Recherche du chemin le plus rapide sur un circuit



KLEINHANS Jules

CONTRAIRES Noé

Table des matières

I) Modélisation et algorithme génétique

a) Modélisation
b) Théories, principes et implémentation
c) Résultats et améliorations

II) Méthode par descente de gradient

III) Conclusion

a) Bilan
b) Annexes

Partie I : Modélisation et Algorithme génétique

a) Modélisation

#-> “Route” du circuit

*-> Départ

+-> Arrivée



Partie I : Modélisation et Algorithme génétique

a) Modélisation

Chaque voiture est représentée
par un carré de couleur qui possède :

-Une position

-Une séquence de déplacement

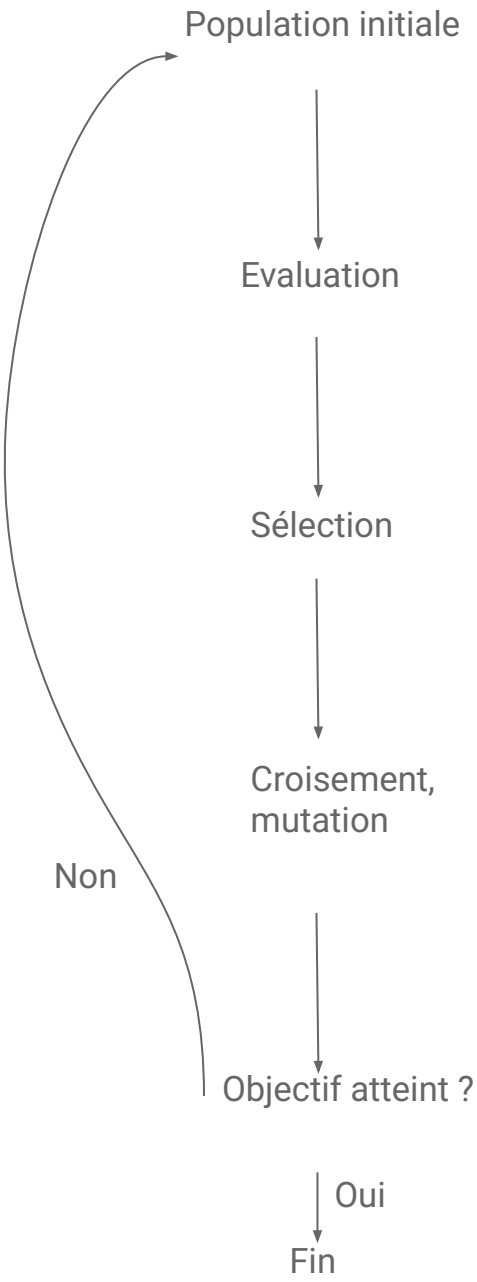
-État de “vie/mort” en fonction de sa position

-Un score

Le circuit a alors la forme suivante :



b) Théorie, principes et implémentation



Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

Les voitures forment la population.

Voiture sans contraintes physiques :

-Haut

-Bas

-Gauche

-Droite

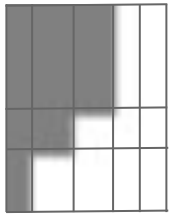
-Rien

Chaque voitures exécute ses commandes jusqu'à :

-Atteindre la fin



-Sortir du circuit

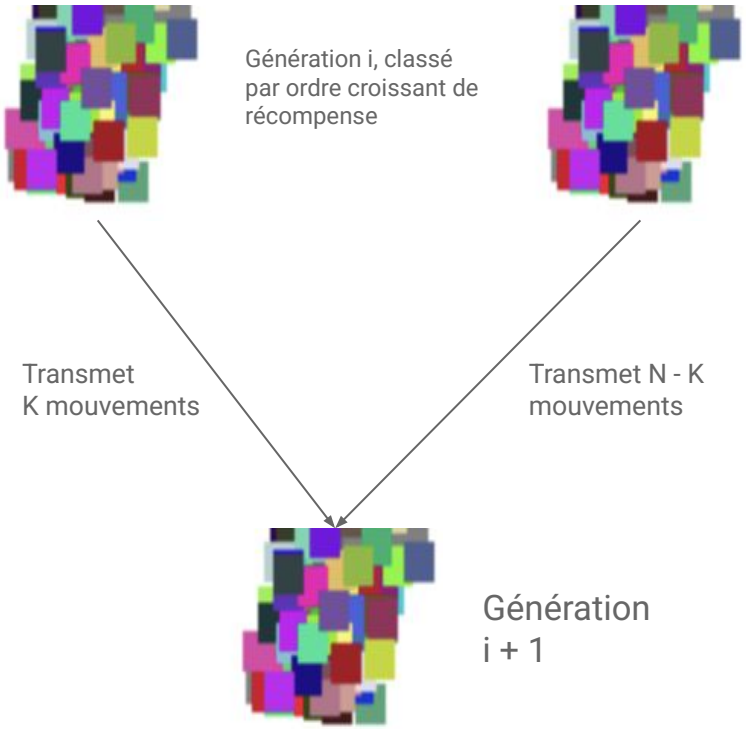


-Avoir épuisé ses mouvements

Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

Croisements et mutations pour les deux types de voitures.

Croisement :



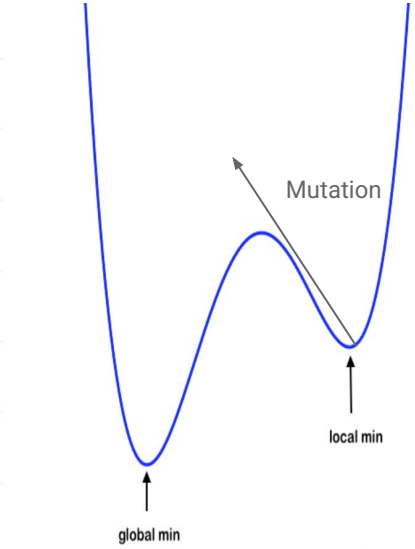
Mutation :

-Intérêt : Evite de rester dans un minimum local

-Probabilité x d'être muté

-Parmis celles mutés y%

des mouvements le sont



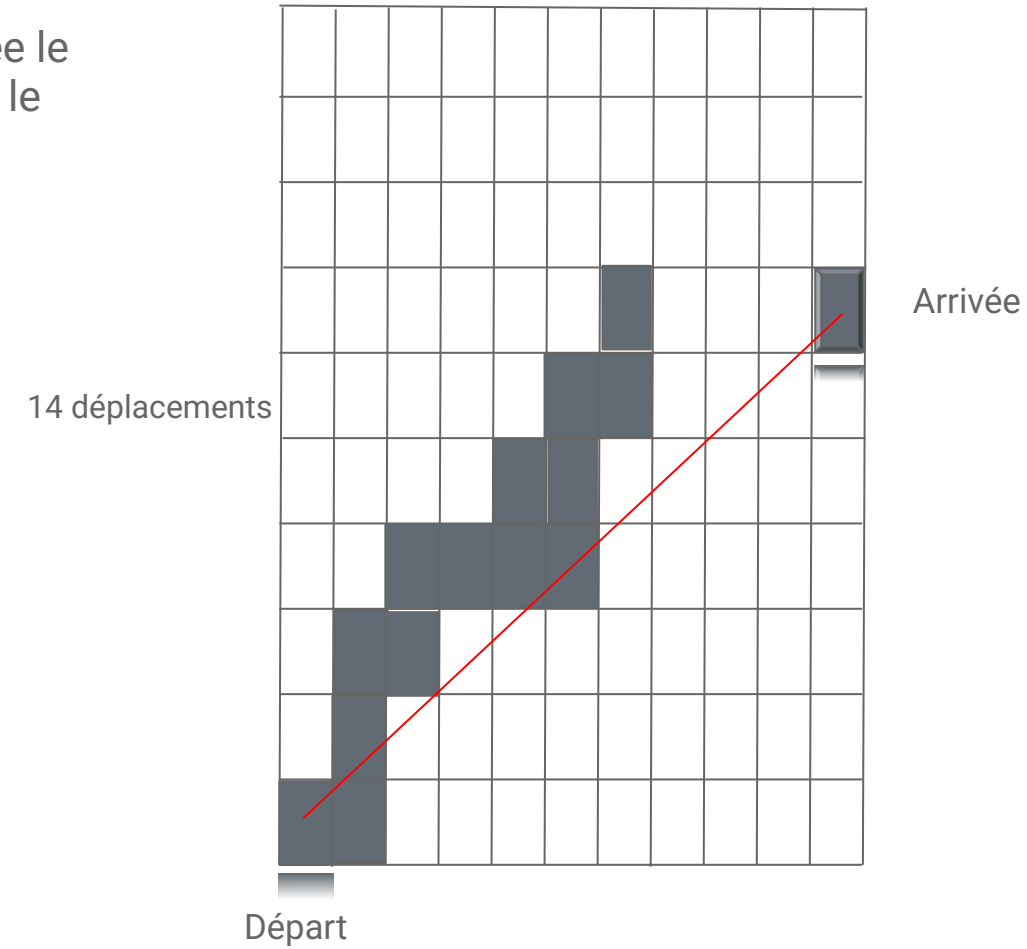
L'objectif est donc d'atteindre l'arrivée le plus rapidement possible, soit ici, en le moins de déplacement possible.

Les paramètres à considérer sont :

-La distance à l'arrivée

-Le "temps" passé sur le circuit

-L'exploration du circuit



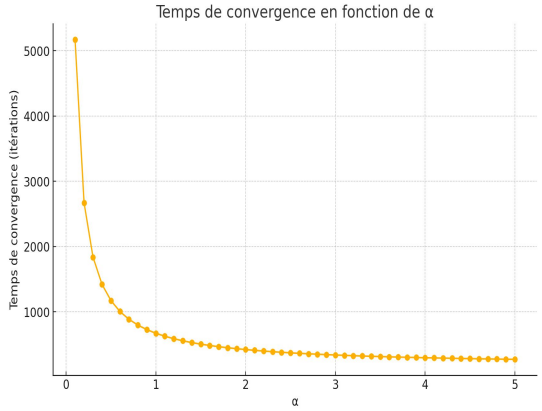
Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

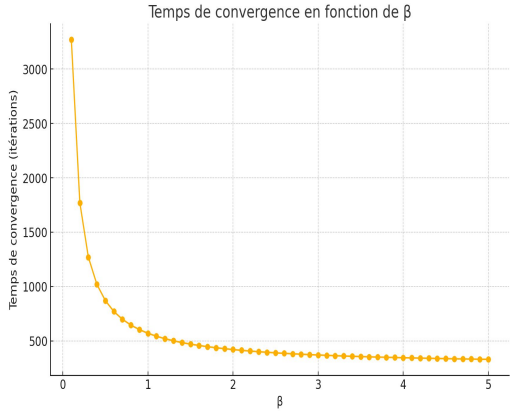
On construit donc en conséquence la fonction de récompense :

