adjust(top=0.92)
cbar_ax = fig.add_axes((0.25, 0.05, 0.5, 0.025))
cbar = fig.colorbar(im, cax=cbar_ax, orientation="horizontal")
cbar.set_ticks([0, 1])
cbar.ax.tick_params(labelsize=20)


plt.savefig(path + "plot_convolution_kernels.png", transparent=True)
plt.clf()

# Growth mapping

mu1, mu2 = 0.3, 0.7
sigma1, sigma2 = 0.05, 0.2
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
interval = np.arange(0, 1, 0.0001)

axs[0].plot(interval, 2 * gauss(interval, mu1, sigma1) - 1)
axs[0].set_title("Exponentielle", fontsize=20, fontweight="bold")
axs[0].text(
    1,
    0.75,
    f"$\\mu = {mu1}, \\sigma = {sigma1}$",
    fontsize=13,
    horizontalalignment="right",
)

axs[1].plot(interval, 2 * indicator(interval, mu2 - sigma2, mu2 + sigma2) - 1)
axs[1].set_title("Rectangulaire", fontsize=20, fontweight="bold")
axs[1].text(
    0,
    0.75,
    f"$\\mu = {mu2}, \\sigma = {sigma2}$",
    fontsize=13,
    horizontalalignment="left",
)

for i in [0, 1]:
    axs[i].axhline(y=0, color="r", linestyle="--", linewidth=2, alpha=0.5)
```

```

plt.savefig(path + "plot_growth_mapping.png", transparent=True)
plt.clf()


# Multi Kernels

A = [0.3, 1, 0.7, 0.2]
gamma = np.random.uniform(0, 1, (len(A),))
delta = np.random.uniform(0, 0.3, (len(A),))
gamma = [0.2, 0.4, 0.6, 0.8]
delta = [0.015, 0.05, 0.01, 0.1]

dist_1d = np.arange(0, 1, 0.001)

step = 1000j
x, y = np.ogrid[-1 : 1 : 2 * step, -1 : 1 : 2 * step]  # grid
dist_norm = ((x) ** 2 + (y) ** 2) ** 0.5

multi_kernel_core = np.zeros_like(dist_1d)
multi_kernel_shell = np.zeros_like(dist_norm)
for i in range(len(A)):
    multi_kernel_core += A[i] * gauss(dist_1d, gamma[i], delta[i])
    multi_kernel_shell += (dist_norm <= 1) * A[i] * gauss(dist_norm, gamma[i], delta[i])


fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].plot(dist_1d, multi_kernel_core)
ax[0].set_xlabel("Distance", fontsize="x-large")
im = ax[1].imshow(multi_kernel_shell, cmap="plasma")
ax[1].axis("off")
plt.colorbar(im, ax=ax[1], cmap="plasma", location="bottom", shrink=0.7)

fig.tight_layout()
plt.savefig(path + "plot_multi_ring_kernel.png", transparent=True)


folder_path = (
    "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/simul/resultLeniaMachineLearning/"
)


def name_image(i: int):
    s = str(i + 1)
    while len(s) != 5:
        s = "0" + s
    return s + ".png"


steps = [0, 3, 5, 7, 10, 20, 30, 40, 50, 75, 100, 150, 200, 300, 400, 500]
image_names = [name_image(i) for i in steps]

num_images = len(image_names)
cols = 4
rows = (num_images + cols - 1) // cols

fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 2.5 * rows))
axes = axes.flatten()

i = 0
for i, image_name in enumerate(image_names):
    img_path = os.path.join(folder_path, image_name)
    img = Image.open(img_path)
    axes[i].imshow(img)
    axes[i].set_title(f"t = {steps[i]}", fontweight="bold", fontsize="x-large")
    axes[i].axis("off")
```

```python

for j in range(i + 1, len(axes)):
    axes[j].axis("off")

plt.tight_layout()
plt.savefig(path + "evolution_machine_learning", transparent=True)
```