

```

1  # Module computing complex numbers
2  # disclaimer : this class is not made to deal with less than 1e-10 values
3
4
5  from math import isclose
6  from typing import List, Union
7
8  from numpy import arctan2, cos, pi, sin, sqrt
9
10
11 class Complex:
12     """Computing complex numbers"""
13
14     def __init__(self, real=0.0, imaginary=0.0):
15         self.re = real # round(real, 15)
16         self.im = imaginary # round(imaginary, 15)
17
18     def __str__(self) -> str:
19         if self.im == 0.0:
20             string = f"{self.re}"
21         elif self.re == 0:
22             string = f"i({self.im})"
23         else:
24             string = f"{self.re} + i({self.im})"
25         return string
26
27     __repr__ = __str__
28
29     def __eq__(self, other) -> bool:
30         return bool(isclose(self.re, other.re) and isclose(self.im, other.im))
31
32     def is_null(self):
33         return isclose(self.re, 0) and isclose(self.im, 0)
34
35     def is_real(self):
36         return isclose(self.im, 0)
37
38     def is_imaginary(self):
39         return isclose(self.re, 0)
40
41     def arg(self):
42         """return the argument of the complex number
43         return None if 0"""
44         if self.is_null():
45             arg = None
46         elif isclose(self.re, 0) and self.im > 0:
47             arg = pi / 2
48         elif isclose(self.re, 0) and self.im < 0:
49             arg = -pi / 2
50         else:
51             arg = round(arctan2(self.im, self.re), 15)
52         return arg
53
54     def module(self):
55         """return the module of the complex number"""
56         return round(sqrt(self.re**2 + self.im**2), 15)
57
58     def conjugate(self):
59         return Complex(self.re, -self.im)
60
61     # arithmetic
62     def __add__(self, other):
63         return Complex(self.re + other.re, self.im + other.im)
64
65     def __sub__(self, other):
66         return Complex(self.re - other.re, self.im - other.im)

```

```

66     return Complex(self.re - other.re, self.im - other.im)
67
68     def __mul__(self, other):
69         real = (self.re * other.re) - (self.im * other.im)
70         imaginary = (self.re * other.im) + (self.im * other.re)
71         return Complex(real, imaginary)
72
73     def __truediv__(self, other):
74         if other.is_null():
75             raise ValueError("Error : dividing by 0")
76         elif other.is_real():
77             return Complex(self.re / other.re, self.im / other.re)
78         else:
79             denominator = (other.re**2) + (other.im**2)
80             real = ((self.re * other.re) + (self.im * other.im)) / denominator
81             imaginary = ((self.im * other.re) - (self.re * other.im)) / denominator
82             return Complex(real, imaginary)
83
84

```

```

85 Num = Union[int, float]
86
87

```

```

88 def addition(
89     *complexes: Complex,
90 ) -> (
91     Complex
92 ): # partially depreciated (can still be usefull for more iterable arguments)
93     """calculate the sum of complex numbers

```

```

94
95     parameters
96     -----

```

```

97         - *complexes : iterable type of Complex
98
99     return
100     -----

```

```

101         - sum of the complex numbers"""
102

```

```

103     res = Complex(0)
104     for number in complexes:
105         res.re += number.re
106         res.im += number.im
107     return res
108
109

```

```

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

```

```

114
115     parameters
116     -----

```

```

117         - cpx1 : Complex number
118         - cpx2 : Complex number to subtract to cpx1 (=Complex(0) by default)
119

```

```

120     return
121     -----

```

```

122         - difference of the two complex numbers"""
123

```

```

124     res = Complex()
125     res.re = cpx1.re - cpx2.re
126     res.im = cpx1.im - cpx2.im
127     return res
128

```

```

129 def product(
130     *complexes: Complex,
131 ) -> (
132     Complex

```

## Listing TIPE

```

133 ): # partially depreciated (can still be usefull for more iterable arguments)
134 """calculate the product of complex numbers
135 parameters
136 -----
137 - *complexes : iterable type of Complex
138
139
140 return
141 -----
142 - product of the complex numbers"""
143 res = Complex(1)
144 for number in complexes:
145     re = res.re * number.re - res.im * number.im
146     im = res.re * number.im + res.im * number.re
147     res.re = re
148     res.im = im
149 return res
150
151
152 def exp_to_literal(arg: float, module: float = 1.0) -> Complex:
153     """return the literal expression of a complex number defined by its argument and module
154 parameters
155 -----
156 - arg : type(float) (should be between 0 and 2pi)
157 - module : type(float) (must have a positive value) (=1 by default)
158
159
160 return
161 -----
162 - Complex number associated"""
163 assert module >= 0, "second-argument(module) must have a positive value"
164 return Complex(module * cos(arg), module * sin(arg))
165
166
167 def nth_root(n: int, cpx: Complex = Complex(1)) -> Complex:
168     """calculate the nth root of a complex number
169 parameters
170 -----
171 - n : type(int)
172 - complex : type(Complex) (=Complex(1) by default) (must not be Complex(0))
173
174
175 return
176 -----
177 - list of the nth roots"""
178 assert (
179     cpx.re != 0 or cpx.im != 0
180 ), "second argument must be a non-zero complex number"
181 module = cpx.module()
182 arg = cpx.arg()
183 if arg is not None:
184     return exp_to_literal((arg / n), module ** (1 / n))
185 else:
186     return Complex(
187         1
188     ) # Not used case but just here to ensure nth_root cannot return None
189
190
191 def nth_roots_unity(n: int) -> list:
192     """calculate the n roots of unity
193 parameter
194 -----
195 - n : type(int) : must be a positive integer
196
197
198 return
199 -----