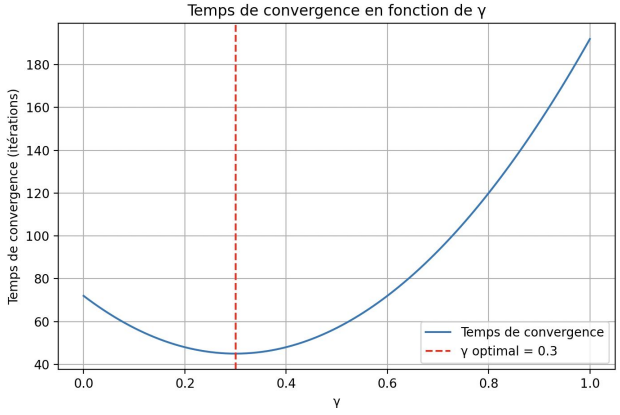
α * (10 000 / (1 + distance_arrivée)) +

β * cases_visitées -

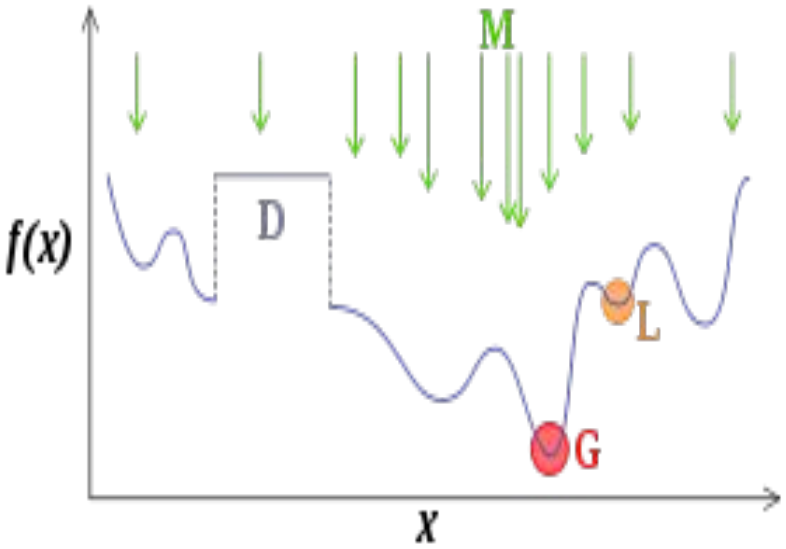
γ * distance_parcourue







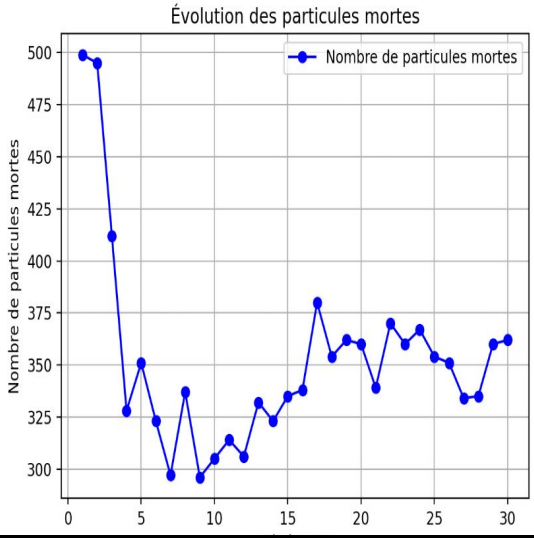
L'implémentation est en python, le langage permet une illustration des déplacements par des bibliothèques tel que pygame plus facile que dans d'autres langages.



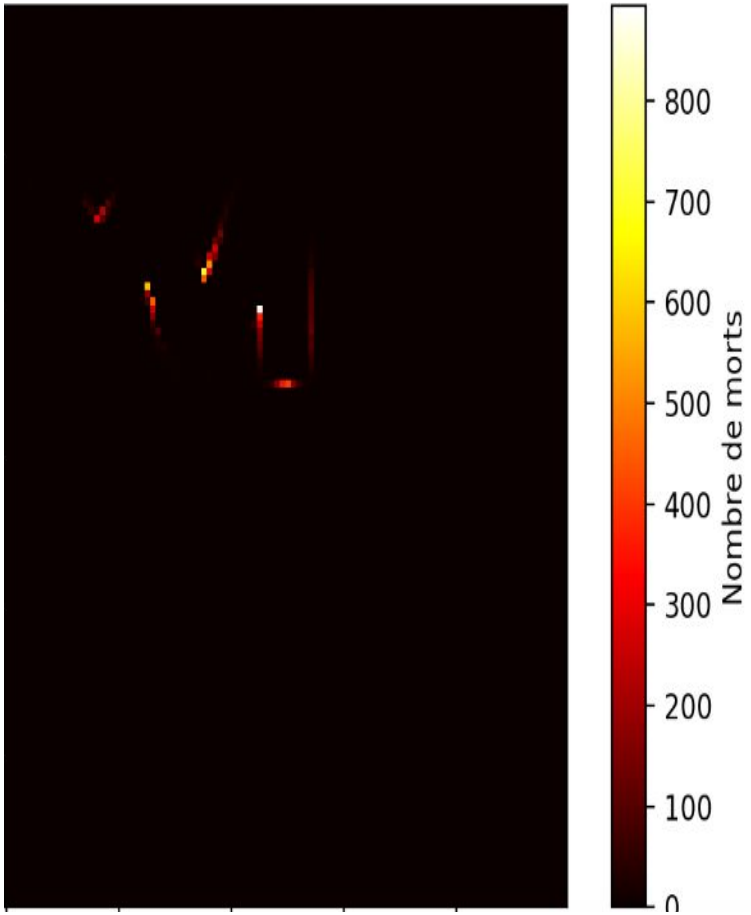
La difficulté de l'algorithme réside dans la recherche d'une bonne fonction de récompense.



Partie I : Modélisation et Algorithme génétique
c) Résultats et Améliorations

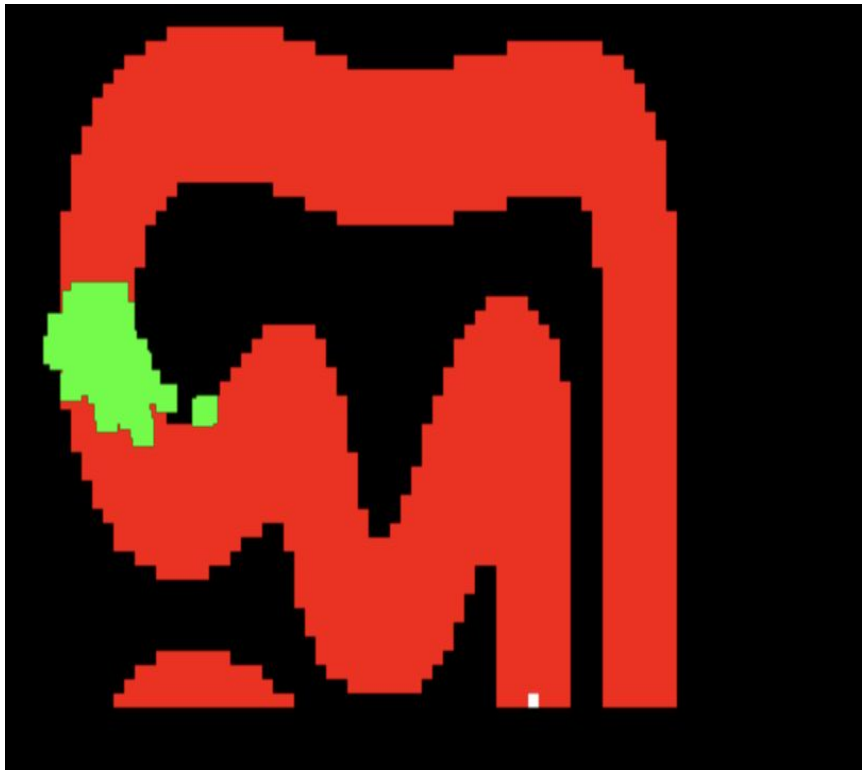






10 000 Mouvements





Paramètres physiques :

Accélération_MAX = 1G

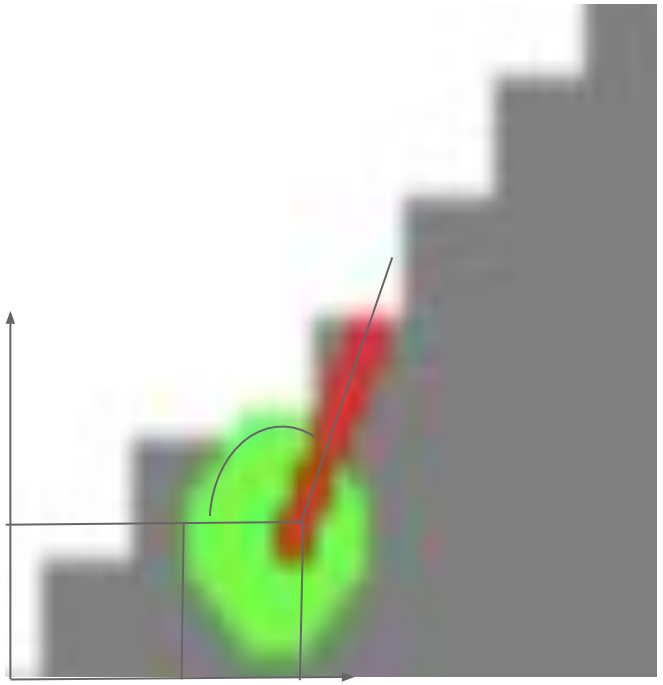
Frein_MAX = 6G

ACCEL_LATERALE_MAX = 2G

Angle de braquage maximum.

Accélération 80% du temps, on freine sinon.

L'accélération est tiré aléatoirement entre 0 et Accélération_MAX, Idem pour le freinage.

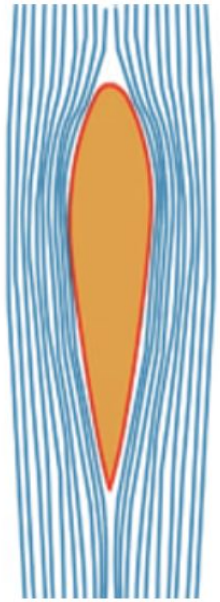


Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

Frottement fluide : K * v

K : Coefficient de frottement dépendant de la géométrie de l'objet

v : Vitesse



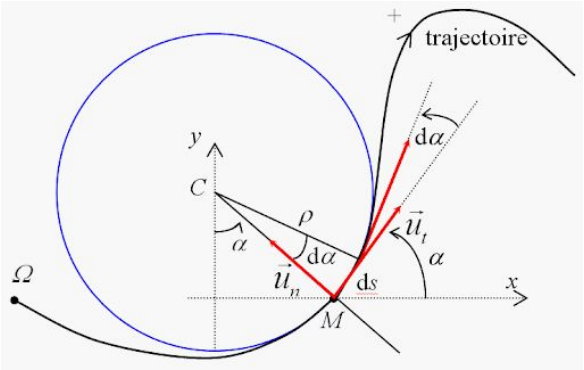
Accélération latérale :

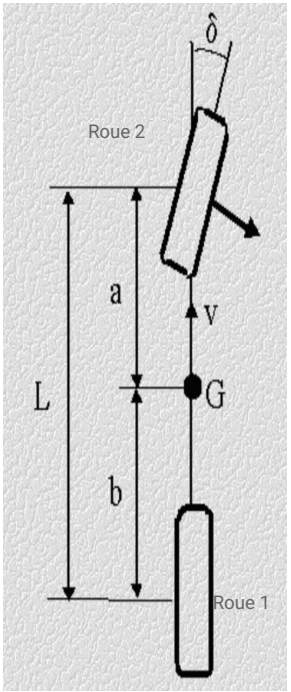
Modèle simplifié dit "2 roues" :

Accélération latérale =

Accélération normale dans un repère de Frenet = v^2/R

R = L / tan (δ)





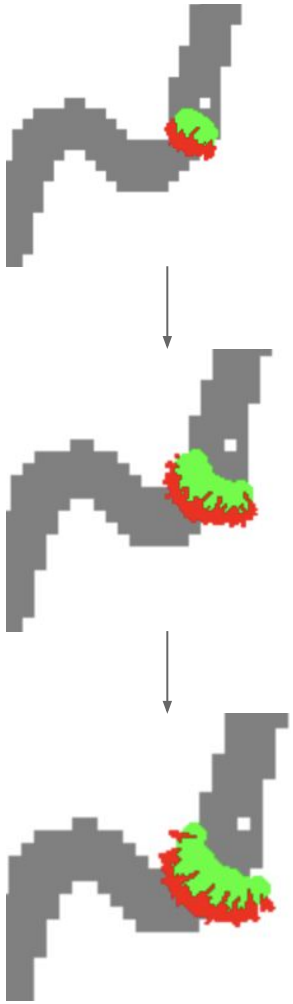
Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

Avant chaque itération on connaît :

- -x, y : position actuelle
- -v : vitesse actuelle
- -θ : orientation actuelle
- -a : accélération (tirée aléatoirement initialement)
- -δ : angle de braquage

On cherche à intégrer :

- -x’ = v * cos(θ)
- -y’ = v * sin(θ)
- -v’ = a
- -θ’ = v/L * tan(δ)



Partie I : Modélisation et Algorithme génétique
b) Théorie, principes et implémentations

Méthode d'intégration d'Euler :

-Revient à approximer la fonction par la méthode des rectangles.

Algorithme d'Euler :

-Initialisation du pas h et de la durée T.

