# Optimisation informatique de la trajectoire d’une voiture sur un circuit

# INTRODUCTION

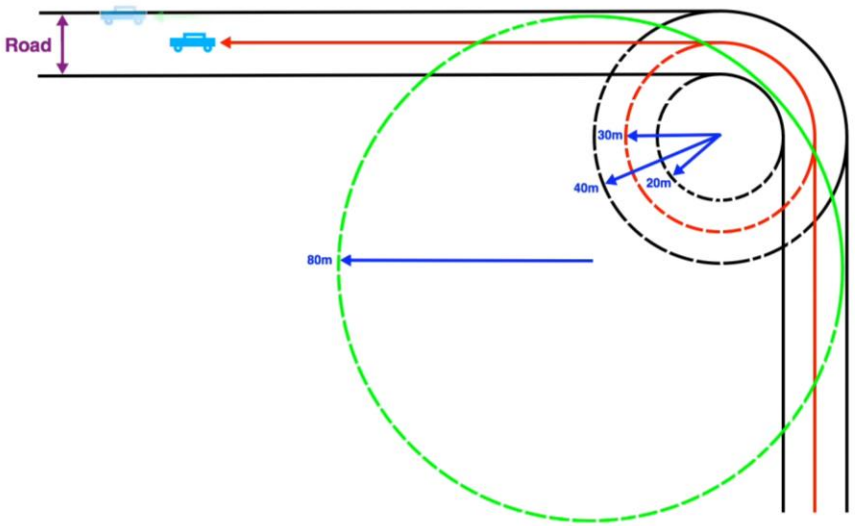
Comment choisir les paramètres de l'algorithme PSO pour obtenir une ligne de course optimale ?


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

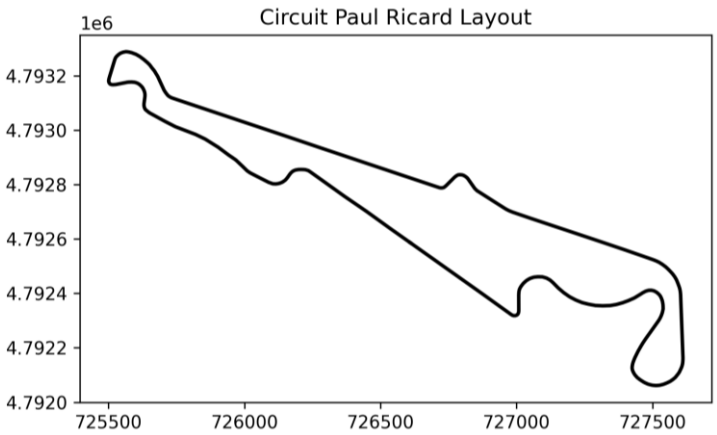
Sur une trajectoire circulaire uniforme on a :
$a = v^2/r$

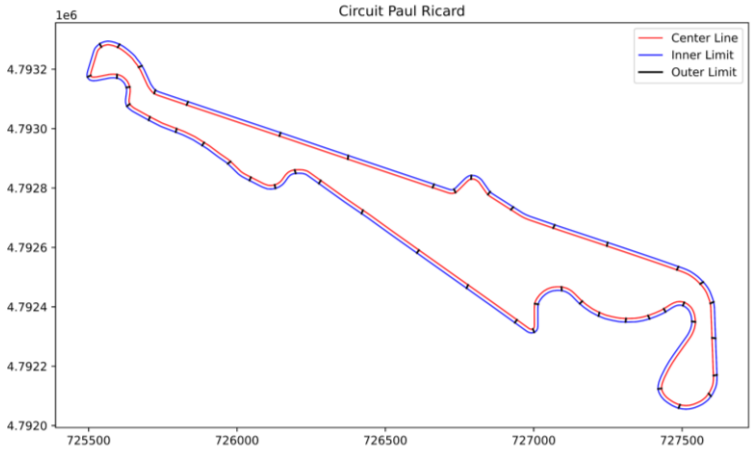
+ le rayon est grand, + on va vite

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

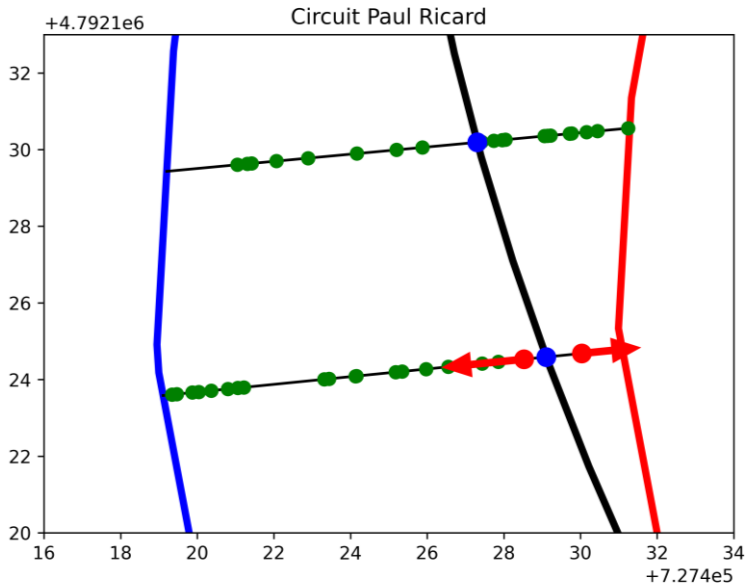
# MODÉLISATION DU CIRCUIT

GeoJson

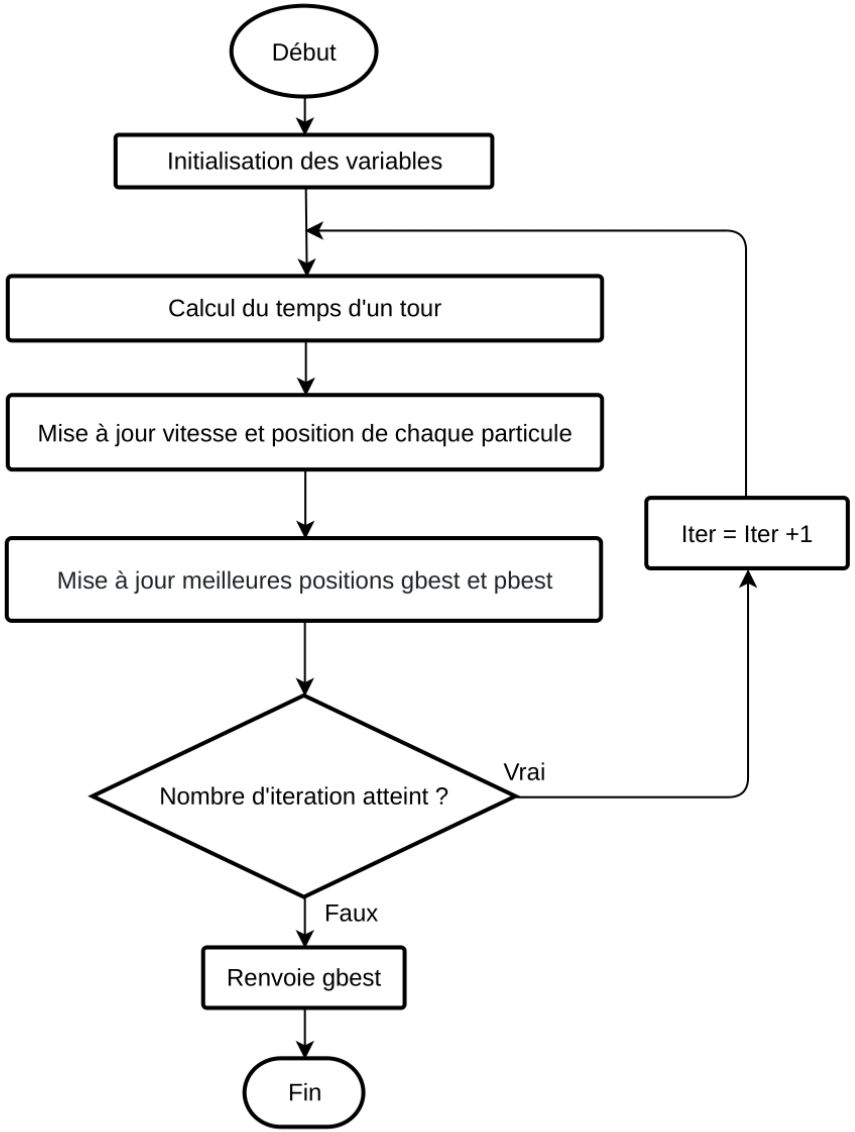


# PARTICLE SWARM OPTIMISATION (PSO)




Particules placées de manière uniforme le long de chaque secteur



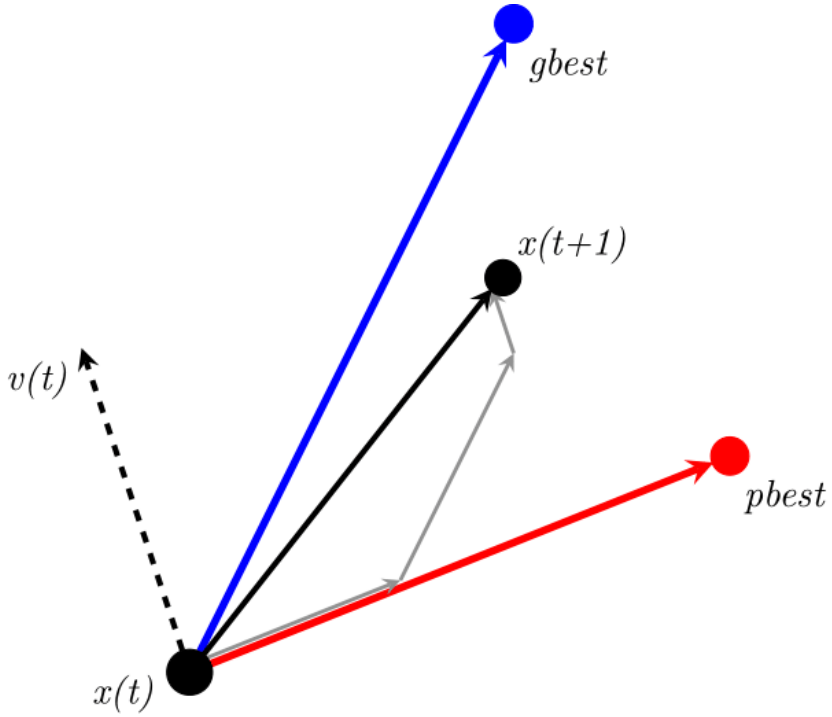
# LES PARAMÈTRES

| **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

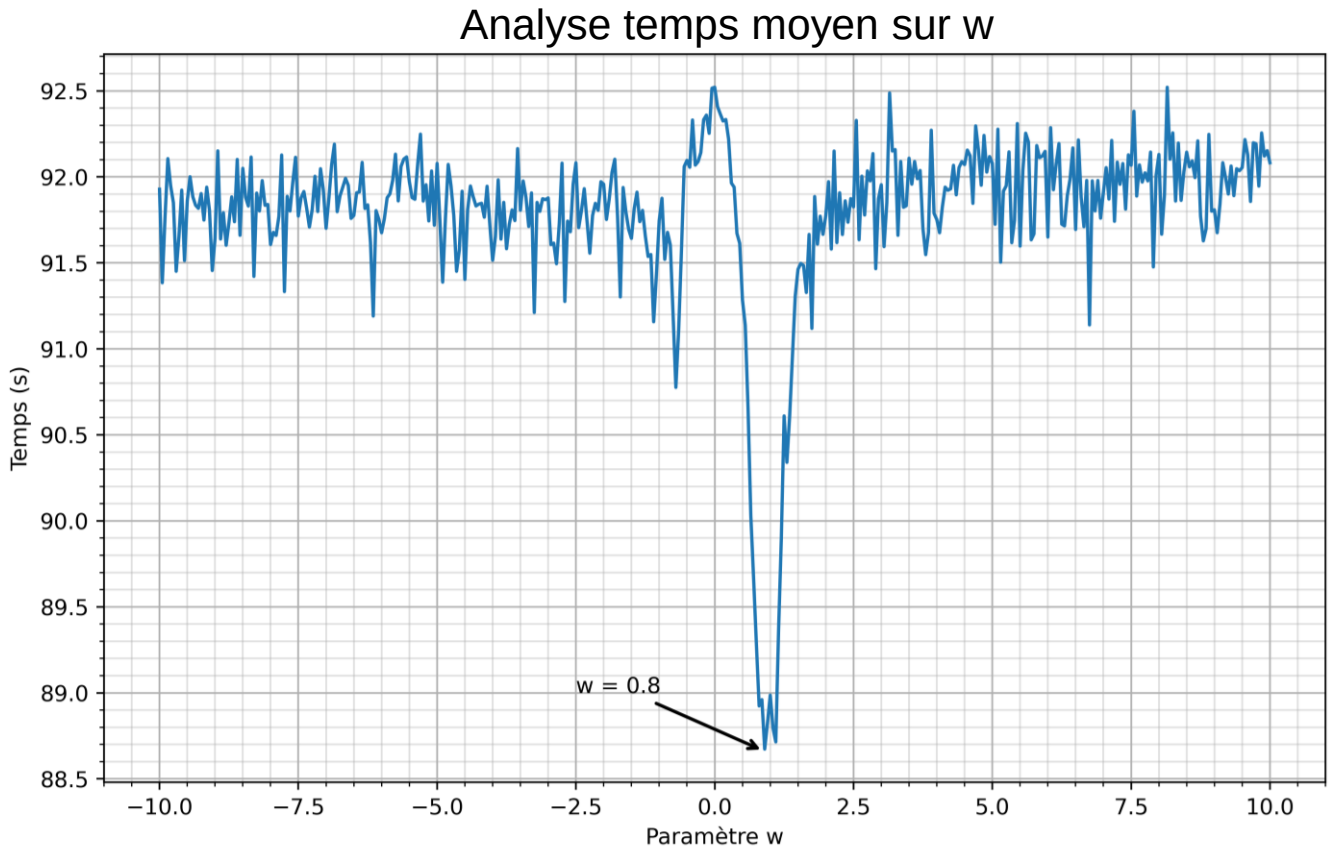
$$v(t+1) = \boxed{w} * v(t) + \boxed{c1} * r1\left(p_{best} - x(t)\right) + \boxed{c2} * r2(g_{best} - x(t)) \quad [1]$$