```

## Listing TIPE

```
199
200     - a list of Complex containing the n roots of unity"""
201     roots = [Complex(1) for i in range(n)]
202     for k in range(0, n):
203         roots[k] = exp_to_literal((2 * k * pi / n), 1.0)
204     return roots
205
206
207 def inverse_nth_roots_unity(n: int) -> list:
208     """calculate the inversed n roots of unity
209
210     parameter
211     -----
212     - n : type(int) : must be a positive integer
213
214     return
215     -----
216     - a list of Complex containing the inversed n roots of unity"""
217     roots = [Complex(1) for i in range(n)]
218     for k in range(0, n):
219         roots[k] = exp_to_literal((-2 * k * pi / n), 1.0)
220     return roots
221
222
223 def make_complex(values: List[Num]) -> List[Complex]:
224     res = []
225     for value in values:
226         res.extend([Complex(value)])
227     return res
228
229
230 if __name__ == "__main__":
231     pass
```

```

1  # fast-fourier transforms
2
3  from cmath import cos, exp, pi
4
5  import complex as cpx
6  from numpy import log2
7
8
9  def FFT(vector: list) -> list:
10     """calculate the fast fourier tranform of a vector
11
12     parameters
13     -----
14     -vector : list of Complex object
15
16     return
17     -----
18     - 1-D fast fourier transform of the vector"""
19     n = len(vector)
20     assert log2(
21         n
22     ).is_integer(), "make sure that the length of the argument is a power of 2"
23     if n == 1:
24         return vector
25     poly_even, poly_odd = vector[::2], vector[1::2]
26     res_even, res_odd = FFT(poly_even), FFT(poly_odd)
27     res = [cpx.Complex(0)] * n
28     for j in range(n // 2):
29         w_j = cpx.exp_to_literal(-2 * pi * j / n)
30         product = w_j * res_odd[j]
31         res[j] = res_even[j] + product
32         res[j + n // 2] = res_even[j] - product
33     return res
34
35
36 def IFFT_aux(vector: list) -> list:
37     """auxiliary function that makes the recursive steps of the IFFT algorithm
38
39     parameters
40     -----
41     -vector : list of Complex object
42
43     return
44     -----
45     - partial inverse of the 1-D fast fourier transform of the vector (lack the division by n)
46     """
47     n = len(vector)
48     assert log2(
49         n
50     ).is_integer(), "make sure that the length of the argument is a power of 2"
51     if n == 1:
52         return vector
53     poly_even, poly_odd = vector[::2], vector[1::2]
54     res_even, res_odd = IFFT_aux(poly_even), IFFT_aux(poly_odd)
55     res = [cpx.Complex(0)] * n
56     for j in range(n // 2):
57         w_j = cpx.exp_to_literal((2 * pi * j) / n)
58         product = w_j * res_odd[j]
59         res[j] = res_even[j] + product
60         res[j + n // 2] = res_even[j] - product
61     return res
62
63 def IFFT(vector: list) -> list:
64     """caclulate the inverse of the fast fourier tranform of a vector (in order to have ifft(fft(poly)) == poly)
65
66     parameters
67     -----
68     -vector : list of Complex object
69
70     return
71     -----
72     - inverse of the 1-D fast fourier transform of the vector"""
73     n = len(vector)
74     res = IFFT_aux(vector)
75     for i in range(n):
76         res[i] = res[i] / cpx.Complex(n)
77     return res

```

```

1 import time as t
2
3 import matplotlib.animation as anim
4 import matplotlib.pyplot as plt
5 import numpy as np
6
7 DPI = 100
8
9
10 def create_movie(
11     X, evolve, path, steps=100, cmap=None, interpolation="bicubic", interval=50
12 ):
13
14     print(f"Rendering {path}")
15     time = t.time()
16     if len(X.shape) == 2 and cmap is None:
17         cmap = "gray_r"
18
19     fig = plt.figure(figsize=(16, 12))
20     im = plt.imshow(X, cmap=cmap, interpolation=interpolation, vmin=0, vmax=1)
21     plt.axis("off")
22
23     def update(i):
24
25         if i % (steps // 10) == 0:
26             print(f"Step {i}/{steps}")
27
28         if i == 0:
29             return (im,)
30         nonlocal X
31         X = evolve(X)
32         im.set_array(X)
33         return (im,)
34
35     ani = anim.FuncAnimation(fig, update, steps, interval=interval, blit=True)
36     ani.save(path, fps=25, dpi=DPI)
37     time = t.time() - time
38     print(f"Done in {time//60}min{time%60}s")
39
40
41 def create_movie_multi(Xs, evolve, path, steps=100, interpolation="bicubic"):
42
43     fig = plt.figure(figsize=(16, 9))
44     im = plt.imshow(np.dstack(Xs), interpolation=interpolation)
45     plt.axis("off")
46
47     def update(i):
48
49         if i % (steps // 10) == 0:
50             print(f"Step {i}/{steps}")
51
52         if i == 0:
53             return (im,)
54         nonlocal Xs
55         Xs = evolve(Xs)
56         im.set_array(np.dstack(Xs))
57         return (im,)
58
59     ani = anim.FuncAnimation(fig, update, steps, interval=50, blit=True)
60     ani.save(path, fps=25, dpi=DPI)

```

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import scipy as sp
4 from movie import *
5
6 import species
7
8 # Utils
9
10 path_simul = (
11     "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/simul/" # absolute path ! careful !
12 )
13 path_graphs = "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/doc/slideshow/img/"
14
15
16 def gauss(x, mu: float, sigma: float):
17     """Return non-normalized gaussian function of expected value mu and
18     variance sigma ** 2"""
19     return np.exp(-0.5 * ((x - mu) / sigma) ** 2)
20
21
22 def polynomial(x, alpha: int):
23     return (4 * x * (1 - x)) ** alpha
24
25
26 # Game of life (GoL)
27
28
29 def evolution_gol(grid):
30     # count the neighbor considering a periodic grid (wrapped around its border)
31     neighbor_count = sum(
32         np.roll(np.roll(grid, i, 0), j, 1)
33         for i in (-1, 0, 1)
34         for j in (-1, 0, 1)
35         if (i != 0 or j != 0)
36     )
37     return (neighbor_count == 3) | (grid & (neighbor_count == 2))
38
39
40 # simulation test
41 grid = np.random.randint(0, 2, (64, 64))
42 # create_movie(
43 #     grid,
44 #     evolution_gol,
45 #     path_simul + "gol_simul.mp4",
46 #     200,
47 #     cmap="plasma",
48 #     interpolation="none",
49 #     interval=200,
50 # )
51
52 # GoL with continuous kernel and growth
53
54 kernel_gol = np.array([[1, 1, 1], [1, 0, 1], [1, 1, 1]])
55
56
57 def growth_gol(neighbor_val):
58     cond1 = (neighbor_val >= 1) & (neighbor_val <= 3)
59     cond2 = (neighbor_val > 3) & (neighbor_val <= 4)
60     return -1 + (neighbor_val - 1) * cond1 + 8 * (1 - neighbor_val / 4) * cond2
61
62
63 def evolution_continuous_gol(grid):
64     neighbor_count = sp.signal.convolve2d(
65         grid, kernel_gol, mode="same", boundary="wrap"
66     )
67     grid = grid + growth_gol(neighbor_count)

```