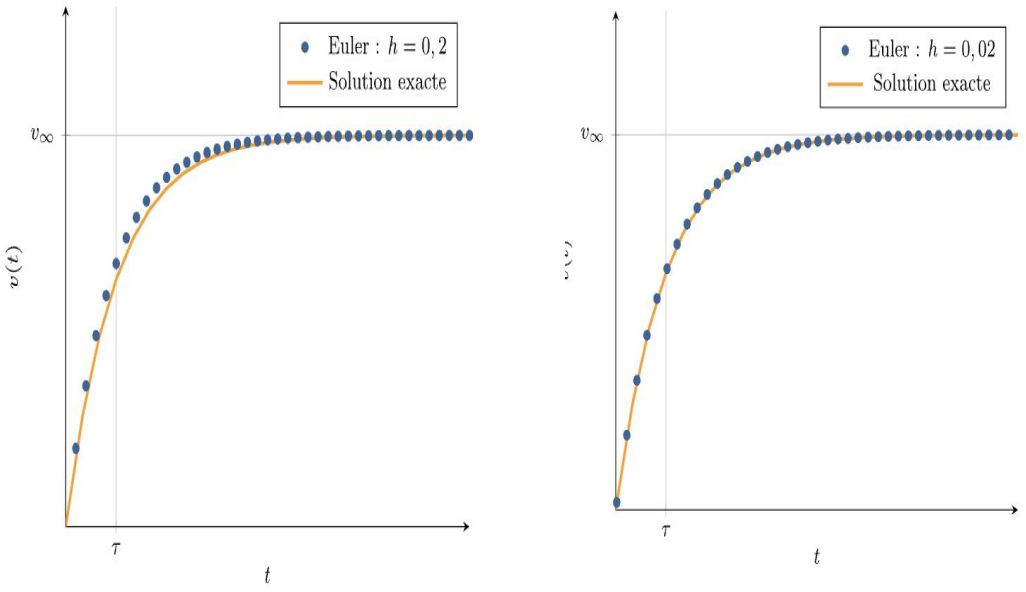
-Initialisation des conditions initiales : t = 0 et y=y(0).

-Tant que t ≤ T faire :

Calcul de k1 = f (t , y)

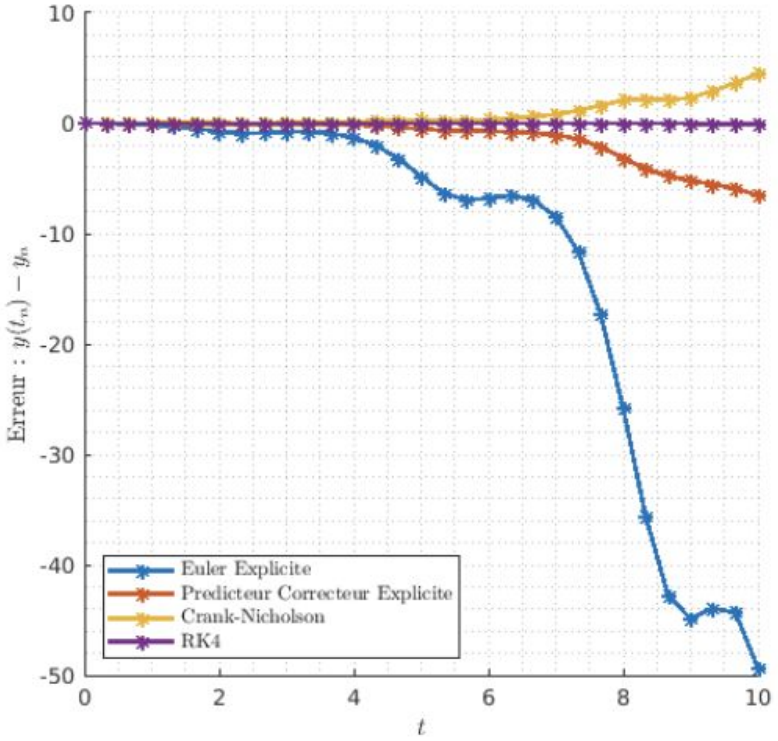
y=y+hk1

t = t + h



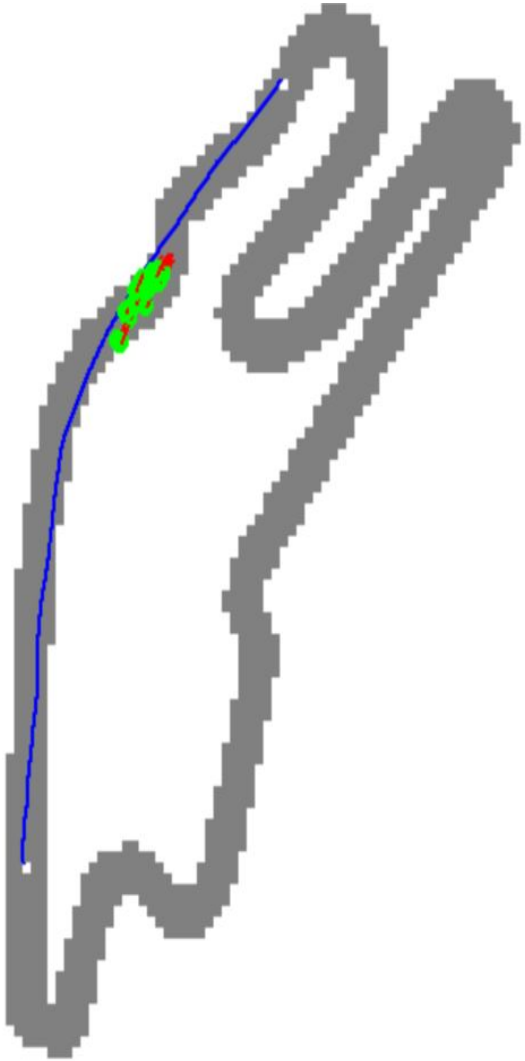
Autre modèle d'intégration : Méthode de Runge–Kutta

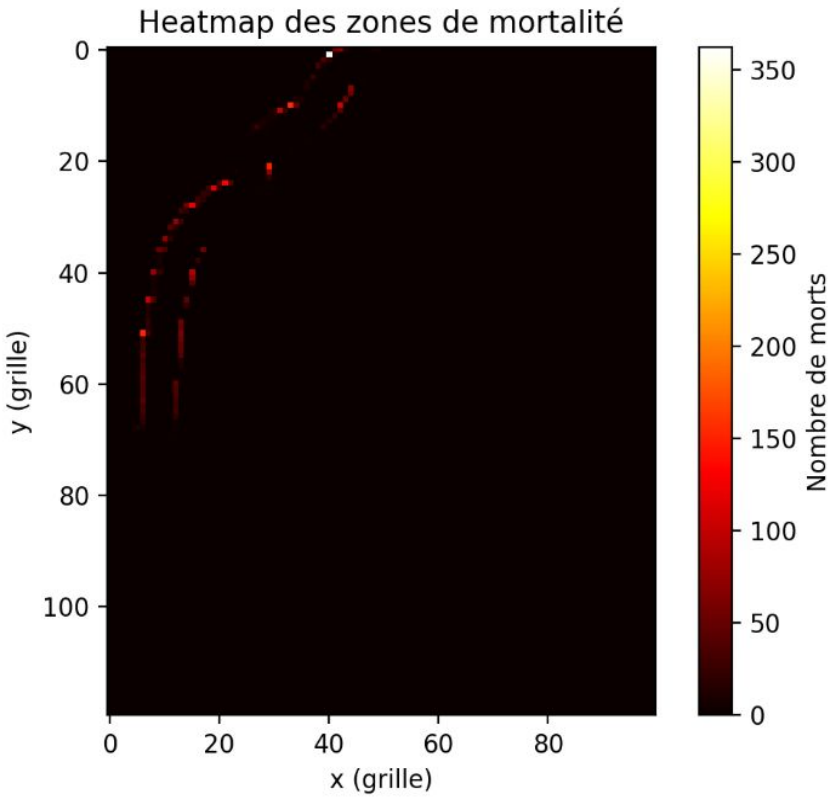
$$k_1 = f\left(t_n, y_n\right)$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$
$$k_4 = f\left(t_n + h, y_n + hk_3\right)$$



Solutions numériques de : y' = (sint(t))^2 * y (avec y(0) = 1)