Inconnues à évaluer



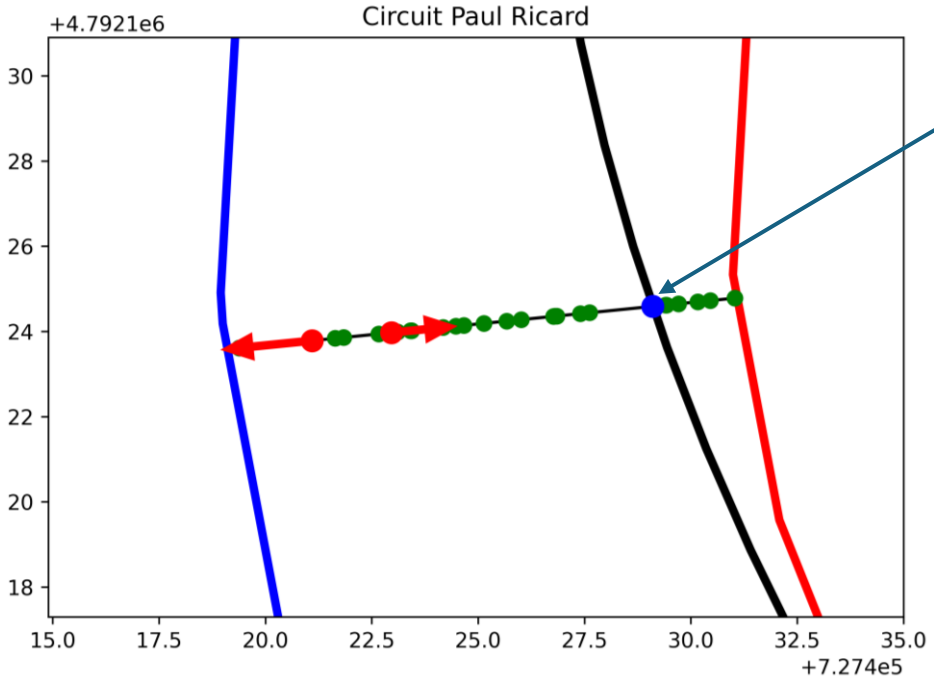
# LE PARAMÈTRE D'INERTIE

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



# GBEST

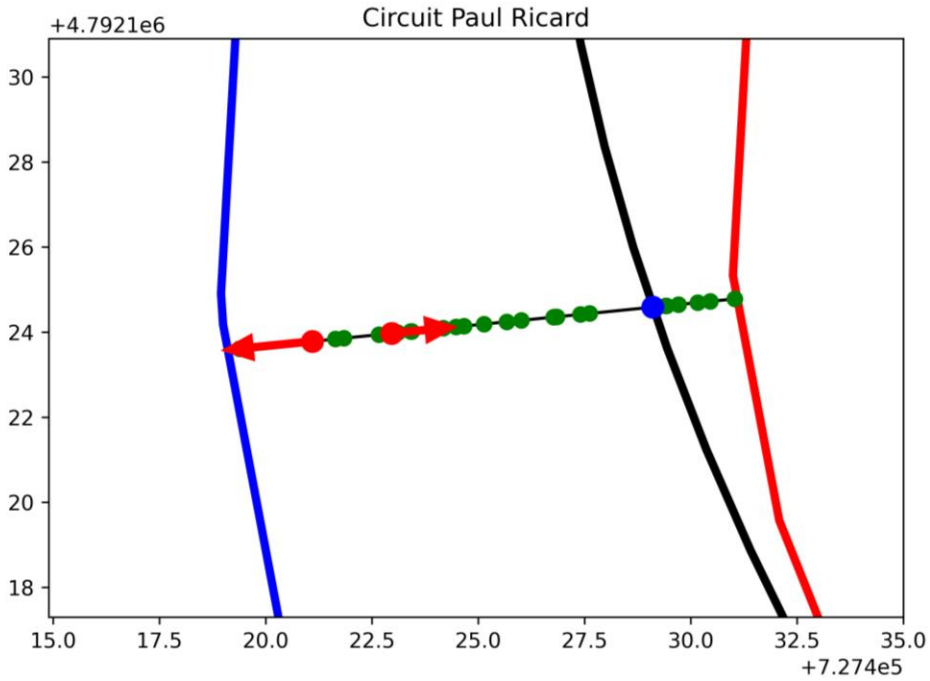
$$v(t+1) = w * v(t) + c1 * r1\left(p_{best} - x(t)\right) + \boxed{c2 * r2\left(g_{best} - x(t)\right)} \quad [1]$$




Meilleure solution trouvée entre toutes les particules

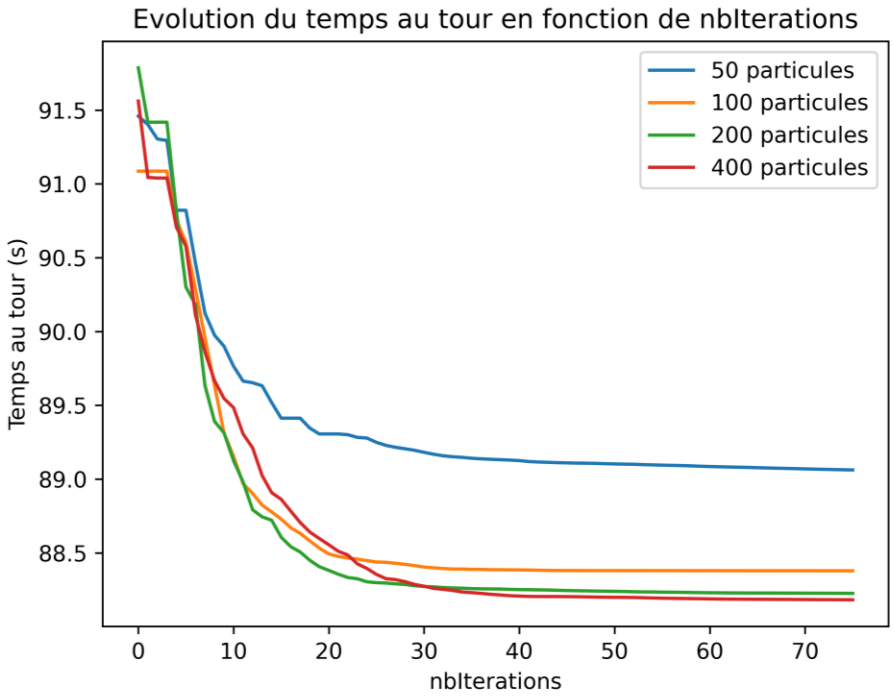
# PBEST

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




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

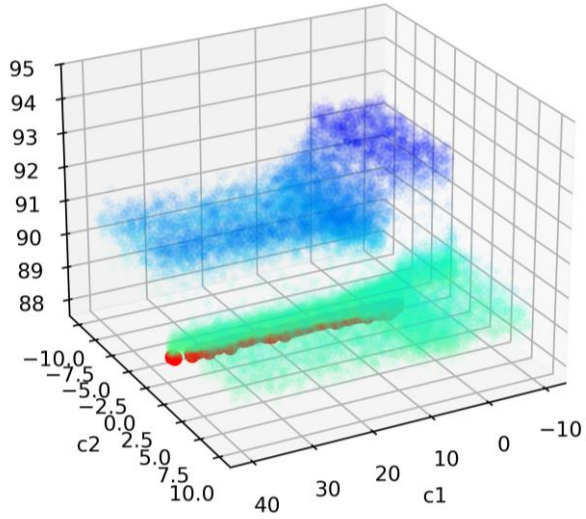
# ESTIMATION DE C1 et C2

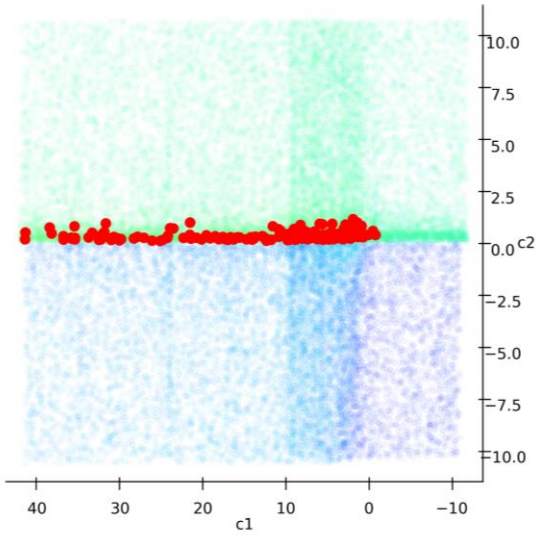
Recherche d’un minimum sur un plan

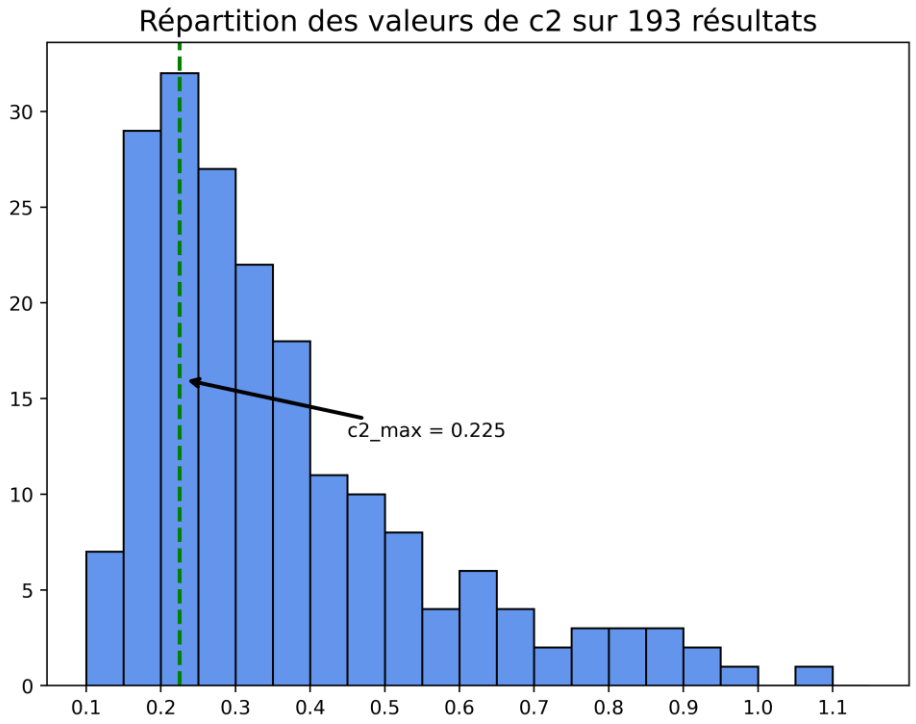
| | ⚠ Recherche exhaustive | ✅ Recuit Simulé |
|---|---|---|
| Temps total | 6,4 ans | 32h |
| Temps multiprocessé | 876h | 32h |
| Nombre de résultats | 1 | 64 |