```

68     grid = np.clip(grid, 0, 1)
69     return grid
70
71
72 # simulation test
73 grid = np.random.randint(0, 2, (64, 64))
74 # create_movie(
75 #     grid,
76 #     evolution_continuous_gol,
77 #     path_simul + "gol_continuous_simul.mp4",
78 #     200,
79 #     cmap="plasma",
80 #     interpolation="none",
81 #     interval=200,
82 # )
83
84 # Lenia
85
86 scale = 1 # scaling factor to speed up rendering when testing
87
88 # Ring filter
89
90 R = 13 # radius of kernel
91 x, y = np.ogrid[-R:R, -R:R] # grid
92 dist_norm = ((1 + x) ** 2 + (1 + y) ** 2) ** 0.5 / R # normalized so that dist(R) = 1
93
94 gamma = 0.5
95 delta = 0.15
96 kernel_shell = (dist_norm <= 1) * gauss(
97     dist_norm, gamma, delta
98 ) # we don't consider neighbor having dist > 1
99 kernel_shell = kernel_shell / np.sum(kernel_shell) # normalizing values
100
101 # show ring
102 plt.figure(figsize=(20, 10))
103 plt.subplot(121)
104 plt.imshow(dist_norm, interpolation="none", cmap="plasma")
105 plt.subplot(122)
106 plt.imshow(kernel_shell, interpolation="none", cmap="plasma")
107 plt.savefig(path_simul + "ring_kernel.png")
108
109 # Growth function
110
111
112 def growth_lenia(region):
113     mu = 0.15
114     sigma = 0.015
115     return -1 + 2 * gauss(region, mu, sigma)
116
117
118 # Evolve function
119
120 dt = 0.1 # set the time step
121
122
123 def evolution_lenia(grid):
124     neighbor = sp.signal.convolve2d(grid, kernel_shell, mode="same", boundary="wrap")
125     grid = grid + dt * growth_lenia(neighbor)
126     grid = np.clip(grid, 0, 1)
127     return grid
128
129
130 # simulation test
131 size = int(256 * scale)
132 mid = size // 2
133 grid = np.ones((size, size))
134
135 # gaussian spot initialization

```



```

136 radius = int(36 * scale)
137 y, x = np.ogrid[-mid:mid, -mid:mid]
138 grid = np.exp(-0.5 * (x**2 + y**2) / radius**2)
139
140 # create_movie(grid, evolution_lenia, path_simul + "lenia_spot.mp4", 700, cmap="plasma")
141
142
143 # Graphs
144
145 # basic example
146
147 # random initialization
148 grid = np.random.rand(size, size)
149
150
151 def plot_basic_lenia():
152     global grid
153     fig, ax = plt.subplots(3, 4)
154     step = 500
155     plotted_steps = [0, 1, 3, 5, 9, 15, 30, 60, 90, 125, 200, 500]
156     k = 0
157     i, j = 0, 0
158     while k <= step:
159         if k in plotted_steps:
160             ax[i, j].imshow(grid, cmap="plasma")
161             ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
162             ax[i, j].axis("off")
163             if j == 3:
164                 i += 1
165                 j = 0
166             else:
167                 j += 1
168             grid = evolution_lenia(grid)
169             k += 1
170
171     fig.tight_layout()
172     plt.savefig(path_graphs + "evolution_lenia_random_init.png", transparent=True)
173
174
175 # plot_basic_lenia()
176
177
178 # example with perturbation
179
180 # random initialization
181 grid = np.random.rand(size, size)
182
183
184 def plot_basic_lenia_with_perturbation():
185     global grid
186     fig, ax = plt.subplots(3, 4)
187     step = 500
188     plotted_steps = [0, 15, 30, 60, 125, 199, 200, 205, 210, 220, 250, 500]
189     step_perturbation = 200
190     k = 0
191     i, j = 0, 0
192     while k <= step:
193         if k == step_perturbation:
194             for x in range(1 * len(grid) // 3, 2 * len(grid) // 3):
195                 for y in range(3 * len(grid[i]) // 6, 5 * len(grid[i]) // 6):
196                     grid[x, y] = 0
197         if k in plotted_steps:
198             ax[i, j].imshow(grid, cmap="plasma")
199             if k == step_perturbation:
200                 ax[i, j].set_title(
201                     f"t = {k} : Perturbation", color="r", fontweight="bold", fontsize=7
202                 )
203             else:

```

```

203         else:
204             ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
205             ax[i, j].axis("off")
206             if j == 3:
207                 i += 1
208                 j = 0
209             else:
210                 j += 1
211
212         grid = evolution_lenia(grid)
213         k += 1
214
215     fig.tight_layout()
216     plt.savefig(
217         path_graphs + "evolution_lenia_random_init_perturbation.png", transparent=True
218     )
219
220
221 # plot_basic_lenia_with_perturbation()
222
223
224 # random initialization
225 grid = np.random.rand(size, size)
226
227 # create_movie(
228 #     grid,
229 #     evolution_lenia,
230 #     path_simul + "lenia_random.mp4",
231 #     300,
232 #     cmap="plasma",
233 #     interpolation="none",)
234
235 # Orbium (gol's glider "equivalent")
236
237 orbium = species.orbium
238
239 plt.imshow(orbium.T, cmap="plasma")
240 plt.savefig(path_simul + "orbium.png")
241 plt.savefig(path_graphs + "orbium.png")
242
243 size = 128
244 grid = np.zeros((size, size))
245 pos = size // 6
246 grid[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T
247
248
249 def plot_orbium():
250
251     size = 64
252     grid_basic = np.zeros((size, size))
253     pos = size // 6
254     grid_basic[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T
255     grid_perturbation = grid_basic.copy()
256
257     fig, ax = plt.subplots(2, 6, figsize=(12, 5))
258     fontsize = 10
259
260     step = 100
261     plotted_steps = [0, 25, 50, 75, 100]
262     step_perturbation = 50
263
264     ax[0, 0].imshow(orbium, cmap="plasma", interpolation="bicubic", vmin=0, vmax=1)
265     ax[0, 0].set_title("Forme", fontsize="x-large")
266     ax[0, 0].axis("off")
267     k = 0
268     j1, j2 = 1, 0
269     while k <= step:
270         if k == step_perturbation:

```

## Listing TIPE