Partie I : Modélisation et Algorithme génétique
c) Résultats et Améliorations

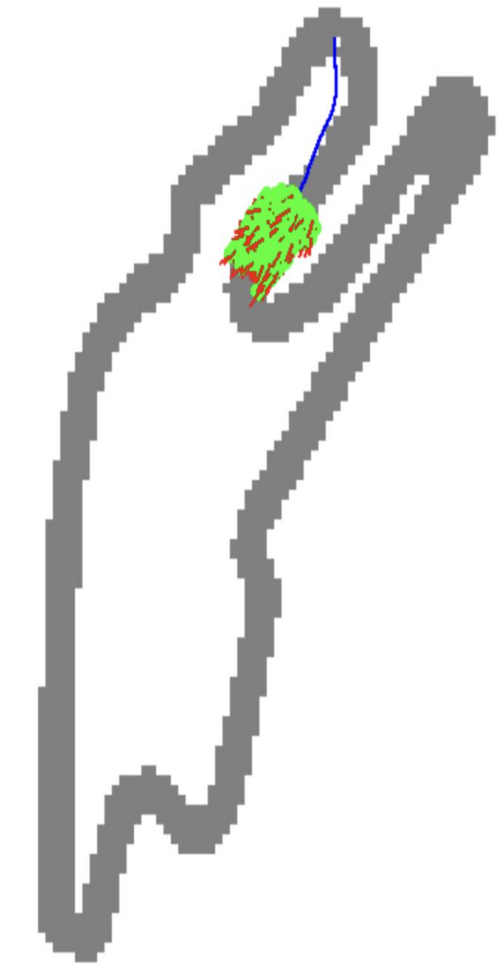


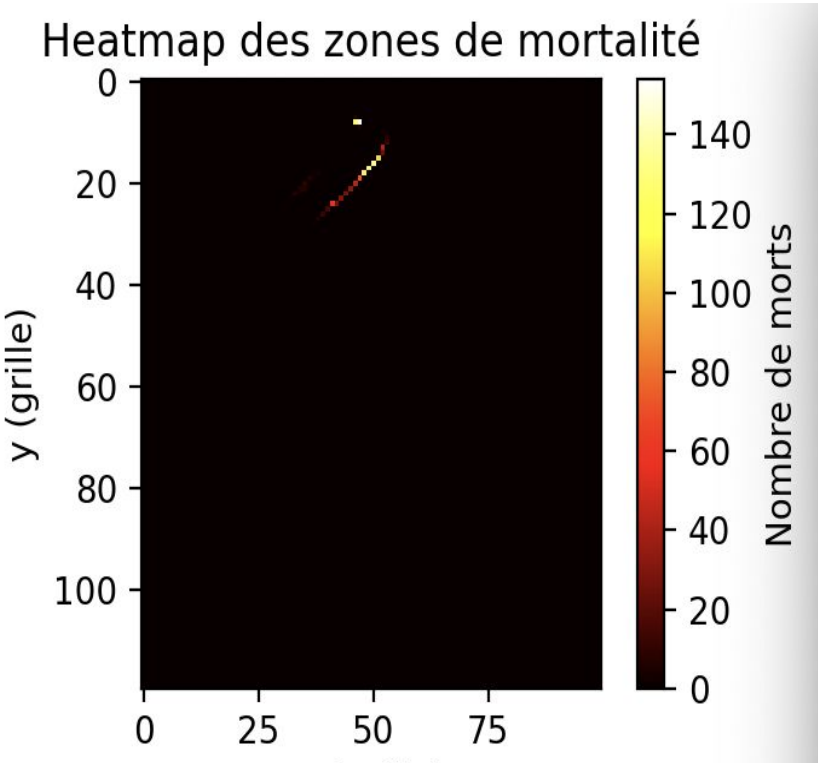


1 000 voitures, 35 000 mouvements

Partie I : Modélisation et Algorithme génétique

c) Résultats et Améliorations

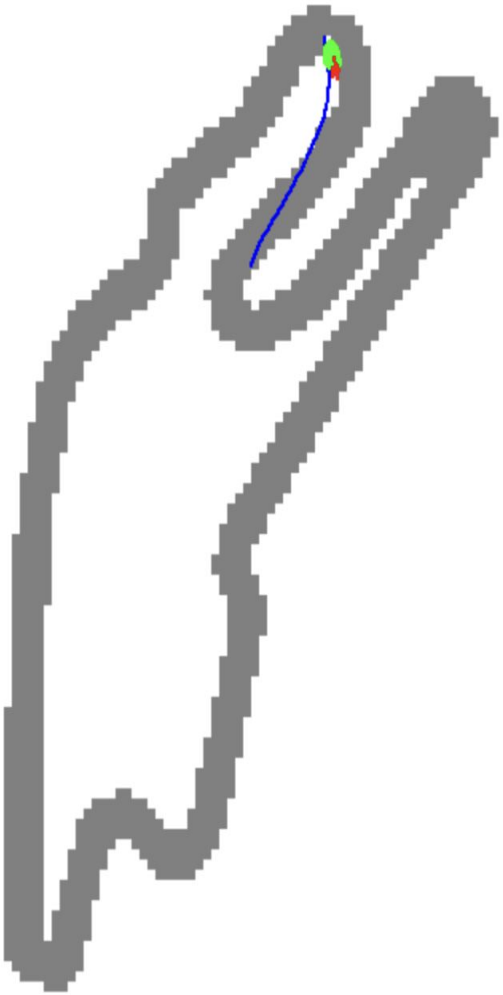


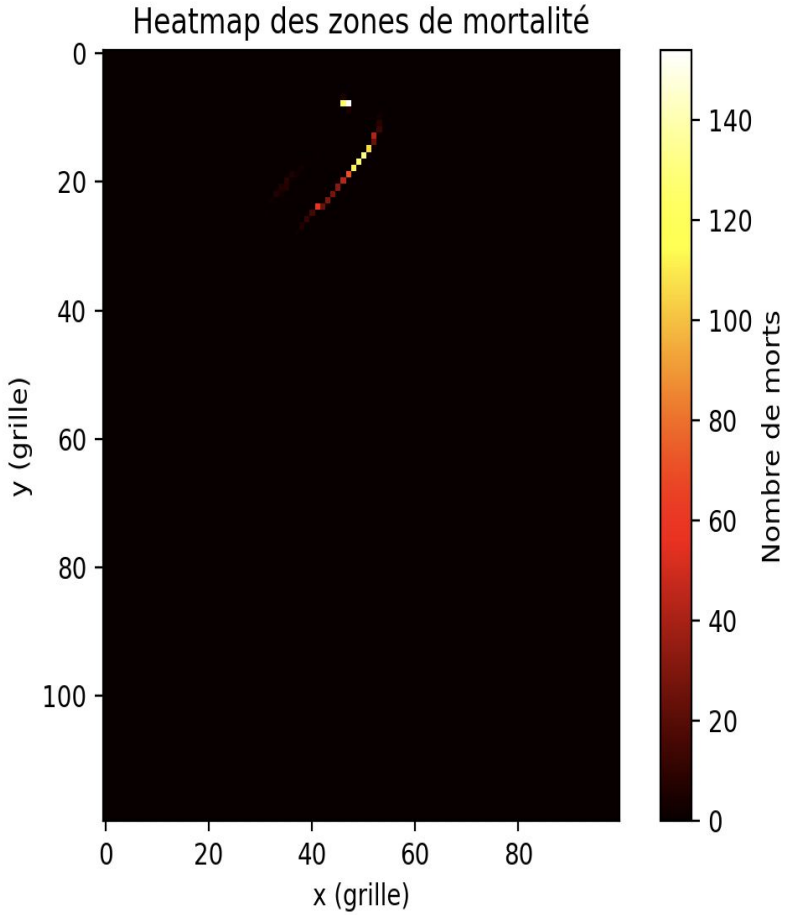


1 000 voitures, 10 000 mouvements

Partie I : Modélisation et Algorithme génétique

c) Résultats et Améliorations

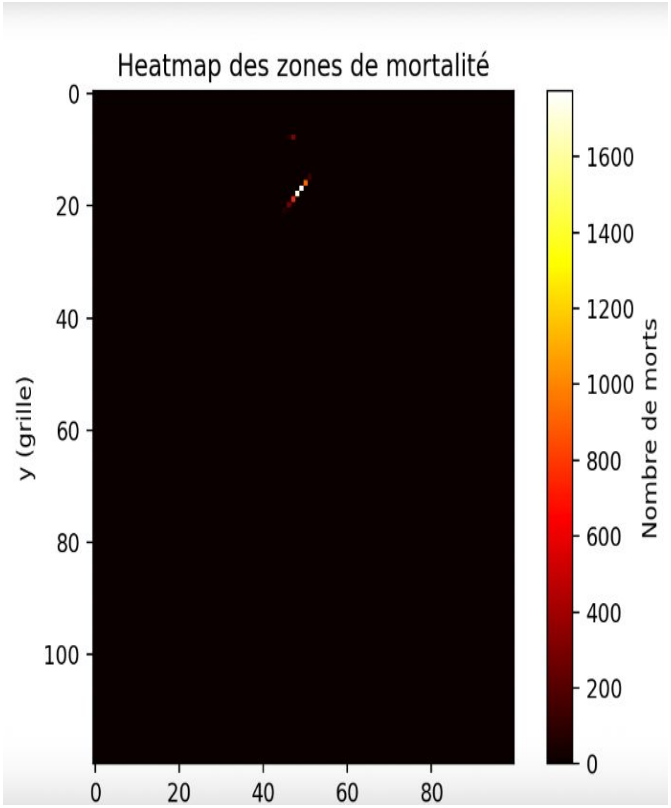


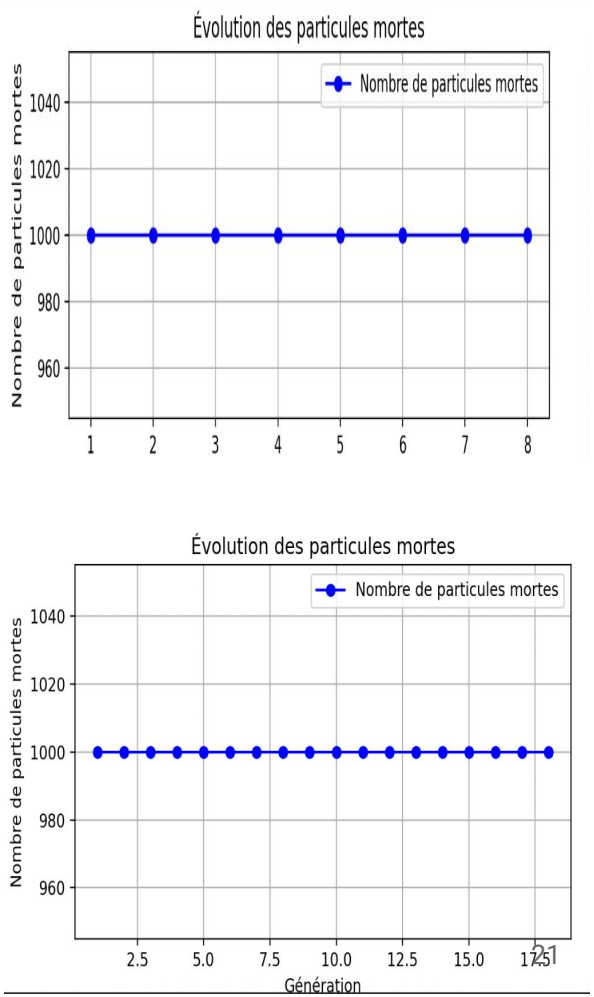


Partie I : Modélisation et Algorithme génétique

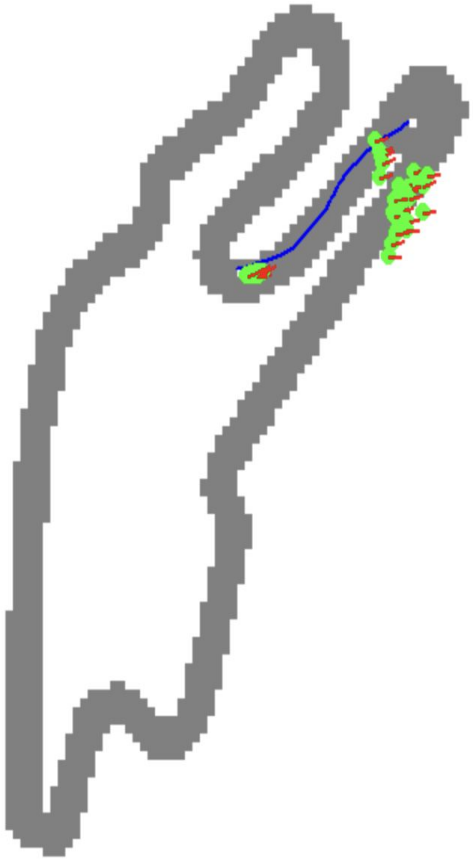
c) Résultats et Améliorations

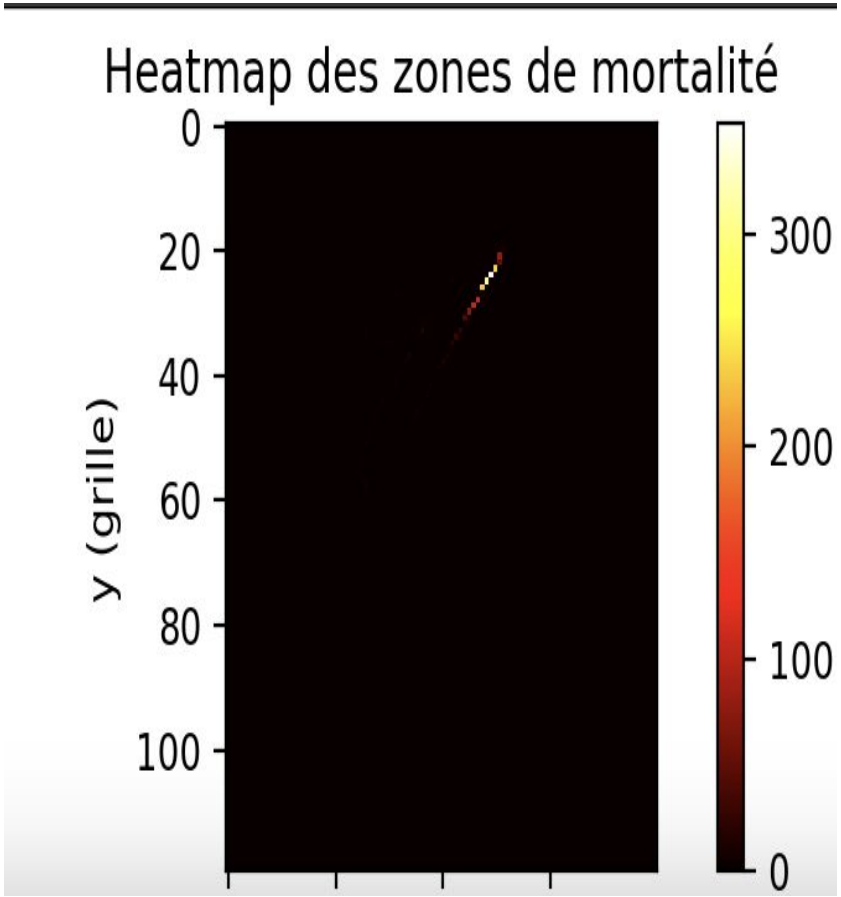






Partie I : Modélisation et Algorithme génétique
c) Résultats et Améliorations

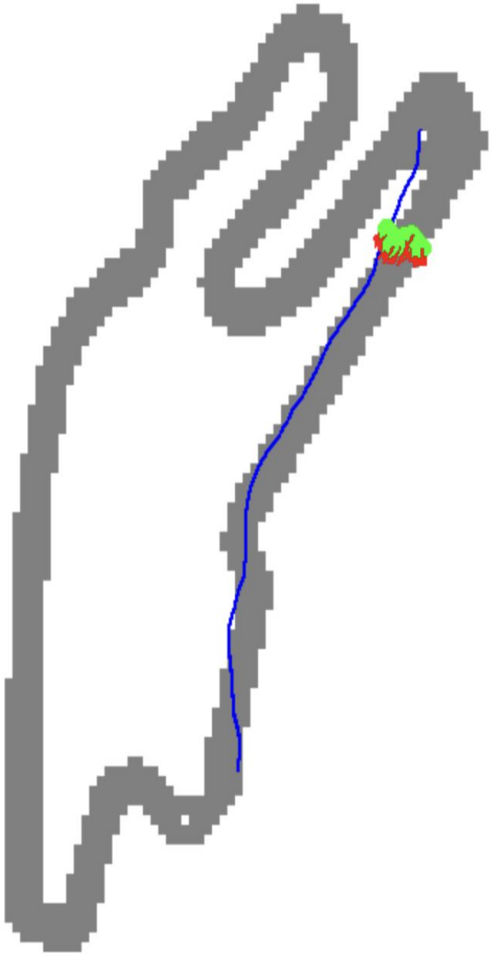


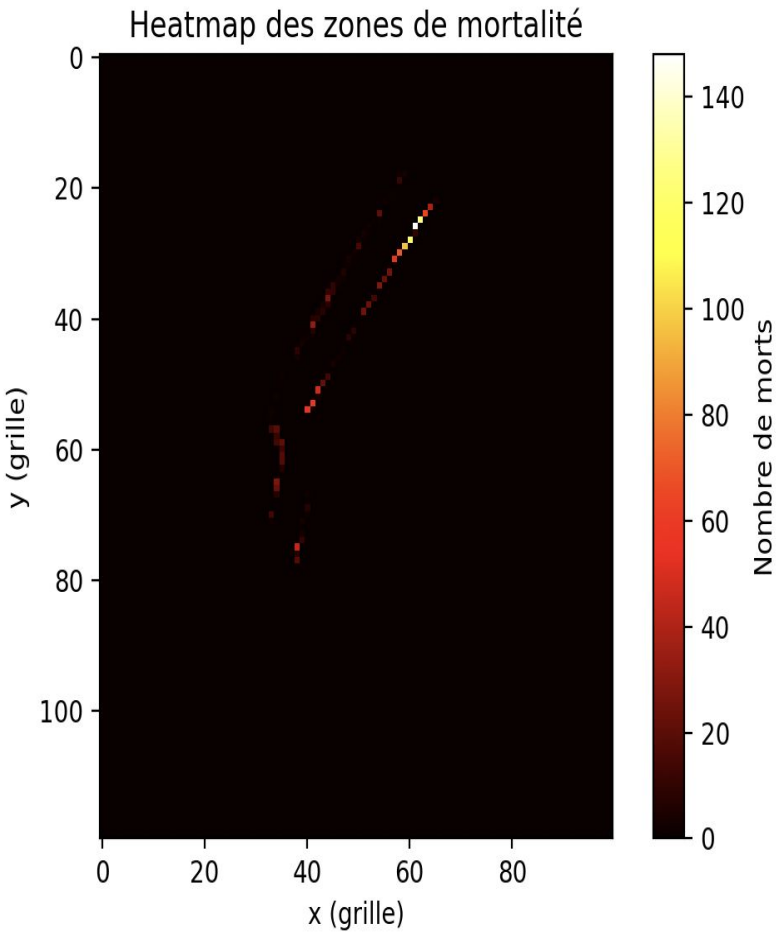


1 000 voitures, 15 000 mouvements

Partie I : Modélisation Algorithme génétique

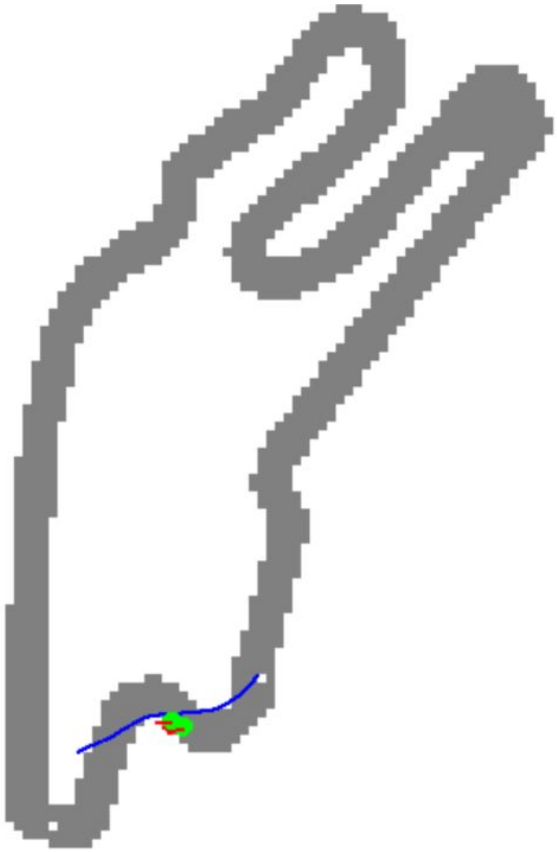