# RÉSULTATS DES RECHERCHES

Analyse des temps sur le plan



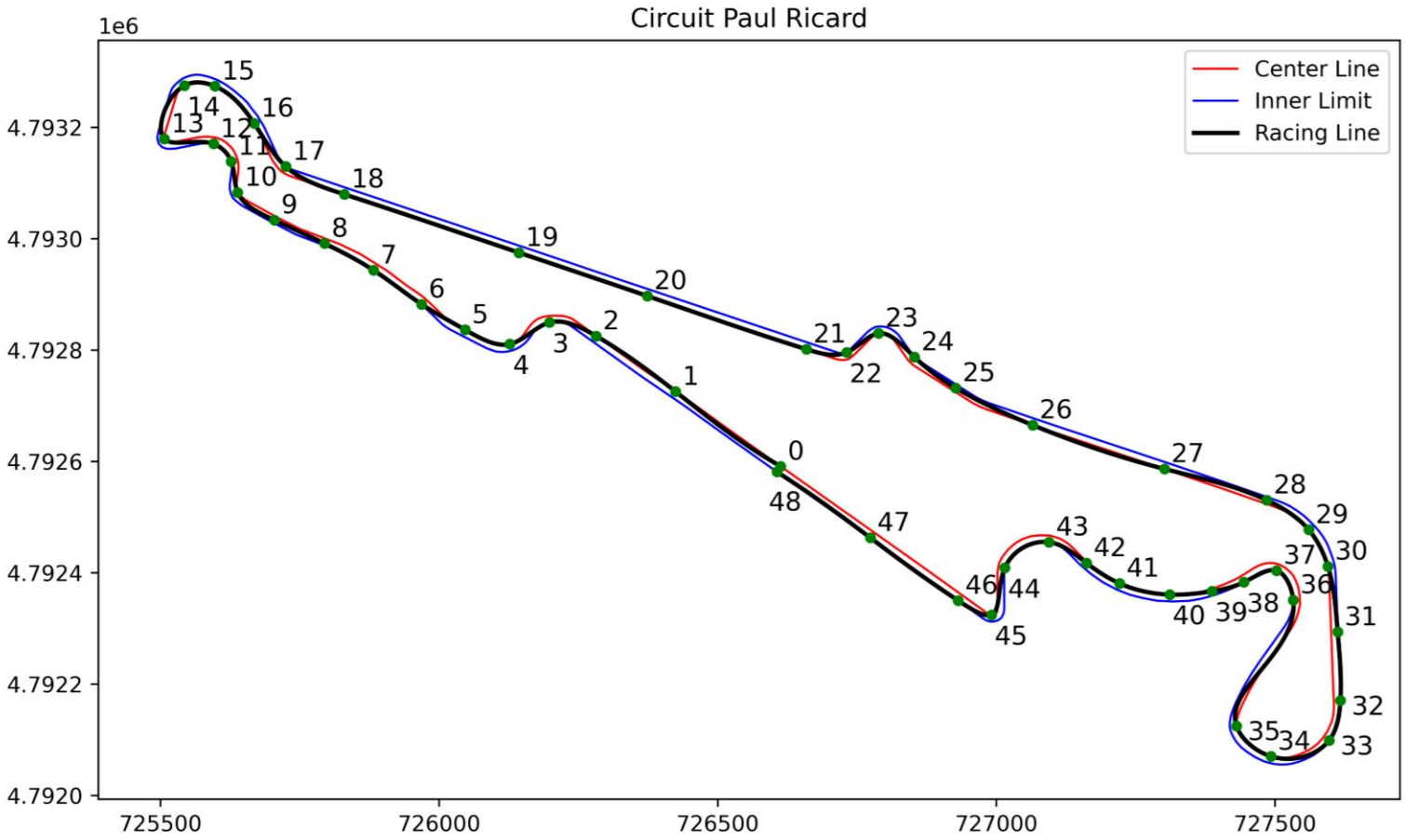



# RÉSULTATS PARAMÈTRES

| |
|---|
| Nombre de secteur = 50 |
| Nombre d’itérations = 50 |
| Nombre de particules<br>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 = v²/r

*Considérons une particule en mouvement circulaire uniforme de rayon r*

*On a l'accelération en coordonnées polaires*:

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

*Or comme r est constant* : $\dot{r} = 0$ *et* $\ddot{r} = 0$
*De plus on a* $\dot{\theta}$ *qui est contant donc* $\ddot{\theta} = 0$
*Donc* : $\vec{a} = -\mathrm{r}\dot{\theta}^2\vec{u}_r$
*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* :

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

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

*Enfin* :

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

# Preuve v_max

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

$De\ plus\ sans\ glissement\ la\ force\ tangentiel\ de\ frottement\ s'ecrit :$

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

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

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

$Donc :$

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

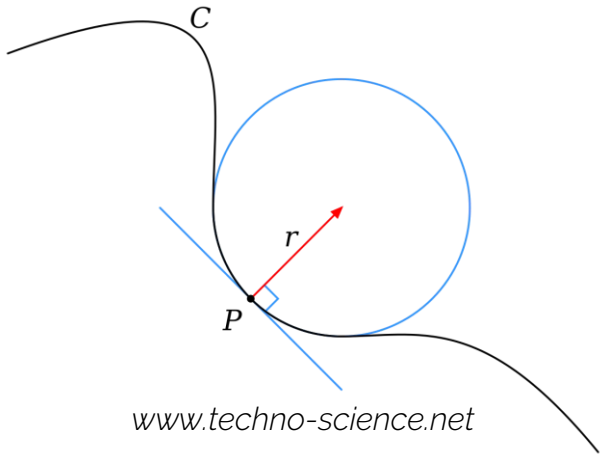
$$\boxed{v_{max} = \sqrt{\mu_{track} * g * r}}$$

# 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

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




$$\begin{cases} v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1\left(p_i^t - x_i^t\right) + c\_2 \cdot r_2\left(g^t - x_i^t\right)\ (1) \\ x_i^{t+1} = x_i^t + v_i^{t+1}\Delta t \qquad (2) \end{cases}$$

