

Optimisation informatique de la trajectoire d'une voiture sur un circuit



INTRODUCTION

Comment choisir les paramètres de l'algorithme PSO pour obtenir une ligne de course optimale ?

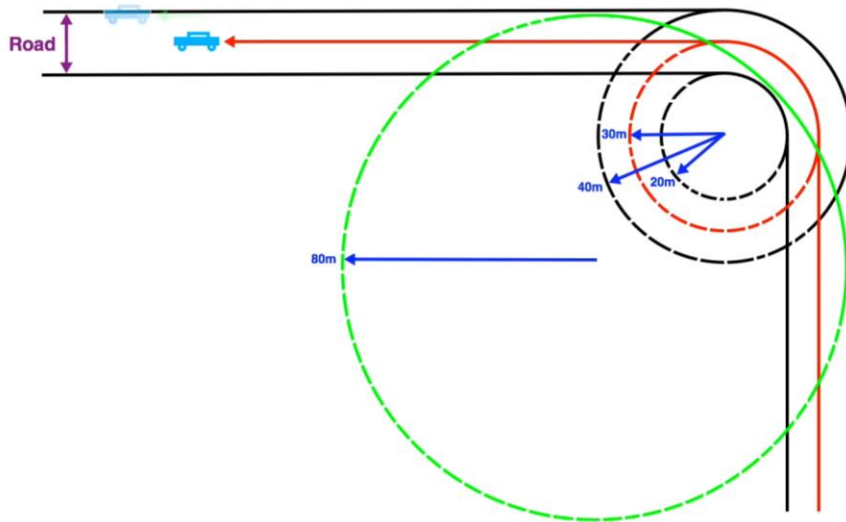


Deux demandes :

- Les données d'une ou plusieurs vraies trajectoires
- Un avis sur des trajectoires que j'ai réussi à générer

DÉFINITIONS

LA LIGNE DE COURSE



motorsport.com

MODÉLISATION

Voiture comme un point

Circuit 2D

Sur une trajectoire circulaire
uniforme on a :

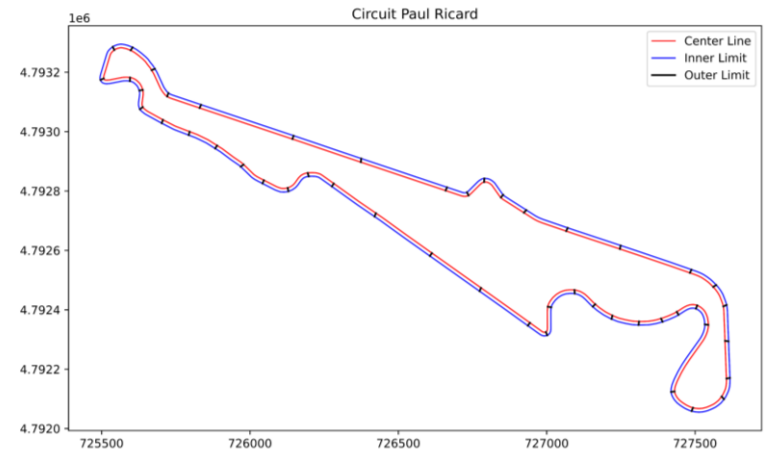
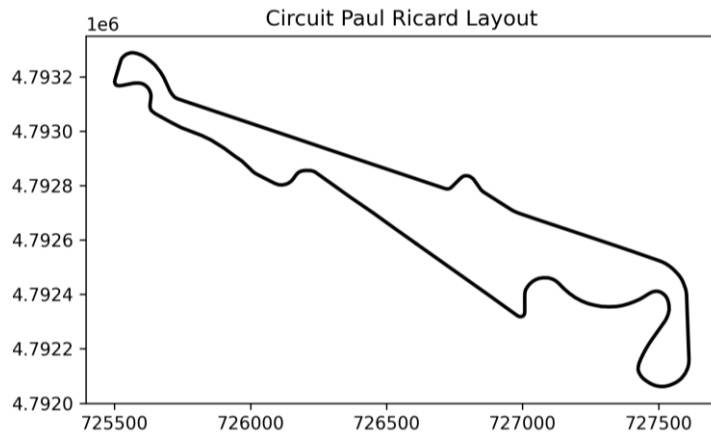
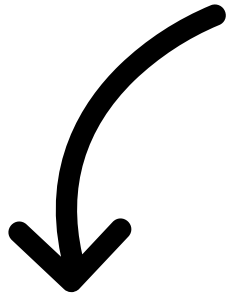
$$a = v^2/r$$

+ le rayon est grand, + on va vite

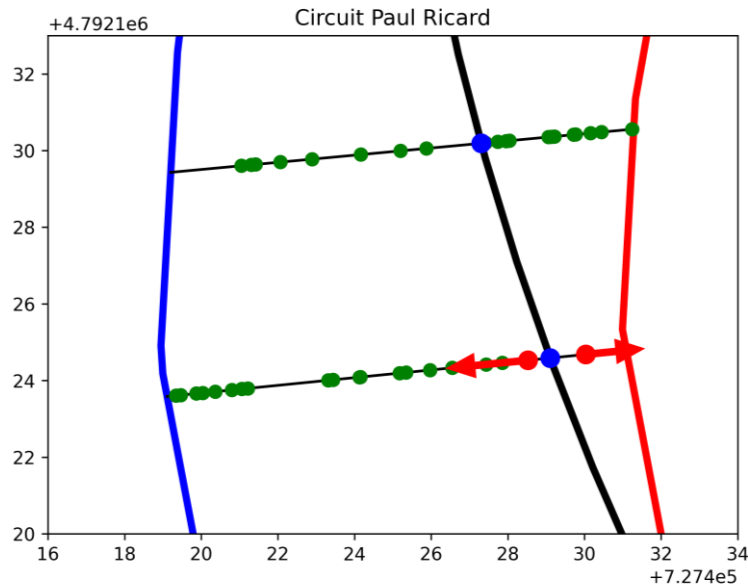
$$v_{max} = \sqrt{\mu_{track} * r * g}$$

MODÉLISATION DU CIRCUIT

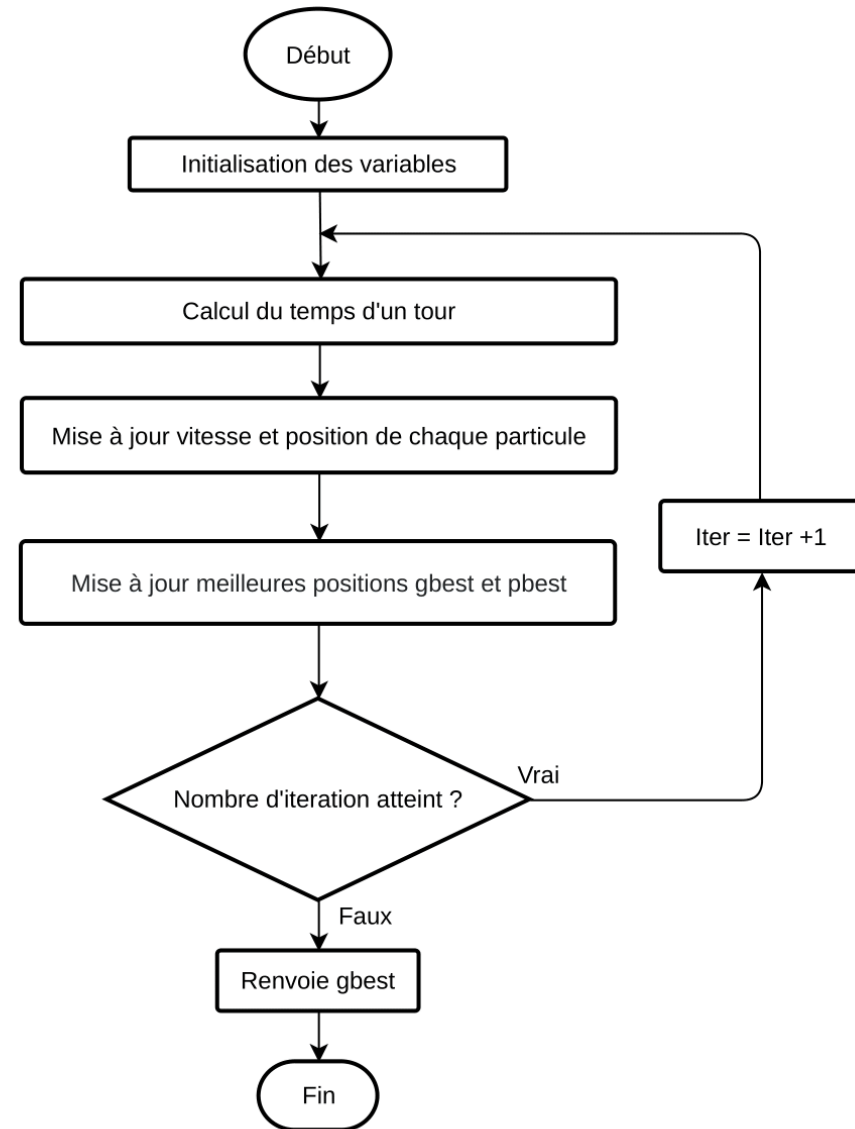
GEOJSON



PARTICLE SWARM OPTIMISATION (PSO)



Particules placées de manière uniforme le long de chaque secteur



LES PARAMÈTRES

Problème en 6 dimensions

Nombre de secteurs (fixé)

Nombre de particules

Nombre d'itérations

w : l'inertie

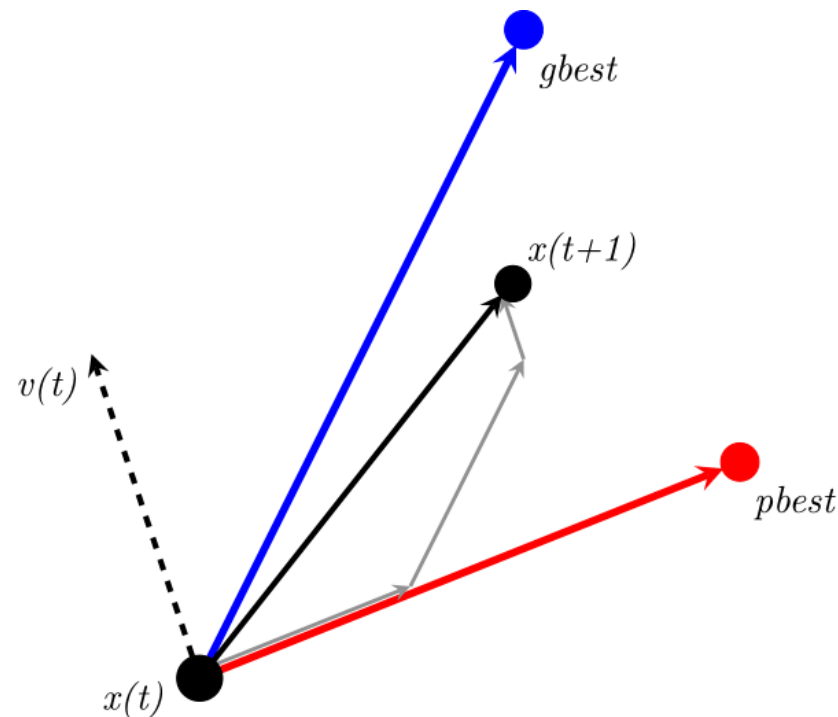
c1 : global best

c2 : personal best

MIS À JOUR DE LA VITESSE

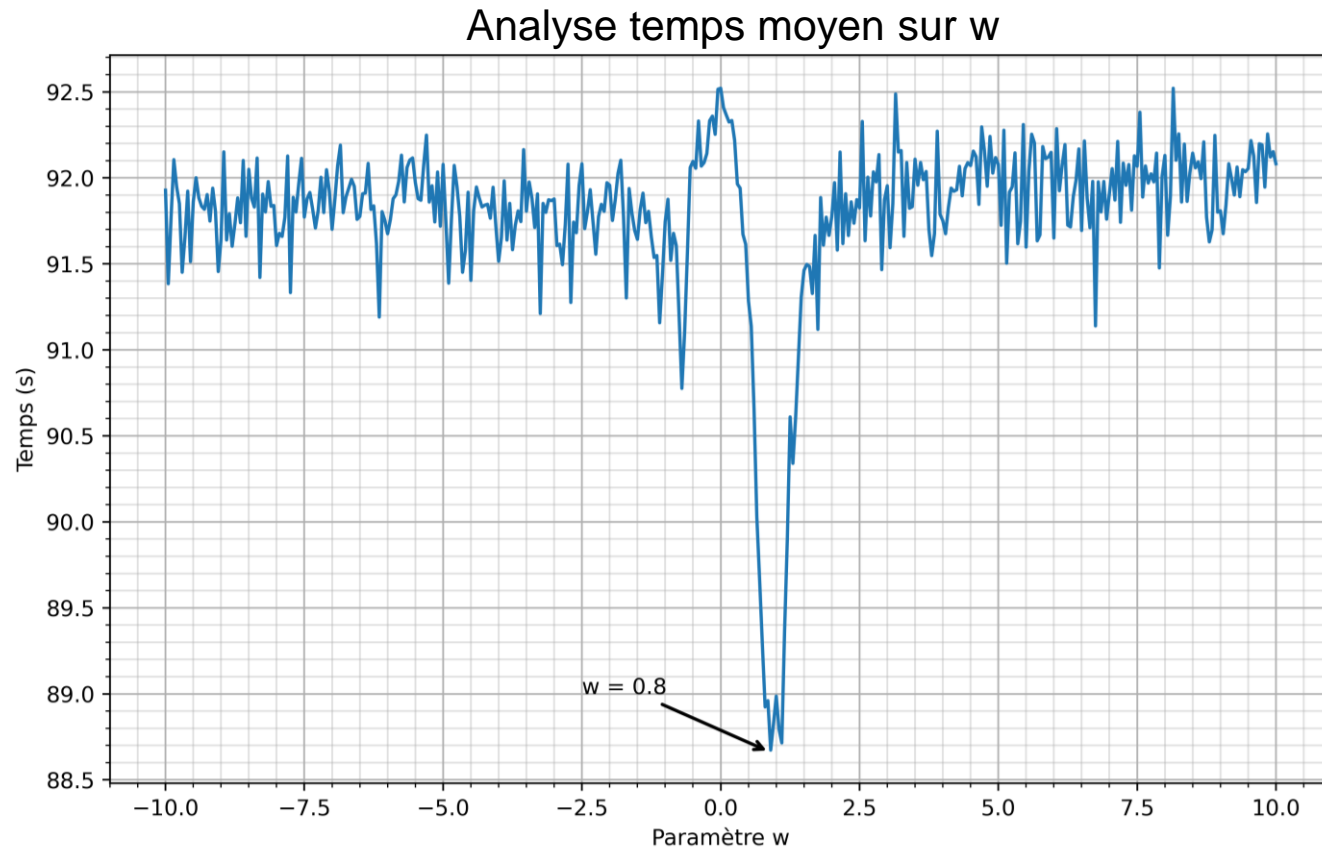
$$v(t+1) = \boxed{w} * v(t) + \boxed{c1} * r1(p_{best} - x(t)) + \boxed{c2} * r2(g_{best} - x(t)) \quad [1]$$