```

271         for x in range(len(grid_perturbation)):
272             for y in range(len(grid_perturbation[x])):
273                 if grid_perturbation[x, y] > 0:
274                     v = np.random.choice(
275                         [0, grid_perturbation[x, y]], p=[1 / 3, 1 - 1 / 3]
276                     )
277                     grid_perturbation[x, y] = v
278     if k in plotted_steps:
279         ax[0, j1].imshow(grid_basic, cmap="plasma")
280         ax[1, j2].imshow(grid_perturbation, cmap="plasma")
281         ax[0, j1].axis("off")
282         ax[1, j2].axis("off")
283         ax[0, j1].set_title(f"t = {k}", fontweight="bold", fontsize=fontsize)
284         if k == step_perturbation:
285             ax[1, j2].set_title(
286                 f"t = {k} : Perturbation",
287                 color="r",
288                 fontweight="bold",
289                 fontsize=fontsize,
290             )
291         else:
292             ax[1, j2].set_title(f"t = {k}", fontweight="bold", fontsize=fontsize)
293         j1 += 1
294         j2 += 1
295
296     if k == step_perturbation - 1:
297         ax[1, j2].imshow(grid_perturbation, cmap="plasma")
298         ax[1, j2].axis("off")
299         ax[1, j2].set_title(f"t = {k}", fontweight="bold", fontsize=7)
300         j2 += 1
301
302     grid_basic = evolution_lenia(grid_basic)
303     if k < step_perturbation:
304         grid_perturbation = grid_basic.copy()
305     else:
306         grid_perturbation = evolution_lenia(grid_perturbation)
307     k += 1
308
309     fig.tight_layout()
310     plt.savefig(path_graphs + "evolution_orbium.png", transparent=True)
311     plt.clf()
312
313
314 # plot_orbium()
315
316
317 # create_movie(
318 #     grid, evolution_lenia, path_simul + "lenia_orbium.mp4", 100, cmap="plasma", interval=50
319 # )
320
321 # Lenia optimization with fft
322
323 size = 128
324 mid = size // 2
325 grid = np.zeros((size, size))
326 pos = size // 6
327 grid[pos : (pos + orbium.shape[1]), pos : (pos + orbium.shape[0])] = orbium.T
328
329 # redefine kernel to meet fft's requirements
330
331 R = 13
332 x, y = np.ogrid[-mid:mid, -mid:mid] # grid
333 dist_norm = (((x**2 + y**2) ** 0.5)) / R # normalized so that dist(R) = 1
334 kernel_shell = (dist_norm <= 1) * gauss(dist_norm, 0.5, 0.15)
335 kernel_shell = kernel_shell / np.sum(kernel_shell)
336 f_kernel = sp.fft.fft2(sp.fft.fftfshift(kernel_shell)) # fft of kernel
337 # show ring fft
338 plt.figure(figsize=(20, 10))

```

```

339 plt.subplot(121)
340 plt.imshow(dist_norm, interpolation="none", cmap="plasma")
341 plt.subplot(122)
342 plt.imshow(kernel_shell, interpolation="none", cmap="plasma")
343 plt.savefig(path_simul + "ring_kernel_fft.png")
344 plt.clf()
345
346
347 def evolution_lenia_fft(grid):
348     neighbor = np.real(sp.fft.ifft2(f_kernel * sp.fft.fft2(grid)))
349     grid = np.clip(grid + dt * growth_lenia(neighbor), 0, 1)
350     return grid
351
352
353 # create_movie(
354 #     grid,
355 #     evolution_lenia_fft,
356 #     path_simul + "lenia_orbium_fft.mp4",
357 #     500,
358 #     cmap="plasma",
359 #     interval=50,
360 # )
361
362
363 # Multi Kernel
364
365 size = 128
366 mid = size // 2
367 x, y = np.ogrid[-mid:mid, -mid:mid]
368 R = 36
369 amplitude = [1, 0.667, 0.333, 0.667]
370 dist_norm = (x**2 + y**2) ** 0.5 / R * len(amplitude)
371
372 kernel_multi_quadrium = np.zeros_like(dist_norm)
373 alpha = 4
374 for i in range(len(amplitude)):
375     kernel_multi_quadrium += (
376         (dist_norm.astype(int) == i) * amplitude[i] * polynomial(dist_norm % 1, alpha)
377     )
378 kernel_multi_quadrium /= np.sum(kernel_multi_quadrium)
379 f_kernel = sp.fft.fft2(sp.fft.fftshift(kernel_multi_quadrium)) # fft of kernel
380
381
382 def growth_quadrium_lenia(region):
383     mu = 0.16
384     sigma = 0.01
385     return 2 * gauss(region, mu, sigma) - 1
386
387
388 def evolution_quadrium_fft(grid):
389     neighbor = np.real(sp.fft.ifft2(f_kernel * sp.fft.fft2(grid)))
390     grid = np.clip(grid + dt * growth_quadrium_lenia(neighbor), 0, 1)
391     return grid
392
393
394 grid = np.zeros((size, size))
395 quadrium = species.quadrium
396 pos = size // 10
397 grid[size - quadrium.shape[0] - pos : size - pos, 0 : quadrium.shape[1]] = quadrium
398
399 # create_movie(
400 #     grid,
401 #     evolution_quadrium_fft,
402 #     path_simul + "quadrium_simul.mp4",
403 #     200,
404 #     cmap="plasma",
405 #     interpolation="none",
406 #     interval=200,

```

```

407 # )
408
409
410 def plot_quadrium():
411     global grid
412     fig, ax = plt.subplots(2, 4)
413     step = 500
414     plotted_steps = [0, 50, 100, 200, 300, 400, 500]
415     k = 0
416     i, j = 0, 1
417     ax[0, 0].imshow(quadrium, cmap="plasma")
418     ax[0, 0].axis("off")
419     ax[0, 0].set_title("Forme", fontsize="x-large")
420     while k <= step:
421         if k in plotted_steps:
422             ax[i, j].imshow(grid, cmap="plasma")
423             ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
424             ax[i, j].axis("off")
425             if j == 3:
426                 i += 1
427                 j = 0
428             else:
429                 j += 1
430             grid = evolution_quadrium_fft(grid)
431             k += 1
432
433     fig.tight_layout()
434     plt.savefig(path_graphs + "evolution_quadrium.png", transparent=True)
435
436
437 # plot_quadrium()
438
439 # Multi-channel Lenia
440
441 kernels_table = species.aquarium["kernels"]
442
443 betas = [k["b"] for k in kernels_table]
444 mus = [k["m"] for k in kernels_table]
445 sigmas = [k["s"] for k in kernels_table]
446 heights = [k["h"] for k in kernels_table]
447 radii = [k["r"] for k in kernels_table]
448 sources = [k["c0"] for k in kernels_table]
449 destinations = [k["c1"] for k in kernels_table]
450
451 gamma = 0.5
452 delta = 0.15
453
454 dt = 0.5
455 R = 12
456 size = 128
457 mid = size // 2
458 x, y = np.ogrid[-mid:mid, -mid:mid]
459
460
461 kernel_shells = []
462 kernels_fft = []
463
464 fig, ax = plt.subplots(1)
465
466 for b, r in zip(betas, radii):
467     r *= R
468     dist_norm = (x**2 + y**2) ** 0.5 / r * len(b)
469     kernel_shell = np.zeros_like(dist_norm)
470     for i in range(len(b)):
471         mask_norm = dist_norm.astype(int) == i
472         kernel_shell += mask_norm * b[i] * gauss(dist_norm % 1, gamma, delta)
473     kernel_shell /= kernel_shell.sum()
474     kernel_shells.append(kernel_shell)

```