# 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

```python
"""
Author: Xayon
file: tipe.py
"""

import numpy as np
import geopandas as gpd
import matplotlib.pyplot as plt
from shapely import LinearRing, get_coordinates, shortest_line, Point
import math
from scipy import interpolate
import time
from pyproj import Proj
from tqdm import tqdm
import os
import matplotlib

# Constants
WIDTH = 12
A = 6378137 # m
F= 1/298.257223563

class Track:

    def __init__(self, filename):

        self.nb_sectors = 70
        self.sectors = []

        self.filename = filename
        self.trackData = gpd.read_file(filename)
        self.trackname = self.trackData['Name'][0]

        self.coords = None
        self.pos_x, self.pos_y = self.getCoords() # useless

        self.centerLine = None
        self.innerLimit = None
        self.outerLimit = None

        self.s_points = None
        self.inside_points = None
        self.outside_points = None

        self.boundaries = []

        self.test = []
        self.test2 = True

        self.generateTrackLimits()

        self.defineSector()

        self.getBoundaries()

        self.racingLine = None
        self.rl2 = None


    def getCoords(self):
        self.coords = interpolator(self.trackData['geometry'][0],1000).flatten()
        x = [self.coords[i] for i in range(len(self.coords)) if i % 2 == 0]
        y = [self.coords[i] for i in range(len(self.coords)) if i % 2 == 1]

        p = Proj(proj='utm',zone=31,ellps='WGS84', preserve_units=False)
        x,y = p(x,y)
        self.coords = np.array([x,y]).T.flatten()
        return x,y

    def plotTrack(self, centerLine=False, innerLimit=True, outerLimit=True,
                  sectors=False,racingLine =False,nb=False, points=False):

        plt.figure(figsize=(11.2,6.3))

        if centerLine:
            plt.plot(*self.centerLine.xy, color='black', linewidth=2)
        if innerLimit:
            # print(self.innerLimit.xy)
            plt.plot(*self.innerLimit.xy, color='red', linewidth=1)
        if outerLimit:
            # print(self.outerLimit.xy)
            plt.plot(*self.outerLimit.xy, color='blue', linewidth=1)
        if racingLine:
            plt.plot(*self.racingLine, color='black', linewidth=2)
        if sectors:
            for i in range(self.nb_sectors):
                plt.plot([self.outside_points[i][0],self.inside_points[i][0]],
                         [self.outside_points[i][1],self.inside_points[i][1]])
        if nb:
            for j in range(len(self.rl2[0])):
                if j == 0:
                    plt.text(self.rl2[0][j]-15,self.rl2[1][j]-34,
                        len(self.rl2[0])-j-1, fontsize=12,
                        horizontalalignment='left', verticalalignment='top')
                elif len(self.rl2[0])-j in [4,5,15,23,39,40,41,45,46]:
                    plt.text(self.rl2[0][j]+5,self.rl2[1][j]-18,
                        len(self.rl2[0])-j-1, fontsize=12,
                        horizontalalignment='left', verticalalignment='top')
                elif len(self.rl2[0])-j-1 in [32,33,35]:
                    plt.text(self.rl2[0][j]+20,self.rl2[1][j]+10,
                         len(self.rl2[0])-j-1, fontsize=12,
                         horizontalalignment='left', verticalalignment='top')
                else:
```

```
                plt.text(self.rl2[0][j]+12,self.rl2[1][j]+12,
                len(self.rl2[0])-j-1, fontsize=12)
    if points:
        plt.plot(self.rl2[0],self.rl2[1], "go", markersize=4)

    plt.gca().set_aspect('equal')
    plt.title(self.trackname)
    plt.legend(['Center Line', 'Inner Limit', 'Outer Limit'])
    plt.show()


def saveTrack(self, centerLine=False, innerLimit=True, outerLimit=True,
          sectors=False,racingLine =False,nb=False, points=False,
          transparent=False, path="Data/plot_output/"):
    try:
        os.mkdir(path)
    except:
        pass

    path_ = f"{path}{self.trackname}{'_racingline'if racingLine else ''}
        {'_centerline'if centerLine else ''}{'_nb' if nb else ''}
        {'_t' if transparent else ''}{'_s' if sectors else ''}.png"
    print(f"saving {path_} ... at {time.strftime('%H:%M:%S')}")

    plt.figure(figsize=(11.2,6.3))

    if centerLine:
        plt.plot(*self.centerLine.xy, color='black', linewidth=2)
    if innerLimit:
        # print(self.innerLimit.xy)
        plt.plot(*self.innerLimit.xy, color='red', linewidth=1)
    if outerLimit:
        # print(self.outerLimit.xy)
        plt.plot(*self.outerLimit.xy, color='blue', linewidth=1)
    if racingLine:
        plt.plot(*self.racingLine, color='black', linewidth=2)

    if sectors:
        for i in range(self.nb_sectors):
            plt.plot([self.outside_points[i][0],self.inside_points[i][0]],
                [self.outside_points[i][1],self.inside_points[i][1]],
                color='black')
    if nb:
        for j in range(len(self.rl2[0])):
            if j == 0:
                plt.text(self.rl2[0][j]-15,self.rl2[1][j]-34,
                len(self.rl2[0])-j-1, fontsize=12,
                horizontalalignment='left', verticalalignment='top')
            elif len(self.rl2[0])-j in [4,5,15,23,39,40,41,45,46]:
                plt.text(self.rl2[0][j]+5,self.rl2[1][j]-18,
                len(self.rl2[0])-j-1, fontsize=12,
                horizontalalignment='left', verticalalignment='top')
            elif len(self.rl2[0])-j-1 in [32,33,35]:
                plt.text(self.rl2[0][j]+20,self.rl2[1][j]+10,
                len(self.rl2[0])-j-1, fontsize=12,
                horizontalalignment='left', verticalalignment='top')
            else:
                plt.text(self.rl2[0][j]+12,self.rl2[1][j]+12,
                len(self.rl2[0])-j-1, fontsize=12)
    if points:
        plt.plot(self.rl2[0],self.rl2[1], "go", markersize=4)

    plt.gca().set_aspect('equal')
    plt.title(self.trackname)
    plt.legend(['Center Line', 'Inner Limit', 'Outer Limit'])
    plt.savefig(path_,dpi=600,transparent=transparent)
    plt.close('all')
    plt.clf()

def generateTrackLimits(self):
    self.centerLine = LinearRing(self.coords.reshape(-1, 2))
    # print(self.coords.reshape(-1,2))
    self.innerLimit = self.centerLine.buffer(WIDTH/2, join_style=2,
                                             mitre_limit=1).interiors[0]
    self.outerLimit = self.centerLine.buffer(WIDTH/2, join_style=2,
                                             mitre_limit=1).exterior


def defineSector(self):
    '''Defines sectors' search space

    Parameters
    ----------
    center_line : LineString
        Center line of the track
    inside_line : LineString
        Inside line of the track
    outside_line : LineString
        Outside line of the track
    n_sectors : int
        Number of sectors

    Returns
    -------
    inside_points : list
        List coordinates corresponding to the internal point of each sector
        segment
    outside_points :
    list
        List coordinates corresponding to the external point of each sector
        segment
    '''
    center_line =
    self.centerLine
    inside_line =
    self.innerLimit
    outside_line =
    self.outerLimit
```

```
        # sect.insert(0,0)

        # sect.insert(1,30)
        # sect.append(0)
        # print("sect : ",sect)
        sect = [0, 35, 69, 81, 95, 102, 114, 128, 140, 158, 172, 183, 196,
        207,
246, 262, 283, 293, 310, 326, 338, 355, 390, 435, 463, 479, 494,
506, 520, 575, 620, 680, 701, 716, 733, 745, 762, 780, 789, 797,
812, 830, 848, 867, 882, 899, 914, 930, 960, 0]
        self.nb_sectors = len(sect)
        n_sectors = self.nb_sectors

        distances = np.linspace(0, center_line.length, n_sectors)


        center_points = [self.pos_x[sect[i]] for i in range(len(sect))],
        [self.pos_y[sect[i]] for i in range(len(sect))]
        center_points = np.array(center_points).T

        distances = np.linspace(0,inside_line.length, 1000)
        inside_border = [inside_line.interpolate(distance)
                    for distance in distances]
        inside_border = np.array([[e.x, e.y] for e in inside_border])
        inside_points = np.array([get_closest_points([center_points[i][0],
                         center_points[i][1]], inside_border)
                         for i in range(len(center_points))])

        distances = np.linspace(0,outside_line.length, 1000)
        outside_border = [outside_line.interpolate(distance)
                         for distance in distances]
        outside_border = np.array([[e.x, e.y] for e in outside_border])
        outside_points = np.array([get_closest_points([inside_points[i][0],
                         inside_points[i][1]], outside_border)
                         for i in range(len(center_points))])


        self.inside_points = inside_points
        self.outside_points = outside_points

    def getBoundaries(self):
        """
        Calculates the boundaries of each sector by computing the Euclidean
        distance between the inside and outside points.
        Stores the boundaries in the 'boundaries' attribute of the object.
        """

        for i in range(self.nb_sectors):
            self.boundaries.append(np.linalg.norm(
                    self.inside_points[i]-self.outside_points[i]))
        # print(self.boundaries)
        self.boundaries = [12]*self.nb_sectors


    def getRacingLine(self, sectors):
        """
        Calculates the racing line for a given set of sectors.

        Args:
            sectors (list): A list of sectors.


        Returns:
          list: A list of points representing the racing line.
        """
        rl = []
        for i in range(len(sectors)):
            x1, y1 = self.inside_points[i][0], self.inside_points[i][1]
            x2, y2 = self.outside_points[i][0],
            self.outside_points[i][1]
            m = (y2-y1)/(x2-x1)

            a = math.cos(math.atan(m)) # angle with x axis
            b = math.sin(math.atan(m)) # angle with x axis

            xp = x1 - sectors[i]*a
            yp = y1 - sectors[i]*b

            if xp < min([x1, x2]) or xp > max([x1,x2]):
                xp = x1 + sectors[i]*a
                yp = y1 + sectors[i]*b

            rl.append([xp, yp])
            if self.test2:
                self.test.append([xp,yp])

        return rl


    next_path(path_pattern):
    def
    """
    Finds the next free path in an sequentially named list of files

    e.g. path_pattern = 'file-%s.txt':

    file-1.txt
    file-2.txt
    file-3.txt

    Runs in log(n) time where n is the number of existing files in
    sequence
    """
    i = 1

    # First do an exponential search
    while os.path.exists(path_pattern % i):
        i = i * 2

    # Result lies somewhere in the interval (i/2..i]
```

```
    # We call this interval (a..b] and narrow it down until a + 1 = b
    a, b = (i // 2, i)
    while a + 1 < b:
        c = (a + b) // 2 # interval midpoint
        a, b = (c, b) if os.path.exists(path_pattern % c) else (a, c)


    return path_pattern % b



def get_closest_points(point, array):

    """
    Returns the closest point(s) in an array to a given point.

     Args:
     point (tuple): A tuple containing the x and y coordinates of the point.
     array (list): A list of tuples, each containing the x and y coordinates
of
     a point.

     Returns:
     list: A list containing the x and y coordinates of the closest point(s)
in
     the array to the given point.
     """
     result = []
     distance = 1000
     for i in range(len(array)):
        temp = math.sqrt((point[0]-array[i][0])**2+(point[1]-array[i][1])**2)
        if temp<distance:
            distance = temp
            result = [array[i][0], array[i][1]]
    return result

def interpolator(linring, nb):
    """
    Interpolates a linear ring to obtain a specified number of points.

    Args:
        linring (shapely.geometry.LinearRing): The linear ring to interpolate.
        nb (int): The number of points to interpolate.
    length = linring.length
    step = length / nb
    Returns:
    points = [] ndarray: An array of interpolated points.
    """
    for i in range(nb):
        points.append(linring.interpolate(i*step).coords[0])
    return np.array(points)

def flatten(l):
    result = []
    for el in l:
        result.extend(el)
    return result

    dist(x1,y1,x2,y2):
def
    return math.sqrt((x1-x2)**2+(y1-y2)**2)



    ---------Particle Swarm Optimization algorithm implementation. --------
"""    -"""

class Particle:
    def __init__(self, dim, boundaries):

        self.position = []
        self.velocity = []

        self.b_position =  []


        for i in range(dim):

            # np.append(self.position,np.random.uniform(0,  boundaries[i]))
            self.position.append(np.random.uniform(0,  boundaries[i]))
            #  np.append(self.velocity,np.random.uniform(-boundaries[i],
            #             boundaries[i]))
            self.velocity.append(np.random.uniform(-boundaries[i],
                                                   boundaries[i]))

         self.position = np.array(self.position)
         self.velocity = np.array(self.velocity)

         self.b_position = self.position

      def setPosition(self,value):
         self.position = value

      def setVelocity(self,value):
         self.velocity = value

      def setBestPosition(self,value):
def pso(fitFunc,dim,boundaries, nbParticle, nbIter, w, a, b, track):
         self.b_position = value
    """
    Particle Swarm Optimization algorithm implementation.

    Parameters:
    fitFunc (function): The fitness function to optimize.
    dim (int): The number of dimensions of the search space.
```

```
    boundaries (list): A list of tuples representing the boundaries of the search space.
    nbParticle (int): The number of particles in the swarm.
    nbIter (int): The number of iterations to run the algorithm.
    w (float): The inertia weight.
    a (float): The cognitive parameter.
    b (float): The social parameter.
    track (Track): The track object to plot the particles' positions.

    Returns:
    tuple: A tuple containing the global best position, the global best fitness value, the history of global best positions, and the history of global best fitness values.
    """

    particles = []
    gs = []
    gs_e = []
    # gs_h = []
    # gs_he = []

    for i in range(nbParticle):
        particles.append(Particle(dim,boundaries))


    gs = particles[0].position
    gs_e = fitFunc(gs,track)
    for p in particles:
        p_eval = fitFunc(p.b_position,track)
        if p_eval < gs_e:
            gs = p.b_position
            gs_e = fitFunc(gs,track)

    # gs_h.append(gs)
    # gs_he.append(gs_e)
    with tqdm(total=nbIter*len(particles)*dim) as pbar:

        for t in range(nbIter):
            # for i in range(nbParticle):

            for p in particles:
                e1 = np.random.uniform(0,1)
                e2 = np.random.uniform(0,1)
                newPosition = []
                newVelocity = []
                for d in range(dim):

                    newVelocity.append(w * p.velocity[d]
                        + a * e1 * (p.b_position[d] - p.position[d] )