Inconnues à évaluer



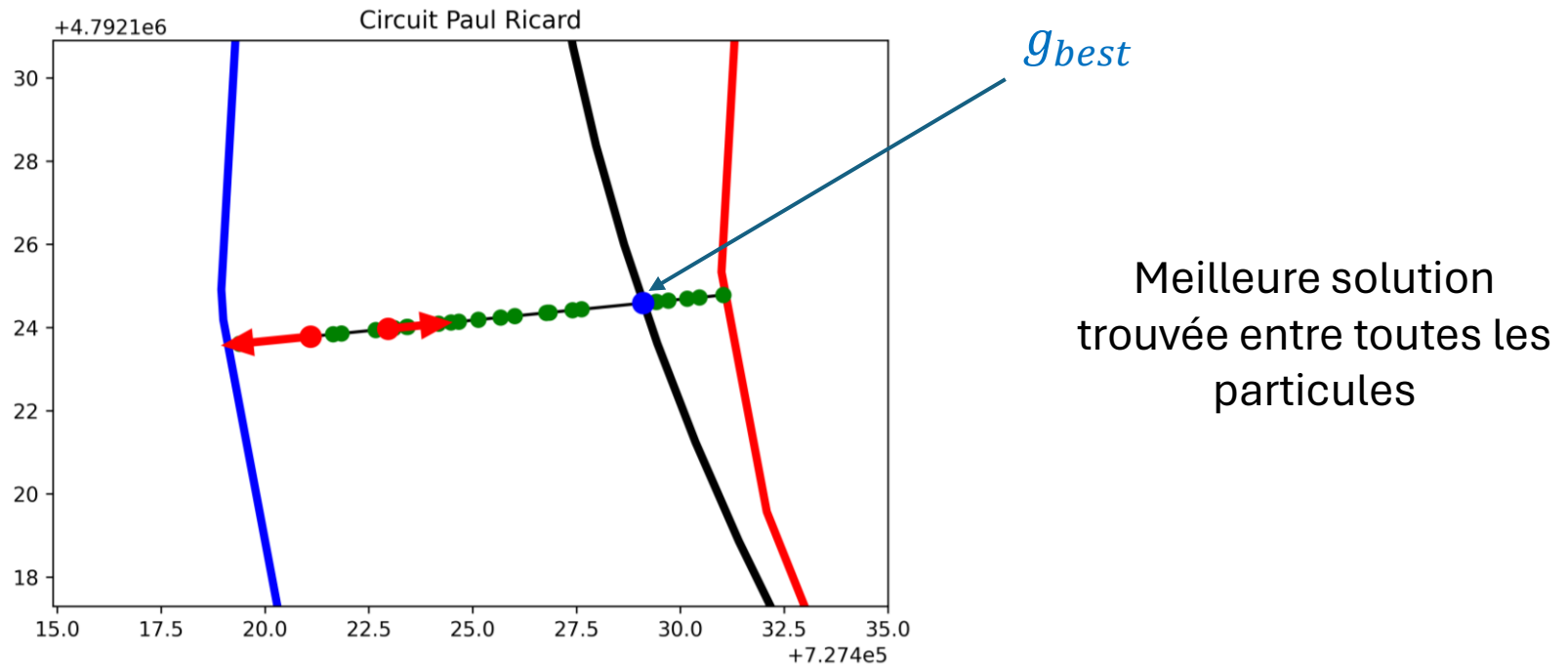
LE PARAMÈTRE D'INERTIE

$$v(t+1) = \boxed{w * v(t)} + c1 * r1(p_{best} - x(t)) + c2 * r2(g_{best} - x(t)) \quad [1]$$



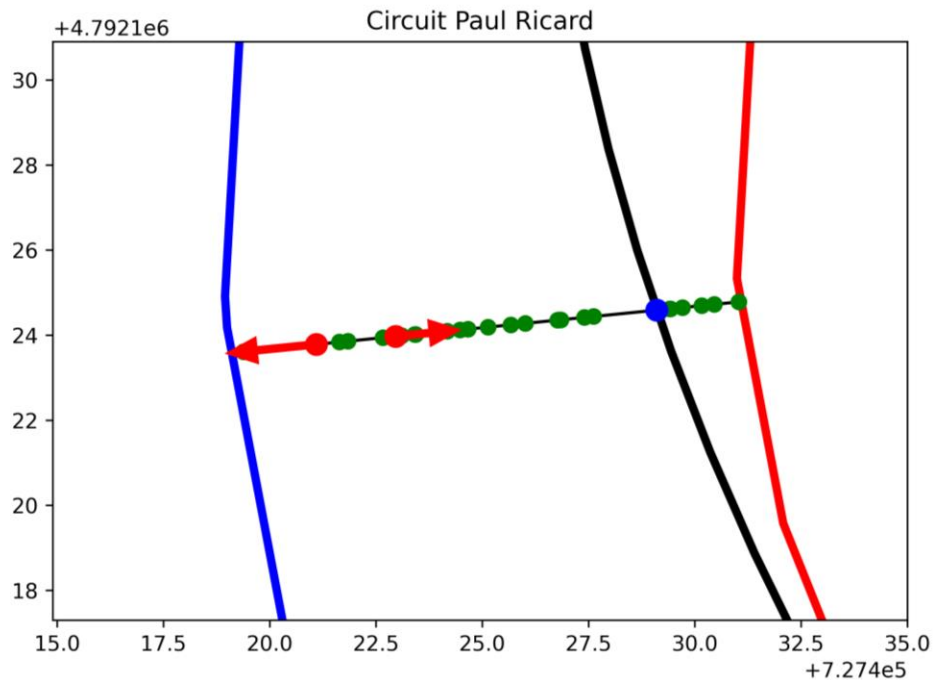
GBEST

$$v(t+1) = w * v(t) + c1 * r1(p_{best} - x(t)) + \boxed{c2 * r2(g_{best} - x(t))} \quad [1]$$



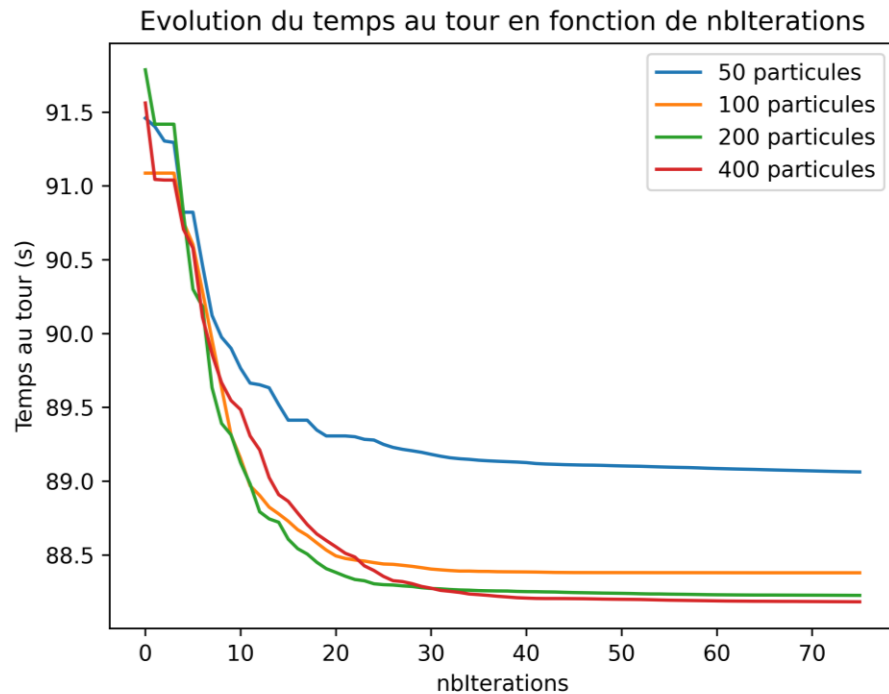
PBEST

$$v(t+1) = w * v(t) + \boxed{c1 * r1(p_{best} - x(t))} + c2 * r2(g_{best} - x(t)) \quad [1]$$



Comme g_best , mais c'est une variable personnelle propre à chaque particule

nbIterations VS nbParticules




nbIterations = 50 convient

Plus le nombre de particules est grand, plus il coûte cher en temps de calcul, bien que cela améliore le temps au tour

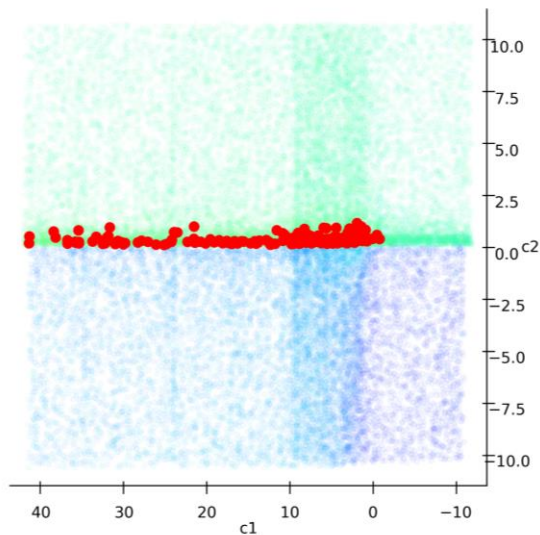
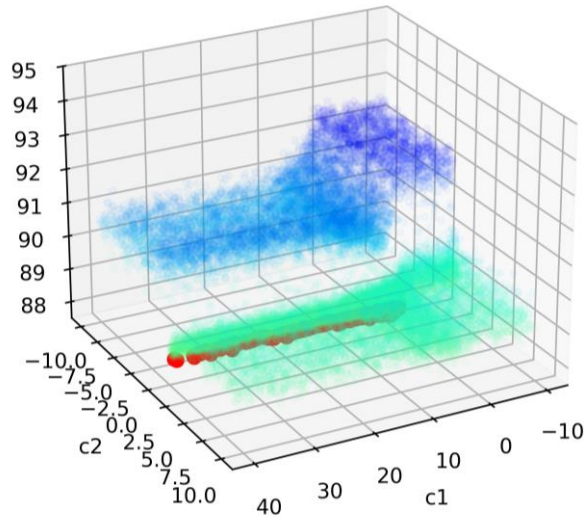
ESTIMATION DE C1 et C2

Recherche d'un minimum sur un plan

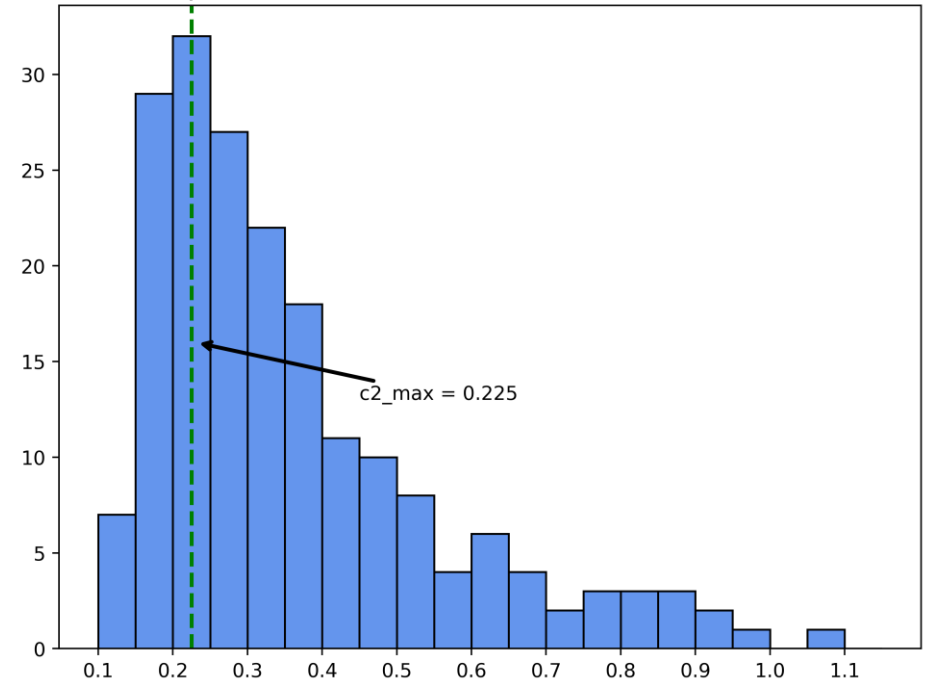
	 Recherche exhaustive	 Recuit Simulé
Temps total	6,4 ans	32h
Temps multiprocessé	876h	32h
Nombre de résultats	1	64

RÉSULTATS DES RECHERCHES

Analyse des temps sur le plan



Répartition des valeurs de c2 sur 193 résultats



RÉSULTATS PARAMÈTRES

Nombre de secteur = 50

Nombre d'itérations = 50

Nombre de particules
dépend du temps de calcul

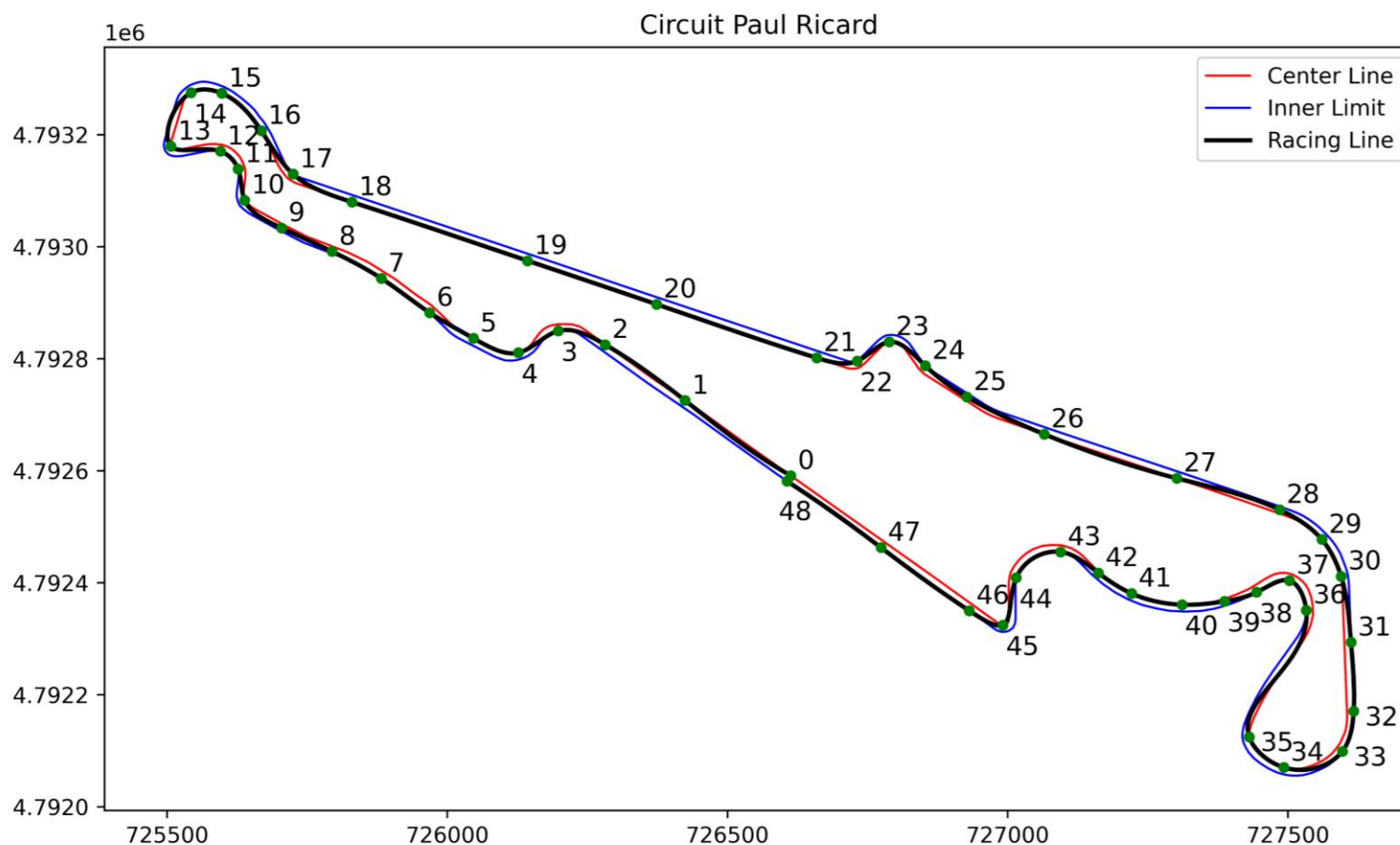
$w = 0.8$

$c1 > 0$

$c2 = 0.225$

TRAJECTOIRE FINALE

	Meilleure trajectoire	Trajectoire centrale	Temps Hamilton
Temps	88.27s	93.64s	88.319s



