```python
"""
                                        2019
                                La Brachistochrone Réelle
                            Un TIPE réalisé par Gautier BEN AÏM
                                    http://tobog.ga
"""

import numpy as np

#
#  I. Calculs physiques
# ======================
#

def generer_ligne(longueur, hauteur, nb_points):
    """
    Renvoie le toboggan ligne droite.

    Un toboggan est représenté par un triplet
    (longueur, hauteur, liste des hauteurs des points intermédiaires)
    longueur  : flottant, distance horizontale entre le départ et l'arrivée
    hauteur   : flottant, distance verticale
    nb_points : entier, nombre total de points
    """
    return (
        longueur,
        hauteur,
        [hauteur * (1. - i / (nb_points - 1)) for i in range(1, nb_points - 1)],
    )


def calculer_temps_segment(distance, v, deriver_v, limite, pas):
    """
    Renvoie le temps et la vitesse après le parcours d'un segment.

    distance  : flottant, distance à parcourir
    v         : flottant, vitesse intiale
    deriver_v : fonction, renvoie la dérivée de la vitesse
    limite    : flottant, limite de temps de parcours
    pas       : flottant, intervalle de temps dt
    """
    t = 0.
    x = 0.
    # On utilise la méthode d'Euler
    while x < distance and t < limite and v >= 0.:
        x += pas * v
        v += pas * deriver_v(v)
        t += pas
    if x >= distance:
        return t, v
    return None, None


def calculer_temps_toboggan(toboggan, appliquer_pfd, limite, pas):
    """
    Renvoie le temps de parcours du toboggan donné.

    toboggan      : triplet
    appliquer_pfd : fonction, renvoie deriver_v
    limite        : flottant, limite de temps de parcours
```

```
    pas           : flottant, intervalle de temps dt
    """
    points = toboggan[2][:]
    points.append(0.)  # On rajoute l'arrivée

    l = len(points)
    section = toboggan[0] / l  # Distance horizontale entre deux points
    section2 = section * section

    temps_total = 0.
    vitesse = 0.

    depart = toboggan[1]
    for i in range(l):
        arrivee = points[i]
        distance = ((depart - arrivee) * (depart - arrivee) + section2) ** 0.5

        # On applique le PFD sur le segment
        deriver_v = appliquer_pfd(section, depart - arrivee)
        temps, vitesse = calculer_temps_segment(
            distance, vitesse, deriver_v, limite, pas
        )

        if temps is None:
            return None

        temps_total += temps
        limite -= temps
        depart = arrivee

    return temps_total


#
#  II. Algorithme hybride
# ========================
#

def generer_evaluateur(appliquer_pfd):
    """
    Renvoie une fonction qui calcule le score (le temps de parcours)
    d'un toboggan.

    appliquer_pfd : fonction, renvoie deriver_v
    """
    return lambda toboggan, limite, pas: (
        calculer_temps_toboggan(toboggan, appliquer_pfd, limite, pas)
    )


def muter_creuser(toboggan, n):
    """ Creuse un intervalle choisi au hasard d'une profondeur au hasard. """
    _, hauteur, points = toboggan
    i = np.random.randint(len(points))
    j = np.random.randint(len(points))
    if i > j:
        i, j = j, i
    h = hauteur / (1. + 0.05 * n)
    v = np.random.uniform(-h, h)
    for k in range(i, j + 1):
```

```
        points[k] += v


def muter_lisser(toboggan, n):
    """ Prend un point au hasard et en fait la moyenne de ses voisins. """
    _, _, points = toboggan
    i = np.random.randint(len(points) - 2)
    points[i + 1] = (points[i] + points[i + 2]) / 2.


def diviser(toboggan, nb_points):
    """ Coupe chaque segment pour augmenter le nombre de points. """
    longueur, hauteur, anciens_points = toboggan
    anciens_points = [hauteur] + anciens_points + [0.]
    ancien_nb_points = len(anciens_points)
    points = []

    for i in range(1, nb_points - 1):
        x = i * (ancien_nb_points - 1) / (nb_points - 1)
        j = int(x)
        t = x % 1
        points.append((1 - t) * anciens_points[j] + t * anciens_points[j + 1])

    return longueur, hauteur, points


def generer_incrementeur(
    evaluateur, nb_points=16, facteur_nb_points=2, pas=0.001, facteur_pas=0.1
):
    """
    Renvoie une fonction qui permet de passer à la génération suivante.