                        + b * e2 * ( gs[d] - p.position[d]))

                    # p.setVelocity(p.velocity + a*e1*(gs -  p.position)
                    #             + b*e2*(p.b_position - p.position))

                    # Check the particles will stay in between the boundaries
                    #  before computing the positions
                    if newVelocity[d] < -boundaries[d]:
                        newVelocity[d] = -boundaries[d]
                    elif newVelocity[d] > boundaries[d]:
                        newVelocity[d] = boundaries[d]

                    newPosition.append(p.position[d] + newVelocity[d])
                    # p.setPosition(p.position + p.velocity)

                    # Check if the particle is still in the boundaries
                    if newPosition[d] < 0.0 :
                        newPosition[d] = 0.0
                    elif  newPosition[d] > boundaries[d]:
                        newPosition[d] = boundaries[d]

                    pbar.update(1)

                p.setVelocity(newVelocity)
                p.setPosition(newPosition)

                pEval = fitFunc(p.position,track)
                if pEval < fitFunc(p.b_position,track):
                    p.setBestPosition(p.position)
                    if pEval < gs_e:
                        gs = p.position
                        gs_e = pEval

            # gs_h.append(gs)
            # gs_he.append(gs_e)
        pbar.close()

    return gs, gs_e

def fitFunc(sectors,track):
    """
    Calculates the lap time for a given set of sectors.

    Returns:
        float: The lap time for the given sectors.
    """

    # if track.test2:
        # print("sectors : ")
        # print(sectors)
        # print(len(sectors))
    a = getLapTime(track.getRacingLine(sectors))[0]
    track.test2 = False
    return a
```

```python
def getLapTime(racingLine):
    """
    Computes the lap time and the (x, y) coordinates of the racing line.

    Parameters:
    racingLine (list): A list of (x, y) coordinates of the racing line.


    Returns:
    tuple: A tuple containing the lap time and the (x, y) coordinates of the
    racing line.
    """
    rl = np.array(racingLine)


    # Find the spline
    tck, _ = interpolate.splprep([rl[:,0], rl[:,1]],s=0, k=3)
    # Evaluate the spline
    x, y = interpolate.splev(np.linspace(0, 1, 2000), tck)
    # Compute the derivative
    dx, dy = np.gradient(x), np.gradient(y)
    d2x, d2y = np.gradient(np.array(dx)), np.gradient(np.array(dy))

    k = np.abs(dx*d2y - dy*d2x) / np.power(dx*dx + dy*dy, 1.5)
    r = 1/k

    # μ = 0.7
    μ = 1.78
    # computing the max speed
    v = np.sqrt(μ * r * 9.81)
    v = np.clip(v,None, 95.66)
    # print(v)

    # computing the lap time
    lapTime = np.sum(np.sqrt(np.power(np.diff(x), 2)
                             + np.power(np.diff(y), 2)) / v[:-1])

    return lapTime, (x, y)

def main():
    """

    Main function to run the Particle Swarm Optimization algorithm.
    """

    nbpart = 400
    nbiter = 300
    path = f"Data/plot_output/{time.strftime('%d-%m-%Y')}/%s/"
    path = next_path(path)
    w = 0.8
    a = 1
    b = 0.2
    print(f"""
- nbIteration : {nbiter}
- nbParticule : {nbpart}
- w : {w}
- c1 : {a}
- c2 : {b}
        """)
    gs, gs_e = pso(fitFunc, track.nb_sectors, track.boundaries, nbpart,
nbiter,
                   w, a, b, track)
    rl = track.getRacingLine(gs)
    gs_e = getLapTime(rl)[0]
    track.racingLine = getLapTime(rl)[1]
    track.rl2 = [[rl[i][0] for i in range(len(rl))],
                 [rl[i][1] for i in range(len(rl))]]

    print(f"""
- Global best position : {gs}
- Lap time : {gs_e}
- Parameters : {w}, {a}, {b}
    """)

    path="Img/"
    track.plotTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
                    points=True)
    track.saveTrack(transparent=False, path=path, sectors=True)
    track.saveTrack(transparent=True, path=path, sectors=True)
    track.saveTrack(centerLine=False, transparent=True, path=path)
    track.saveTrack(centerLine=False, transparent=False, path=path)


    track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
                    points=True,transparent=True, path=path)
    track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
                    points=True,transparent=False, path=path)
    track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=False,
                    points=True,transparent=False, path=path)

    with open(f"{path}data.txt","w") as file:
        file.write(str([gs_e,gs,rl]))
        file.close()

    main2():
def
    """
    Main function to run the Particle Swarm Optimization algorithm
    multiple
    times.
    """
    track = Track("Data/Track_data/cbg.geojson")
    nbpart = 50
    nbiter = 50
    print(f"nbiter : {nbiter}, nbpart : {nbpart}")
    path = f"Data/plot_output/{time.strftime('%d-%m-%Y')}/"
    try:
```

```
        os.mkdir(path)


    except:
        pass
    n = 100
    with tqdm(total=n,position=0) as pbar:
        with tqdm(total=0, position=1, bar_format='{desc}') as pbar2:

            for i in range(n):
                # print(f"i : {i}")
                pbar.set_description(f" n°{i} ")
                path_ = next_path(f"{path}%s/")
                try:
                    os.mkdir(path_)
                except:

                    pass
                pbar2.set_description(f"Calcul de la meilleure trajectoire de
                                      {path_}...")
                gs ,gs_e = pso_2(fitFunc,track.nb_sectors,track.boundaries,
                                 nbpart,nbiter,0.8,1,0.20,track)
                rl = track.getRacingLine(gs)
                track.racingLine = getLapTime(rl)[1]
                track.rl2 = [[rl[i][0] for i in range(len(rl))],
                             [rl[i][1] for i in range(len(rl))]]
                pbar2.set_description(f"Sauvegarde de {path_}data.txt ...")
                with open(f"{path_}data.txt","w") as file:
                    file.write(str([gs_e,gs,rl]))
                    file.close()
                pbar2.set_description(f"Sauvegarde des plots de {path_} ...")
                # track.plotTrack(centerLine=False,sectors=False,
                #                 racingLine=True,nb=True,points=True)
                track.saveTrack(centerLine=False,sectors=False,racingLine=True,
                        nb=True,points=True,transparent=False,  path=path_)

                track.saveTrack(centerLine=False,sectors=False,racingLine=True,
                         nb=False,points=True,transparent=False,path=path_)
                pbar2.set_description(f"Sauvegarde des plots n°{i} ... OK")
                pbar.update(1)
    """
def plot the track with the racing line from the data file.
    openData():
    file = open("Data/Results/2/2/Circuit Paul Ricard.txt","r")
    n,gs_e,gs,rl = list(eval(file.read())[7])
    print(n,gs_e)
    track.racingLine = getLapTime(rl)[1]
    track.rl2 = [[rl[i][0] for i in range(len(rl))],
                 [rl[i][1] for i in range(len(rl))]]
    track.plotTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
                    points=True)
    track.saveTrack(centerLine=False,sectors=False,racingLine=True,nb=True,
                    points=True,transparent=False)
```

```
            track.saveTrack(centerLine=False,sectors=False,racingLine=True,n
b=True,
                    points=True,transparent=False)
    file.close()
            track.saveTrack(centerLine=False,sectors=False,racingLine=True,n
b=False,
if __name__ == "__main__":
    track = Track("Data/Track_data/cbg.geojson")
    main()
    # main2()



"""
Author: Xayon
file: annealing.py
"""
# from scipy.optimize import dual_annealing
import dual_annealing
import time
import tipe
import numpy as np
from tqdm import tqdm
import os

N_THREAD = 2
NB_PARTICLE = 150
NB_ITER = 80
MAX_ITER = 500

track = tipe.Track("Data/Track_data/cbg.geojson")
pbar = tqdm(total=NB_ITER*NB_PARTICLE*track.nb_sectors,
position=3,
            desc="Threads")

date = time.strftime("%d_%m_%y_%H:%M",time.localtime())
path = f"Data/annealing_output/{date}"
w = 0.8
d = []
a = []

def func(W):
    l = []
    W = list(W)
    c1 = round(W[0],2)
    c2 = round(W[1],2)
    W = [c1,c2]
    # print(f"Starting with c1={c1}, c2={c2} at {time.ctime()}")
    cur_param.set_description_str(f"Current parameters : c1={c1}, c2={c2}
                                  at {time.ctime()}")
```

```
    file = open(f"{path}/data_{id}.txt", "r")
    d = list(eval(file.read()))
    file.close()
    for c1_,c2_,lmin_,lmean_,lstd_ in d:
        if c1_ == c1 and c2_ == c2:
            # print(f"{c1},{c2} already computed")
          res.set_description_str(f"c1={c1}, c2={c2} already computed :
                        {lmin_},{lmean_},{lstd_}")
            a.append(1)
            dual_annealing.pbar.refresh()
            return lmin_
    for i in range(N_THREAD):
        gs, gs_e = tipe.pso_(tipe.fitFunc, track.nb_sectors, track.boundaries,
                             NB_PARTICLE, NB_ITER, w, c1, c2, track,pbar)
        pbar.reset()
        l.append(gs_e)

    l = np.array(l)
    lmin = l.min()
    lmean = l.mean()
    lstd = l.std()


    d.append([c1, c2, lmin, lmean, lstd])
    # print(f"c1={c1},c2={c2} : {lmin},{lmean}")
    res.set_description_str(f"c1={c1}, c2={c2} : {lmin},{lmean},{lstd}")

    file = open(f"{path}/data_{id}.txt", "w")
    # print(str(d))
    file.write(str(d))
    file.close()
    a.append(0)
    dual_annealing.pbar.refresh()

    return lmin

def main(id=id):
    try:
        os.mkdir(path)
    except:
        pass


    file = open(f"{path}/data_{id}.txt","w")
    file.write("[]")
    file.close()
    d = []

    dual_annealing.pbar = tqdm(total=MAX_ITER, position=0, desc="Iterations")
    dual_annealing.pbar.write(f"""Starting calculation with :
        \033[96mNB_PARTICLE\033[00m=\033[92m{NB_PARTICLE}\033[00m,
        \033[96mNB_ITER\033[00m=\033[92m{NB_ITER}\033[00m,
        \033[96mN_THREAD\033[00m=\033[92m{N_THREAD}\033[00m,
        \033[96mMAX_ITER\033[00m=\033[92m{MAX_ITER}\033[00m""")
    r_min, r_max = -10., 10.
    bounds = [[r_min, r_max], [r_min, r_max]]
    result = dual_annealing.dual_annealing(func, bounds,maxiter=MAX_ITER)

    # summarize the result
    # print('Status : %s' % result['message'])
    # print('Total Evaluations: %d' % result['nfev'])

    # evaluate solution
    solution = result['x']
    solution = solution.tolist()
    evaluation = func(solution)
    # print('Solution: f(%s) = %.5f' % (solution, evaluation))

    cur_param.write(f"Status : {result['message']}")
    dual_annealing.pbar.close()
    pbar.close()
    cur_param.set_description_str(f"Total Evaluations: {result['nfev']}")
    res.set_description_str(f"Solution found : f({solution}) =
    {evaluation}")
    cur_param.close()
    res.close()

    file = open(f"{path}/data_{id}.txt","r")
    data = file.read()
    file.close()

    file = open(f"{path}/data{id}.txt","w")
    file.write(data)
    file.close()

    file = open(f"{path}/info{id}.txt","w")
    file.write(str((w,r_min,r_max,NB_PARTICLE,NB_ITER,MAX_ITER))+"\n")
    file.write(str([solution,evaluation,result['message'],
result['nfev']])+"\n")
    file.write(str(a)+"\n")
    file.write(str(dual_annealing.r)+"\n")
    file.close()
    print(f"\nSaving OK")
    file =
    open(f"{path}/data_{id}.txt","w")
    file.close()
    os.remove(f"{path}/data_{id}.txt")
if __name__ == "__main__":
    cur_param = tqdm(total=0, position=1, bar_format='{desc}')
    res = tqdm(total=0, position=2, bar_format='{desc}')

    main(id)
```

```

"""
Author: Xayon
file: process_annealing.py
"""
import dual_annealing
import time
import tipe
import numpy as np
from tqdm import tqdm
import os

import sys

N_THREAD = 2
NB_PARTICLE = 1
NB_ITER = 1
MAX_ITER = 20

if not os.path.exists("Data/Track_data"):
    os.mkdir("Data/Track_data")
if not os.path.exists("Data/Track_data/cbg.geojson"):
    os.system("wget -O Data/Track_data/cbg.geojson \
               https://raw.githubusercontent.com/bacinger/\
               f1-circuits/master/circuits/fr-1969.geojson")

track = tipe.Track("Data/Track_data/cbg.geojson")
pbar = tqdm(total=NB_ITER*NB_PARTICLE*track.nb_sectors, position=3,
            desc="Threads")
if not os.path.exists("Data/annealing_output"):
    os.mkdir("Data/annealing_output")

date = time.strftime("%d_%m_%y_%H:%",time.localtime())
path = f"Data/annealing_output/{date}"

d = []
w = 0.8


a = []

def func(W):
    l = []
    W = list(W)
    c1 = round(W[0],2)
    c2 = round(W[1],2)
    W = [c1,c2]
    # print(f"Starting with c1={c1}, c2={c2} at {time.ctime()}")
    cur_param.set_description_str(f"Current parameters : c1={c1},\
                                    c2={c2} at {time.ctime()}")

    file = open(f"{path}/data_{id}.txt", "r")
    d = list(eval(file.read()))
    file.close()
    for c1_,c2_,lmin_,lmean_,lstd_ in d:
        if c1_ == c1 and c2_ == c2:
            # print(f"{c1},{c2} already computed")
            res.set_description_str(f"c1={c1}, c2={c2} already computed :\
                    {lmin_},{lmean_},{lstd_}")
            a.append(1)
            dual_annealing.pbar.refresh()
            return lmin_
    for i in range(N_THREAD):
        gs, gs_e = tipe.pso_(tipe.fitFunc, track.nb_sectors,
        track.boundaries,
                        NB_PARTICLE, NB_ITER, w, c1, c2, track,pbar)
        pbar.reset()
        l.append(gs_e)


    l = np.array(l)
    lmin = l.min()
    lmean = l.mean()
    lstd = l.std()

    d.append([c1, c2, lmin, lmean, lstd])
    # print(f"c1={c1},c2={c2} : {lmin},{lmean}")
    res.set_description_str(f"c1={c1}, c2={c2} : {lmin},{lmean},{lstd}")

    file = open(f"{path}/data_{id}.txt", "w")
    # print(str(d))
    file.write(str(d))
    file.close()
    a.append(0)
    dual_annealing.pbar.refresh()

    return lmin


main(id=id):
def

try:
    os.mkdir(path)
except:
    pass

file = open(f"{path}/data_{id}.txt","w")
file.write("[]")
file.close()


d = []
```

```
    dual_annealing.pbar = tqdm(total=MAX_ITER, position=0, desc="Iterations")
    dual_annealing.pbar.write(f"""Starting calculation with :
        \033[96mNB_PARTICLE\033[00m=\033[92m{NB_PARTICLE}\033[00m,
        \033[96mNB_ITER\033[00m=\033[92m{NB_ITER}\033[00m,
        \033[96mN_THREAD\033[00m=\033[92m{N_THREAD}\033[00m,
        \033[96mMAX_ITER\033[00m=\033[92m{MAX_ITER}\033[00m""")
    r_min, r_max = -10., 10.
    # bounds = [[r_min, r_max], [r_min, r_max]]
    bounds = [(-10,40),(-10,10)]
    result = dual_annealing.dual_annealing(func, bounds,maxiter=MAX_ITER)


    # summarize the result
    # print('Status : %s' % result['message'])
    # print('Total Evaluations: %d' % result['nfev'])

    # evaluate solution
    solution = result['x']
    solution = solution.tolist()
    evaluation = func(solution)
    # print('Solution: f(%s) = %.5f' % (solution, evaluation))


    cur_param.write(f"Status : {result['message']}")
    dual_annealing.pbar.close()
    pbar.close()
    cur_param.set_description_str(f"Total Evaluations: {result['nfev']}")
    res.set_description_str(f"Solution found : f({solution}) = {evaluation}")