```

475     kernels_fft.append(sp.fft.fft2(sp.fft.fftshift(kernel_shell)))
476
477 colors = {0: "r", 1: "g", 2: "b"}
478 for i, k in enumerate(kernel_shells):
479     ax.plot(k[size // 2, :], color=colors[sources[i]], label=f"canal {sources[i]}")
480     ax.xaxis.set_visible(False)
481
482 fig.tight_layout()
483 # plt.savefig(path_graphs + "plot_kernel_multi_channel.png", transparent=True)
484
485 grids = [np.zeros((size, size)) for _ in range(3)]
486
487
488 def evolution_multi_channel(grids):
489     grids_fft = [sp.fft.fft2(grid) for grid in grids]
490     potentials = [
491         np.real(np.fft.ifft2(kernel_fft * grids_fft[source]))
492         for kernel_fft, source in zip(kernels_fft, sources)
493     ]
494     growths_potential = [
495         2 * gauss(potential, mus[i], sigmas[i]) - 1
496         for i, potential in enumerate(potentials)
497     ]
498     growths = np.zeros_like(grids)
499     for destination, height, growth in zip(destinations, heights, growths_potential):
500         growths[destination] += height * growth
501     grids = [np.clip(grid + dt * growth, 0, 1) for grid, growth in zip(grids, growths)]
502     return grids
503
504
505 aquarium = [np.array(species.aquarium["cells"][c]) for c in range(3)]
506
507 for c in range(3):
508     grids[c][mid : mid + aquarium[c].shape[0], mid : mid + aquarium[c].shape[1]] = (
509         aquarium[c]
510     )
511
512
513 def plot_aquarium():
514     global grids
515     fig, ax = plt.subplots(3, 4)
516     step = 600
517     plotted_steps = [0, 10, 20, 50, 100, 150, 200, 300, 400, 500, 600]
518     k = 0
519     i, j = 0, 1
520     ax[0, 0].imshow(np.dstack(aquarium))
521     ax[0, 0].axis("off")
522     ax[0, 0].set_title("Forme", fontsize="x-large")
523     while k <= step:
524         if k in plotted_steps:
525             ax[i, j].imshow(np.dstack(grids))
526             ax[i, j].set_title(f"t = {k}", fontweight="bold", fontsize=7)
527             ax[i, j].axis("off")
528             if j == 3:
529                 i += 1
530                 j = 0
531             else:
532                 j += 1
533             grids = evolution_multi_channel(grids)
534             k += 1
535
536     fig.tight_layout()
537     plt.savefig(path_graphs + "evolution_aquarium.png", transparent=True)
538
539
540 # plot_aquarium()

```

```

1 import torch
2 from gymnasium import Dict
3
4 # This code is part of bigger system, that I haven't code myself,
5 # here is only the imgep algorithm run, that I did code myself (inspired ofc
6 # from the work made by INRIA bordeaux)
7
8
9 def execute_imgep_exploration(self, exploration_runs, resume_existing_run=False):
10     retry = True
11
12     while retry:
13         print("STARTING NEW INITIALIZATION")
14         print("Exploration: ")
15
16         if resume_existing_run:
17             current_run = len(self.policy_archive)
18         else:
19             self.policy_archive = []
20             self.goal_archive = torch.empty((0,) + self.goal_space.shape)
21             current_run = 0
22
23         alive_randoms = 0
24
25         while current_run < exploration_runs:
26             policy_params = Dict.fromkeys(["init_state", "update_strategy"])
27
28             if len(self.policy_archive) < self.config.initial_random_runs:
29                 target = None
30                 selected_policy = None
31                 goal_achieved = torch.ones(19)
32
33                 policy_params["init_state"] = self.system.init_space.sample()
34                 policy_params["update_strategy"] = self.system.strategy_space.sample()
35                 policy_params["update_strategy"].h /= 3
36
37                 self.system.reset(
38                     initialization_parameters=policy_params["init_state"],
39                     update_rule_parameters=policy_params["update_strategy"],
40                 )
41
42                 with torch.no_grad():
43                     self.system.random_obstacle(8)
44                     self.system.generate_init_state()
45                     results = self.system.run()
46                     goal_achieved = self.goal_space.map(results)
47
48                 is_failed = goal_achieved[0] > 0.9 or goal_achieved[1] < -0.5
49                 if not is_failed:
50                     alive_randoms += 1
51
52                 optimization_steps = 0
53                 distance_to_goal = None
54
55             else:
56                 if len(self.policy_archive) - self.config.initial_random_runs < 8:
57                     target = torch.ones(3) * -10
58                     target[0] = 0.065
59                     target[2] = (
60                         0.19
61                         - (len(self.policy_archive) - self.config.initial_random_runs)
62                         * 0.06
63                     )
64                     target[1] = 0
65                 else:

```

## Listing TIPE