ANNEXE - références

Github fichiers GeoJSON : <https://github.com/bacinger/f1-circuits>

[1] : J. KENNEDY, R.C. EBERHART : Particle swarm optimization : Proceedings of ICNN'95 - International Conference on Neural Networks, pp. 1942-1948 vol.4, (1995), Perth, Australia, doi: 10.1109/ICNN.1995.4889

[2] : A. GRAY, E. ABBENA ,S. SALAMON : Modern Differential Geometry of Curves and Surfaces with Mathematica Third Edition : Chapman & Hall/CRC, 2006, ISBN: 1584884487

Preuve $a = v^2/r$

Considérons une particule en mouvement circulaire uniforme de rayon r

On a l'accélération en coordonnées polaires:

$$\vec{a} = (\ddot{r} - r\dot{\theta}^2)\vec{u}_r + (r\ddot{\theta} + 2\dot{r}\dot{\theta})\vec{u}_\theta$$

Or comme r est constant : $\dot{r} = 0$ et $\ddot{r} = 0$

De plus on a $\dot{\theta}$ qui est constant donc $\ddot{\theta} = 0$

Donc : $\vec{a} = -r\dot{\theta}^2\vec{u}_r$

Aussi on a :

$$\vec{v} = \frac{dr\vec{u}_r}{dt} = r \frac{d\vec{u}_r}{dt} = r \frac{d\vec{u}_r}{d\theta} \frac{d\theta}{dt}$$

Or :

$$\frac{d\vec{u}_r}{d\theta} = \vec{u}_\theta$$

$$\text{Donc } \vec{v} = r\dot{\theta}\vec{u}_\theta$$

Enfin :

$$\boxed{a = r\dot{\theta}^2 = v^2/r}$$

Preuve v_max

On a $a = \frac{v^2}{r}$

De plus sans glissement la force tangentielle de frottement s'écrit :

$$\|\vec{T}\| = \mu_{track} * \|\vec{N}\| \quad \text{avec} \quad \|\vec{N}\| = mg$$

Donc par le principe fondamental de la dynamique, on a selon \hat{u}_r :

$$ma = \mu_{track} * m * g = \frac{mv^2}{r}$$

Donc :

$$v_{max}^2 = \mu_{track} * g * r$$

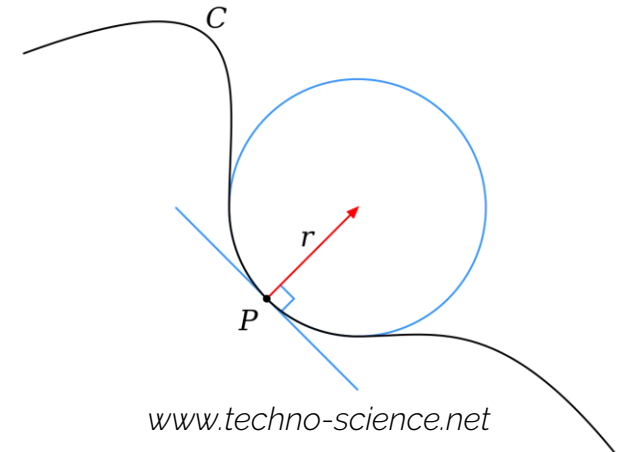
$$v_{max} = \sqrt{\mu_{track} * g * r}$$

ANNEXE

$$\begin{cases} \mu_{slick} \in [1,7; 1,9] (1,78) \\ \mu_{voiture} = 0,8 \end{cases}$$

Avec 1,9 on a un temps de 86,603s soit une différence de 1,667s

$$r = \frac{1}{k} \text{ avec } k = \frac{\left| \frac{dx}{dt} \frac{d^2y}{dt^2} - \frac{dy}{dt} \frac{d^2x}{dt^2} \right|}{\left(\frac{dx^2}{dt} + \frac{dy^2}{dt} \right)^{\frac{3}{2}}} \quad [2]$$



$$\begin{cases} v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1(p_i^t - x_i^t) + c_2 \cdot r_2(g^t - x_i^t) & (1) \\ x_i^{t+1} = x_i^t + v_i^{t+1} \Delta t & (2) \end{cases}$$

CONTACT

- BWT (Best Water Technologies) : sponsorise Alpine f1 ainsi que d'autre catégorie de sport auto (Porsche Cup,f2,...)
- Alpine f1 : demande de données gps ainsi que l'évaluation de trajectoires
- Mon père travaille à STMicroelectronics : accès à une ferme de calcul
- D'autres écuries sans réponses

```

1 """
2 Author: Xayon
3 file: tipe.py
4 """
5
6 import numpy as np
7 import geopandas as gpd
8 import matplotlib.pyplot as plt
9 from shapely import LinearRing, get_coordinates, shortest_line, Point
10 import math
11 from scipy import interpolate
12 import time
13 from pyproj import Proj
14 from tqdm import tqdm
15 import os
16 import matplotlib
17
18 # Constants
19 WIDTH = 12
20 A = 6378137 # m
21 F= 1/298.257223563
22
23 class Track:
24
25     def __init__(self, filename):
26
27         self.nb_sectors = 70
28         self.sectors = []
29
30         self.filename = filename
31         self.trackData = gpd.read_file(filename)
32         self.trackname = self.trackData['Name'][0]
33
34         self.coords = None
35         self.pos_x, self.pos_y = self.getCoords() # useless
36
37         self.centerLine = None
38         self.innerLimit = None
39         self.outerLimit = None
40
41         self.s_points = None
42         self.inside_points = None
43         self.outside_points = None
44
45         self.boundaries = []
46
47         self.test = []
48         self.test2 = True
49
50         self.generateTrackLimits()
51
52         self.defineSector()

```

```

52
53     self.getBoundaries()
54
55     self.racingLine = None
56     self.rl2 = None
57
58
59     def getCoords(self):
60         self.coords = interpolator(self.trackData['geometry'][0],1000).flatten()
61         x = [self.coords[i] for i in range(len(self.coords)) if i % 2 == 0]
62         y = [self.coords[i] for i in range(len(self.coords)) if i % 2 == 1]
63
64         p = Proj(proj='utm',zone=31,ellps='WGS84', preserve_units=False)
65         x,y = p(x,y)
66         self.coords = np.array([x,y]).T.flatten()
67         return x,y
68
69     def plotTrack(self, centerLine=False, innerLimit=True, outerLimit=True,
70                 sectors=False,racingLine =False,nb=False, points=False):
71
72         plt.figure(figsize=(11.2,6.3))
73
74         if centerLine:
75             plt.plot(*self.centerLine.xy, color='black', linewidth=2)
76         if innerLimit:
77             # print(self.innerLimit.xy)
78             plt.plot(*self.innerLimit.xy, color='red', linewidth=1)
79         if outerLimit:
80             # print(self.outerLimit.xy)
81             plt.plot(*self.outerLimit.xy, color='blue', linewidth=1)
82         if racingLine:
83             plt.plot(*self.racingLine, color='black', linewidth=2)
84         if sectors:
85             for i in range(self.nb_sectors):
86                 plt.plot([self.outside_points[i][0],self.inside_points[i][0]],
87                         [self.outside_points[i][1],self.inside_points[i][1]])
88         if nb:
89             for j in range(len(self.rl2[0])):
90                 if j == 0:
91                     plt.text(self.rl2[0][j]-15,self.rl2[1][j]-34,
92                             len(self.rl2[0])-j-1, fontsize=12,
93                             horizontalalignment='left', verticalalignment='top')
94                 elif len(self.rl2[0])-j in [4,5,15,23,39,40,41,45,46]:
95                     plt.text(self.rl2[0][j]+5,self.rl2[1][j]-18,
96                             len(self.rl2[0])-j-1, fontsize=12,
97                             horizontalalignment='left', verticalalignment='top')
98                 elif len(self.rl2[0])-j-1 in [32,33,35]:
99                     plt.text(self.rl2[0][j]+20,self.rl2[1][j]+10,
100                             len(self.rl2[0])-j-1, fontsize=12,
101                             horizontalalignment='left', verticalalignment='top')
102             else:

```

```

103         plt.text(self.r12[0][j]+12,self.r12[1][j]+12,
104                 len(self.r12[0])-j-1, fontsize=12)
105     if points:
106         plt.plot(self.r12[0],self.r12[1], "go", markersize=4)
107
108     plt.gca().set_aspect('equal')
109     plt.title(self.trackname)
110     plt.legend(['Center Line', 'Inner Limit', 'Outer Limit'])
111     plt.show()
112
113
114 def saveTrack(self, centerLine=False, innerLimit=True, outerLimit=True,
115             sectors=False, racingLine=False, nb=False, points=False,
116             transparent=False, path="Data/plot_output/"):
117     try:
118         os.mkdir(path)
119     except:
120         pass
121
122     path_ = f"{path}{self.trackname}{'_racingline' if racingLine else ''}
123             {'_centerline' if centerLine else ''}{'_nb' if nb else ''}
124             {'_t' if transparent else ''}{'_s' if sectors else ''}.png"
125     print(f"saving {path_} ... at {time.strftime('%H:%M:%S')}")
126
127     plt.figure(figsize=(11.2,6.3))
128
129     if centerLine:
130         plt.plot(*self.centerLine.xy, color='black', linewidth=2)
131     if innerLimit:
132         # print(self.innerLimit.xy)
133         plt.plot(*self.innerLimit.xy, color='red', linewidth=1)
134     if outerLimit:
135         # print(self.outerLimit.xy)
136         plt.plot(*self.outerLimit.xy, color='blue', linewidth=1)
137     if racingLine:
138         plt.plot(*self.racingLine, color='black', linewidth=2)
139
140     if sectors:
141         for i in range(self.nb_sectors):
142             plt.plot([self.outside_points[i][0],self.inside_points[i][0]],
143                     [self.outside_points[i][1],self.inside_points[i][1]],
144                     color='black')
145     if nb:
146         for j in range(len(self.r12[0])):
147             if j == 0:
148                 plt.text(self.r12[0][j]-15,self.r12[1][j]-34,
149                         len(self.r12[0])-j-1, fontsize=12,
150                         horizontalalignment='left', verticalalignment='top')
151             elif len(self.r12[0])-j in [4,5,15,23,39,40,41,45,46]:
152                 plt.text(self.r12[0][j]+5,self.r12[1][j]-18,
153                         len(self.r12[0])-j-1, fontsize=12,
154                         horizontalalignment='left', verticalalignment='top')

```

```

154         elif len(self.r12[0])-j-1 in [32,33,35]:
155             plt.text(self.r12[0][j]+20,self.r12[1][j]+10,
156                     len(self.r12[0])-j-1, fontsize=12,
157                     horizontalalignment='left', verticalalignment='top')
158         else:
159             plt.text(self.r12[0][j]+12,self.r12[1][j]+12,
160                     len(self.r12[0])-j-1, fontsize=12)
161     if points:
162         plt.plot(self.r12[0],self.r12[1], "go", markersize=4)
163
164     plt.gca().set_aspect('equal')
165     plt.title(self.trackname)
166     plt.legend(['Center Line', 'Inner Limit', 'Outer Limit'])
167     plt.savefig(path_,dpi=600,transparent=transparent)
168     plt.close('all')
169     plt.clf()
170
171 def generateTrackLimits(self):
172     self.centerLine = LinearRing(self.coords.reshape(-1, 2))
173     # print(self.coords.reshape(-1,2))
174     self.innerLimit = self.centerLine.buffer(WIDTH/2, join_style=2,
175                                             mitre_limit=1).interiors[0]
176     self.outerLimit = self.centerLine.buffer(WIDTH/2, join_style=2,
177                                             mitre_limit=1).exterior
178
179
180 def defineSector(self):
181     '''Defines sectors' search space
182
183     Parameters
184     -----
185     center_line : LineString
186         Center line of the track
187     inside_line : LineString
188         Inside line of the track
189     outside_line : LineString
190         Outside line of the track
191     n_sectors : int
192         Number of sectors
193
194     Returns
195     -----
196     inside_points : list
197         List coordinates corresponding to the internal point of each sector
198     segment
199     outside_points :
200         list coordinates corresponding to the external point of each sector
201     segment
202     '''
203     center_line =
204     self.centerLine
205     inside_line =
206     self.innerLimit
207     outside_line =
208     self.outerLimit

```

```

205     # sect.insert(0,0)
206
207     # sect.insert(1,30)
208     # sect.append(0)
209     # print("sect : ",sect)
210     sect = [0, 35, 69, 81, 95, 102, 114, 128, 140, 158, 172, 183, 196,
211            207,
212            246, 262, 283, 293, 310, 326, 338, 355, 390, 435, 463, 479, 494,
213            506, 520, 575, 620, 680, 701, 716, 733, 745, 762, 780, 789, 797,
214            812, 830, 848, 867, 882, 899, 914, 930, 960, 0]
215     self.nb_sectors = len(sect)
216     n_sectors = self.nb_sectors
217
218     distances = np.linspace(0, center_line.length, n_sectors)
219
220     center_points = [self.pos_x[sect[i]] for i in range(len(sect))],
221     [self.pos_y[sect[i]] for i in range(len(sect))]
222     center_points = np.array(center_points).T
223
224     distances = np.linspace(0,inside_line.length, 1000)
225     inside_border = [inside_line.interpolate(distance)
226                      for distance in distances]
227     inside_border = np.array([e.x, e.y] for e in inside_border])
228     inside_points = np.array([get_closest_points([center_points[i][0],
229            center_points[i][1]], inside_border)
230            for i in range(len(center_points))])
231
232     distances = np.linspace(0,outside_line.length, 1000)
233     outside_border = [outside_line.interpolate(distance)
234                      for distance in distances]
235     outside_border = np.array([e.x, e.y] for e in outside_border])
236     outside_points = np.array([get_closest_points([inside_points[i][0],
237            inside_points[i][1]], outside_border)
238            for i in range(len(center_points))])
239
240     self.inside_points = inside_points
241     self.outside_points = outside_points
242
243     def getBoundaries(self):
244         """
245         Calculates the boundaries of each sector by computing the Euclidean
246         distance between the inside and outside points.
247         Stores the boundaries in the 'boundaries' attribute of the object.
248         """
249
250         for i in range(self.nb_sectors):
251             self.boundaries.append(np.linalg.norm(
252                 self.inside_points[i]-self.outside_points[i]))
253             # print(self.boundaries)
254             self.boundaries = [12]*self.nb_sectors
255
256     def getRacingLine(self, sectors):