    cur_param.close()
    res.close()

    file = open(f"{path}/data_{id}.txt","r")
    data = file.read()
    file.close()

    file = open(f"{path}/data{id}.txt","w")
    file.write(data)
    file.close()

    file = open(f"{path}/info{id}.txt","w")
    file.write(str((w,bounds,NB_PARTICLE,NB_ITER,MAX_ITER))+"\n")
     file.write(str([solution,evaluation,result['message'],
                    result['nfev']])+"\n")
     file.write(str(a)+"\n")
     file.write(str(dual_annealing.r)+"\n")
     file.close()
     print(f"\nSaving OK")
     file = open(f"{path}/data_{id}.txt","w")
     file.close()
if __name__ == "__main__":
    os.remove(f"{path}/data_{id}.txt")
```

```
    if len(sys.argv) > 1:
        id = int(sys.argv[1])
    else:
        id = 0
    if (os.path.exists(f"Data/annealing_output/{date}/data_{id}.txt")
      or os.path.exists(f"Data/annealing_output/{date}/data{id}.txt")):
        print(f"data_{id}.txt already exists")
        id_list = []
        for file_name in os.listdir(f"Data/annealing_output/{date}"):
            if file_name.startswith("data_") and
            file_name.endswith(".txt"):
                id_list.append(int(file_name.split("_")[1].split(".")[0]))
            elif file_name.startswith("data") and file_name.endswith(".txt"):
                id_list.append(int(file_name[4:].split(".")[0]))
        if id_list:
            biggest_id = max(id_list)
        else:
            biggest_id = 0
        print(f"Starting with id={biggest_id+1}")
        id = biggest_id+1

    cur_param = tqdm(total=0, position=1,
    bar_format='{desc}')
    res = tqdm(total=0, position=2, bar_format='{desc}')

    main(id)



"""
Author: Xayon
file: analyse_annealing.py
"""
import numpy as np
import json
import tkinter as tk
from tkinter import filedialog, Tk
import os
import random
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def analyse_data_1(directory):
    # Load the data
    with open(f"{directory}/data.json", "r") as f:
        data = list(eval(f.read()))
        f.close()

    # Create a list of all the keys
    x,y,z,z2,z3 = [],[],[],[],[]
    #   x.append(int(key.split(",")[0]))
    # for key in data.keys():
```

```
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      fig2.subplots_adjust(bottom=0.2)
1047      ax2.set_title("Temps moyen")
1048      ax2.set_xlabel("c1")
1049      ax2.set_ylabel("c2")
1050      ax2.set_zlabel("Temps (s)")
1051      ax2.set_zlim3d(bottom=87.5,top=95)
1052      ax2.scatter(x, y, z2, c='b', marker='o')
1053      plt.show()
1054
1055      fig3 = plt.figure()
1056      ax3 = fig3.add_subplot(111, projection='3d')
1057      fig3.subplots_adjust(bottom=0.2)
1058      ax3.set_title("ecart type")
1059      ax3.set_xlabel("c1")
1060      ax3.set_ylabel("c2")
1061      ax3.scatter(x, y, z3, c='b', marker='o')
1062      plt.show()
1063
1064
1065  def analyse_info_1(directory):
1066      with open(f"{directory}/info.json", "r") as f:
1067          parameters = tuple(eval(f.readline()))
1068          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
1087      plt.plot(rate)
1088      plt.title("vitesse de calcul")
1089      plt.show()
1090
1091      print(f"parameters= {parameters}")
1092      print(f"result= {result}")
1093
1094
1095
1096      analyse_1():
1097  def
1098      root = tk.Tk()
1099      root.withdraw()
          directory =
          filedialog.askdirectory(initialdir="Data/annealing_output",
1100                                              title="Select the directory to
                                                open")
1101      analyse_data_1(directory)
1102      analyse_info_1(directory)
1103
1104  import time as test
1105
1106  def analyse_data_2(directory_list):
1107      # Create empty lists for x, y, z, z2, z3
1108      x, y, z, z2, z3 = [], [], [], [], []
1109      nb_files = 0
1110      for directory in directory_list:
1111          if not os.path.exists(f"{directory}/data.json"):
1112              prefixed = [filename for filename in
                    os.listdir(f'{directory}/.')
1113                  if filename.startswith("data")]
1114              for filename in prefixed:
1115                  nb_files += 1
1116                  f = open(f"{directory}/{filename}", "r")
1117                  data = list(eval(f.read()))
1118                  f.close()
1119                  for c1, c2, lmin, lmean, lstd in data:
```

```
                    # if (c1, c2) in zip(x, y):
                #   index = list(zip(x, y)).index((c1, c2))

                #   removed += 1
                #   if lmin < z[index]:
                #       z[index] = lmin
                #       z2[index] = lmean
                #       z3[index] = lstd
                  # else:

                    x.append(c1)
                    y.append(c2)
                    z.append(lmin)
                    z2.append(lmean)
                    z3.append(lstd)

        else:
            with open(f"{directory}/data.json", "r") as f:
                data = list(eval(f.read()))
                f.close()
                for c1, c2, lmin, lmean, lstd in data:
                    if (c1, c2) in zip(x, y):
                        index = list(zip(x, y)).index((c1, c2))
                        if lmin < z[index]:
                            z[index] = lmin
                            z2[index] = lmean
                            z3[index] = lstd
                    else:
                        x.append(c1)
                        y.append(c2)
                        z.append(lmin)
                        z2.append(lmean)
                        z3.append(lstd)

    print(len(x))
    print(f"nb_files = {nb_files}")

    print("-----Starting to remove randomly-----")
    time1 = test.time()
    # Remove half of the data points randomly
    num_points = len(x)
    # num_points_to_remove = num_points - 30000 if num_points > 20000 else 0
    num_to_keep = 30000
    time2 = test.time()
    # indices_to_remove =random.sample(range(num_points),
num_points_to_remove)
    indices_to_keep = random.sample(range(num_points), num_to_keep)
    print("-----Data removed----- in ", round(test.time() - time1,2))
    time3 = test.time()
    x = [x[i] for i in indices_to_keep]
    y = [y[i] for i in indices_to_keep]
    z = [z[i] for i in indices_to_keep]
    z2 = [z2[i] for i in indices_to_keep]
      z3 = [z3[i] for i in indices_to_keep]
    print("-----Data removed----- in ", round(test.time() -
time3,2))

    # print(x)
    print(len(x))
    # print(y)
    print(len(y))
    print("-----Data removed----- in ", round(test.time() -
time1,2))

    # Plot the data
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    fig.subplots_adjust(bottom=0.2)
    fig.suptitle("Analyse des données en 3D")
    ax.set_title("Temps min")
    ax.set_xlabel("c1")
    ax.set_ylabel("c2")
    ax.set_zlabel("Temps (s)")
    ax.set_zlim3d(bottom=87.5, top=95)
    ax.scatter(x, y, z, c='r', marker='o')
    plt.show()

    fig2 = plt.figure()
    ax2 = fig2.add_subplot(111, projection='3d')
    fig2.subplots_adjust(bottom=0.2)
    ax2.set_title("Temps moyen")
    ax2.set_xlabel("c1")
    ax2.set_ylabel("c2")
    ax2.set_zlabel("Temps (s)")
    ax2.set_zlim3d(bottom=87.5, top=95)
    ax2.scatter(x, y, z2, c='b', marker='o')
    plt.show()

    fig3 = plt.figure()
    ax3 = fig3.add_subplot(111, projection='3d')
    fig3.subplots_adjust(bottom=0.2)
    ax3.set_title("ecart type")
    ax3.set_xlabel("c1")
    ax3.set_ylabel("c2")
    ax3.scatter(x, y, z3, c='b', marker='o')
    plt.show()

    print("-----Starting to import results-----")
      1213
    nb_al_computed = []
    rate = []
    parameters = []
    results = []

    for directory in directory_list:
        if not os.path.exists(f"{directory}/info.json"):
```

```
            prefixed = [filename for filename in os.listdir(f'{directory}/.')
                    if filename.startswith("info")]
            for filename in prefixed:
                f = open(f"{directory}/{filename}", "r")
                parameters.append(tuple(eval(f.readline())))
                results.append(list(eval(f.readline())))
                nb_al_computed.extend(list(eval(f.readline())))
                rate.extend(list(eval(f.readline())))
                f.close()
        else:
            with open(f"{directory}/info.json", "r") as f:
                parameters.append(tuple(eval(f.readline())))
                results.append(list(eval(f.readline())))
                nb_al_computed.extend(list(eval(f.readline())))
                rate.extend(list(eval(f.readline())))
                f.close()

    nb = len(nb_al_computed)
    print(f"nb = {nb}")
    # plt.plot(nb_al_computed)
    # plt.title("Calcul déjà effectué")
    # plt.show()

    # nb_al_computed2 = []
    # for i in range(nb):
    #    nb_al_computed2.append(sum(nb_al_computed[:i]))

    # plt.plot(nb_al_computed2)
    # plt.title("Calcul déjà effectué")
    # plt.show()

    # plt.plot(rate)
    # plt.title("Vitesse de calcul")
    # plt.show()

    print(f"parameters = {parameters}")
    # print(f"result = {results}")
    x_res, y_res, z_res = [], [], []
    iters = []
    for coord,time,message,iter in results:
        x_res.append(coord[0])
        y_res.append(coord[1])
        z_res.append(time)
        iters.append(iter)

    print(f" results = {z_res}")
    iter_total = sum(iters)
    print(f"Total iterations = {iter_total}")


    print("-----Data imported-----")

    print("-----Starting to plot-----")
    fig4 = plt.figure(figsize=(8,15))
    # fig4.subplots_adjust(bottom=0.2, top=0.8)
    ax = fig4.add_subplot(211, projection='3d')
    fig4.subplots_adjust(wspace=0 , hspace=0)
    fig4.subplots_adjust(top=0.95)
    # fig4.tight_layout()
    fig4.set_constrained_layout_pads(w_pad=0, h_pad=0, hspace=0, wspace=0)
    fig4.suptitle("Analyse des données en 3D",y=0.95,fontsize=15)
    ax.set_xlabel("c1")
    ax.set_ylabel("c2")
    # ax.set_zlabel("Temps (s)")
    newcmp = ListedColormap(plt.cm.get_cmap('jet_r')(np.linspace(.5, .9
,256)))

    ax.set_zlim3d(bottom=87.5, top=95)
    ax.scatter(x, y, z, c='b', marker='o', alpha=0.03, s=10)
    # ax.scatter(x, y, z, c=z, marker='o', alpha=0.03, s=10,cmap=newcmp)
    ax.scatter(x_res, y_res, z_res, c='r', marker='o' , alpha=1, s=40)
    ax.view_init(elev=24, azim=62, roll=0)

    ax2 = fig4.add_subplot(212, projection='3d')
    fig4.subplots_adjust(wspace=0 , hspace=0)
    fig4.subplots_adjust(top=0.95)
    ax2.set_xlabel("c1")
    ax2.set_ylabel("c2")
    # ax.set_zlabel("Temps (s)")
    ax2.set_zlim3d(bottom=87.5, top=95)
    ax2.scatter(x, y, z, c='b', marker='o', alpha=0.03, s=10)
    # ax2.scatter(x, y, z, c=z, marker='o', alpha=0.03, s=10,cmap=newcmp)
    ax2.scatter(x_res, y_res, z_res, c='r', marker='o' , alpha=1, s=35)
    ax2.view_init(elev=-90, azim=90, roll=0)

    ax2.grid(False)
    ax2.xaxis.pane.fill = False
    ax2.yaxis.pane.fill = False
    ax2.zaxis.pane.fill = False
    ax2.xaxis.pane.set_edgecolor('w')
    ax2.yaxis.pane.set_edgecolor('w')
    ax2.zaxis.pane.set_edgecolor('w')
    ax2.zaxis.line.set_lw(0.)
    ax2.set_zticks([])

    plt.savefig("Img/analyse_3D.png", dpi=300, bbox_inches='tight')
    # plt.show()



def analyse_2():
    import_=input("Do you want to import the data from import file ?
    (y/n): ")
```

```
    dirs = []
    dir = None
    if import_ == "y":
        with open("import.txt", "r") as f:
            for line in f:
                dirs.append(line.strip())
    else:
        root = Tk()
        while dir != ():
         dir = filedialog.askdirectory(initialdir="Data/annealing_output",
                                       title="Select the directory to open")
            print(dir)
            if dir != ():
                dirs.append(dir)
    analyse_data_2(dirs)


def analyse_results(directory_list):
    print("-----Starting to import results-----")
    nb_al_computed = []
    rate = []
    parameters = []
    results = []

    for directory in directory_list:
        if not os.path.exists(f"{directory}/info.json"):
        prefixed = [filename for filename in os.listdir(f'{directory}/.')
                if filename.startswith("info")]
            for filename in prefixed:
                f = open(f"{directory}/{filename}", "r")
                parameters.append(tuple(eval(f.readline())))
                results.append(list(eval(f.readline())))
                nb_al_computed.extend(list(eval(f.readline())))
                rate.extend(list(eval(f.readline())))
                f.close()
        else:
            with open(f"{directory}/info.json", "r") as f:
                parameters.append(tuple(eval(f.readline())))
                results.append(list(eval(f.readline())))
                nb_al_computed.extend(list(eval(f.readline())))
                rate.extend(list(eval(f.readline())))
                f.close()

    nb = len(nb_al_computed)
    print(f"nb = {nb}")
    # plt.plot(nb_al_computed)
    # plt.title("Calcul déjà effectué")
    # plt.show()


    # nb_al_computed2 = []
    # for i in range(nb):
    #   nb_al_computed2.append(sum(nb_al_computed[:i]))
        # plt.plot(nb_al_computed2)
    # plt.title("Calcul déjà effectué")
    # plt.show()

    # plt.plot(rate)
    # plt.title("Vitesse de calcul")
    # plt.show()

    # print(f"parameters = {parameters}")
    # print(f"result = {results}")
    # x_res, y_res, z_res = [], [], []
    # iters = []
    # for coord,time,message,iter in results:
    #   x_res.append(round(coord[0],2))
    #   y_res.append(round(coord[1],2))
    #   z_res.append(time)
    #   iters.append(iter)

    y_res = [round(result[0][1],2) for result in results]
    moy = np.mean(y_res)
    med = np.median(y_res)
    print(f"""
results = {y_res}
moyenne = {moy}
ecart type = {np.std(y_res)}
mediane = {med}
""")

    print("-----Data imported-----")
    print("-----Starting to plot-----")
    # faire un histogramme sur la variable y_res
    fig = plt.figure(figsize=(12,9))
    ax = fig.add_subplot(111)

    bins = np.arange(round(min(y_res),1),round(max(y_res),1)+0.1,0.05)
    print(bins)
    print(len(bins))
    ax.hist(y_res, bins=bins,color='cornflowerblue',edgecolor='black',
            linewidth=1)
    print(np.histogram(y_res,  bins=bins))
    y,x = np.histogram(y_res, bins=bins)

    total = sum(y)
    max_ = round(x[np.argmax(y)],2)
    print(f'max : {max_}')
    print(total)
    #ax.hist(y_res,bins=20,range=(round(min(y_res),1),round(max(y_res),1)+0
    .1)
)
    plt.xticks(ticks=np.arange(start=round(min(y_res),1),
                               step=round(np.ptp(y_res)/13,1),
```

```
                                stop=round(np.max(y_res),1)+0.1))

    plt.axvline(x = med, color = 'r', label = 'mediane',
                linestyle = 'dashed',linewidth=2)
    plt.annotate(f'médiane = {round(med,2)}', xy =(med+0.01, 29),
                xytext =(med+0.23, 31),
                arrowprops = dict(facecolor ='black',arrowstyle = "->",lw =
2))

    plt.axvline(x = max_+0.025, color = 'g', label = 'max',
                linestyle = 'dashed',linewidth=2)
    plt.annotate(f'x_max = {max_+0.025}', xy =(max_+0.03, 16),
                xytext =(max_+0.25, 13),