```

66         target = self.sample_interesting_goal()
67
68     if len(self.policy_archive) - self.config.initial_random_runs >= 2:
69         print(f"Run {current_run}, optimizing towards goal: ")
70         print("TARGET =", str(target))
71
72     selected_policy_idx = self.find_source_policy(target)
73     selected_policy = self.policy_archive[selected_policy_idx]
74
75     if (
76         len(self.policy_archive) - self.config.initial_random_runs < 8
77         or len(self.policy_archive) % 5 == 0
78     ):
79         policy_params["init_state"] = deepcopy(
80             selected_policy["init_state"]
81         )
82         policy_params["update_strategy"] = deepcopy(
83             selected_policy["update_strategy"]
84         )
85         self.system.reset(
86             initialization_parameters=policy_params["init_state"],
87             update_rule_parameters=policy_params["update_strategy"],
88         )
89         iterations = self.config.goal_optimizer.steps
90     else:
91         iterations = 15
92         mutation_failed = True
93         while mutation_failed:
94             policy_params["init_state"] = self.system.init_space.mutate(
95                 selected_policy["init_state"]
96             )
97             policy_params["update_strategy"] = (
98                 self.system.strategy_space.mutate(
99                     selected_policy["update_strategy"]
100                 )
101             )
102             self.system.reset(
103                 initialization_parameters=policy_params["init_state"],
104                 update_rule_parameters=policy_params["update_strategy"],
105             )
106             with torch.no_grad():
107                 self.system.generate_init_state()
108                 results = self.system.run()
109                 goal_achieved = self.goal_space.map(results)
110
111             if (
112                 results.states[-1, :, :, 0].sum() > 10
113                 or goal_achieved[0] > 0.11
114             ):
115                 mutation_failed = False
116
117     if (
118         isinstance(self.system, torch.nn.Module)
119         and self.config.goal_optimizer.steps > 0
120     ):
121         optimizer_class = eval(
122             f"torch.optim.{self.config.goal_optimizer.name}"
123         )
124         self.goal_optimizer = optimizer_class(
125             [
126                 {
127                     "params": self.system.init_state.parameters(),
128                     **self.config.goal_optimizer.init_cppn.parameters,
129                 },
130                 {
131                     "params": self.system.step.parameters(),

```



## Listing TIPE

```

132         **self.config.goal_optimizer.step.parameters,
133     },
134 ],
135     **self.config.goal_optimizer.parameters,
136 )
137
138 last_failed = False
139 for optimization_steps in range(1, iterations):
140     self.system.random_obstacle(8)
141     self.system.generate_init_state()
142     results = self.system.run()
143     goal_achieved = self.goal_space.map(results)
144
145     x = torch.arange(self.system.config.SX)
146     y = torch.arange(self.system.config.SY)
147     xx = x.view(-1, 1).repeat(1, self.system.config.SY)
148     yy = y.repeat(self.system.config.SX, 1)
149     X = (
150         xx - (target[1] + 0.5) * self.system.config.SX
151     ).float() / 35
152     Y = (
153         yy - (target[2] + 0.5) * self.system.config.SY
154     ).float() / 35
155     D = torch.sqrt(X**2 + Y**2)
156     mask = 0.85 * (D < 0.5).float() + 0.15 * (D < 1).float()
157
158     loss = (
159         (0.9 * mask - results.states[-1, :, :, 0])
160         .pow(2)
161         .sum()
162         .sqrt()
163     )
164
165     self.goal_optimizer.zero_grad()
166     loss.backward()
167     self.goal_optimizer.step()
168
169     self.system.step.compute_kernel()
170
171     failed = results.states[-1, :, :, 0].sum() < 10
172     if failed and last_failed:
173         self.goal_optimizer.zero_grad()
174         break
175     last_failed = failed
176
177 if (
178     len(self.policy_archive) >= self.config.initial_random_runs
179     and len(self.policy_archive) - self.config.initial_random_runs
180     < 2
181 ):
182     if loss > 19.5:
183         break
184     elif (
185         len(self.policy_archive) - self.config.initial_random_runs
186         == 2
187     ):
188         retry = False
189
190 self.system.update_initialization_parameters()
191 self.system.update_update_rule_parameters()
192 policy_params["init_state"] = self.system.initialization_parameters
193 policy_params["update_strategy"] = (
194     self.system.update_rule_parameters
195 )
196 distance_to_goal = loss.item()
197

```

## Listing TIPE

```

198     goal_achieved = torch.zeros(3).cpu()
199     with torch.no_grad():
200         for _ in range(20):
201             self.system.random_obstacle(8)
202             self.system.generate_init_state()
203             results = self.system.run()
204             if results.states[-1, :, :, 0].sum() < 10:
205                 goal_achieved[0] = 10
206                 break
207             goal_achieved = (
208                 goal_achieved + self.goal_space.map(results).cpu() / 20
209             )
210
211         if len(self.policy_archive) - self.config.initial_random_runs >= 2:
212             print("reached=", str(goal_achieved))
213
214     goal_achieved = goal_achieved.cpu()
215     self.db.add_run_data(
216         id=current_run,
217         policy_parameters=policy_params,
218         observations=results,
219         source_policy_idx=selected_policy_idx,
220         target_goal=target,
221         reached_goal=goal_achieved,
222         n_optim_steps_to_reach_goal=optimization_steps,
223         dist_to_target=distance_to_goal,
224     )
225
226     self.policy_archive.append(policy_params)
227     self.goal_archive = torch.cat(
228         [
229             self.goal_archive,
230             goal_achieved.reshape(1, -1).to(self.goal_archive.device).detach(),
231         ]
232     )
233
234     if len(self.policy_archive) >= self.config.initial_random_runs:
235         plt.imshow(self.system.init_wall.cpu())
236         plt.scatter(
237             (
238                 (self.goal_archive[:, 0] < 0.11).float()
239                 * (self.goal_archive[:, 2] > -0.5).float()
240                 * (self.goal_archive[:, 2] + 0.5)
241                 * self.system.config.SY
242             ).cpu(),
243             (
244                 (self.goal_archive[:, 0] < 0.11).float()
245                 * (self.goal_archive[:, 1] > -0.5).float()
246                 * (self.goal_archive[:, 1] + 0.5)
247                 * self.system.config.SX
248             ).cpu(),
249         )
250         plt.show()
251
252     current_run += 1
253
254     if len(self.policy_archive) == self.config.initial_random_runs:
255         if alive_randoms < 2:
256             break
257         print(current_run)
258
259     if len(self.policy_archive) == exploration_runs - 1:
260         retry = False

```