c) Résultats et Améliorations

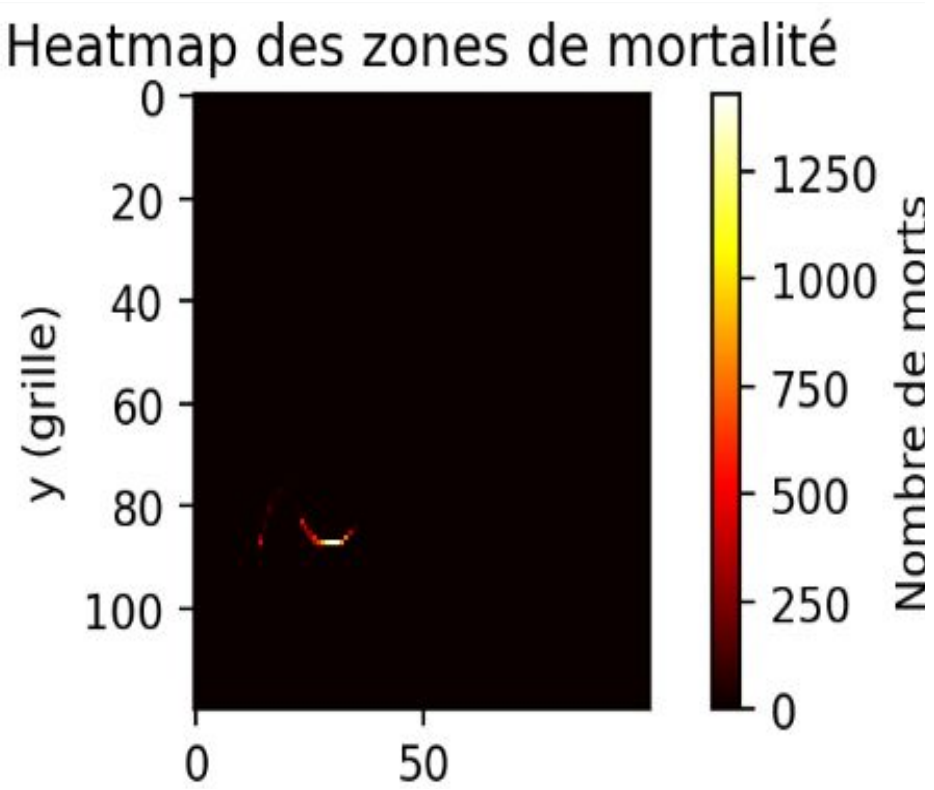




Partie I : Modélisation et Algorithme génétique

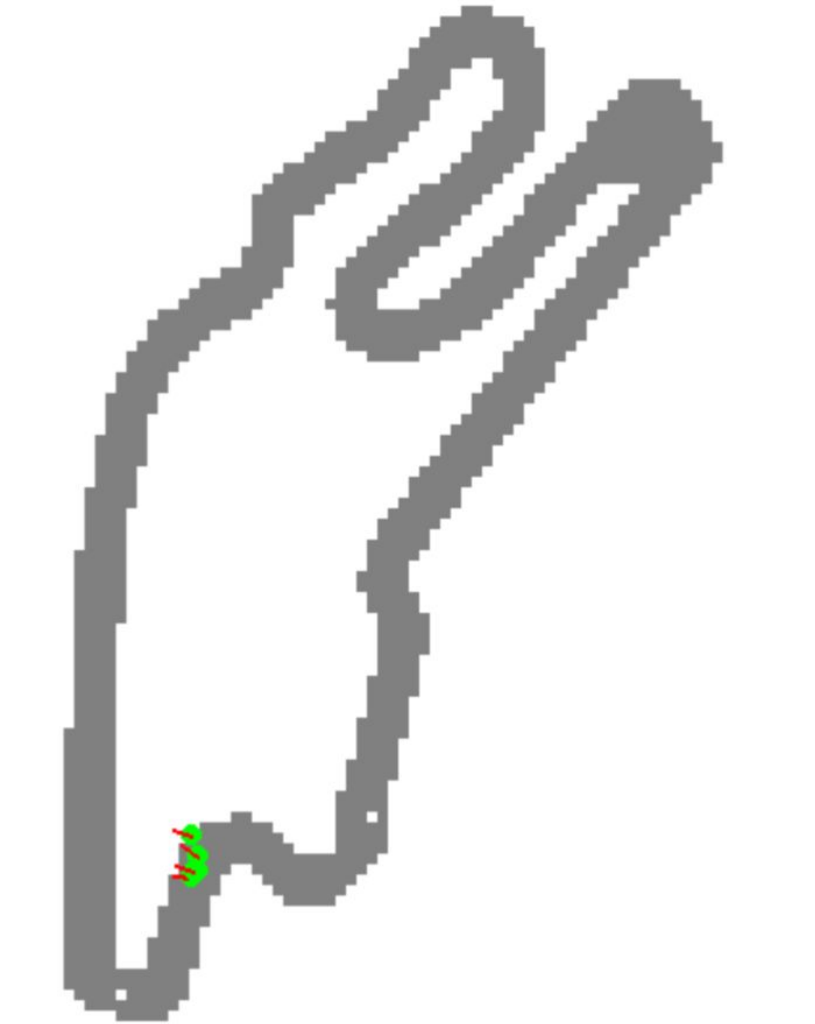
c) Résultats et Améliorations

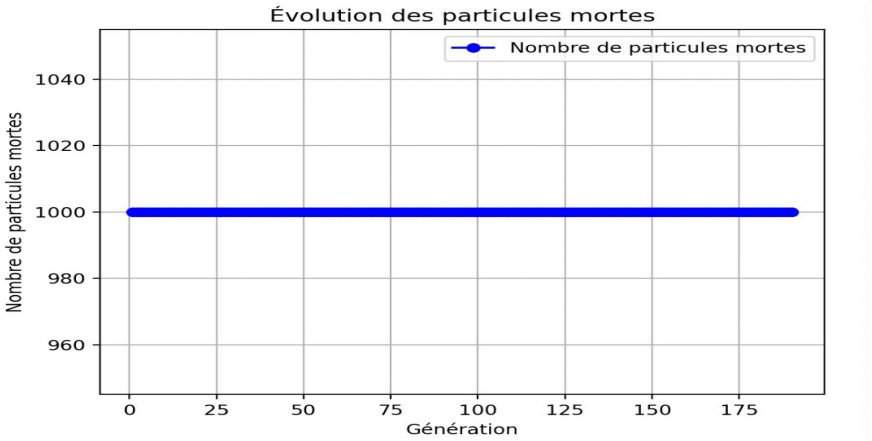


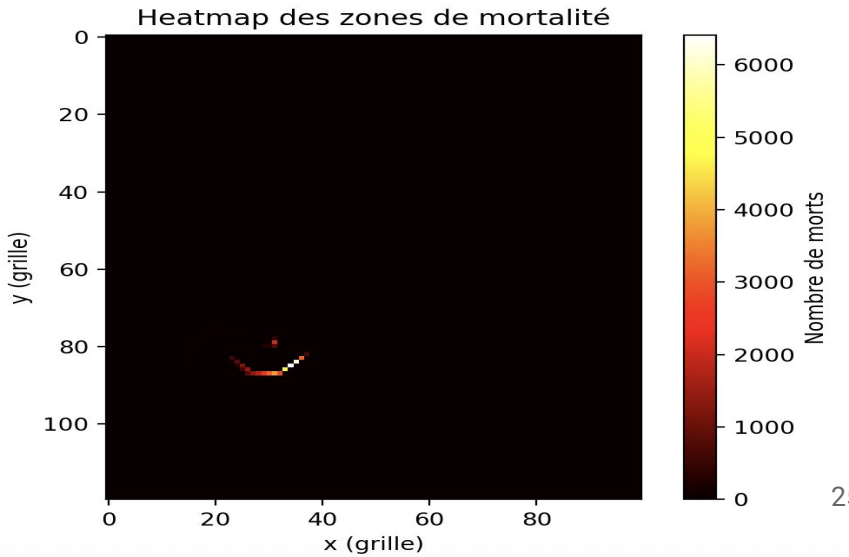


1000 voitures 10 000 mouvemnets, génération 50

Partie I : Modélisation et Algorithme génétique
c) Résultats et Améliorations







II) Méthode par descente de gradient

Recherche de minimum par suivi de la pente

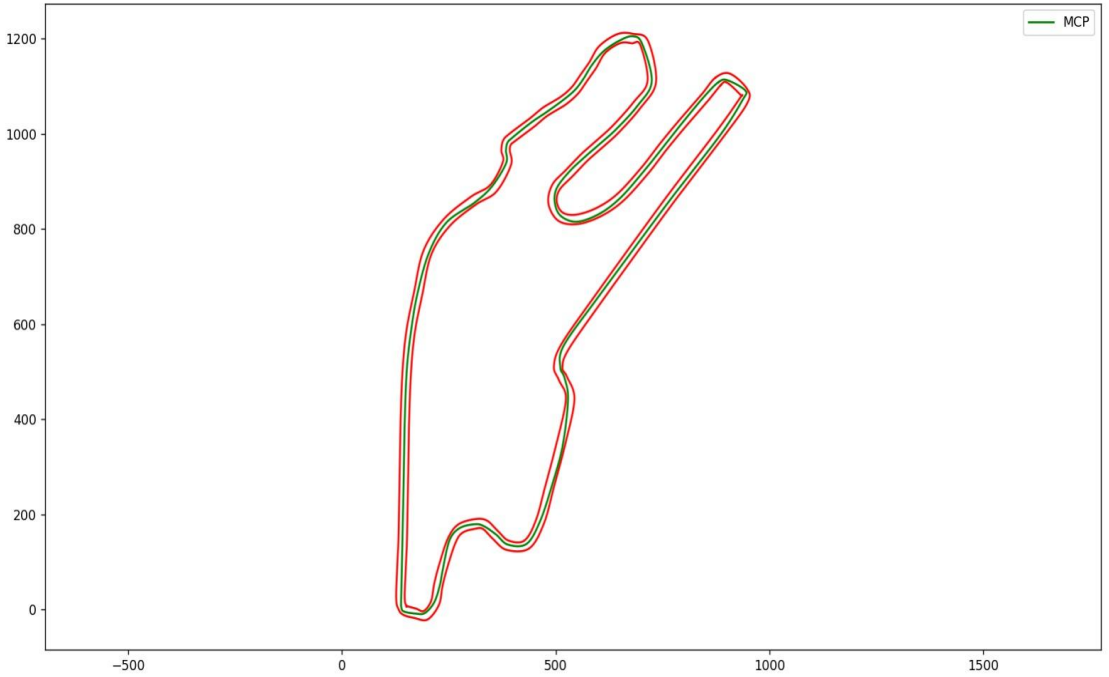
$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

On cherche à optimiser :

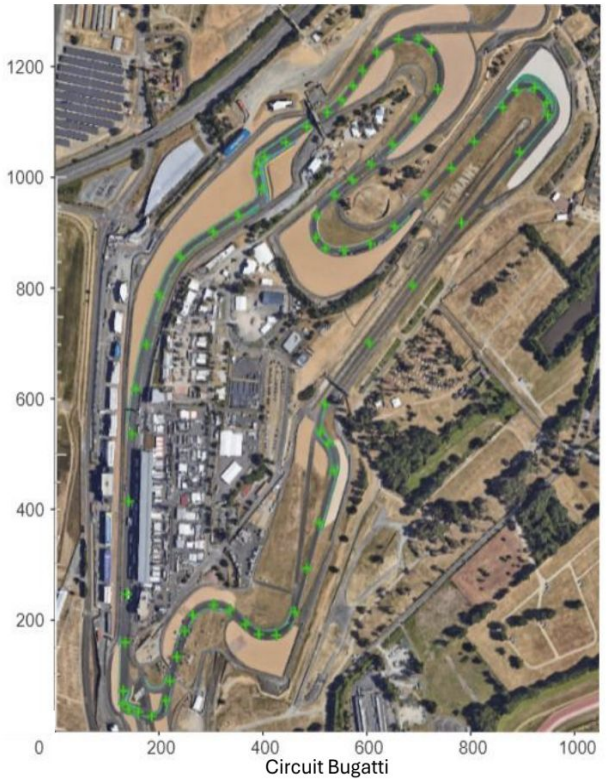
- Le temps

-La distance

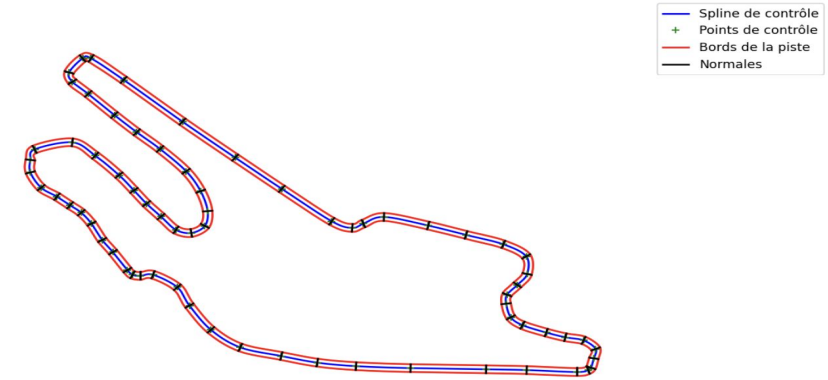
-La courbure

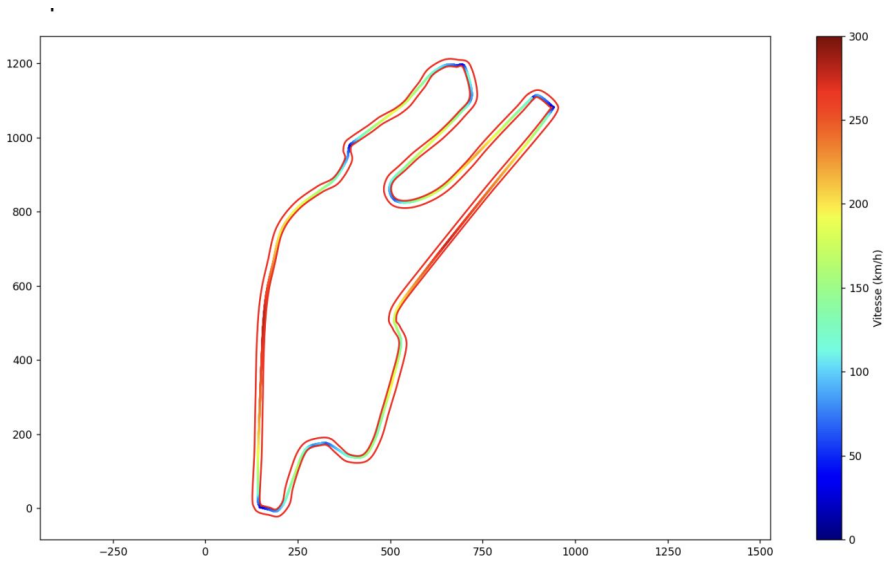


II) Méthode par descente de gradient



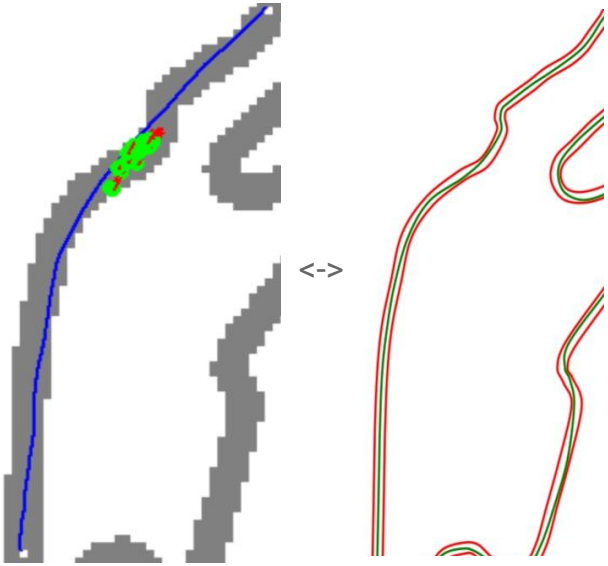
Circuit Bugatti

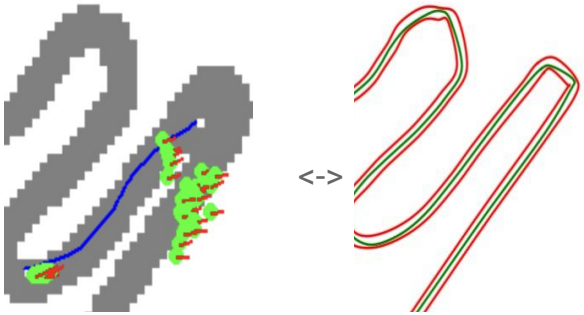


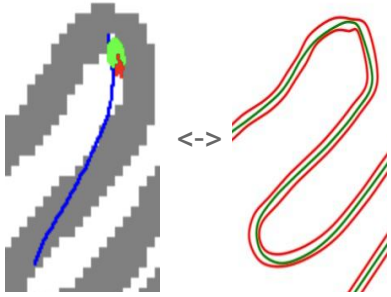


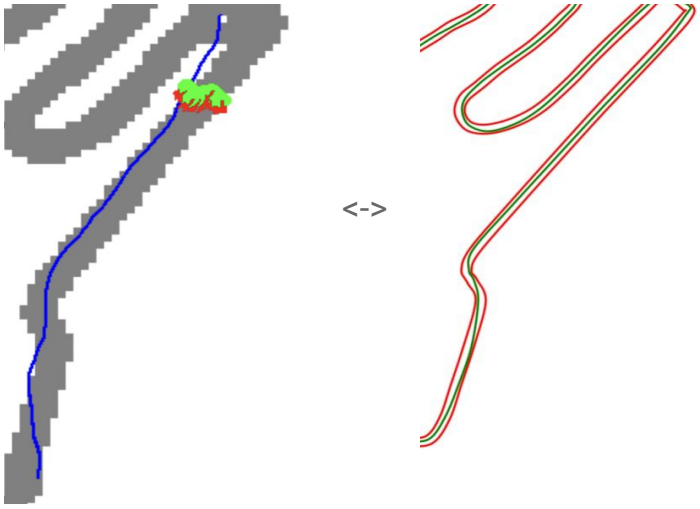