                arrowprops = dict(facecolor ='black',arrowstyle = "->",lw =
2))

    plt.title(f"Répartition des valeurs de c2 sur {total} résultats",
                fontsize=15)
    # plt.legend()
    plt.savefig("Img/histogramme/histogramme_y_res.png",  dpi=500,
                bbox_inches='tight')
    # plt.show()

def analyse_3():
    import_=input("Do you want to import the data from import file ? (y/n) :
")


    dirs = []
    dir = None
    if import_ == "y":
        with open("import.txt", "r") as f:
            for line in f:
                dirs.append(line.strip())
    else:
        root = Tk()
        while dir != ():
            dir = filedialog.askdirectory(initialdir="Data/annealing_output",
                                          title="Select the directory to open")
            print(dir)
            if dir != ():
                dirs.append(dir)
    analyse_results(dirs)


if __name__ == "__main__":
`~numpy.random.Generator`}
    # analyse_1()
    analyse_2()

    # analyse_3()


# Dual Annealing implementation.
```

```
# Copyright (c) 2018 Sylvain Gubian <sylvain.gubian@pmi.com>,
# Yang Xiang <yang.xiang@pmi.com>
# Author: Sylvain Gubian, Yang Xiang, PMP S.A.
# Modifications by: Xayon

"""
A Dual Annealing global optimization algorithm
"""

import numpy as np
from scipy.optimize import OptimizeResult
from scipy.optimize import minimize, Bounds
from scipy.special import gammaln
from scipy._lib._util import check_random_state
from scipy.optimize._constraints import new_bounds_to_old
import time
from tqdm import tqdm

__all__ = ['dual_annealing']

pbar = None
r = []


class VisitingDistribution:
    """
    Class used to generate new coordinates based on the distorted
    Cauchy-Lorentz distribution. Depending on the steps within the
    strategy
    chain, the class implements the strategy for generating new location
    changes.

    Parameters
    ----------
    lb : array_like
        A 1-D NumPy ndarray containing lower bounds of the generated
        components. Neither NaN or inf are allowed.
    ub : array_like
        A 1-D NumPy ndarray containing upper bounds for the generated
        components. Neither NaN or inf are allowed.
    visiting_param : float
        Parameter for visiting distribution. Default value is 2.62.
        Higher values give the visiting distribution a heavier tail, this
        makes the algorithm jump to a more distant region.
        The value range is (1, 3]. Its value is fixed for the life of the
        object.
    rand_gen : {`~numpy.random.RandomState`,

        A `~numpy.random.RandomState`, `~numpy.random.Generator` object
        for using the current state of the created random generator
        container.

    """
    TAIL_LIMIT = 1.e8
    MIN_VISIT_BOUND = 1.e-10
```

```
    def __init__(self, lb, ub, visiting_param, rand_gen):
        # if you wish to make _visiting_param adjustable during the life of
        # the object then _factor2, _factor3, _factor5, _d1, _factor6 will
        # have to be dynamically calculated in `visit_fn`. They're factored
        # out here so they don't need to be recalculated all the time.
        self._visiting_param = visiting_param
        self.rand_gen = rand_gen
        self.lower = lb
        self.upper = ub
        self.bound_range = ub - lb

        # these are invariant numbers unless visiting_param changes
        self._factor2 = np.exp((4.0 - self._visiting_param) * np.log(
            self._visiting_param - 1.0))
        self._factor3 = np.exp((2.0 - self._visiting_param) * np.log(2.0)
                               / (self._visiting_param - 1.0))
        self._factor4_p = np.sqrt(np.pi) * self._factor2 / (self._factor3 * (
            3.0 - self._visiting_param))

        self._factor5 = 1.0 / (self._visiting_param - 1.0) - 0.5
        self._d1 = 2.0 - self._factor5
        self._factor6 = np.pi * (1.0 - self._factor5) / np.sin(
          np.pi * (1.0 - self._factor5)) / np.exp(gammaln(self._d1))

    def visiting(self, x, step, temperature):
        """ Based on the step in the strategy chain, new coordinates are
        generated by changing all components is the same time or only
        one of them, the new values are computed with visit_fn method
         """
        dim = x.size
        if step < dim:

            # Changing all coordinates with a new visiting value
            visits = self.visit_fn(temperature, dim)
            upper_sample, lower_sample = self.rand_gen.uniform(size=2)
            visits[visits > self.TAIL_LIMIT] = self.TAIL_LIMIT * upper_sample
            visits[visits < -self.TAIL_LIMIT] = -self.TAIL_LIMIT *
lower_sample
            x_visit = visits + x
            a = x_visit - self.lower
            b = np.fmod(a, self.bound_range) + self.bound_range
            x_visit = np.fmod(b, self.bound_range) + self.lower
            x_visit[np.fabs(
              x_visit - self.lower) < self.MIN_VISIT_BOUND] += 1.e-10
        else:
            # Changing only one coordinate at a time based on strategy
            # chain step
            x_visit = np.copy(x)
            visit = self.visit_fn(temperature, 1)[0]
            if visit > self.TAIL_LIMIT:
                visit = self.TAIL_LIMIT * self.rand_gen.uniform()
            elif visit < -self.TAIL_LIMIT:
              visit = -self.TAIL_LIMIT * self.rand_gen.uniform()
            index = step - dim
            x_visit[index] = visit + x[index]
            a = x_visit[index] - self.lower[index]
        b = np.fmod(a, self.bound_range[index]) + self.bound_range[index]
            x_visit[index] = np.fmod(b, self.bound_range[
                index]) + self.lower[index]
            if np.fabs(x_visit[index] - self.lower[
                    index]) < self.MIN_VISIT_BOUND:
                x_visit[index] += self.MIN_VISIT_BOUND
        return x_visit

    def visit_fn(self, temperature, dim):
        """ Formula Visita from p. 405 of reference [2] """
        x, y = self.rand_gen.normal(size=(dim, 2)).T

    factor1 = np.exp(np.log(temperature) / (self._visiting_param - 1.0))
        factor4 = self._factor4_p * factor1

        # sigmax
        x *= np.exp(-(self._visiting_param - 1.0) * np.log(
          self._factor6 / factor4) / (3.0 - self._visiting_param))

        den = np.exp((self._visiting_param - 1.0) * np.log(np.fabs(y)) /
                     (3.0 - self._visiting_param))

        return x / den

   1600 class EnergyState:
    """
    Class used to record the energy state. At any time, it knows what is
    the
    currently used coordinates and the most recent best location.

    Parameters
    ----------
    lower : array_like
    A 1-D NumPy ndarray containing lower bounds for generating an initial
        random components in the `reset` method.
    upper : array_like
    A 1-D NumPy ndarray containing upper bounds for generating an initial
        random components in the `reset` method
        components. Neither NaN or inf are allowed.
    callback : callable, ``callback(x, f, context)``, optional
        A callback function which will be called for all minima found.
        ``x`` and ``f`` are the coordinates and function value of the
        latest minimum found, and `context` has value in [0, 1, 2]
    """
    # Maximum number of trials for generating a valid starting point
    MAX_REINIT_COUNT = 1000
```

```
    def __init__(self, lower, upper, callback=None):
        self.ebest = None
        self.current_energy = None
        self.current_location = None
        self.xbest = None

        self.lower = lower
        self.upper = upper
        self.callback = callback
    def reset(self, func_wrapper, rand_gen, x0=None):
        """
        Initialize current location is the search domain. If `x0` is
        not
        provided, a random location within the bounds is generated.
        """
        if x0 is None:
            self.current_location = rand_gen.uniform(self.lower,
self.upper,
                                                     size=len(self.lower))
         else:
            self.current_location =
np.copy(x0)
        init_error = True
        reinit_counter = 0
        while init_error:
            if reinit_counter >= EnergyState.MAX_REINIT_COUNT:
                init_error = False
                message = (
                    'Stopping algorithm because function '
                    'create NaN or (+/-) infinity values even with '

                    'trying new random parameters'
                )
                raise ValueError(message)
            self.current_location = rand_gen.uniform(self.lower,
                                                     self.upper,
                                                     size=self.lower.size)
            reinit_counter += 1
        else:
            init_error = False
        # If first time reset, initialize ebest and xbest
        if self.ebest is None and self.xbest is None:
            self.ebest = self.current_energy
            self.xbest = np.copy(self.current_location)
        # Otherwise, we keep them in case of reannealing reset


    def update_best(self, e, x, context):
        self.ebest = e
        self.xbest = np.copy(x)
        if self.callback is not None:
            val = self.callback(x, e, context)
            if val is not None:
                if val:
                    return ('Callback function requested to stop early by '
                            'returning
                            True')

    def update_current(self, e, x):
        self.current_energy = e
        self.current_location = np.copy(x)
class StrategyChain:
    """
    Class that implements within a Markov chain the strategy for
    location
    acceptance and local search decision making.

    Parameters

    ----------

    acceptance_param : float

        Parameter for acceptance distribution. It is used to control
        the
        probability of acceptance. The lower the acceptance parameter, the
        smaller the probability of acceptance. Default value is -5.0 with
        a range (-1e4, -5].
    visit_dist : VisitingDistribution
        Instance of `VisitingDistribution` class.
    func_wrapper : ObjectiveFunWrapper
        Instance of `ObjectiveFunWrapper` class.
    minimizer_wrapper: LocalSearchWrapper
        Instance of `LocalSearchWrapper` class.
     rand_gen : {None, int, `numpy.random.Generator`,
                    `numpy.random.RandomState`},  optional

        If `seed` is None (or `np.random`), the
        `numpy.random.RandomState`
        singleton is used.
        If `seed` is an int, a new ``RandomState`` instance is used,
        seeded with `seed`.
        If `seed` is already a ``Generator`` or ``RandomState`` instance then
        that instance is used.
    energy_state: EnergyState
        Instance of `EnergyState` class.

    """

    def __init__(self, acceptance_param, visit_dist,
    func_wrapper,minimizer_wrapper, rand_gen, energy_state):
        # Local strategy chain minimum energy and location
        self.emin = energy_state.current_energy
        self.xmin =
        np.array(energy_state.current_location)
        # Global optimizer state
        self.energy_state = energy_state
        # Acceptance parameter
        self.acceptance_param = acceptance_param
```

```
        # Visiting distribution instance
        self.visit_dist = visit_dist
        # Wrapper to objective function
        self.func_wrapper = func_wrapper
        # Wrapper to the local minimizer
        self.minimizer_wrapper =
        minimizer_wrapper
        self.not_improved_idx = 0
        self.not_improved_max_idx = 1000
        self._rand_gen = rand_gen
        self.temperature_step = 0
        self.K = 100 * len(energy_state.current_location)

    def accept_reject(self, j, e, x_visit):
        r =
self._rand_gen.uniform()
        pqv_temp = 1.0 - ((1.0 - self.acceptance_param)
        * (e - self.energy_state.current_energy) /
self.temperature_step)
        if r <= pqv:
            # We accept the new location and update state
            self.energy_state.update_current(e, x_visit)
            self.xmin = np.copy(self.energy_state.current_location)
        # No improvement for a long time
        if self.not_improved_idx >= self.not_improved_max_idx:
            if j == 0 or self.energy_state.current_energy < self.emin:
                self.emin = self.energy_state.current_energy
                self.xmin = np.copy(self.energy_state.current_location)

    def run(self, step, temperature):
        self.temperature_step = temperature / float(step + 1)
        self.not_improved_idx += 1
        for j in range(self.energy_state.current_location.size * 2):
            if step == 0:
                self.energy_state_improved = True
            else:
                self.energy_state_improved =
False
            x_visit =
self.visit_dist.visiting(
                self.energy_state.current_location, j,
temperature)
            # Calling the objective
function
            e =
self.func_wrapper.fun(x_visit)
            if e <
self.energy_state.current_energy:
                # We have got a better energy value
                self.energy_state.update_current(e, x_visit)
                if e < self.energy_state.ebest:
                    val = self.energy_state.update_best(e, x,
None)
                    if val is not None:
                        if val:
                            return val
                    self.energy_state_improved = True
                    self.not_improved_idx = 0
            else:
                # We have not improved but do we accept the new
                location?
                self.accept_reject(j, e, x_visit)
            if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
                return ('Maximum number of function call reached '
                        'during
annealing') # End of
        StrategyChain loop

    def local_search(self):
        # Decision making for performing a local search
        # based on strategy chain results
        # If energy has been improved or no improvement since too
        long,
        # performing a local search with the best strategy chain
        location
        if self.energy_state_improved:
            # Global energy has improved, let's see if LS improves further
            e, x =
self.minimizer_wrapper.local_search(self.energy_state.xbest,
                                                self.energy_state.ebest
)
            if e < self.energy_state.ebest:
                self.not_improved_idx = 0
                val = self.energy_state.update_best(e, x,
                1)
                if val is not None:
                    if val:
                        return val
                self.energy_state.update_current(e, x)
            if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
                return ('Maximum number of function call reached '
                                'during local search')
        # Check probability of a need to perform a LS even if no
        improvement
        do_ls = False
        if pls >= np.exp(self(
                self.energy_state.ebest - self.energy_state.current_energy) /