```

```

256     """
257     Calculates the racing line for a given set of sectors.
258
259     Args:
260         sectors (list): A list of sectors.
261
262     Returns:
263         list: A list of points representing the racing line.
264     """
265     r1 = []
266     for i in range(len(sectors)):
267         x1, y1 = self.inside_points[i][0], self.inside_points[i][1]
268         x2, y2 = self.outside_points[i][0], self.outside_points[i][1]
269         m = (y2-y1)/(x2-x1)
270
271         a = math.cos(math.atan(m)) # angle with x axis
272         b = math.sin(math.atan(m)) # angle with x axis
273
274         xp = x1 - sectors[i]*a
275         yp = y1 - sectors[i]*b
276
277         if xp < min([x1, x2]) or xp > max([x1,x2]):
278             xp = x1 + sectors[i]*a
279             yp = y1 + sectors[i]*b
280
281         r1.append([xp, yp])
282         if self.test2:
283             self.test.append([xp,yp])
284
285     return r1
286
287     next_path(path_pattern):
288     def
289     """
290     Finds the next free path in an sequentially named list of files
291
292     e.g. path_pattern = 'file-%s.txt':
293
294     file-1.txt
295     file-2.txt
296     file-3.txt
297
298     Runs in log(n) time where n is the number of existing files in
299     sequence
300     """
301     i = 1
302
303     # First do an exponential search
304     while os.path.exists(path_pattern % i):
305         i = i * 2
306
307     # Result lies somewhere in the interval (i/2..i]

```

```

307 # We call this interval (a..b] and narrow it down until a + 1 = b
308 a, b = (i // 2, i)
309 while a + 1 < b:
310     c = (a + b) // 2 # interval midpoint
311     a, b = (c, b) if os.path.exists(path_pattern % c) else (a, c)
312
313 return path_pattern % b
314
315
316
317 def get_closest_points(point, array):
318     """
319     Returns the closest point(s) in an array to a given point.
320
321     Args:
322     point (tuple): A tuple containing the x and y coordinates of the point.
323     array (list): A list of tuples, each containing the x and y coordinates
324     of
325     a point.
326
327     Returns:
328     list: A list containing the x and y coordinates of the closest point(s)
329     in
330     the array to the given point.
331     """
332     result = []
333     distance = 1000
334     for i in range(len(array)):
335         temp = math.sqrt((point[0]-array[i][0])**2+(point[1]-array[i][1])**2)
336         if temp<distance:
337             distance = temp
338             result = [array[i][0], array[i][1]]
339     return result
340
341 def interpolator(linring, nb):
342     """
343     Interpolates a linear ring to obtain a specified number of points.
344
345     Args:
346     linring (shapely.geometry.LinearRing): The linear ring to interpolate.
347     nb (int): The number of points to interpolate.
348
349     length = linring.length
350     step = length / nb
351     points = []
352     for i in range(nb):
353         points.append(linring.interpolate(i*step).coords[0])
354     return np.array(points)
355
356
357 def flatten(l):

```

```

358 result = []
359 for el in l:
360     result.extend(el)
361 return result
362
363 dist(x1,y1,x2,y2):
364 def
365 return math.sqrt((x1-x2)**2+(y1-y2)**2)
366
367
368 -----Particle Swarm Optimization algorithm implementation. -----
369 """
370 class Particle:
371     def __init__(self, dim, boundaries):
372
373         self.position = []
374         self.velocity = []
375
376         self.b_position = []
377
378         for i in range(dim):
379
380             # np.append(self.position,np.random.uniform(0, boundaries[i]))
381             self.position.append(np.random.uniform(0, boundaries[i]))
382             # np.append(self.velocity,np.random.uniform(-boundaries[i],
383             # boundaries[i]))
384             self.velocity.append(np.random.uniform(-boundaries[i],
385             boundaries[i]))
386
387         self.position = np.array(self.position)
388         self.velocity = np.array(self.velocity)
389
390         self.b_position = self.position
391
392     def setPosition(self,value):
393         self.position = value
394
395     def setVelocity(self,value):
396         self.velocity = value
397
398     def setPosition(self,value):
399         self.b_position = value
400
401     """
402     Particle Swarm Optimization algorithm implementation.
403
404     Parameters:
405     fitFunc (function): The fitness function to optimize.
406     dim (int): The number of dimensions of the search space.

```



```

408     boundaries (list): A list of tuples representing the boundaries of the
409     search space.
410     nbParticle (int): The number of particles in the swarm.
411     nbIter (int): The number of iterations to run the algorithm.
412     a(float): The cognitive parameter.
413     b(float): The social parameter.
414     track (Track): The track object to plot the particles' positions.
415
416     Returns:
417     tuple: A tuple containing the global best position, the global best fitness
418     value, the history of global best positions, and the history of global
419     best fitness values.
420     """
421
422     particles = []
423     gs = []
424     gs_e = []
425     # gs_h = []
426     # gs_he = []
427
428     for i in range(nbParticle):
429         particles.append(Particle(dim,boundaries))
430
431     gs = particles[0].position
432     gs_e = fitFunc(gs,track)
433     for p in particles:
434         p_eval = fitFunc(p.b_position,track)
435         if p_eval < gs_e:
436             gs = p.b_position
437             gs_e = fitFunc(gs,track)
438
439     # gs_h.append(gs)
440     # gs_he.append(gs_e)
441     with tqdm(total=nbIter*len(particles)*dim) as pbar:
442
443         for t in range(nbIter):
444             # for i in range(nbParticle):
445
446                 for p in particles:
447                     e1 = np.random.uniform(0,1)
448                     e2 = np.random.uniform(0,1)
449                     newPosition = []
450                     newVelocity = []
451                     for d in range(dim):
452                         newVelocity.append(w * p.velocity[d]
453                             + a * e1 * (p.b_position[d] - p.position[d])
454                             + b * e2 * (gs[d] - p.position[d]))
455
456                         # p.setVelocity(p.velocity + a*e1*(gs - p.position)
457                         # + b*e2*(p.b_position - p.position))
458

```

```

459         # Check the particles will stay in between the
460         boundaries
461         # before computing the positions
462         if newVelocity[d] < -boundaries[d]:
463             newVelocity[d] = -boundaries[d]
464         elif newVelocity[d] > boundaries[d]:
465             newVelocity[d] = boundaries[d]
466
467         newPosition.append(p.position[d] + newVelocity[d])
468         # p.setPosition(p.position + p.velocity)
469
470         # Check if the particle is still in the boundaries
471         if newPosition[d] < 0.0 :
472             newPosition[d] = 0.0
473         elif newPosition[d] > boundaries[d]:
474             newPosition[d] = boundaries[d]
475
476         pbar.update(1)
477
478         p.setVelocity(newVelocity)
479         p.setPosition(newPosition)
480
481         pEval = fitFunc(p.position,track)
482         if pEval <
483             fitFunc(p.b_position,track):
484                 p.setBestPosition(p.position)
485                 if pEval < gs_e:
486                     gs = p.position
487                     gs_e = pEval
488
489         # gs_h.append(gs)
490         # gs_he.append(gs_e)
491         pbar.close()
492
493     return gs, gs_e
494
495     fitFunc(sectors,track):
496     def
497     """
498     Calculates the lap time for a given set of
499     sectors.
500
501     Returns:
502     float: The lap time for the given sectors.
503     """
504
505     # if track.test2:
506         # print("sectors : ")
507         # print(sectors)
508         # print(len(sectors))
509         a = getLapTime(track.getRacingLine(sectors))[0]
510         track.test2 = False
511     return a

```

```

510 def getLapTime(racingLine):
511     """
512     Computes the lap time and the (x, y) coordinates of the racing line.
513
514     Parameters:
515     racingLine (list): A list of (x, y) coordinates of the racing line.
516
517     Returns:
518     tuple: A tuple containing the lap time and the (x, y) coordinates of the
519     racing line.
520     """
521     rl = np.array(racingLine)
522
523
524     # Find the spline
525     tck, _ = interpolate.splprep([rl[:,0], rl[:,1]],s=0, k=3)
526     # Evaluate the spline
527     x, y = interpolate.splev(np.linspace(0, 1, 2000), tck)
528     # Compute the derivative
529     dx, dy = np.gradient(x), np.gradient(y)
530     d2x, d2y = np.gradient(np.array(dx)), np.gradient(np.array(dy))
531
532     k = np.abs(dx*d2y - dy*d2x) / np.power(dx*dx + dy*dy, 1.5)
533     r = 1/k
534
535     #  $\mu = 0.7$ 
536      $\mu = 1.78$ 
537     # computing the max speed
538     v = np.sqrt( $\mu * r * 9.81$ )
539     v = np.clip(v, None, 95.66)
540     # print(v)
541
542     # computing the lap time
543     lapTime = np.sum(np.sqrt(np.power(np.diff(x), 2)
544                                + np.power(np.diff(y), 2)) / v[:-1])
545
546     return lapTime, (x, y)
547
548
549 def main():
550     """
551     Main function to run the Particle Swarm Optimization algorithm.
552
553     nbpart = 400
554     nbiter = 300
555     path = f"Data/plot_output/{time.strftime('%d-%m-%Y')}/%s/"
556     path = next_path(path)
557     w = 0.8
558     a = 1
559     b = 0.2
560     print(f"""

```

```

561 - nbIteration : {nbiter}
562 - nbParticule : {nbpart}
563 - w : {w}
564 - c1 : {a}
565 - c2 : {b}
566     """
567     gs, gs_e = pso(fitFunc, track.nb_sectors, track.boundaries, nbpart,
568                    nbiter,
569                    w, a, b, track)
570     rl = track.getRacingLine(gs)
571     gs_e = getLapTime(rl)[0]
572     track.racingLine = getLapTime(rl)[1]
573     track.rl2 = [[rl[i][0] for i in range(len(rl))],
574                 [rl[i][1] for i in range(len(rl))]]
575     print(f"""
576 - Global best position : {gs}
577 - Lap time : {gs_e}
578 - Parameters : {w}, {a}, {b}
579
580     path="Img/"
581     track.plotTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
582                    points=True)
583     track.saveTrack(transparent=False, path=path, sectors=True)
584     track.saveTrack(transparent=True, path=path, sectors=True)
585     track.saveTrack(centerLine=False, transparent=True, path=path)
586     track.saveTrack(centerLine=False, transparent=False, path=path)
587
588     track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
589                    points=True,transparent=True, path=path)
590     track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
591                    points=True,transparent=False, path=path)
592     track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=False,
593                    points=True,transparent=False, path=path)
594
595     with open(f"{path}data.txt","w") as file:
596         file.write(str([gs_e,gs,rl]))
597         file.close()
598
599     main2():
600     def
601     """
602     Main function to run the Particle Swarm Optimization algorithm
603     multiple
604     times.
605     """
606     track = Track("Data/Track_data/cbg.geojson")
607     nbpart = 50
608     nbiter = 50
609     print(f"nbiter : {nbiter}, nbpart : {nbpart}")
610     path = f"Data/plot_output/{time.strftime('%d-%m-%Y')}/"
611     try:

```

```

612         os.mkdir(path)

613     except:
614         n = 0
615     with tqdm(total=n, position=0) as pbar:
616         with tqdm(total=0, position=1, bar_format='{desc}') as pbar2:
617             for i in range(n):
618                 # print(f"i : {i}")
619                 pbar.set_description(f" n°{i} ")
620                 path_ = next_path(f"{path}%s/")
621                 try:
622                     os.mkdir(path_)
623                 except:
624                     pass
625                 pbar2.set_description(f"Calcul de la meilleure trajectoire de
626                                     {path_}...")
627                 gs ,gs_e = pso_2(fitFunc,track.nb_sectors,track.boundaries,
628                                 nbpart,nbiter,0.8,1,0.20,track)
629                 r1 = track.getRacingLine(gs)
630                 track.racingLine = getLapTime(r1)[1]
631                 track.r12 = [[r1[i][0] for i in range(len(r1))],
632                             [r1[i][1] for i in range(len(r1))]]
633                 pbar2.set_description(f"Sauvegarde de {path_}data.txt ...")
634                 with open(f"{path_}data.txt","w") as file:
635                     file.write(str([gs_e,gs,r1]))
636                     file.close()
637                 pbar2.set_description(f"Sauvegarde des plots de {path_} ...")
638                 # track.plotTrack(centerLine=False,sectors=False,
639                                     racingLine=True,nb=True,points=True)
640                 track.saveTrack(centerLine=False,sectors=False,racingLine=True,
641                                 nb=True,points=True,transparent=False, path=path_)

642                 track.saveTrack(centerLine=False,sectors=False,racingLine=True,
643                                 nb=False,points=True,transparent=False,path=path_)
644                 pbar2.set_description(f"Sauvegarde des plots n°{i} ... OK")
645                 pbar.update(1)

646     """
647     def plot the track with the racing line from the data file.
648     """
649     def plotData():
650         file = open("Data/Results/2/2/Circuit Paul Ricard.txt","r")
651         n,gs_e,gs,r1 = list(eval(file.read()))[7]
652         print(n,gs_e)
653         track.racingLine = getLapTime(r1)[1]
654         track.r12 = [[r1[i][0] for i in range(len(r1))],
655                     [r1[i][1] for i in range(len(r1))]]
656         track.plotTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
657                         points=True)
658         track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
659                         points=True,transparent=False)

```

```

660         track.saveTrack(centerLine=False,sectors=False,racingLine=True,n
661                         b=True,
662                         points=True,transparent=True)
663         file.close()
664         track.saveTrack(centerLine=False,sectors=False,racingLine=True,n
665                         b=False,
666                         if __name__ == "__main__":
667                             track = Track("Data/Track_data/cbg.geojson")
668                             main()
669                             # main2()
670                             """
671                             Author: Xayon
672                             file: annealing.py
673                             """
674                             # from scipy.optimize import dual_annealing
675                             import dual_annealing
676                             import time
677                             import tipe
678                             import numpy as np
679                             from tqdm import tqdm
680                             import os
681                             N_THREAD = 2
682                             NB_PARTICLE = 150
683                             NB_ITER = 80
684                             MAX_ITER = 500
685                             track = tipe.Track("Data/Track_data/cbg.geojson")
686                             pbar = tqdm(total=NB_ITER*NB_PARTICLE*track.nb_sectors,
687                                         position=3,
688                                         desc="Threads")
689                             date = time.strftime("%d_%m_%y_%H:%M",time.localtime())
690                             path = f"Data/annealing_output/{date}"
691                             w = 0.8
692                             d = []
693                             a = []
694                             def func(w):
695                                 l = []
696                                 W = list(w)
697                                 c1 = round(W[0],2)
698                                 c2 = round(W[1],2)
699                                 W = [c1,c2]
700                                 # print(f"Starting with c1={c1}, c2={c2} at {time.ctime()}")
701                                 cur_param.set_description_str(f"Current parameters : c1={c1}, c2={c2}
702                                                                at {time.ctime()}")