Partie III : Conclusion

a) Bilan









Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```python
import pygame
import random
import matplotlib.pyplot as plt
import math

RESOLUTION = (500, 600)
WINDOW = pygame.display.set_mode(RESOLUTION)
BLACK = (255, 255, 255) #Défini la couleur de l'cran (modifié en blanc car plus esthétique)
GO = True

import numpy as np

HEATMAP_SCALE_X = 5
HEATMAP_SCALE_Y = 5
heatmap_width = RESOLUTION[0] // HEATMAP_SCALE_X
heatmap_height = RESOLUTION[1] // HEATMAP_SCALE_Y
death_heatmap = np.zeros((heatmap_height, heatmap_width))


NB_VOITURE = 1000
GRADED_RETAIN_PERCENT = 0.05
NUMBER_OF_GRADED_RETAIN = int(NB_VOITURE * GRADED_RETAIN_PERCENT)
PARENTS_GARDES = 0.1 #Pourcentage de parents non gardés retenus
NUMBER_OF_MOVES = 20000

#L'idée est ici qu'on a 2% d'être muté et parmis ces 2%, 30% des mouvements sont modifiés (idée implémenté dans la fct mutate)
CHANCE_DE_MUTATION = 0.02
POURCENTAGE_DE_MUTATION = 0.02
NOMBRE_DE_MUTATION = int(POURCENTAGE_DE_MUTATION * NUMBER_OF_MOVES)