                self.temperature_step)
            if pls >= self._rand_gen.uniform():
                do_ls = True

        # Global energy not improved, let's see what LS gives
        # on the best strategy chain location
        if self.not_improved_idx >= self.not_improved_max_idx:
            do_ls = True

        if do_ls:
            e, x = self.minimizer_wrapper.local_search(self.xmin,
            self.emin)
            self.xmin = np.copy(x)
```

```
            self.not_improved_max_idx =
self.energy_state.current_location.size
            if e < self.energy_state.ebest:
                val = self.energy_state.update_best(
                    self.emin, self.xmin, 2)
                if val is not None:
                    if val:
                        return val
                self.energy_state.update_current(e, x)
            if self.func_wrapper.nfev >= self.func_wrapper.maxfun:
                return ('Maximum number of function call reached '
                        'during dual annealing')

class ObjectiveFunWrapper:

    def __init__(self, func, maxfun=1e7, *args):

        self.func = func
        self.args = args
        # Number of objective function evaluations
        self.nfev = 0
        # Number of gradient function evaluation if used
        self.ngev = 0
        # Number of hessian of the objective function if used
        self.nhev = 0
        self.maxfun = maxfun

    def fun(self, x):
        self.nfev += 1
        return self.func(x, *self.args)


class LocalSearchWrapper:
    """
    Class used to wrap around the minimizer used for local search
    Default local minimizer is SciPy minimizer L-BFGS-B
    """

    LS_MAXITER_RATIO = 6
    LS_MAXITER_MIN = 100
    LS_MAXITER_MAX = 1000

    def __init__(self, search_bounds, func_wrapper, *args, **kwargs):
        self.func_wrapper = func_wrapper
        self.kwargs = kwargs
        self.jac = self.kwargs.get('jac', None)
        self.minimizer = minimize
        bounds_list = list(zip(*search_bounds))
        self.lower = np.array(bounds_list[0])
        self.upper = np.array(bounds_list[1])

        # If no minimizer specified, use SciPy minimize with 'L-BFGS-B'
        method
        if not self.kwargs:
            n = len(self.lower)
            ls_max_iter = min(max(n * self.LS_MAXITER_RATIO,
                                  self.LS_MAXITER_MIN),
                              self.LS_MAXITER_MAX)
            self.kwargs['method'] = 'L-BFGS-B'
            self.kwargs['options'] = {
                'maxiter': ls_max_iter,
            }
            self.kwargs['bounds'] = list(zip(self.lower, self.upper))
        elif callable(self.jac):
            def wrapped_jac(x):
                return self.jac(x, *args)
            self.kwargs['jac'] = wrapped_jac

    def local_search(self, x, e):
        # Run local search from the given x location where energy value
        is e
        x_tmp = np.copy(x)
        mres = self.minimizer(self.func_wrapper.fun, x, **self.kwargs)
        if 'njev' in mres:
            self.func_wrapper.ngev += mres.njev
        if 'nhev' in mres:
            self.func_wrapper.nhev += mres.nhev
        # Check if is valid value
        is_finite = np.all(np.isfinite(mres.x)) and np.isfinite(mres.fun)
        in_bounds = np.all(mres.x >= self.lower) and np.all(
            mres.x <= self.upper)
        is_valid = is_finite and in_bounds

        # Use the new point only if it is valid and return a better
        results
        if is_valid and mres.fun < e:
            return mres.fun, mres.x
        else:
            return e, x_tmp

from process_annealing import pbar as pbar2
def dual_annealing(func, bounds, args=(), maxiter=1000,
                   minimizer_kwargs=None, initial_temp=5230.,
                   restart_temp_ratio=2.e-5, visit=2.62, accept=-5.0,
                   maxfun=1e7, seed=None, no_local_search=False,
                   callback=None, x0=None):
    """
    Find the global minimum of a function using Dual Annealing.

    Parameters
    ----------
    func : callable
        The objective function to be minimized. Must be in the form
        ``f(x, *args)``, where ``x`` is the argument in the form of a 1-D
array
        and ``args`` is a  tuple of any additional fixed parameters
        needed to
        completely specify the function.
```

```
    bounds : sequence or `Bounds`
        Bounds for variables. There are two ways to specify the bounds:

Instance of `Bounds` class.
Sequence of ``(min, max)`` pairs for each element in `x`.

    args : tuple, optional
        Any additional fixed parameters needed to completely specify the
        objective function.
    maxiter : int, optional
    The maximum number of global search iterations. Default value is 1000.
    minimizer_kwargs : dict, optional
        Extra keyword arguments to be passed to the local minimizer
        (`minimize`). Some important options could be:
        ``method`` for the minimizer method to use and ``args`` for
        objective function additional arguments.
    initial_temp : float, optional
        The initial temperature, use higher values to facilitates a wider
        search of the energy landscape, allowing dual_annealing to escape
        local minima that it is trapped in. Default value is 5230. Range is
        (0.01, 5.e4].
    restart_temp_ratio : float, optional
        During the annealing process, temperature is decreasing, when it
    reaches ``initial_temp * restart_temp_ratio``, the reannealing process
        is triggered. Default value of the ratio is 2e-5. Range is (0, 1).
    visit : float, optional

        Parameter for visiting distribution. Default value is 2.62. Higher
        values give the visiting distribution a heavier tail, this makes
        the algorithm jump to a more distant region. The value range is (1,
3].
    accept : float, optional
        Parameter for acceptance distribution. It is used to control the

        probability of acceptance. The lower the acceptance parameter, the

        smaller the probability of acceptance. Default value is -5.0 with

        a range (-1e4, -5].
    maxfun : int, optional

        Soft limit for the number of objective function calls. If the

        algorithm is in the middle of a local search, this number will be
        exceeded, the algorithm will stop just after the local search is
        done. Default value is 1e7.
    seed : {None, int, `numpy.random.Generator`, `numpy.random.RandomState`},
optional
        If `seed` is None (or `np.random`), the `numpy.random.RandomState`
        singleton is used.
        If `seed` is an int, a new ``RandomState`` instance is used,
        seeded with `seed`.
        If `seed` is already a ``Generator`` or ``RandomState`` instance then
        that instance is used.
        Specify `seed` for repeatable minimizations. The random numbers
        generated with this seed only affect the visiting distribution

function
        and new coordinates generation.
    no_local_search : bool, optional
        If `no_local_search` is set to True, a traditional Generalized
        Simulated Annealing will be performed with no local search
        strategy applied.
    callback : callable, optional
        A callback function with signature ``callback(x, f, context)``,
        which will be called for all minima found.
        ``x`` and ``f`` are the coordinates and function value of the
    latest minimum found, and ``context`` has value in [0, 1, 2], with the
        following meaning:

            - 0: minimum detected in the annealing process.
            - 1: detection occurred in the local search process.
            - 2: detection done in the dual annealing process.

    If the callback implementation returns True, the algorithm will stop.
    x0 : ndarray, shape(n,), optional
        Coordinates of a single N-D starting point.

    Returns
    -------
    res : OptimizeResult
        The optimization result represented as a `OptimizeResult` object.
    Important attributes are: ``x`` the solution array, ``fun`` the value
        of the function at the solution, and ``message`` which describes
        the
        cause of the termination.
        See `OptimizeResult` for a description of other attributes.

    Notes
    -----
    This function implements the Dual Annealing optimization. This
stochastic
    approach derived from [3]_ combines the generalization of CSA
(Classical
    Simulated Annealing) and FSA (Fast Simulated Annealing) [1]_ [2]_
coupled
    to a strategy for applying a local search on accepted locations [4]_.
    An alternative implementation of this same algorithm is described in
[5]_
         and benchmarks are presented in [6]_. This approach introduces an
                                                                  advanced
    method to refine the solution found by the generalized annealing
    process. This algorithm uses a distorted Cauchy-Lorentz visiting
    distribution, with its shape controlled by the parameter :math:`q_{v}`

    .. math::

        g_{q_{v}}(\\Delta x(t)) \\propto \\frac{ \\
        \\left[T_{q_{v}}(t) \\right]^{-\\frac{D}{3-q_{v}}}}{ \\
        \\left[{1+(q_{v}-1)\\frac{(\\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}}}

     Where :math:`t` is the artificial time. This visiting distribution is
                                                                  used
    to generate a trial jump distance :math:`\\Delta x(t)` of variable
```

```
    :math:`x(t)` under artificial temperature :math:`T_{q_{v}}(t)`.

    From the starting point, after calling the visiting distribution
    function, the acceptance probability is computed as follows:


    .. math::

        p_{q_{a}} = \\min{\\{1,\\left[1-(1-q_{a}) \\beta \\Delta E \\right]^{
\\
        \\frac{1}{1-q_{a}}}\\}}


    Where :math:`q_{a}` is a acceptance parameter. For :math:`q_{a}<1`, zero
    acceptance probability is assigned to the cases where

    .. math::

        [1-(1-q_{a}) \\beta \\Delta E] < 0

    The artificial temperature :math:`T_{q_{v}}(t)` is decreased according to

    .. math::

        T_{q_{v}}(t) = T_{q_{v}}(1) \\frac{2^{q_{v}-1}-1}{\\left( \\
+ t\\right)^{q_{v}-1}-1}

    Where :math:`q_{v}` is the visiting parameter.

    .. versionadded:: 1.2.0

    References
    ----------
    .. [1] Tsallis C. Possible generalization of Boltzmann-Gibbs
        statistics. Journal of Statistical Physics, 52, 479-487 (1998).
    .. [2] Tsallis C, Stariolo DA. Generalized Simulated Annealing.
        Physica A, 233, 395-406 (1996).

    .. [3] Xiang Y, Sun DY, Fan W, Gong XG. Generalized Simulated
        Annealing Algorithm and Its Application to the Thomson Model.
        Physics Letters A, 233, 216-220 (1997).
    .. [4] Xiang Y, Gong XG. Efficiency of Generalized Simulated
        Annealing. Physical Review E, 62, 4473 (2000).
    .. [5] Xiang Y, Gubian S, Suomela B, Hoeng J. Generalized
        Simulated Annealing for Efficient Global Optimization: the GenSA
        Package for R. The R Journal, Volume 5/1 (2013).
    .. [6] Mullen, K. Continuous Global Optimization in R. Journal of
        Statistical Software, 60(6), 1 - 45, (2014).
        :doi:`10.18637/jss.v060.i06`

    Examples
    --------
    The following example is a 10-D problem, with many local minima.
    The function involved is called Rastrigin
    (https://en.wikipedia.org/wiki/Rastrigin_function)

    >>> import numpy as np
    >>> from scipy.optimize import dual_annealing
    >>> func = lambda x: np.sum(x*x - 10*np.cos(2*np.pi*x)) +
    10*np.size(x)
    >>> lw = [-5.12] * 10
    >>> up = [5.12] * 10
    >>> ret = dual_annealing(func, bounds=list(zip(lw, up)))
    >>> 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)
```

```
    # Initialization of random Generator for reproducible runs if seed
provided
    rand_state = check_random_state(seed)
    # Initialization of the energy state
    energy_state = EnergyState(lower, upper, callback)
    energy_state.reset(func_wrapper, rand_state, x0)
    # Minimum value of annealing temperature reached to perform
    # re-annealing
    temperature_restart = initial_temp * restart_temp_ratio
    # VisitingDistribution instance
    visit_dist = VisitingDistribution(lower, upper, visit, rand_state)
    # Strategy chain instance
    strategy_chain = StrategyChain(accept, visit_dist, func_wrapper,
                                   minimizer_wrapper, rand_state,
energy_state)
    need_to_stop = False
    iteration = 0
    message = []
    # OptimizeResult object to be returned

    optimize_res = OptimizeResult()
    optimize_res.success = True
    optimize_res.status = 0

    t1 = np.exp((visit - 1) * np.log(2.0)) - 1.0
    # Run the search loop
    start_time = time.time()
    pbar2.write(f"Starting search at {time.strftime('%H:%M:%S',
time.localtime())}")

    while not need_to_stop:
        for i in range(maxiter):
            # Compute temperature for this step
            s = float(i) + 2.0
            t2 = np.exp((visit - 1) * np.log(s)) - 1.0
            temperature = initial_temp * t1 / t2
            if iteration >= maxiter:
              message.append("Maximum number of iteration reached")
                need_to_stop = True
                break
            # Need a re-annealing process?
            if temperature < temperature_restart:
                energy_state.reset(func_wrapper, rand_state)
                break
            # starting strategy chain
            val = strategy_chain.run(i, temperature)
            if val is not None:
                message.append(val)
                need_to_stop = True
                optimize_res.success = False
                break
          # Possible local search at the end of the strategy chain
            if not no_local_search:
                val = strategy_chain.local_search()
                if val is not None:
                    message.append(val)
                    need_to_stop = True
                    optimize_res.success = False
                    break
            iteration += 1
            pbar.update(1)
            rate = pbar.format_dict['rate']
            if rate is None:
                r.append(0)
            else:
                r.append(1/rate)

            # print("iteration: ", iteration)
            # elapsed_time = time.time_ns() - start_time
            # elapsed_time_seconds = elapsed_time / 1e9
            # estimated_time = elapsed_time_seconds / iteration *
            (maxiter -
 iteration)
            # estimated_finish_time = time.strftime('%H:%M:%S',
            # time.localtime(time.time() + estimated_time))
            # print(f"ETA: {estimated_finish_time}")

    # Setting the OptimizeResult values
    optimize_res.x = energy_state.xbest
    optimize_res.fun = energy_state.ebest
    optimize_res.nit = iteration
    optimize_res.nfev = func_wrapper.nfev
    optimize_res.njev = func_wrapper.ngev
    optimize_res.nhev = func_wrapper.nhev
    optimize_res.message = message

    pbar2.write(f"Finished search at {time.ctime()} in\
          {time.strftime('%H:%M:%S',time.gmtime(time.time()  -
 start_time))}")

    return optimize_res
```