```

```

714
715 file = open(f"{path}/data_{id}.txt", "r")
716 d = list(eval(file.read()))
717 file.close()
718 for c1_,c2_,lmin_,lmean_,lstd_ in d:
719     if c1_ == c1 and c2_ == c2:
720         # print(f"{c1},{c2} already computed")
721         res.set_description_str(f"c1={c1}, c2={c2} already computed :
722                               {lmin_},{lmean_},{lstd_}")
723         a.append(1)
724         dual_annealing.pbar.refresh()
725         return lmin_
726 for i in range(N_THREAD):
727     gs, gs_e = tipe.pso_(tipe.fitFunc, track.nb_sectors, track.boundaries,
728                         NB_PARTICLE, NB_ITER, w, c1, c2, track.pbar)
729     pbar.reset()
730     l.append(gs_e)
731
732 l = np.array(l)
733 lmin = l.min()
734 lmean = l.mean()
735 lstd = l.std()
736
737 d.append([c1, c2, lmin, lmean, lstd])
738 # print(f"c1={c1},c2={c2} : {lmin},{lmean}")
739 res.set_description_str(f"c1={c1}, c2={c2} : {lmin},{lmean},{lstd}")
740
741 file = open(f"{path}/data_{id}.txt", "w")
742 # print(str(d))
743 file.write(str(d))
744 file.close()
745 a.append(0)
746 dual_annealing.pbar.refresh()
747
748 return lmin
749
750 def main(id=id):
751     try:
752         os.mkdir(path)
753     except:
754         pass
755
756 file = open(f"{path}/data_{id}.txt", "w")
757 file.write("[]")
758 file.close()
759 d = []
760
761 dual_annealing.pbar = tqdm(total=MAX_ITER, position=0, desc="Iterations")
762 dual_annealing.pbar.write(f""Starting calculation with :

```

```

763
764 \033[96mNB_PARTICLE\033[00m=\033[92m{NB_PARTICLE}\033[00m,
765 \033[96mNB_ITER\033[00m=\033[92m{NB_ITER}\033[00m,
766 \033[96mN_THREAD\033[00m=\033[92m{N_THREAD}\033[00m,
767 \033[96mMAX_ITER\033[00m=\033[92m{MAX_ITER}\033[00m""")
768 r_min, r_max = -10., 10.
769 bounds = [[r_min, r_max], [r_min, r_max]]
770 result = dual_annealing.dual_annealing(func, bounds,maxiter=MAX_ITER)
771
772 # summarize the result
773 # print('Status : %s' % result['message'])
774 # print('Total Evaluations: %d' % result['nfev'])
775
776 # evaluate solution
777 solution = result['x']
778 solution = solution.tolist()
779 evaluation = func(solution)
780 # print('Solution: f(%s) = %.5f' % (solution, evaluation))
781
782 cur_param.write(f"Status : {result['message']}")
783 dual_annealing.pbar.close()
784 pbar.close()
785 cur_param.set_description_str(f"Total Evaluations: {result['nfev']}")
786 res.set_description_str(f"Solution found : f({solution}) =
787 {evaluation}")
788 cur_param.close()
789 res.close()
790
791 file = open(f"{path}/data_{id}.txt", "r")
792 data = file.read()
793 file.close()
794
795 file = open(f"{path}/data_{id}.txt", "w")
796 file.write(data)
797 file.close()
798
799 file = open(f"{path}/info_{id}.txt", "w")
800 file.write(str((w,r_min,r_max,NB_PARTICLE,NB_ITER,MAX_ITER))+ "\n")
801 file.write(str(f"Solution,evaluation,result['message']"),
802 file.write(str(a)+"\n")
803 file.write(str(dual_annealing.r)+"\n")
804 file.close()
805 print(f"\nSaving OK")
806 file =
807 open(f"{path}/data_{id}.txt", "w")
808 file.close()
809 os.remove(f"{path}/data_{id}.txt")
810 if __name__ == "__main__":
811     cur_param = tqdm(total=0, position=1, bar_format='{desc}')
812     res = tqdm(total=0, position=2, bar_format='{desc}')
813
814     main(id)

```

```

815
816
817
818 """
819 Author: Xayon
820 file: process_annealing.py
821 """
822 import dual_annealing
823 import time
824 import tipe
825 import numpy as np
826 from tqdm import tqdm
827 import os

828 import sys
829
830 N_THREAD = 2
831 NB_PARTICLE = 1
832 NB_ITER = 1
833 MAX_ITER = 20
834
835 if not os.path.exists("Data/Track_data"):
836     os.mkdir("Data/Track_data")
837 if not os.path.exists("Data/Track_data/cbg.geojson"):
838     os.system("wget -O Data/Track_data/cbg.geojson \
839         https://raw.githubusercontent.com/bacinger/\
840             f1-circuits/master/circuits/fr-1969.geojson")
841
842 track = tipe.Track("Data/Track_data/cbg.geojson")
843 pbar = tqdm(total=NB_ITER*NB_PARTICLE*track.nb_sectors, position=3,
844             desc="Threads")
845 if not os.path.exists("Data/annealing_output"):
846     os.mkdir("Data/annealing_output")
847
848 date = time.strftime("%d_%m_%y_%H:%M", time.localtime())
849 path = f"Data/annealing_output/{date}"
850
851 d = []
852 w = 0.8

853
854 a = []
855
856 def func(w):
857     l = []
858     W = list(w)
859     c1 = round(W[0], 2)
860     c2 = round(W[1], 2)
861     W = [c1, c2]
862     # print(f"Starting with c1={c1}, c2={c2} at {time.ctime()}")
863     cur_param.set_description_str(f"Current parameters : c1={c1},\
864                                   c2={c2} at {time.ctime()}")
865

```

```

866 file = open(f"{path}/data_{id}.txt", "r")
867 d = list(eval(file.read()))
868 file.close()
869 for c1_, c2_, lmin_, lmean_, lstd_ in d:
870     if c1_ == c1 and c2_ == c2:
871         # print(f"{c1},{c2} already computed")
872         res.set_description_str(f"c1={c1}, c2={c2} already computed :\
873                                 {lmin_},{lmean_},{lstd_}")
874         a.append(1)
875         dual_annealing.pbar.refresh()
876         return lmin_
877 for i in range(N_THREAD):
878     gs, gs_e = tipe.pso_(tipe.fitFunc, track.nb_sectors,
879                          track.boundaries,
880                          NB_PARTICLE, NB_ITER, w, c1, c2, track, pbar)
881     pbar.reset()
882     l.append(gs_e)
883
884 l = np.array(l)
885 lmin = l.min()
886 lmean = l.mean()
887 lstd = l.std()
888
889 d.append([c1, c2, lmin, lmean, lstd])
890 # print(f"c1={c1}, c2={c2} : {lmin},{lmean}")
891 res.set_description_str(f"c1={c1}, c2={c2} : {lmin},{lmean},{lstd}")
892
893 file = open(f"{path}/data_{id}.txt", "w")
894 # print(str(d))
895 file.write(str(d))
896 file.close()
897 a.append(0)
898 dual_annealing.pbar.refresh()
899
900 return lmin
901
902
903 main(id=id):
904 def
905 try:
906     os.mkdir(path)
907 except:
908     pass
909
910 file = open(f"{path}/data_{id}.txt", "w")
911 file.write("[]")
912 file.close()
913
914 d = []
915

```

```

916 dual_annealing.pbar = tqdm(total=MAX_ITER, position=0, desc="Iterations")
917 dual_annealing.pbar.write(f"Starting calculation with :
918 \033[96mNB_PARTICLE\033[00m=\033[92m{NB_PARTICLE}\033[00m,
919 \033[96mNB_ITER\033[00m=\033[92m{NB_ITER}\033[00m,
920 \033[96mN_THREAD\033[00m=\033[92m{N_THREAD}\033[00m,
921 \033[96mMAX_ITER\033[00m=\033[92m{MAX_ITER}\033[00m""")
922 r_min, r_max = -10., 10.
923 # bounds = [[r_min, r_max], [r_min, r_max]]
924 bounds = [(-10,40),(-10,10)]
925 result = dual_annealing.dual_annealing(func, bounds,maxiter=MAX_ITER)
926
927 # summarize the result
928 # print('Status : %s' % result['message'])
929 # print('Total Evaluations: %d' % result['nfev'])
930
931 # evaluate solution
932 solution = result['x']
933 solution = solution.tolist()
934 evaluation = func(solution)
935 # print('Solution: f(%s) = %.5f' % (solution, evaluation))
936
937 cur_param.write(f"Status : {result['message']}")
938 dual_annealing.pbar.close()
939 pbar.close()
940 cur_param.set_description_str(f"Total Evaluations: {result['nfev']}")
941 res.set_description_str(f"Solution found : f({solution}) = {evaluation}")
942
943 cur_param.close()
944 res.close()
945
946 file = open(f"{path}/data_{id}.txt", "r")
947 data = file.read()
948 file.close()
949
950 file = open(f"{path}/data{id}.txt", "w")
951 file.write(data)
952 file.close()
953
954 file = open(f"{path}/info{id}.txt", "w")
955 file.write(str((w,bounds,NB_PARTICLE,NB_ITER,MAX_ITER))+ "\n")
956 file.write(str([solution,evaluation,result['message'],
957 result['nfev']]))+ "\n")
958 file.write(str(a)+ "\n")
959 file.write(str(dual_annealing.r)+ "\n")
960 file.close()
961 print(f"\nSaving OK")
962 file = open(f"{path}/data_{id}.txt", "w")
963 file.close()
964 if __name__ == '__main__':
965     remove(f"{path}/data_{id}.txt")

```

```

967 if len(sys.argv) > 1:
968     id = int(sys.argv[1])
969 else:
970     id = 0
971 if (os.path.exists(f"Data/annealing_output/{date}/data_{id}.txt")
972 or os.path.exists(f"Data/annealing_output/{date}/data{id}.txt")):
973     print(f"data_{id}.txt already exists")
974     id_list = []
975     for file_name in os.listdir(f"Data/annealing_output/{date}"):
976         if file_name.startswith("data_") and
977         file_name.endswith(".txt"):
978             id_list.append(int(file_name.split("_")[1].split(".")[0]))
979             elif file_name.startswith("data") and file_name.endswith(".txt"):
980                 id_list.append(int(file_name[4:].split(".")[0]))
981 if id_list:
982     biggest_id = max(id_list)
983 else:
984     biggest_id = 0
985 print(f"Starting with id={biggest_id+1}")
986 id = biggest_id+1
987
988 cur_param = tqdm(total=0, position=1,
989 bar_format='{desc}')
990 res = tqdm(total=0, position=2, bar_format='{desc}')
991
992 main(id)
993
994 Author: Xayon
995 file: analyse_annealing.py
996
997 import numpy as np
998 import json
999 import tkinter as tk
1000 from tkinter import filedialog, Tk
1001 import os
1002 import random
1003 import matplotlib.pyplot as plt
1004 from matplotlib.colors import ListedColormap
1005
1006 def analyse_data_1(directory):
1007     # Load the data
1008     with open(f"{directory}/data.json", "r") as f:
1009         data = list(eval(f.read()))
1010         f.close()
1011
1012     # Create a list of all the keys
1013     x,y,z,z2,z3 = [],[],[],[],[]
1014     # x.append(int(key.split(",")[0]))
1015     # for append in data[key.split(",")[1]]:

```

```

1018
1019 for c1,c2,lmin,lmean,lstd in data:
1020     x.append(c1)
1021     y.append(c2)
1022     z.append(lmin)
1023     z2.append(lmean)
1024     z3.append(lstd)
1025
1026 fig = plt.figure()
1027 ax = fig.add_subplot(111, projection='3d')
1028 fig.subplots_adjust(bottom=0.2)
1029
1030 # Affichage des données
1031 fig.suptitle("Analyse des données en 3D")
1032
1033 # Paramètres
1034 # param1, param2 = np.meshgrid(param1, param1)
1035
1036 ax.set_title("Temps min")
1037 ax.set_xlabel("c1")
1038 ax.set_ylabel("c2")
1039 ax.set_zlabel("Temps (s)")
1040 ax.set_zlim3d(bottom=87.5,top=95)
1041 ax.scatter(x, y, z, c='r', marker='o')
1042 plt.show()
1043
1044 fig2 = plt.figure()
1045 ax2 = fig2.add_subplot(111, projection='3d')
1046
1047 fig2.subplots_adjust(bottom=0.2)
1048 ax2.set_title("Temps moyen")
1049 ax2.set_xlabel("c1")
1050
1051 ax2.set_ylabel("c2")
1052
1053 ax2.set_zlabel("Temps (s)")
1054 ax2.set_zlim3d(bottom=87.5,top=95)
1055 ax2.scatter(x, y, z2, c='b', marker='o')
1056 plt.show()
1057
1058 fig3 = plt.figure()
1059 ax3 = fig3.add_subplot(111, projection='3d')
1060 fig3.subplots_adjust(bottom=0.2)
1061 ax3.set_title("ecart type")
1062 ax3.set_xlabel("c1")
1063 ax3.set_ylabel("c2")
1064 ax3.scatter(x, y, z3, c='b', marker='o')
1065
1066 plt.show()
1067
1068 def analyse_info_1(directory):
1069     with open(f"{directory}/info.json", "r") as f:
1070         parameters = tuple(eval(f.readline()))
1071         result = list(eval(f.readline()))

```

```

1069     nb_al_computed = list(eval(f.readline()))
1070     rate = list(eval(f.readline()))
1071     f.close()
1072     nb = len(nb_al_computed)
1073     print(f"nb= {nb}")
1074     plt.plot(nb_al_computed)
1075     plt.title("Cacul deja effectué")
1076     plt.show()
1077
1078 nb_al_computed2 = []
1079 for i in range(nb):
1080     nb_al_computed2.append(sum(nb_al_computed[:i]))
1081
1082 plt.plot(nb_al_computed2)
1083 plt.title("Cacul deja effectué")
1084 plt.show()
1085
1086 plt.plot(rate)
1087 plt.title("vitesse de calcul")
1088 plt.show()
1089
1090 print(f"parameters= {parameters}")
1091 print(f"result= {result}")
1092
1093
1094
1095 analyse_1():
1096 def
1097 root = tk.Tk()
1098 root.withdraw()
1099 directory =
1100     filedialog.askdirectory(initialdir="Data/annealing_output",
1101                             title="Select the directory to
1102                                 open")
1103 analyse_data_1(directory)
1104 analyse_info_1(directory)
1105
1106 import time as test
1107
1108 def analyse_data_2(directory_list):
1109     # Create empty lists for x, y, z, z2, z3
1110     x, y, z, z2, z3 = [], [], [], [], []
1111     nb_files = 0
1112     for directory in directory_list:
1113         if not os.path.exists(f"{directory}/data.json"):
1114             prefixed = [filename for filename in
1115                         os.listdir(f'{directory}/.')]
1116             if filename.startswith("data")]
1117             for filename in prefixed:
1118                 nb_files += 1
1119                 f = open(f"{directory}/{filename}", "r")
1120                 data = list(eval(f.read()))
1121                 f.close()
1122                 for c1, c2, lmin, lmean, lstd in data:

```

```

1120         # if (c1, c2) in zip(x, y):
1121         #     index = list(zip(x, y)).index((c1, c2))
1122
1123         #     removed += 1
1124         #     if lmin < z[index]:
1125         #         z[index] = lmin
1126         #         z2[index] = lmean
1127         #         z3[index] = lstd
1128         #     else:
1129
1130         x.append(c1)
1131         y.append(c2)
1132         z.append(lmin)
1133         z2.append(lmean)
1134         z3.append(lstd)
1135
1136     else:
1137         with open(f"{directory}/data.json", "r") as f:
1138             data = list(eval(f.read()))
1139             f.close()
1140             for c1, c2, lmin, lmean, lstd in data:
1141                 if (c1, c2) in zip(x, y):
1142                     index = list(zip(x, y)).index((c1, c2))
1143                     if lmin < z[index]:
1144                         z[index] = lmin
1145                         z2[index] = lmean
1146                         z3[index] = lstd
1147                     else:
1148                         x.append(c1)
1149                         y.append(c2)
1150                         z.append(lmin)
1151                         z2.append(lmean)
1152                         z3.append(lstd)
1153
1154     print(len(x))
1155     print(f"nb_files = {nb_files}")
1156
1157     print("-----Starting to remove randomly-----")
1158     time1 = test.time()
1159     # Remove half of the data points randomly
1160     num_points = len(x)
1161     # num_points_to_remove = num_points - 30000 if num_points > 20000 else 0
1162     num_to_keep = 30000
1163     time2 = test.time()
1164     # indices_to_remove = random.sample(range(num_points),
1165     num_points_to_remove)
1166     indices_to_keep = random.sample(range(num_points), num_to_keep)
1167     print("-----Data removed----- in ", round(test.time() - time1, 2))
1168     time3 = test.time()
1169     x = [x[i] for i in indices_to_keep]
1170     y = [y[i] for i in indices_to_keep]
1171     z = [z[i] for i in indices_to_keep]
1172     z2 = [z2[i] for i in indices_to_keep]
1173
1174     z3 = [z3[i] for i in indices_to_keep]
1175
1176     print("-----Data removed----- in ", round(test.time() -
1177     time3, 2))
1178
1179     # Plot the data
1180     fig = plt.figure()
1181     ax = fig.add_subplot(111, projection='3d')
1182     fig.subplots_adjust(bottom=0.2)
1183     fig.suptitle("Analyse des données en 3D")
1184     ax.set_title("Temps min")
1185     ax.set_xlabel("c1")
1186     ax.set_ylabel("c2")
1187     ax.set_zlabel("Temps (s)")
1188     ax.set_zlim3d(bottom=87.5, top=95)
1189     ax.scatter(x, y, z, c='r', marker='o')
1190     plt.show()
1191
1192     fig2 = plt.figure()
1193     ax2 = fig2.add_subplot(111, projection='3d')
1194     fig2.subplots_adjust(bottom=0.2)
1195     ax2.set_title("Temps moyen")
1196     ax2.set_xlabel("c1")
1197     ax2.set_ylabel("c2")
1198     ax2.set_zlabel("Temps (s)")
1199     ax2.set_zlim3d(bottom=87.5, top=95)
1200     ax2.scatter(x, y, z2, c='b', marker='o')
1201     plt.show()
1202
1203     fig3 = plt.figure()
1204     ax3 = fig3.add_subplot(111, projection='3d')
1205     fig3.subplots_adjust(bottom=0.2)
1206     ax3.set_title("ecart type")
1207     ax3.set_xlabel("c1")
1208     ax3.set_ylabel("c2")
1209     ax3.scatter(x, y, z3, c='b', marker='o')
1210     plt.show()
1211
1212     print("-----Starting to import results-----")
1213
1214     nb_al_computed = []
1215     rate = []
1216     parameters = []
1217     results = []
1218
1219     for directory in directory_list:
1220         if not os.path.exists(f"{directory}/info.json"):

```



```

1221     prefixed = [filename for filename in os.listdir(f'{directory}/.')]
1222     if filename.startswith("info")]
1223     for filename in prefixed:
1224         f = open(f"{directory}/{filename}", "r")
1225         parameters.append(tuple(eval(f.readline())))
1226         results.append(list(eval(f.readline())))
1227         nb_al_computed.extend(list(eval(f.readline())))
1228         rate.extend(list(eval(f.readline())))
1229         f.close()
1230     else:
1231         with open(f"{directory}/info.json", "r") as f:
1232             parameters.append(tuple(eval(f.readline())))
1233             results.append(list(eval(f.readline())))
1234             nb_al_computed.extend(list(eval(f.readline())))
1235             rate.extend(list(eval(f.readline())))
1236             f.close()
1237
1238     nb = len(nb_al_computed)
1239     print(f"nb = {nb}")
1240     # plt.plot(nb_al_computed)
1241     # plt.title("Calcul déjà effectué")
1242     # plt.show()
1243
1244     # nb_al_computed2 = []
1245     # for i in range(nb):
1246     #     nb_al_computed2.append(sum(nb_al_computed[:i]))
1247
1248     # plt.plot(nb_al_computed2)
1249     # plt.title("Calcul déjà effectué")
1250     # plt.show()
1251
1252     # plt.plot(rate)
1253     # plt.title("Vitesse de calcul")
1254     # plt.show()
1255
1256     print(f"parameters = {parameters}")
1257     # print(f"result = {results}")
1258     x_res, y_res, z_res = [], [], []
1259     iters = []
1260     for coord,time,message,iter in results:
1261         x_res.append(coord[0])
1262         y_res.append(coord[1])
1263         z_res.append(time)
1264         iters.append(iter)
1265
1266     print(f" results = {z_res}")
1267     iter_total = sum(iters)
1268     print(f"Total iterations = {iter_total}")
1269
1270     print("-----Data imported-----")
1271

```

```

1272     print("-----Starting to plot-----")
1273     fig4 = plt.figure(figsize=(8,15))
1274     # fig4.subplots_adjust(bottom=0.2, top=0.8)
1275     ax = fig4.add_subplot(211, projection='3d')
1276     fig4.subplots_adjust(wspace=0, hspace=0)
1277     fig4.subplots_adjust(top=0.95)
1278     # fig4.tight_layout()
1279     fig4.set_constrained_layout_pads(w_pad=0, h_pad=0, hspace=0, wspace=0)
1280     fig4.suptitle("Analyse des données en 3D",y=0.95,fontsize=15)
1281     ax.set_xlabel("c1")
1282     ax.set_ylabel("c2")
1283     # ax.set_zlabel("Temps (s)")
1284     newcmp = ListedColormap(plt.cm.get_cmap('jet_r')(np.linspace(.5, .9
1285     ,256)))
1286     ax.set_zlim3d(bottom=87.5, top=95)
1287     ax.scatter(x, y, z, c='b', marker='o', alpha=0.03, s=10)
1288     # ax.scatter(x, y, z, c=z, marker='o', alpha=0.03, s=10,cmap=newcmp)
1289     ax.scatter(x_res, y_res, z_res, c='r', marker='o', alpha=1, s=40)
1290     ax.view_init(elev=24, azim=62, roll=0)
1291
1292     ax2 = fig4.add_subplot(212, projection='3d')
1293     fig4.subplots_adjust(wspace=0, hspace=0)
1294     fig4.subplots_adjust(top=0.95)
1295     ax2.set_xlabel("c1")
1296     ax2.set_ylabel("c2")
1297     # ax.set_zlabel("Temps (s)")
1298     ax2.set_zlim3d(bottom=87.5, top=95)
1299     ax2.scatter(x, y, z, c='b', marker='o', alpha=0.03, s=10)
1300     # ax2.scatter(x, y, z, c=z, marker='o', alpha=0.03, s=10,cmap=newcmp)
1301     ax2.scatter(x_res, y_res, z_res, c='r', marker='o', alpha=1, s=35)
1302     ax2.view_init(elev=-90, azim=90, roll=0)
1303
1304     ax2.grid(False)
1305     ax2.xaxis.pane.fill = False
1306     ax2.yaxis.pane.fill = False
1307     ax2.zaxis.pane.fill = False
1308     ax2.xaxis.pane.set_edgecolor('w')
1309     ax2.yaxis.pane.set_edgecolor('w')
1310     ax2.zaxis.pane.set_edgecolor('w')
1311     ax2.zaxis.line.set_lw(0.)
1312     ax2.set_zticks([])
1313
1314     plt.savefig("Img/analyse_3D.png", dpi=300, bbox_inches='tight')
1315     # plt.show()
1316
1317
1318     analyse_2():
1319     import _input("Do you want to import the data from import file ?
1320     (y/n): ")
1321

```

```

1321     dirs = []
1322     dir = None
1323     if import_ == "y":
1324         with open("import.txt", "r") as f:
1325             for line in f:
1326                 dirs.append(line.strip())
1327     else:
1328         root = Tk()
1329         while dir != ():
1330             dir = filedialog.askdirectory(initialdir="Data/annealing_output",
1331                                           title="Select the directory to open")
1332             print(dir)
1333             if dir != ():
1334                 dirs.append(dir)
1335 analyse_data_2(dirs)
1336
1337
1338 def analyse_results(directory_list):
1339     print("-----Starting to import results-----")
1340     nb_al_computed = []
1341     rate = []
1342     parameters = []
1343     results = []
1344
1345     for directory in directory_list:
1346         if not os.path.exists(f"{directory}/info.json"):
1347             prefixed = [filename for filename in os.listdir(f'{directory}/.')]
1348             if filename.startswith("info")]
1349                 for filename in prefixed:
1350                     f = open(f"{directory}/{filename}", "r")
1351                     parameters.append(tuple(eval(f.readline())))
1352                     results.append(list(eval(f.readline())))
1353                     nb_al_computed.extend(list(eval(f.readline())))
1354                     rate.extend(list(eval(f.readline())))
1355                     f.close()
1356         else:
1357             with open(f"{directory}/info.json", "r") as f:
1358                 parameters.append(tuple(eval(f.readline())))
1359                 results.append(list(eval(f.readline())))
1360                 nb_al_computed.extend(list(eval(f.readline())))
1361                 rate.extend(list(eval(f.readline())))
1362                 f.close()
1363
1364     nb = len(nb_al_computed)
1365     print(f"nb = {nb}")
1366     # plt.plot(nb_al_computed)
1367     # plt.title("Calcul déjà effectué")
1368     # plt.show()
1369
1370     # nb_al_computed2 = []
1371     # for i in range(nb):

```

```

1372     # nb_al_computed2.append(sum(nb_al_computed[:i]))
1373     # plt.plot(nb_al_computed2)
1374     # plt.title("Calcul déjà effectué")
1375     # plt.show()
1376
1377     # plt.plot(rate)
1378     # plt.title("Vitesse de calcul")
1379     # plt.show()
1380
1381     # print(f"parameters = {parameters}")
1382     # print(f"result = {results}")
1383     # x_res, y_res, z_res = [], [], []
1384     # iters = []
1385     # for coord,time,message,iter in results:
1386     #     x_res.append(round(coord[0],2))
1387     #     y_res.append(round(coord[1],2))
1388     #     z_res.append(time)
1389     #     iters.append(iter)
1390
1391     y_res = [round(result[0][1],2) for result in results]
1392     moy = np.mean(y_res)
1393     med = np.median(y_res)
1394     print(f"""
1395     results = {y_res}
1396     moyenne = {moy}
1397     ecart type = {np.std(y_res)}
1398     mediane = {med}
1399     """)
1400
1401     print("-----Data imported-----")
1402     print("-----Starting to plot-----")
1403     # faire un histogramme sur la variable y_res
1404     fig = plt.figure(figsize=(12,9))
1405     ax = fig.add_subplot(111)
1406
1407     bins = np.arange(round(min(y_res),1),round(max(y_res),1)+0.1,0.05)
1408     print(bins)
1409     print(len(bins))
1410     ax.hist(y_res, bins=bins,color='cornflowerblue',edgecolor='black',
1411            linewidth=1)
1412     print(np.histogram(y_res, bins=bins))
1413     y,x = np.histogram(y_res, bins=bins)
1414
1415     total = sum(y)
1416     max_ = round(x[np.argmax(y)],2)
1417     print(f'max : {max_}')
1418     print(total)
1419     #ax.hist(y_res,bins=20,range=(round(min(y_res),1),round(max(y_res),1)+0
1420     .1)
1421
1422     plt.xticks(ticks=np.arange(start=round(min(y_res),1),
1423                               step=round(np.ptp(y_res)/13,1),

```

```

1422         stop=round(np.max(y_res),1)+0.1))
1423
1424     plt.axvline(x = med, color = 'r', label = 'mediane',
1425               linestyle = 'dashed',linewidth=2)
1426     plt.annotate(f'médiane = {round(med,2)}', xy =(med+0.01, 29),
1427               xytext =(med+0.23, 31),
1428               arrowprops = dict(facecolor='black',arrowstyle="->",lw =
1429                               2))
1430
1431     plt.axvline(x = max_+0.025, color = 'g', label = 'max',
1432               linestyle = 'dashed',linewidth=2)
1433     plt.annotate(f'x_max = {max_+0.025}', xy =(max_+0.03, 16),
1434               xytext =(max_+0.25, 13),
1435               arrowprops = dict(facecolor='black',arrowstyle="->",lw =
1436                               2))
1437
1438     plt.title(f"Répartition des valeurs de c2 sur {total} résultats",
1439             fontsize=15)
1440     # plt.legend()
1441     plt.savefig("Img/histogramme/histogramme_y_res.png", dpi=500,
1442             bbox_inches='tight')
1443     # plt.show()
1444
1445 def analyse_3():
1446     import_=input("Do you want to import the data from import file ? (y/n) :
1447     ")
1448
1449     dirs = []
1450     dir = None
1451     if import_ == "y":
1452         with open("import.txt", "r") as f:
1453             for line in f:
1454                 dirs.append(line.strip())
1455     else:
1456         root = Tk()
1457         while dir != ():
1458             dir = filedialog.askdirectory(initialdir="Data/annealing_output",
1459                                         title="Select the directory to open")
1460         print(dir)
1461         if dir != ():
1462             dirs.append(dir)
1463     analyse_results(dirs)
1464
1465 if __name__ == "__main__":
1466     ~numpy.random.Generator` }
1467     # analyse_1()
1468     analyse_2()
1469
1470     # analyse_3()
1471
1472 # Dual Annealing implementation.