SPEED_X = 1
SPEED_Y = 1
VOITURES_LIST = []
SIZE = 10
COUNT = 0

# Paramètres dynamiques
ACCEL_MAX = 10.0 #1G
FREIN_MAX = -60.0 #6G
DIRECTION_MAX = math.radians(80)
EMPATTEMENT = 2.5
ACCEL_LATERALE_MAX = 20 #2G
FPS = 60
DELTA_TEMPS = 1 / FPS
STEER_MAX = 1.0
OMEGA_MAX = math.radians(2)
```

```python
# Listes pour stocker l'évolution des générations et le nombre de particules mortes

generation_history = []
dead_history = []
plt.ion()  # Mode interactif pour matplotlib

#Il faut générer une position aléatoire pour chaque particule sinon elles se superposent
def get_random(a, b):
    return random.randint(a, b)

# Génération de population
def generate_population():
    for i in range(NB_VOITURE):
        VOITURES_LIST.append(VoitureDynamique(pygame.Rect(get_random(0, RESOLUTION[0] - SIZE), get_random(0, RESOLUTION[1] - SIZE), SIZE, SIZE)))

def draw_voitures(self, WINDOW):
    nouvelles_voitures = []
    for voiture in self.voitures:
        if voiture.est_sur_circuit(self.circuit.carres):  # Vérifie si la voiture est sur la route
            couleur = get_random(0, 255)  # Vert pour les voitures en course
            nouvelles_voitures.append(voiture)  # On garde la voiture
        else:
            couleur = (255, 255, 255)

        pygame.draw.rect(WINDOW, couleur, (voiture.rect.x, voiture.rect.y, 5, 5))  # Dessiner la voiture

    self.voitures = nouvelles_voitures  # Mise à jour de la liste des voitures
    print(f"Nombre de voitures restantes : {len(self.voitures)}")

def voiture_bouge():
    for Voiture_classe in VOITURES_LIST:
        Voiture_classe.move(Voiture_classe.moves[COUNT][0], Voiture_classe.moves[COUNT][1])

def bouge_aléatoire():
    for Voiture_classe in VOITURES_LIST:
        Voiture_classe.gen_random_mouvements()

def quitAlgo ():
    pygame.quit()
    quit()
```

Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```python
class VoitureDynamique:
    # Classe représentant une voiture dynamique avec intégration RK4
    def __init__(self, rect, ID=None):
        self.rect = rect
        self.ID = ID
        self.x = float(rect.x)
        self.y = float(rect.y)
        self.v = 0.0             # vitesse (m/s)
        self.ori = 21        # orientation (rad)
        self.moves = []
        self.on_track_time = 0
        self.distance_parcourue = 0.0
        self.visited = set()
        self.vie = True
        self.distance_with_end = 1
        self.path = [(self.x, self.y)]
        # paramètres physiques
        self.longueur = EMPATTEMENT    # empattement
        self.dt = DELTA_TEMPS
        self.coeff_frottement = 0.1

    def est_sur_circuit(self, carres, other_carres):
        for car in carres + other_carres:
            if self.rect.colliderect(car): return True
        return False

    def move(self, carres, other_carres, end_pos, count):
        if not self.vie: return
        # commandes du pas
        a_cmd, steer_cmd = self.moves[count]
        # calcul angle de braquage instantané
        delta = steer_cmd * DIRECTION_MAX
        # centripète avant intégration
        if abs(delta) > 1e-3 and self.v > 0:
            R = self.longueur / math.tan(delta)
            if (self.v**2 / abs(R)) > ACCEL_LATERALE_MAX:
                self.vie = False
                return
        # intégration RK4 de (x, y, ori, v)
        self.rk4_step(self.dt, a_cmd, delta)
        # frottement après déplacement
        self.v = max(0, self.v - self.coeff_frottement * self.v * self.dt)
        # mise à jour du rect et traces
        self.rect.x, self.rect.y = int(self.x), int(self.y)
        self.distance_parcourue += abs(self.v) * self.dt
        self.visited.add((self.rect.x//10, self.rect.y//10))
        self.path.append((self.x, self.y))
        # hors fenêtre
        if (self.rect.x<0 or self.rect.right>RESOLUTION[0] or
            self.rect.y<0 or self.rect.bottom>RESOLUTION[1]):
            gy, gx = self.rect.y//HEATMAP_SCALE_Y, self.rect.x//HEATMAP_SCALE_X
            if 0<=gy<heatmap_height and 0<=gx<heatmap_width:
                death_heatmap[gy,gx]+=1
            self.vie=False; return
        # hors piste
        if not self.est_sur_circuit(carres, other_carres):
            gy, gx = self.rect.y//HEATMAP_SCALE_Y, self.rect.x//HEATMAP_SCALE_X
            if 0<=gy<heatmap_height and 0<=gx<heatmap_width:
                death_heatmap[gy,gx]+=1
            self.vie=False; return
        # temps sur piste et distance à l'arrivée
        self.on_track_time += 1
        ex, ey = end_pos
        self.distance_with_end = math.hypot(self.rect.x-ex, self.rect.y-ey)
```

```python
def generer_commandes(self):
    self.moves = []
    for _ in range(NUMBER_OF_MOVES):
        if random.random()<0.6:
            a=random.uniform(0,ACCEL_MAX)
        else:
            a=random.uniform(FREIN_MAX,0)
        s=random.uniform(-1,1)
        self.moves.append((a,s))
```

```python
def rk4_step(self, dt, a, delta):
    def f(state):
        x, y, theta, v = state
        dx = v * math.cos(theta)
        dy = v * math.sin(theta)
        dtheta = v / self.longueur * math.tan(delta)
        dv = a
        return [dx, dy, dtheta, dv]
    state=[self.x, self.y, self.ori, self.v]
    k1=f(state)
    k2=f([state[i]+0.5*dt*k1[i] for i in range(4)])
    k3=f([state[i]+0.5*dt*k2[i] for i in range(4)])
    k4=f([state[i]+dt*k3[i] for i in range(4)])
    for i in range(4): state[i]+=(dt/6)*(k1[i]+2*k2[i]+2*k3[i]+k4[i])
    self.x, self.y, self.ori, self.v = state
```

Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```
class Circuit_classe:
    def __init__(self, fichier):
        self.fichier = fichier
        self.carres = []
        self.other_carres = []
        self.start_pos = (0, 0)
        self.end_pos = (0, 0)
        self.scale_x = 5
        self.scale_y = 5
        self.distances = []

    def generate(self):
        with open(self.fichier, "r") as fichier_texte:
            contenu = fichier_texte.read()

        j = 0
        k = 0
        for i in contenu:
            if i == "\n":
                j += 1
                k = 0
            elif i == "#":
                self.carres.append(pygame.Rect(k * self.scale_x, j *
self.scale_y, self.scale_x, self.scale_y))
            elif i == "*":
                self.start_pos = (k * self.scale_x, j * self.scale_y)
                self.other_carres.append(pygame.Rect(k * self.scale_x, j *
self.scale_y, self.scale_x, self.scale_y))
            elif i == "+":
                self.end_pos = (k * self.scale_x, j * self.scale_y)
                self.other_carres.append(pygame.Rect(k * self.scale_x, j *
self.scale_y, self.scale_x, self.scale_y))
            k += 1
```

```
    def toutes_les_voitures_sont_mortes(self):
        return all(not voiture.vie for voiture in
self.voitures)


    def draw(self):
        for i in self.carres:
            pygame.draw.rect(WINDOW, (128, 128, 128), i)
        for i in self.other_carres:
            pygame.draw.rect(WINDOW, (255, 255, 255), i)
```

Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```python
class Gen_dynamiques :
    # Gestion d'une population de VoitureDynamique avec sélection génétique,
    # en utilisant la distance Euclidienne (vol d'oiseau) jusqu'à l'arrivée.

    def __init__(self, circuit):
        self.circuit = circuit
        self.voitures = []
        self.generation = 1
        self.best_paths = []

    def all_dead(self):
        return all(not v.vie for v in self.voitures)

    def generate_first_voitures(self):
        self.voitures = []
        for i in range(NB_VOITURE):
            start_rect = pygame.Rect(
                self.circuit.start_pos[0],
                self.circuit.start_pos[1],
                SIZE, SIZE
            )
            v = VoitureDynamique(start_rect, ID=i)
            v.generer_commandes()
            self.voitures.append(v)
        return self.voitures

    def draw_voitures(self, WINDOW):
        for v in self.voitures:
            if v.vie:
                cx, cy = v.rect.center
                r = SIZE // 2
                pygame.draw.circle(WINDOW, (0,255,0), (cx,cy), r)
                dx = math.cos(v.ori) * (r+4)
                dy = math.sin(v.ori) * (r+4)
                pygame.draw.line(WINDOW, (255,0,0), (cx,cy), (cx+dx, cy+dy), 2)

    def bouge_voitures(self, count):
        """
        Déplace toutes les voitures, en passant
        carres, other_carres et end_pos.
        """
        for v in self.voitures:
            v.move(
                self.circuit.carres,
                self.circuit.other_carres,
                self.circuit.end_pos,
                count
            )
        # on ne garde que les vivantes
        self.voitures = [v for v in self.voitures if v.vie]
        # mise à jour des distances / trace heatmap
        self.calcul_distance()
        print(f"Nombre de voitures restantes : {len(self.voitures)}")

    def clean_dead_voitures(self):
        self.voitures = [v for v in self.voitures if v.vie]

    def draw_window(self, WINDOW):
        WINDOW.fill(BLACK)
        self.circuit.draw()
        if self.best_paths:
            pts = [(int(x),int(y)) for x,y in self.best_paths[-1]]
            if len(pts) > 1:
                pygame.draw.lines(WINDOW, (0,0,255), False, pts, 2)
        self.draw_voitures(WINDOW)
        pygame.display.update()

    def get_grade_population(self):
        grades = []
        for v in self.voitures:
            if not v.vie or v.on_track_time < 300:
                reward = -1000
            else:
                distance_score    = 10000 / (1 + v.distance_with_end)
                time_score        = v.on_track_time
                exploration_score = 2 * len(v.visited)
                penalty           = 0.8 * v.distance_parcourue
                reward = distance_score + time_score + exploration_score -
penalty
            grades.append((v.ID, reward, v.moves, v.rect))
        grades.sort(key=lambda elt: -elt[1])
        return grades

    def calcul_distance(self):
        end_x, end_y = self.circuit.end_pos
        for v in self.voitures:
            dx = v.rect.x - end_x
            dy = v.rect.y - end_y
            v.distance_with_end = math.hypot(dx, dy)
```

Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```python
def generate_children(self, parents):
    children = []
    if len(parents) < 2:
        print("Pas assez de parents, réinitialisation…")
        return self.generate_first_voitures()

    start_id = len(parents)
    while len(children) + len(parents) < NB_VOITURE:
        p1, p2 = random.sample(parents, 2)
        child_rect = pygame.Rect(
            self.circuit.start_pos[0],
            self.circuit.start_pos[1],
            SIZE, SIZE
        )
        child = VoitureDynamique(child_rect, ID=start_id)
        start_id += 1
        pivot = random.randrange(NUMBER_OF_MOVES)
        child.moves = p1.moves[:pivot] + p2.moves[pivot:]
        children.append(child)

    return children

def add_diversity(self, graded, parents):
    start_id = len(parents)
    for proto in graded[NUMBER_OF_GRADED_RETAIN:]:
        if random.random() < PARENTS_GARDES:
            _, _, moves, rect = proto
            v = VoitureDynamique(rect.copy(), ID=start_id)
            start_id += 1
            v.moves = moves.copy()
            parents.append(v)

def mutate(self, parents):
    for p in parents:
        if random.random() < CHANCE_DE_MUTATION:
            for _ in range(NOMBRE_DE_MUTATION):
                i = random.randint(0, NUMBER_OF_MOVES - 1)
                p.moves[i] = (
                    random.uniform(FREIN_MAX, ACCEL_MAX),
                    random.uniform(-DIRECTION_MAX, DIRECTION_MAX)
                )
```