```

1  # Rendering the different graphs used for the project
2  import os
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from matplotlib import colors
7  from matplotlib.gridspec import GridSpec, GridSpecFromSubplotSpec
8  from mpl_toolkits.axes_grid1.inset_locator import mark_inset, zoomed_inset_axes
9  from PIL import Image
10
11 path = "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/doc/slideshow/img/"
12
13
14 ## Game of Life
15
16 # evolution step
17
18 gol_cmap = colors.ListedColormap(["#960c6b", "#000066", "#cdd300"])
19 bounds = [-1.5, -0.5, 0.5, 1.5]
20 norm = colors.BoundaryNorm(bounds, gol_cmap.N)
21
22
23 def evolution_gol(grid, step=1):
24     assert step >= 0, "step argument must be >= 0"
25     evolved = grid.copy()
26     evolved = evolved.astype(int)
27     for _ in range(step):
28         # count the neighbor considering a periodic grid (wrapped around its border)
29         neighbor_count = sum(
30             np.roll(np.roll(evolved, i, 0), j, 1)
31             for i in (-1, 0, 1)
32             for j in (-1, 0, 1)
33             if (i != 0 or j != 0)
34         )
35         evolved = (neighbor_count == 3) | (evolved & (neighbor_count == 2))
36     return evolved
37
38
39 # simulation test
40 np.random.seed(69)
41 grid = np.random.choice([0, 1], (64, 64), True, p=[0.8, 0.2])
42 evolved = evolution_gol(grid)
43 grid[42, 5] = -1
44 evolved[42, 5] = -1
45
46 fig, axs = plt.subplots(1, 2, figsize=(10, 5))
47
48 axs[0].imshow(grid, interpolation="none", cmap=gol_cmap, norm=norm)
49 axs[0].tick_params(which="minor", bottom=False, left=False)
50 axs[0].invert_yaxis()
51 axs[0].set_title("Situation Initiale", fontsize=20)
52
53
54 axs[1].imshow(evolved, interpolation="none", cmap=gol_cmap, norm=norm)
55 axs[1].tick_params(which="minor", bottom=False, left=False)
56 axs[1].invert_yaxis()
57 axs[1].set_title("Après une Évolution", fontsize=20)
58
59 ax_zoom = zoomed_inset_axes(axs[0], zoom=8, loc="upper right")
60 ax_zoom.imshow(grid, cmap=gol_cmap, norm=norm)
61
62 # subregion of the original image
63 x1, x2, y1, y2 = 3.5, 6.5, 40.5, 43.5
64 ax_zoom.set_xlim(x1, x2)
65 ax_zoom.set_ylim(y1, y2) # fix the number of ticks on the inset Axes

```

## Listing TIPE

```

66 ax_zoom.yaxis.get_major_locator().set_params(nbins=4)
67 ax_zoom.xaxis.get_major_locator().set_params(nbins=4)
68 ax_zoom.set_xticks(np.arange(3.5, 7, 1), minor=True)
69 ax_zoom.set_yticks(np.arange(40.5, 44, 1), minor=True)
70
71 ax_zoom.tick_params(labelleft=False, labelbottom=False)
72 ax_zoom.grid(which="minor", color="black", linewidth=2)
73
74 # draw a bbox of the region of the inset Axes in the parent Axes and
75 # connecting lines between the bbox and the inset Axes area
76 mark_inset(
77     axs[0], ax_zoom, loc1=2, loc2=4, fc="none", ec="0.5", color="red", linewidth=3
78 )
79
80 plt.savefig(path + "plot_evolution_gol.png", transparent=True)
81 plt.clf()
82
83
84 # species
85
86 grid_block = np.zeros((6, 6))
87 coord_block = [(2, 2), (2, 3), (3, 2), (3, 3)]
88 for coord in coord_block:
89     grid_block[coord] = 1
90
91 grid_blinker = np.zeros((5, 5))
92 coord_blinker = [(2, 1), (2, 2), (2, 3)]
93 for coord in coord_blinker:
94     grid_blinker[coord] = 1
95
96 grid_glider = np.zeros((16, 16))
97 coord_glider = [(1, 1), (2, 2), (2, 3), (3, 1), (3, 2)]
98 for coord in coord_glider:
99     grid_glider[coord] = 1
100
101
102 fig = plt.figure(figsize=(10, 6))
103 outer_gs = GridSpec(
104     2,
105     2,
106     figure=fig,
107     height_ratios=[1, 1],
108     width_ratios=[1, 1.5],
109     hspace=0.1,
110     wspace=0.2,
111 )
112
113
114 def add_centered_pcolor(sub_gs, data_list, plot_titles, line_title):
115     num_plots = len(data_list)
116     ax_line = fig.add_subplot(sub_gs)
117     ax_line.text(
118         0.5,
119         1,
120         line_title,
121         ha="center",
122         va="center",
123         fontsize=20,
124         fontweight="bold",
125         transform=ax_line.transAxes,
126     )
127     ax_line.axis("off")
128
129     inner_gs = GridSpecFromSubplotSpec(1, num_plots, subplot_spec=sub_gs, wspace=0.1)
130     for i, (data, plot_title) in enumerate(zip(data_list, plot_titles)):
131         ax = fig.add_subplot(inner_gs[i])

```

## Listing TIPE

```

132     ax.pcolor(data, cmap="plasma", edgecolor="grey", linewidth=0.5)
133     ax.set_aspect("equal")
134     ax.invert_yaxis()
135     ax.set_title(plot_title, fontsize=12)
136     ax.tick_params(left=False, bottom=False, labelleft=False, labelbottom=False)
137
138
139 data_block = [grid_block, evolution_gol(grid_block)]
140 data_blinker = [evolution_gol(grid_blinker, i) for i in range(3)]
141 data_glider = [evolution_gol(grid_glider, 11 * i) for i in range(5)]
142
143 titles_block = ["t = 0", "t = 1"]
144 titles_blinker = ["t = 0", "t = 1", "t = 2"]
145 titles_glider = [f"t = {11 * i}" for i in range(5)]
146
147 plot_block = outer_gs[0, 0]
148 add_centered_pcolor(plot_block, data_block, titles_block, "Block")
149
150 plot_blinker = outer_gs[0, 1]
151 add_centered_pcolor(plot_blinker, data_blinker, titles_blinker, "Blinker")
152
153 plot_glider = outer_gs[1, :]
154 add_centered_pcolor(plot_glider, data_glider, titles_glider, "Glider")
155
156 plt.savefig(path + "plot_species_gol.png", transparent=True)
157 plt.clf()
158
159
160 # Kernels
161 def indicator(arr, lower_bound: float = 0, upper_bound: float = 1):
162     if type(arr) in ["float", "int"]:
163         return int(lower_bound <= arr <= upper_bound)
164     else:
165         arr = np.copy(arr)
166         mask = (arr >= lower_bound) & (arr <= upper_bound)
167         arr[mask] = 1
168         arr[~mask] = 0
169         return arr
170
171
172 def gauss(x, gamma: float = 0.5, delta: float = 0.15):
173     return np.exp(-0.5 * ((x - gamma) / delta) ** 2)
174
175
176 def polynomial(x, alpha: int = 4):
177     return (4 * x * (1 - x)) ** alpha
178
179
180 fig = plt.figure(figsize=(10, 13))
181
182 subfigs = fig.subfigures(1, 2)
183
184 dist_1d = np.arange(0, 1, 0.001)
185 step = 100j
186 x, y = np.ogrid[-1 : 1 : 2 * step, -1 : 1 : 2 * step] # grid
187 dist_norm = ((x) ** 2 + (y) ** 2) ** 0.5
188
189 kernel_gauss = (dist_norm <= 1) * (gauss(dist_norm))
190 kernel_polynomial = (dist_norm <= 1) * (polynomial(dist_norm))
191 kernel_rectangle = (dist_norm <= 1) * (indicator(dist_norm, 1 / 3, 2 / 3))
192
193 ax1 = subfigs[0].subplots(2, 1)
194 ax1[0].plot(dist_1d, gauss(dist_1d))
195 ax1[0].text(
196     0.5,
197     0,

```