```

```

1470 # Copyright (c) 2018 Sylvain Gubian <sylvain.gubian@pmi.com>,
1471 # Yang Xiang <yang.xiang@pmi.com>
1472 # Author: Sylvain Gubian, Yang Xiang, PMP S.A.
1473 # Modifications by: Xayon
1474
1475 """
1476 A Dual Annealing global optimization algorithm
1477 """
1478
1479 import numpy as np
1480 from scipy.optimize import OptimizeResult
1481 from scipy.optimize import minimize, Bounds
1482 from scipy.special import gammaln
1483 from scipy._lib._util import check_random_state
1484 from scipy.optimize._constraints import new_bounds_to_old
1485 import time
1486 from tqdm import tqdm
1487
1488 __all__ = ['dual_annealing']
1489
1490 pbar = None
1491 r = []
1492
1493 class VisitingDistribution:
1494     """
1495     Class used to generate new coordinates based on the distorted
1496     Cauchy-Lorentz distribution. Depending on the steps within the
1497     strategy
1498     chain, the class implements the strategy for generating new location
1499     changes.
1500
1501     Parameters
1502     -----
1503     lb : array_like
1504         A 1-D NumPy ndarray containing lower bounds of the generated
1505         components. Neither NaN or inf are allowed.
1506     ub : array_like
1507         A 1-D NumPy ndarray containing upper bounds for the generated
1508         components. Neither NaN or inf are allowed.
1509     visiting_param : float
1510         Parameter for visiting distribution. Default value is 2.62.
1511         Higher values give the visiting distribution a heavier tail, this
1512         makes the algorithm jump to a more distant region.
1513         The value range is (1, 3]. Its value is fixed for the life of the
1514         object.
1515     rand_gen : {~numpy.random.RandomState`,
1516
1517         A ~numpy.random.RandomState`, ~numpy.random.Generator` object
1518         for using the current state of the created random generator
1519         container.
1520
1521     """
1522     TAIL_LIMIT = 1.e8
1523     MIN_VISIT_BOUND = 1.e-10

```

```

1521
1522 def __init__(self, lb, ub, visiting_param, rand_gen):
1523     # if you wish to make _visiting_param adjustable during the life of
1524     # the object then _factor2, _factor3, _factor5, _d1, _factor6 will
1525     # have to be dynamically calculated in `visit_fn`. They're factored
1526     # out here so they don't need to be recalculated all the time.
1527     self._visiting_param = visiting_param
1528     self.rand_gen = rand_gen
1529     self.lower = lb
1530     self.upper = ub
1531     self.bound_range = ub - lb
1532
1533     # these are invariant numbers unless visiting_param changes
1534     self._factor2 = np.exp((4.0 - self._visiting_param) * np.log(
1535         self._visiting_param - 1.0))
1536     self._factor3 = np.exp((2.0 - self._visiting_param) * np.log(2.0)
1537         / (self._visiting_param - 1.0))
1538     self._factor4_p = np.sqrt(np.pi) * self._factor2 / (self._factor3 * (
1539         3.0 - self._visiting_param))
1540
1541     self._factor5 = 1.0 / (self._visiting_param - 1.0) - 0.5
1542     self._d1 = 2.0 - self._factor5
1543     self._factor6 = np.pi * (1.0 - self._factor5) / np.sin(
1544         np.pi * (1.0 - self._factor5)) / np.exp(gammain(self._d1))
1545
1546 def visiting(self, x, step, temperature):
1547     """ Based on the step in the strategy chain, new coordinates are
1548     generated by changing all components is the same time or only
1549     one of them, the new values are computed with visit_fn method
1550     """
1551     dim = x.size
1552     if step < dim:
1553
1554         # Changing all coordinates with a new visiting value
1555         visits = self.visit_fn(temperature, dim)
1556         upper_sample, lower_sample = self.rand_gen.uniform(size=2)
1557         visits[visits > self.TAIL_LIMIT] = self.TAIL_LIMIT * upper_sample
1558         visits[visits < -self.TAIL_LIMIT] = -self.TAIL_LIMIT *
lower_sample
1559         x_visit = visits + x
1560         a = x_visit - self.lower
1561         b = np.fmod(a, self.bound_range) + self.bound_range
1562         x_visit = np.fmod(b, self.bound_range) + self.lower
1563         x_visit[np.fabs(
1564             x_visit - self.lower) < self.MIN_VISIT_BOUND] += 1.e-10
1565     else:
1566         # Changing only one coordinate at a time based on strategy
1567         # chain step
1568         x_visit = np.copy(x)
1569         visit = self.visit_fn(temperature, 1)[0]
1570         if visit > self.TAIL_LIMIT:
1571             visit = self.TAIL_LIMIT * self.rand_gen.uniform()

```

```

1571         elif visit < -self.TAIL_LIMIT:
1572             visit = -self.TAIL_LIMIT * self.rand_gen.uniform()
1573             index = step - dim
1574             x_visit[index] = visit + x[index]
1575             a = x_visit[index] - self.lower[index]
1576             b = np.fmod(a, self.bound_range[index]) + self.bound_range[index]
1577             x_visit[index] = np.fmod(b, self.bound_range[
1578                 index]) + self.lower[index]
1579             if np.fabs(x_visit[index] - self.lower[
1580                 index]) < self.MIN_VISIT_BOUND:
1581                 x_visit[index] += self.MIN_VISIT_BOUND
1582             return x_visit
1583
1584 def visit_fn(self, temperature, dim):
1585     """ Formula Visita from p. 405 of reference [2] """
1586     x, y = self.rand_gen.normal(size=(dim, 2)).T
1587
1588     factor1 = np.exp(np.log(temperature) / (self._visiting_param - 1.0))
1589     factor4 = self._factor4_p * factor1
1590
1591     # sigma_x
1592     x *= np.exp(-(self._visiting_param - 1.0) * np.log(
1593         self._factor6 / factor4) / (3.0 - self._visiting_param))
1594
1595     den = np.exp((self._visiting_param - 1.0) * np.log(np.fabs(y)) /
1596         (3.0 - self._visiting_param))
1597
1598     return x / den
1599
1600 class EnergyState:
1601     """
1602     Class used to record the energy state. At any time, it knows what is
1603     the
1604     currently used coordinates and the most recent best location.
1605
1606     Parameters
1607     -----
1608     lower : array_like
1609         A 1-D NumPy ndarray containing lower bounds for generating an initial
1610         random components in the `reset` method.
1611     upper : array_like
1612         A 1-D NumPy ndarray containing upper bounds for generating an initial
1613         random components in the `reset` method
1614     components. Neither NaN or inf are allowed.
1615     callback : callable, ``callback(x, f, context)`` , optional
1616         A callback function which will be called for all minima found.
1617         ``x`` and ``f`` are the coordinates and function value of the
1618         latest minimum found, and `context` has value in [0, 1, 2]
1619
1620     """
1621     # Maximum number of trials for generating a valid starting point
1622     MAX_REINIT_COUNT = 1000

```

```

1622 def __init__(self, lower, upper, callback=None):
1623     self.ebest = None
1624     self.current_energy = None
1625     self.current_location = None
1626     self.xbest = None
1627
1628     self.lower = lower
1629     self.upper = upper
1630     self.callback = callback
1631 def reset(self, func_wrapper, rand_gen, x0=None):
1632     """
1633     Initialize current location is the search domain. If `x0` is
1634     not
1635     provided, a random location within the bounds is generated.
1636     """
1637     if x0 is None:
1638         self.current_location = rand_gen.uniform(self.lower,
1639                                                  self.upper,
1640                                                  size=len(self.lower))
1641     else:
1642         self.current_location = func_wrapper.fun(self.current_location)
1643         if self.current_energy is None:
1644             init_error = True
1645             reinit_counter = 0
1646             while init_error:
1647                 if np.isfinite(self.current_energy) or np.isnan(
1648                     self.current_energy):
1649                     if reinit_counter >= EnergyState.MAX_REINIT_COUNT:
1650                         init_error = False
1651                         message = (
1652                             'Stopping algorithm because function '
1653                             'create NaN or (+/-) infinity values even with '
1654                             'trying new random parameters'
1655                         )
1656                 self.current_location = rand_gen.uniform(self.lower,
1657                                                         self.upper,
1658                                                         size=self.lower.size)
1659                 reinit_counter += 1
1660             else:
1661                 init_error = False
1662                 # If first time reset, initialize ebest and xbest
1663                 if self.ebest is None and self.xbest is None:
1664                     self.ebest = self.current_energy
1665                     self.xbest = np.copy(self.current_location)
1666                 # Otherwise, we keep them in case of reannealing reset
1667
1668 def update_best(self, e, x, context):
1669     self.ebest = e
1670     self.xbest = np.copy(x)
1671     if self.callback is not None:

```

```

1673         val = self.callback(x, e, context)
1674         if val is not None:
1675             if val:
1676                 return ('Callback function requested to stop early by '
1677                         'returning True')
1678
1679 def update_current(self, e, x):
1680     self.current_energy = e
1681     self.current_location = np.copy(x)
1682 class StrategyChain:
1683     """
1684     Class that implements within a Markov chain the strategy for
1685     location
1686     acceptance and local search decision making.
1687
1688     Parameters
1689     -----
1690     acceptance_param : float
1691         Parameter for acceptance distribution. It is used to control
1692         Instance of `VisitingDistribution` class.
1693     visit_dist : VisitingDistribution
1694         Instance of `VisitingDistribution` class.
1695     rand_gen : {None, int, `numpy.random.Generator`,
1696                 `numpy.random.RandomState`}
1697         If `seed` is None (or `np.random`), the
1698         `numpy.random.RandomState`
1699         singleton is used.
1700         If `seed` is an int, a new ``RandomState`` instance is used,
1701         If seeded with already a ``Generator`` or ``RandomState`` instance then
1702         that instance is used.
1703     energy_state: EnergyState
1704         Instance of `EnergyState` class.
1705
1706     """
1707
1708 def __init__(self, acceptance_param, visit_dist,
1709              func_wrapper, minimizer_wrapper, rand_gen, energy_state):
1710     # Local strategy chain minimum energy and location
1711     self.emin = energy_state.current_energy
1712     self.xmin =
1713     np.array(energy_state.current_location)
1714     # Global optimizer state
1715     self.energy_state = energy_state
1716     # Acceptance parameter
1717     self.acceptance_param = acceptance_param

```

```

1724     # Visiting distribution instance
1725     self.visit_dist = visit_dist
1726     # Wrapper to objective function
1727     self.func_wrapper = func_wrapper
1728     # Wrapper to the local minimizer
1729     self.minimizer_wrapper =
1730     minimizer_wrapper
1731     self.not_improved_idx = 0
1732     self.not_improved_max_idx = 1000
1733     self._rand_gen = rand_gen
1734     self.temperature_step = 0
1735     self.K = 100 * len(energy_state.current_location)

1736     def accept_reject(self, j, e, x_visit):
1737         r =
1738         self._rand_gen.uniform()
1739         1738         pqv = np.exp(np.log(1 - self.acceptance_param)
1740         1744         1. - self.acceptance_param))
1741         (e - self.energy_state.current_energy) /
1742         self.temperature_step)
1743         if r < 1738:
1744             # We accept the new location and update state
1745             if pqv <=
1746             self.energy_state.update_current(e, x_visit)
1747             1748             self.xmin = np.copy(self.energy_state.current_location)
1749             1749             1792
1750             else:
1751             # No improvement for a long time
1752             if self.not_improved_idx >= self.not_improved_max_idx:
1753             if j < self.min(self.energy_state.current_energy, self.emin):
1754             self.xmin = np.copy(self.energy_state.current_location)
1755             1756
1757     def run(self, step, temperature):
1758         self.temperature_step = temperature / float(step + 1)

1759         self.not_improved_idx += 1
1760         for i in range(self.energy_state.current_location.size * 2):
1761             if step == 0:
1762                 self.energy_state_improved = True
1763             else:
1764                 self.energy_state_improved =
1765                 False
1766                 x_visit =
1767                 self.visit_dist.visiting(
1768                 1815                 self.energy_state.current_location, j,
1769                 temperature)
1770                 1816
1771                 # Calling the objective
1772                 function
1773                 1817
1774                 e =
1775                 self.func_wrapper.fun(x_visit)
1776                 1818
1777                 if e <
1778                 self.energy_state.current_energy:
1779                 1819
1780                 # We have got a better energy value
1781
1782                 self.energy_state.update_current(e, x_visit)

```

```

1775         if val is not None:
1776             if val:
1777                 return val
1778                 self.energy_state_improved = True
1779                 self.not_improved_idx = 0
1780             else:
1781                 # We have not improved but do we accept the new
1782                 location?
1783                 self.accept_reject(j, e, x_visit)
1784                 if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
1785                     return ('Maximum number of function call reached '
1786                     'during
1787                     annealing') # End of
1788                     StrategyChain loop

1789     def local_search(self):
1790         # Decision making for performing a local search

1791         # Global energy has improved, let's see if LS improves further
1792         e, x =
1793         self.minimizer_wrapper.local_search(chain(results,
1794         self.energy_state.ebest,
1795         self.energy_state.ebest
1796         ))
1797         # If energy has been improved or no improvement since too
1798         long,
1799         if e < self.energy_state.ebest:
1800             # performing a local search with the best strategy chain
1801             location val = self.energy_state.update_best(e, x,
1802             self.energy_state_improved:
1803             if val is not None:
1804                 if val:
1805                     self.energy_state.update_current(e, x)
1806                     if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
1807                         return ('Maximum number of function call reached '
1808                         'during local search')
1809                     # Check probability of a need to perform a LS even if no
1810                     improvement
1811                     do_ls = False
1812                     if pls = np.exp(self.K * (self.energy_state.current_location):
1813                     self.energy_state.ebest - self.energy_state.current_energy) /
1814                     self.temperature_step)
1815                     if pls >= self._rand_gen.uniform():
1816                         do_ls = True

1817         # Global energy not improved, let's see what LS gives

1818         # on the best strategy chain location

1819         if self.not_improved_idx >= self.not_improved_max_idx:

1820             do_ls = True

1821         if do_ls:

1822         e, x = self.minimizer_wrapper.local_search(self.xmin,
1823         self.emin)
1824         self.xmin = np.copy(x)

```

```

1824         self.not_improved_max_idx =
self.energy_state.current_location.size
1825         if e < self.energy_state.ebest:
1826             val = self.energy_state.update_best(
1827                 self.emin, self.xmin, 2)
1828             if val is not None:
1829                 if val:
1830                     return val
1831             self.energy_state.update_current(e, x)
1832         if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
1833             return ('Maximum number of function call reached '
1834                     'during dual annealing')
1835
1836 class ObjectiveFunWrapper:
1837
1838     def __init__(self, func, maxfun=1e7, *args):
1839
1840         self.func = func
1841         self.args = args
1842         # Number of objective function evaluations
1843         self.nfev = 0
1844         # Number of gradient function evaluation if used
1845         self.ngev = 0
1846         # Number of hessian of the objective function if used
1847         self.nhev = 0
1848         self.maxfun = maxfun
1849
1850     def fun(self, x):
1851         self.nfev += 1
1852         return self.func(x, *self.args)
1853
1854 class LocalSearchWrapper:
1855     """
1856     Class used to wrap around the minimizer used for local search
1857     Default local minimizer is SciPy minimizer L-BFGS-B
1858     """
1859     LS_MAXITER_RATIO = 6
1860     LS_MAXITER_MIN = 100
1861     LS_MAXITER_MAX = 1000
1862
1863     def __init__(self, search_bounds, func_wrapper, *args, **kwargs):
1864         self.func_wrapper = func_wrapper
1865         self.kwargs = kwargs
1866         self.jac = self.kwargs.get('jac', None)
1867         self.minimizer = minimize
1868         bounds_list = list(zip(*search_bounds))
1869         self.lower = np.array(bounds_list[0])
1870         self.upper = np.array(bounds_list[1])
1871
1872         # If no minimizer specified, use SciPy minimize with 'L-BFGS-B'
1873         method
1874         if not self.kwargs:

```

```

1874         n = len(self.lower)
1875         ls_max_iter = min(max(n * self.LS_MAXITER_RATIO,
1876                               self.LS_MAXITER_MIN),
1877                             self.LS_MAXITER_MAX)
1878         self.kwargs['method'] = 'L-BFGS-B'
1879         self.kwargs['options'] = {
1880             'maxiter': ls_max_iter,
1881         }
1882         self.kwargs['bounds'] = list(zip(self.lower, self.upper))
1883         if callable(self.jac):
1884             def wrapped_jac(x):
1885                 return self.jac(x, *args)
1886             self.kwargs['jac'] = wrapped_jac
1887
1888     def local_search(self, x, e):
1889         # Run local search from the given x location where energy value
1890         # is e
1891         x_tmp = np.copy(x)
1892         mres = self.minimizer(self.func_wrapper.fun, x, **self.kwargs)
1893         if 'njev' in mres:
1894             self.func_wrapper.ngev += mres.njev
1895         if 'nhev' in mres:
1896             self.func_wrapper.nhev += mres.nhev
1897         # Check if is valid value
1898         is_finite = np.all(np.isfinite(mres.x)) and np.isfinite(mres.fun)
1899         in_bounds = np.all(mres.x >= self.lower) and np.all(
1900             mres.x <= self.upper)
1901         is_valid = is_finite and in_bounds
1902
1903         # Use the new point only if it is valid and return a better
1904         # results
1905         if is_valid and mres.fun < e:
1906             return mres.fun, mres.x
1907         else:
1908             return e, x_tmp
1909
1910 from process_annealing import pbar as pbar2
1911 def dual_annealing(func, bounds, args=(), maxiter=1000,
1912                   minimizer_kwargs=None, initial_temp=5230.,
1913                   restart_temp_ratio=2.e-5, visit=2.62, accept=-5.0,
1914                   maxfun=1e7, seed=None, no_local_search=False,
1915                   callback=None, x0=None):
1916     """
1917     Find the global minimum of a function using Dual Annealing.
1918
1919     Parameters
1920     -----
1921     func : callable
1922         The objective function to be minimized. Must be in the form
1923         ``f(x, *args)``, where ``x`` is the argument in the form of a 1-D
1924         array
1925         and ``args`` is a tuple of any additional fixed parameters
1926         needed to
1927         completely specify the function.