```python
def generate_next_generation(self):
    # Garde en cas d'extinction complète
    if not self.voitures:
        print("Toutes les voitures sont mortes, réinitialisation de la population.")
        self.generate_first_voitures()
        return self.voitures

    self.calcul_distance()
    graded = self.get_grade_population()
    if not graded:
        print("Aucune note disponible, réinitialisation de la population.")
        self.generate_first_voitures()
        return self.voitures

    best_id, _, _, _ = graded[0]
    for v in self.voitures:
        if v.ID == best_id:
            self.best_paths.append(v.path.copy())
            break

    parents_proto = graded[:NUMBER_OF_GRADED_RETAIN]
    if len(parents_proto) < 2:
        parents_proto = graded[:2]

    parents = []
    for ID, _, moves, _ in parents_proto:
        start_rect = pygame.Rect(
            self.circuit.start_pos[0],
            self.circuit.start_pos[1],
            SIZE, SIZE
        )
        v = VoitureDynamique(start_rect, ID=ID)
        v.moves = moves.copy()
        v.on_track_time = 0
        v.distance_parcourue = 0.0
        v.visited.clear()
        v.vie = True
        v.distance_with_end = 1
        parents.append(v)

    self.add_diversity(graded, parents)
    self.mutate(parents)
    children = self.generate_children(parents)

    self.voitures = parents + children
    self.generation += 1
    return self.voitures
```

Partie III : Conclusion

b) Annexes : voitures_dynamiques.py

```
circuit = Circuit_classe("bugatti_final.txt")
circuit.generate()

gen = Gen_dynamiques(circuit)
gen.generate_first_voitures()
gen.calcul_distance()


# Boucle principale de la simulation
while GO:
   for event in pygame.event.get():
       if event.type == pygame.QUIT: # Si appuie sur quitter, on ferme la fenêtre
           GO = False

   gen.bouge_voitures(COUNT)
   COUNT += 1
   if COUNT >= NUMBER_OF_MOVES:
       COUNT = 0
       # Calcul du nombre de voitures mortes pour la génération écoulée
       dead_count = NB_VOITURE - len(gen.voitures)
       generation_history.append(gen.generation)
       dead_history.append(dead_count)

       if gen.generation % 3 == 0:  # Affiche toutes les 10 générations
           plt.figure(2)
           plt.clf()
           plt.title("Heatmap des zones de mortalité")
           plt.imshow(death_heatmap, cmap='hot', interpolation='nearest',
origin='upper')
           plt.colorbar(label="Nombre de morts")
           plt.xlabel("x (grille)")
           plt.ylabel("y (grille)")
           plt.pause(0.001)

       # Tracé du graphe toutes les 5 générations
       if gen.generation % 2 == 0:
           plt.figure(1)
           plt.clf()
           plt.plot(generation_history, dead_history,
marker="o", linestyle="-", color="b", label="Nombre de
particules mortes")
           plt.xlabel("Génération")
           plt.ylabel("Nombre de particules mortes")
           plt.title("Évolution des particules mortes")
           plt.legend()
           plt.grid(True)
           plt.pause(0.001)
       gen.voitures = gen.generate_next_generation()
       gen.generation += 1
   gen.draw_window(WINDOW)

quitAlgo()

plt.figure()
plt.title("Heatmap finale des zones de mortalité")
plt.imshow(death_heatmap, cmap='hot',
interpolation='nearest', origin='upper')
plt.colorbar(label="Nombre de morts")
plt.xlabel("x (grille)")
plt.ylabel("y (grille)")
plt.savefig("heatmap_finale.png")
```

Partie III : Conclusion

b) Annexes : voitures_statiques.py

```
import pygame
import random
import matplotlib.pyplot as plt
import math
import numpy as np

# --- Constantes et paramètres ---
RESOLUTION = (500, 600)
FPS = 60
NUMBER_OF_MOVES = 10000
NB_VOITURE = 1000
GRADED_RETAIN_PERCENT = 0.05
NUMBER_OF_GRADED_RETAIN = int(NB_VOITURE * GRADED_RETAIN_PERCENT)
PARENTS_GARDES = 0.1
CHANCE_DE_MUTATION = 0.02
POURCENTAGE_DE_MUTATION = 0.02
NOMBRE_DE_MUTATION = int(POURCENTAGE_DE_MUTATION * NUMBER_OF_MOVES)
SIZE = 10

HEATMAP_SCALE_X = 1
HEATMAP_SCALE_Y = 1
heatmap_width  = RESOLUTION[0] // HEATMAP_SCALE_X
heatmap_height = RESOLUTION[1] // HEATMAP_SCALE_Y
death_heatmap  = np.zeros((heatmap_height, heatmap_width))

# --- Classe VoitureStatique ---
class VoitureStatique:
    def __init__(self, start_pos, ID=None):
        self.rect = pygame.Rect(start_pos[0], start_pos[1], SIZE, SIZE)
        self.ID = ID
        self.moves = []
        self.on_track_time = 0
        self.distance_parcourue = 0.0
        self.distance_with_end = 1.0
        self.visited = set()
        self.vie = True
        self.path = [(float(self.rect.x), float(self.rect.y))]
        self.color = (0, 0, 255)

    def gen_random_mouvements(self):
        self.moves = [
            random.choice([(-SIZE,0),(SIZE,0),(0,-SIZE),(0,SIZE),(0,0)])
            for _ in range(NUMBER_OF_MOVES)
        ]
```

```
    def move(self, carres, other_carres, end_pos, step):
        if not self.vie:
            return
        dx, dy = self.moves[step]
        self.rect.x += dx
        self.rect.y += dy
        self.distance_parcourue += math.hypot(dx, dy)
        self.path.append((float(self.rect.x), float(self.rect.y)))

        # collision / hors limites
        if (self.rect.x < 0 or self.rect.right > RESOLUTION[0] or
            self.rect.y < 0 or self.rect.bottom > RESOLUTION[1] or
            not any(self.rect.colliderect(c) for c in carres +
other_carres)):
            gy = min(max(self.rect.y // HEATMAP_SCALE_Y, 0),
heatmap_height-1)
            gx = min(max(self.rect.x // HEATMAP_SCALE_X, 0),
heatmap_width-1)
            death_heatmap[gy, gx] += 1
            self.vie = False
            self.color = (255, 0, 0)
            return

        self.on_track_time += 1
        self.visited.add((self.rect.x//10, self.rect.y//10))
        ex, ey = end_pos
        self.distance_with_end = math.hypot(self.rect.x - ex,
self.rect.y - ey)
```

Partie III : Conclusion

b) Annexes : voitures_statiques.py

```
# --- Classe Circuit ---
class Circuit_classe:
   def __init__(self, fichier):
       self.fichier = fichier
       self.carres = []
       self.other_carres = []
       self.start_pos = (0, 0)
       self.end_pos = (0, 0)
       self.scale_x = 5
       self.scale_y = 5

   def generate(self):
       with open(self.fichier, "r") as f:
           contenu = f.read()
       j = k = 0
       for char in contenu:
           if char == "\n":
               j += 1; k = 0
           elif char == "#":
               self.carres.append(
                   pygame.Rect(k*self.scale_x, j*self.scale_y,
self.scale_x, self.scale_y)
               )
           elif char == "*":
               self.start_pos = (k*self.scale_x, j*self.scale_y)
               self.other_carres.append(
                   pygame.Rect(k*self.scale_x, j*self.scale_y,
self.scale_x, self.scale_y)
               )
           elif char == "+":
               self.end_pos = (k*self.scale_x, j*self.scale_y)
               self.other_carres.append(
                   pygame.Rect(k*self.scale_x, j*self.scale_y,
self.scale_x, self.scale_y)
               )
           k += 1
```

```
   def draw(self, surface):
       for rect in self.carres:
           pygame.draw.rect(surface,
(128,128,128), rect)
       for rect in self.other_carres:
           pygame.draw.rect(surface,
(255,255,255), rect)
```

Partie III : Conclusion

b) Annexes : voitures_statiques.py

```python
# --- Classe génétique ---
class GenStatique:
    def __init__(self, circuit):
        self.circuit = circuit
        self.voitures = []
        self.generation = 0
        self.best_path = []

    def initialize(self):
        self.voitures = []
        for i in range(NB_VOITURE):
            v = VoitureStatique(self.circuit.start_pos, ID=i)
            v.gen_random_mouvements()
            self.voitures.append(v)
        self.generation = 1

    def step(self, step_idx):
        for v in self.voitures:
            v.move(self.circuit.carres,
self.circuit.other_carres,
                   self.circuit.end_pos, step_idx)
        self.voitures = [v for v in self.voitures if v.vie]
```

```python
def evaluate_and_breed(self):
    graded = [( (10000/(1+v.distance_with_end)
                 + v.on_track_time
                 + 2*len(v.visited)
                 - v.distance_parcourue), v )
              for v in self.voitures]
    graded.sort(key=lambda x: -x[0])

    if graded:
        _, best = graded[0]
        self.best_path.append(best.path.copy())

    parents = [v for _,v in graded[:NUMBER_OF_GRADED_RETAIN]]
    # diversification
    for _,v in graded[NUMBER_OF_GRADED_RETAIN:]:
        if random.random() < PARENTS_GARDES:
            clone = VoitureStatique(self.circuit.start_pos)
            clone.moves = v.moves.copy()
            parents.append(clone)

    # mutation
    for v in parents:
        if random.random() < CHANCE_DE_MUTATION:
            for _ in range(NOMBRE_DE_MUTATION):
                i = random.randrange(NUMBER_OF_MOVES)
                v.moves[i] = random.choice([(-SIZE,0),(SIZE,0),(0,-SIZE),(0,SIZE),(0,0)])

    # crossover avec garde si population petite
    children = []
    needed = NB_VOITURE - len(parents)
    while len(children) < needed:
        if len(parents) >= 2:
            p1, p2 = random.sample(parents, 2)
        elif parents:
            p1 = parents[0]; p2 = random.choice(parents)
        else:
            # réinitialisation
            self.initialize()
            return
        child = VoitureStatique(self.circuit.start_pos)
        child.moves = [
            random.choice([p1.moves[i], p2.moves[i]])
            for i in range(NUMBER_OF_MOVES)
        ]
        children.append(child)

    self.voitures = parents + children
    self.generation += 1
```

Partie III : Conclusion

b) Annexes : voitures_statiques.py

```
# --- Boucle principale ---
pygame.init()
window = pygame.display.set_mode(RESOLUTION)
clock = pygame.time.Clock()
GO = True
COUNT = 0

circuit = Circuit_classe("bugatti_final.txt")
circuit.generate()

gen = GenStatique(circuit)
gen.initialize()
gen.calcul_distance = lambda: None  # stub pour compatibilité

generation_history = []
dead_history = []

plt.ion()
# Figures exist déjà en fig1, fig2

while GO:
   for event in pygame.event.get():
       if event.type == pygame.QUIT:
           GO = False

   # 1) déplacement
   if COUNT < NUMBER_OF_MOVES:
       gen.step(COUNT)
       COUNT += 1
   else:
       # statistiques génération
       dead_count = NB_VOITURE - len(gen.voitures)
       generation_history.append(gen.generation)
       dead_history.append(dead_count)

       # heatmap toutes les 3 générations
       if gen.generation % 3 == 0:
           plt.figure(2)
           plt.clf()
           plt.title("Heatmap des zones de mortalité")
           plt.imshow(death_heatmap, cmap='hot', origin='upper')
           plt.colorbar(label="Morts")
           plt.pause(0.001)

       # graphe toutes les 2 générations
       if gen.generation % 2 == 0:
           plt.figure(1)
           plt.clf()
           plt.plot(generation_history, dead_history, marker='o',
linestyle='-')
           plt.title("Morts par génération")
           plt.xlabel("Génération")
           plt.ylabel("Nombre de morts")
           plt.grid(True)
           plt.pause(0.001)

       # nouvelle génération
       gen.evaluate_and_breed()
       gen.generation += 1
       COUNT = 0

   # affichage Pygame
   window.fill((30,30,30))
   circuit.draw(window)
   for v in gen.voitures:
       pygame.draw.rect(window, v.color, v.rect)
   pygame.display.flip()
   clock.tick(FPS)

quitAlgo()

plt.figure()
plt.title("Heatmap finale des zones de mortalité")
plt.imshow(death_heatmap, cmap='hot', origin='upper')
plt.colorbar(label="Morts")
plt.xlabel("x (grille)")
plt.ylabel("y (grille)")
plt.savefig("heatmap_finale.png")

pygame.quit()
```