    evaluateur        : fonction, renvoyée par generer_evaluateur
    nb_points         : entier, nombre de points initial
    facteur_nb_points : flottant, coefficient multiplicateur
    pas               : flottant, pas initial
    facteur_pas       : flottant, coefficient multiplicateur
    """

    def premiere_generation(meilleur_candidat):
        """ Lorsque incrementer_generation est appelée pour la première fois. """

        def calculer_score(toboggan, limite):
            return evaluateur(toboggan, limite, pas)

        meilleur_score = calculer_score(meilleur_candidat, 10.)
        if meilleur_score is None:
            raise Exception("Le candidat proposé ne fonctionne pas")
        return meilleur_candidat, meilleur_score, calculer_score

    def incrementer_generation(generation, meilleur_candidat, meilleur_score=None):
        """ Passe à la génération suivante. """
        if generation == 0:
            return premiere_generation(meilleur_candidat)

        nouveau_pas = pas * facteur_pas ** generation

        def calculer_score(toboggan, limite):
            return evaluateur(toboggan, limite, nouveau_pas)

```

```
        meilleur_candidat = diviser(
            meilleur_candidat, (nb_points - 1) * facteur_nb_points ** generation + 1
        )

        score = calculer_score(meilleur_candidat, 2 * meilleur_score)
        if not score is None:
            meilleur_score = score

        return meilleur_candidat, meilleur_score, calculer_score

    return incrementer_generation


def evoluer(
    toboggan,
    incrementer_generation,
    nb_generations,
    signaler_fin,
    rafraichir=None,
    periode_lisser=8,
    generation_suivante=150,
):
    """
    Améliore itérativement le toboggan donné en argument.

    toboggan               : triplet
    incrementer_generation : fonction, appelée au changement de génération
    nb_generations         : entier, maximum de modifications des paramètres
    signaler_fin           : fonction, commande l'arrêt de la fonction
    rafraichir             : fonction, appelée à chaque amélioration
    periode_lisser         : entier, période entre deux lissages
    generation_suivante    : entier, individus à tester avant de passer
    """

    generation = 0
    meilleur_candidat, meilleur_score, calculer_score = incrementer_generation(
        generation, toboggan
    )

    # Nombre de candidats générés, dernier progrès enregistré
    n = 0
    dernier_progres = 0
    nb_progres = 0
    print("Initialisation, score : {:f}".format(meilleur_score))

    while not signaler_fin():
        n += 1

        # Si l'algorithme ne progresse plus, on augmente la finesse
        if (
            n - dernier_progres >= generation_suivante
            and generation < nb_generations - 1
        ):
            generation += 1
            dernier_progres = n
            meilleur_candidat, meilleur_score, calculer_score = incrementer_generation(
                generation, meilleur_candidat, meilleur_score
            )
            print(
                "Génération {} ({}), score : {:f}".format(generation, n, meilleur_score)
```

```
            )

        # On prend un nouveau candidat
        candidat = (meilleur_candidat[0], meilleur_candidat[1], meilleur_candidat[2][:])

        # On le mute
        if n % periode_lisser == 0:
            muter_lisser(candidat, n)
        else:
            muter_creuser(candidat, n)

        # Et enfin on le teste
        score = calculer_score(candidat, meilleur_score)
        if not score is None and score < meilleur_score:

            nb_progres += 1
            dernier_progres = n
            meilleur_candidat = candidat
            meilleur_score = score

            if not rafraichir is None:
                rafraichir(meilleur_candidat, meilleur_score)

    print(("{} individus testés, {} conservés").format(n, nb_progres))
    return meilleur_candidat


#
#  III. Génération d'une cycloïde
# ================================
#

def generer_cycloide(longueur, hauteur, nb_points):
    """ Renvoie le toboggan cycloïde. """

    def trouver_zero(f, a, b, precision=1e-9):
        """ Recherche dichotomique du zéro de f entre a et b. """
        fa = f(a)
        while b - a > precision:
            m = (a + b) / 2.