```



```

1924     bounds : sequence or `Bounds`
1925         Bounds for variables. There are two ways to specify the bounds:
1926
1927 Instance of `Bounds` class.
1928 Sequence of ``(min, max)`` pairs for each element in `x`.
1929
1930     args : tuple, optional
1931         Any additional fixed parameters needed to completely specify the
1932         objective function.
1933     maxiter : int, optional
1934         The maximum number of global search iterations. Default value is 1000.
1935     minimizer_kwargs : dict, optional
1936         Extra keyword arguments to be passed to the local minimizer
1937         (``minimize``). Some important options could be:
1938         ``method`` for the minimizer method to use and ``args`` for
1939         objective function additional arguments.
1940     initial_temp : float, optional
1941         The initial temperature, use higher values to facilitates a wider
1942         search of the energy landscape, allowing dual_annealing to escape
1943         local minima that it is trapped in. Default value is 5230. Range is
1944         (0.01, 5.e4].
1945     restart_temp_ratio : float, optional
1946         During the annealing process, temperature is decreasing, when it
1947         reaches ``initial_temp * restart_temp_ratio``, the reannealing process
1948         is triggered. Default value of the ratio is 2e-5. Range is (0, 1).
1949     visit : float, optional
1950
1951         Parameter for visiting distribution. Default value is 2.62. Higher
1952         values give the visiting distribution a heavier tail, this makes
1953         the algorithm jump to a more distant region. The value range is (1,
1954         3].
1955     accept : float, optional
1956         Parameter for acceptance distribution. It is used to control the
1957
1958         probability of acceptance. The lower the acceptance parameter, the
1959
1960         smaller the probability of acceptance. Default value is -5.0 with
1961
1962         a range (-1e4, -5].
1963     maxfun : int, optional
1964
1965         Soft limit for the number of objective function calls. If the
1966
1967         algorithm is in the middle of a local search, this number will be
1968         exceeded, the algorithm will stop just after the local search is
1969         done. Default value is 1e7.
1970     seed : {None, int, `numpy.random.Generator`, `numpy.random.RandomState`},
1971         optional
1972         If `seed` is None (or `np.random`), the `numpy.random.RandomState`
1973         singleton is used.
1974         If `seed` is an int, a new ``RandomState`` instance is used,
1975         seeded with `seed`.
1976         If `seed` is already a ``Generator`` or ``RandomState`` instance then
1977         that instance is used.
1978         Specify `seed` for repeatable minimizations. The random numbers
1979         generated with this seed only affect the visiting distribution
1980
1981 function

```

```

1972         and new coordinates generation.
1973     no_local_search : bool, optional
1974         If `no_local_search` is set to True, a traditional Generalized
1975         Simulated Annealing will be performed with no local search
1976         strategy applied.
1977     callback : callable, optional
1978         A callback function with signature ``callback(x, f, context)``,
1979         which will be called for all minima found.
1980         ``x`` and ``f`` are the coordinates and function value of the
1981         latest minimum found, and ``context`` has value in [0, 1, 2], with the
1982         following meaning:
1983
1984         - 0: minimum detected in the annealing process.
1985         - 1: detection occurred in the local search process.
1986         - 2: detection done in the dual annealing process.
1987
1988     If the callback implementation returns True, the algorithm will stop.
1989     x0 : ndarray, shape(n,), optional
1990         Coordinates of a single N-D starting point.
1991
1992 Returns
1993 -----
1994     res : OptimizeResult
1995         The optimization result represented as a `OptimizeResult` object.
1996         Important attributes are: ``x`` the solution array, ``fun`` the value
1997         of the function at the solution, and ``message`` which describes the
1998         cause of the termination.
1999         See `OptimizeResult` for a description of other attributes.
2000
2001 Notes
2002 ----
2003     This function implements the Dual Annealing optimization. This
2004     stochastic
2005     approach derived from [3]_ combines the generalization of CSA
2006     (Classical
2007     Simulated Annealing) and FSA (Fast Simulated Annealing) [1]_ [2]_
2008     coupled
2009     to a strategy for applying a local search on accepted locations [4]_.
2010     An alternative implementation of this same algorithm is described in
2011     [5]_
2012     and benchmarks are presented in [6]_. This approach introduces an
2013     advanced
2014     method to refine the solution found by the generalized annealing
2015     process. This algorithm uses a distorted Cauchy-Lorentz visiting
2016     distribution, with its shape controlled by the parameter :math:`q_{\{v\}}`
2017
2018     .. math::
2019
2020         g_{\{q_{\{v\}}\}}(\Delta x(t)) \propto \frac{1}{\left[T_{\{q_{\{v\}}\}}(t) \right]^{-\frac{D}{3-q_{\{v\}}}} \left\{ \frac{1+(q_{\{v\}}-1)\Delta x(t)^2}{\left[T_{\{q_{\{v\}}\}}(t) \right]^{\frac{2}{3-q_{\{v\}}}}} \right\}^{\frac{1}{q_{\{v\}}-1} + \frac{D-1}{2}}}
2021
2022     Where :math:`t` is the artificial time. This visiting distribution is
2023     used
2024     to generate a trial jump distance :math:`\Delta x(t)` of variable

```



```

2023 :math:`x(t)` under artificial temperature :math:`T_{q_v}(t)`.
2024
2025 From the starting point, after calling the visiting distribution
2026 function, the acceptance probability is computed as follows:
2027
2028 .. math::
2029
2030     p_{q_a} = \min\{\{1, \left[1-(1-q_a) \beta \Delta E \right]^{\frac{1}{1-q_a}}\}\}
2031
2032 Where :math:`q_a` is a acceptance parameter. For :math:`q_a < 1`, zero
2033 acceptance probability is assigned to the cases where
2034
2035 .. math::
2036
2037     [1-(1-q_a) \beta \Delta E] < 0
2038
2039 The artificial temperature :math:`T_{q_v}(t)` is decreased according to
2040
2041 .. math::
2042
2043     T_{q_v}(t) = T_{q_v}(1) \frac{2^{q_v-1}-1}{\left[2^{q_v-1}-1\right] + t}
2044 + t \frac{2^{q_v-1}-1}{\left[2^{q_v-1}-1\right]}
2045
2046 Where :math:`q_v` is the visiting parameter.
2047
2048 .. versionadded:: 1.2.0
2049
2050 References
2051 -----
2052 .. [1] Tsallis C. Possible generalization of Boltzmann-Gibbs
2053 statistics. Journal of Statistical Physics, 52, 479-487 (1998).
2054 .. [2] Tsallis C, Stariolo DA. Generalized Simulated Annealing.
2055 Physica A, 233, 395-406 (1996).
2056
2057 .. [3] Xiang Y, Sun DY, Fan W, Gong XG. Generalized Simulated
2058 Annealing Algorithm and Its Application to the Thomson Model.
2059 Physics Letters A, 233, 216-220 (1997).
2060 .. [4] Xiang Y, Gong XG. Efficiency of Generalized Simulated
2061 Annealing. Physical Review E, 62, 4473 (2000).
2062 .. [5] Xiang Y, Gubian S, Suomela B, Hoeng J. Generalized
2063 Simulated Annealing for Efficient Global Optimization: the GenSA
2064 Package for R. The R Journal, Volume 5/1 (2013).
2065 .. [6] Mullen, K. Continuous Global Optimization in R. Journal of
2066 Statistical Software, 60(6), 1 - 45, (2014).
2067 :doi:`10.18637/jss.v060.i06`
2068
2069 Examples
2070 -----
2071 The following example is a 10-D problem, with many local minima.
2072 The function involved is called Rastrigin

```

```

2073 (https://en.wikipedia.org/wiki/Rastrigin_function)
2074
2075 >>> import numpy as np
2076 >>> from scipy.optimize import dual_annealing
2077 >>> func = lambda x: np.sum(x*x - 10*np.cos(2*np.pi*x)) +
2078 10*np.size(x)
2079 >>> lw = [-5.12] * 10
2080 >>> up = [5.12] * 10
2081 >>> ret = dual_annealing(func, bounds=list(zip(lw, up)))
2082 >>> ret.x
array([-4.26437714e-09, -3.91699361e-09, -1.86149218e-09, -3.97165720e-
09,
-6.29151648e-09, -6.53145322e-09, -3.93616815e-09, -6.55623025e-09,
-6.05775280e-09, -5.00668935e-09]) # random
>>> ret.fun
0.000000
"""
pbar2.write("Starting dual annealing...")

if isinstance(bounds, Bounds):
    bounds = new_bounds_to_old(bounds.lb, bounds.ub, len(bounds.lb))

if x0 is not None and not len(x0) == len(bounds):
    raise ValueError('Bounds size does not match x0')

lu = list(zip(*bounds))
lower = np.array(lu[0])
upper = np.array(lu[1])
# Check that restart temperature ratio is correct
if restart_temp_ratio <= 0. or restart_temp_ratio >= 1.:
    raise ValueError('Restart temperature ratio has to be in range (0,
1)')

# Checking bounds are valid
if (np.any(np.isinf(lower)) or np.any(np.isinf(upper)) or np.any(
np.isnan(lower)) or np.any(np.isnan(upper))):
    raise ValueError('Some bounds values are inf values or nan
values')

# Checking that bounds are consistent
if not np.all(lower < upper):
    raise ValueError('Bounds are not consistent min < max')

# Checking that bounds are the same length
if not len(lower) == len(upper):
    raise ValueError('Bounds do not have the same dimensions')

# Wrapper for the objective function
func_wrapper = ObjectiveFunWrapper(func, maxfun, *args)

# minimizer_kwargs has to be a dict, not None
minimizer_kwargs = minimizer_kwargs or {}

minimizer_wrapper = LocalSearchWrapper(
    bounds, func_wrapper, *args, **minimizer_kwargs)

```

```

2123 # Initialization of random Generator for reproducible runs if seed
provided
2124 rand_state = check_random_state(seed)
2125 # Initialization of the energy state
2126 energy_state = EnergyState(lower, upper, callback)
2127 energy_state.reset(func_wrapper, rand_state, x0)
2128 # Minimum value of annealing temperature reached to perform
2129 # re-annealing
2130 temperature_restart = initial_temp * restart_temp_ratio
2131 # VisitingDistribution instance
2132 visit_dist = VisitingDistribution(lower, upper, visit, rand_state)
2133 # Strategy chain instance
2134 strategy_chain = StrategyChain(accept, visit_dist, func_wrapper,
2135                               minimizer_wrapper, rand_state,
energy_state)
2136 need_to_stop = False
2137 iteration = 0
2138 message = []
2139 # OptimizeResult object to be returned

2140 optimize_res = OptimizeResult()
2141 optimize_res.success = True
2142 optimize_res.status = 0
2143
2144 t1 = np.exp((visit - 1) * np.log(2.0)) - 1.0
2145 # Run the search loop
2146 start_time = time.time()
2147 pbar2.write(f"Starting search at {time.strftime('%H:%M:%S',
time.localtime())}")
2148
2149 while not need_to_stop:
2150     for i in range(maxiter):
2151         # Compute temperature for this step
2152         s = float(i) + 2.0
2153         t2 = np.exp((visit - 1) * np.log(s)) - 1.0
2154         temperature = initial_temp * t1 / t2
2155         if iteration >= maxiter:
2156             message.append("Maximum number of iteration reached")
2157             need_to_stop = True
2158             break
2159         # Need a re-annealing process?
2160         if temperature < temperature_restart:
2161             energy_state.reset(func_wrapper, rand_state)
2162             break
2163         # starting strategy chain
2164         val = strategy_chain.run(i, temperature)
2165         if val is not None:
2166             message.append(val)
2167             need_to_stop = True
2168             optimize_res.success = False
2169             break
2170 # Possible local search at the end of the strategy chain

```

```

2171 if not no_local_search:
2172     val = strategy_chain.local_search()
2173     if val is not None:
2174         message.append(val)
2175         need_to_stop = True
2176         optimize_res.success = False
2177         break
2178     iteration += 1
2179     pbar.update(1)
2180     rate = pbar.format_dict['rate']
2181     if rate is None:
2182         r.append(0)
2183     else:
2184         r.append(1/rate)
2185
2186 # print("iteration: ", iteration)
2187 # elapsed_time = time.time_ns() - start_time
2188 # elapsed_time_seconds = elapsed_time / 1e9
2189 # estimated_time = elapsed_time_seconds / iteration *
(maxiter -
iteration)
2190 # estimated_finish_time = time.strftime('%H:%M:%S',
2191 # time.localtime(time.time() + estimated_time))
2192 # print(f"ETA: {estimated_finish_time}")
2193
2194 # Setting the OptimizeResult values
2195 optimize_res.x = energy_state.xbest
2196 optimize_res.fun = energy_state.ebest
2197 optimize_res.nit = iteration
2198 optimize_res.nfev = func_wrapper.nfev
2199 optimize_res.njev = func_wrapper.njev
2200 optimize_res.nhev = func_wrapper.nhev
2201 optimize_res.message = message
2202
2203 pbar2.write(f"Finished search at {time.ctime()} in\
2204 {time.strftime('%H:%M:%S',time.gmtime(time.time() -
start_time))}")
2205
2206 return optimize_res

```