```

198     r"$\gamma = 0.5, \delta = 0.15$",
199     fontsize=20,
200     horizontalalignment="center",
201 )
202 ax1[0].set_xlabel("Distance", fontsize="x-large")
203 im = ax1[1].imshow(kernel_gauss, interpolation="none", cmap="plasma")
204 ax1[1].axis("off")
205 subfigs[0].suptitle("Exponentiel ", fontsize=30, fontweight="bold")
206
207 ax2 = subfigs[1].subplots(2, 1)
208 ax2[0].plot(
209     dist_1d,
210     indicator(dist_1d, 2 / 4, 3 / 4),
211     label=r"$[a, b] = [\frac{2}{4}, \frac{3}{4}]$",
212 )
213 ax2[0].text(
214     0,
215     0.5,
216     r"$[a, b] = [\frac{2}{4}, \frac{3}{4}]$",
217     fontsize=20,
218 )
219 ax2[0].set_xlabel("Distance", fontsize="x-large")
220 ax2[1].imshow(kernel_rectangle, interpolation="none", cmap="plasma")
221 ax2[1].axis("off")
222 subfigs[1].suptitle("Rectangle", fontsize=30, fontweight="bold")
223
224 fig.tight_layout(pad=4, h_pad=1, w_pad=4)
225 fig.subplots_adjust(top=0.92)
226 cbar_ax = fig.add_axes((0.25, 0.05, 0.5, 0.025))
227 cbar = fig.colorbar(im, cax=cbar_ax, orientation="horizontal")
228 cbar.set_ticks([0, 1])
229 cbar.ax.tick_params(labelsize=20)
230
231
232 plt.savefig(path + "plot_convolution_kernels.png", transparent=True)
233 plt.clf()
234
235 # Growth mapping
236
237 mu1, mu2 = 0.3, 0.7
238 sigma1, sigma2 = 0.05, 0.2
239 fig, axs = plt.subplots(1, 2, figsize=(10, 5))
240 interval = np.arange(0, 1, 0.0001)
241
242 axs[0].plot(interval, 2 * gauss(interval, mu1, sigma1) - 1)
243 axs[0].set_title("Exponentielle", fontsize=20, fontweight="bold")
244 axs[0].text(
245     1,
246     0.75,
247     f"$\\mu = {mu1}, \\sigma = {sigma1}$",
248     fontsize=13,
249     horizontalalignment="right",
250 )
251
252 axs[1].plot(interval, 2 * indicator(interval, mu2 - sigma2, mu2 + sigma2) - 1)
253 axs[1].set_title("Rectangulaire", fontsize=20, fontweight="bold")
254 axs[1].text(
255     0,
256     0.75,
257     f"$\\mu = {mu2}, \\sigma = {sigma2}$",
258     fontsize=13,
259     horizontalalignment="left",
260 )
261
262 for i in [0, 1]:
263     axs[i].axhline(y=0, color="r", linestyle="--", linewidth=2, alpha=0.5)

```

```

264
265 plt.savefig(path + "plot_growth_mapping.png", transparent=True)
266 plt.clf()
267
268
269 # Multi Kernels
270
271 A = [0.3, 1, 0.7, 0.2]
272 gamma = np.random.uniform(0, 1, (len(A),))
273 delta = np.random.uniform(0, 0.3, (len(A),))
274 gamma = [0.2, 0.4, 0.6, 0.8]
275 delta = [0.015, 0.05, 0.01, 0.1]
276
277 dist_1d = np.arange(0, 1, 0.001)
278
279 step = 1000j
280 x, y = np.ogrid[-1 : 1 : 2 * step, -1 : 1 : 2 * step] # grid
281 dist_norm = ((x) ** 2 + (y) ** 2) ** 0.5
282
283 multi_kernel_core = np.zeros_like(dist_1d)
284 multi_kernel_shell = np.zeros_like(dist_norm)
285 for i in range(len(A)):
286     multi_kernel_core += A[i] * gauss(dist_1d, gamma[i], delta[i])
287     multi_kernel_shell += (dist_norm <= 1) * A[i] * gauss(dist_norm, gamma[i], delta[i])
288
289
290 fig, ax = plt.subplots(1, 2, figsize=(10, 5))
291 ax[0].plot(dist_1d, multi_kernel_core)
292 ax[0].set_xlabel("Distance", fontsize="x-large")
293 im = ax[1].imshow(multi_kernel_shell, cmap="plasma")
294 ax[1].axis("off")
295 plt.colorbar(im, ax=ax[1], cmap="plasma", location="bottom", shrink=0.7)
296
297 fig.tight_layout()
298 plt.savefig(path + "plot_multi_ring_kernel.png", transparent=True)
299
300
301 folder_path = (
302     "/Users/arsnm/Documents/cpge/mp2/tipe-mp2/simul/resultLeniaMachineLearning/"
303 )
304
305
306 def name_image(i: int):
307     s = str(i + 1)
308     while len(s) != 5:
309         s = "0" + s
310     return s + ".png"
311
312
313 steps = [0, 3, 5, 7, 10, 20, 30, 40, 50, 75, 100, 150, 200, 300, 400, 500]
314 image_names = [name_image(i) for i in steps]
315
316 num_images = len(image_names)
317 cols = 4
318 rows = (num_images + cols - 1) // cols
319
320 fig, axes = plt.subplots(rows, cols, figsize=(5 * cols, 2.5 * rows))
321 axes = axes.flatten()
322
323 i = 0
324 for i, image_name in enumerate(image_names):
325     img_path = os.path.join(folder_path, image_name)
326     img = Image.open(img_path)
327     axes[i].imshow(img)
328     axes[i].set_title(f"t = {steps[i]}", fontweight="bold", fontsize="x-large")
329     axes[i].axis("off")

```

## Listing TIPE

```
330
331 for j in range(i + 1, len(axes)):
332     axes[j].axis("off")
333
334 plt.tight_layout()
335 plt.savefig(path + "evolution_machine_learning", transparent=True)
```