            fm = f(m)
            if fm == 0.:
                return m
            elif fm * fa > 0.:
                a = m
                fa = f(a)
            else:
                b = m
        return m

    # Valeur de thêta du point d'arrivée
    theta = trouver_zero(
        lambda t: hauteur / longueur - (1. - np.cos(t)) / (t - np.sin(t)),
        0.001,
        2 * np.pi,
    )
    # Rayon de la cycloïde reliant le départ et l'arrivée
    r = hauteur / (1. - np.cos(theta))

    # Points de la courbe paramétrée
```

```
    courbe = []
    for i in range(2 * nb_points + 1):
        t = theta * i / (2 * nb_points)
        x = r * (t - np.sin(t))
        y = r * (np.cos(t) - 1.) + hauteur
        courbe.append((x, y))

    # Points intermédiaires du toboggan
    points = []
    j = 0
    for i in range(1, nb_points - 1):
        x = longueur * i / (nb_points - 1)
        while courbe[j][0] < x:
            j += 1
        a = (courbe[j][1] - courbe[j - 1][1]) / (courbe[j][0] - courbe[j - 1][0])
        b = courbe[j][1] - a * courbe[j][0]
        points.append(a * x + b)

    return longueur, hauteur, points


#
#  IV. Génération de la meilleure courbe
# ======================================
#

if __name__ == "__main__":

    import sys
    import matplotlib.pyplot as plt
    from time import time

    debut = time()

    # Paramètres de l'expérience
    longueur = 1.2
    hauteur = 0.5
    nb_generations = 4
    nb_points = 121  # Départ + intermédiaires + arrivée
    nb_points_initial = 16
    pas = 0.000001  # Intervalle de temps dt
    pas_initial = 0.0004
    facteur_pas = 0.2
    temps_de_calcul = int(sys.argv[1]) if len(sys.argv) >= 2 else 60

    def appliquer_pfd(x, y):
        """ PFD au point parcourant le toboggan. """
        g_sin_theta = 9.81 * y / (y * y + x * x) ** 0.5
        fg_cos_theta = 0.3263 * 9.81 * x / (y * y + x * x) ** 0.5
        a = g_sin_theta - fg_cos_theta
        # Renvoie la dérivée de la vitesse v exprimée en fonction d'elle-même
        return lambda v: a - 0.0026 * v - 0.4748 * v * v

    # Calcul pour la cycloïde
    cycloide = generer_cycloide(longueur, hauteur, nb_points)
    calculer_score = generer_evaluateur(appliquer_pfd)
    temps_cycloide = calculer_score(cycloide, 10., pas)

    # Point de départ de l'algorithme
    ligne = generer_ligne(longueur, hauteur, nb_points_initial)
```

```
361
362     # Affichage
363     plt.figure("Toboggan", figsize=(8, 6), dpi=72)
364     plt.plot(
365         np.linspace(0., longueur, nb_points),
366         [hauteur] + cycloide[2] + [0.],
367         "#363737",
368         dashes=[3, 2],
369         label="cycloïde"
370         if temps_cycloide is None
371         else "cycloïde ({:f} s)".format(temps_cycloide),
372     )
373     graphe, = plt.plot(
374         np.linspace(0., longueur, nb_points_initial),
375         [hauteur] + ligne[2] + [0.],
376         "#ef4026",
377         linewidth=2,
378         label="toboggan",
379     )
380     plt.title("La brachistochrone réelle")
381     plt.xlabel("Longueur (m)")
382     plt.ylabel("Hauteur (m)")
383     plt.axis("equal")
384     plt.legend()
385     plt.draw()
386     plt.pause(0.001)
387
388     def generer_chronometre():
389         """ Renvoie toutes les fonctions dépendantes du temps. """
390
391         debut = time()
392
393         def temps_ecoule():
394             """ Temps écoulé. """
395             return time() - debut
396
397         def signaler_fin():
398             """ Signal de fin. """
399             return temps_ecoule() > temps_de_calcul
400
401         def rafraichir(toboggan, temps):
402             """ Met à jour le graphe à chaque amélioration. """
403             t = temps_ecoule()
404             nb_points = len(toboggan[2]) + 2
405             if len(graphe.get_xdata()) != nb_points:
406                 graphe.set_xdata(np.linspace(0., longueur, nb_points))
407             graphe.set_ydata([hauteur] + toboggan[2] + [0.])
408             graphe.set_label("toboggan ({:f} s)".format(temps))
409             plt.title(
410                 "La brachistochrone réelle après {:d}min{:0>2d}s de calcul".format(
411                     int(t / 60), int(t % 60)
412                 )
413             )
414             if temps_cycloide is None or temps <= temps_cycloide:
415                 graphe.set_color("#0165fc")
416             plt.legend()
417             plt.draw()
418             plt.pause(0.001)
419
420         return signaler_fin, rafraichir
```

```
421
422     signaler_fin, rafraichir = generer_chronometre()
423
424     # Appel de l'algorithme hybride
425     toboggan = evoluer(
426         ligne,
427         generer_incrementeur(
428             calculer_score,
429             nb_points=nb_points_initial,
430             pas=pas_initial,
431             facteur_pas=facteur_pas,
432         ),
433         nb_generations,
434         signaler_fin,
435         rafraichir,
436     )
437     temps = calculer_score(toboggan, 10., pas)
438
439     rafraichir(toboggan, temps)
440     print("Temps sur le toboggan optimisé : {:f} secondes".format(temps))
441
442     if not temps_cycloide is None:
443         print(
444             (
445                 "Temps sur la cycloïde ........ : {:f} secondes\n" +
446                 "Différence de temps .......... : {:f} secondes"
447             ).format(temps_cycloide, abs(temps_cycloide - temps))
448         )
449     else:
450         print("La cycloïde ne permet pas de rejoindre les deux points")
451
452     # Temps d'exécution
453     print("Calculé en {:f} secondes".format(time() - debut))
454
455     if len(sys.argv) >= 3 and sys.argv[2] == "svg":
456         plt.savefig("toboggan.svg")
457     plt.show()
```