

```

1  """
2
3      2019
4      La Brachistochrone Réelle
5      Un TIPE réalisé par Gautier BEN AÏM
6      http://tobog.ga
7
8  import numpy as np
9
10 #
11 # I. Calculs physiques
12 # =====
13 #
14
15 def generer_ligne(longueur, hauteur, nb_points):
16     """
17     Renvoie le toboggan ligne droite.
18
19     Un toboggan est représenté par un triplet
20     (longueur, hauteur, liste des hauteurs des points intermédiaires)
21     longueur : flottant, distance horizontale entre le départ et l'arrivée
22     hauteur   : flottant, distance verticale
23     nb_points : entier, nombre total de points
24     """
25     return (
26         longueur,
27         hauteur,
28         [hauteur * (1. - i / (nb_points - 1)) for i in range(1, nb_points - 1)],
29     )
30
31
32 def calculer_temps_segment(distance, v, deriver_v, limite, pas):
33     """
34     Renvoie le temps et la vitesse après le parcours d'un segment.
35
36     distance : flottant, distance à parcourir
37     v         : flottant, vitesse initiale
38     deriver_v : fonction, renvoie la dérivée de la vitesse
39     limite    : flottant, limite de temps de parcours
40     pas       : flottant, intervalle de temps dt
41     """
42     t = 0.
43     x = 0.
44     # On utilise la méthode d'Euler
45     while x < distance and t < limite and v >= 0.:
46         x += pas * v
47         v += pas * deriver_v(v)
48         t += pas
49     if x >= distance:
50         return t, v
51     return None, None
52
53
54 def calculer_temps_toboggan(toboggan, appliquer_pfd, limite, pas):
55     """
56     Renvoie le temps de parcours du toboggan donné.
57
58     toboggan      : triplet
59     appliquer_pfd : fonction, renvoie deriver_v
60     limite        : flottant, limite de temps de parcours

```

```

61     pas : flottant, intervalle de temps dt
62     """
63     points = toboggan[2][:]
64     points.append(0.) # On rajoute l'arrivée
65
66     l = len(points)
67     section = toboggan[0] / l # Distance horizontale entre deux points
68     section2 = section * section
69
70     temps_total = 0.
71     vitesse = 0.
72
73     depart = toboggan[1]
74     for i in range(l):
75         arrivee = points[i]
76         distance = ((depart - arrivee) * (depart - arrivee) + section2) ** 0.5
77
78         # On applique le PFD sur le segment
79         deriver_v = appliquer_pfd(section, depart - arrivee)
80         temps, vitesse = calculer_temps_segment(
81             distance, vitesse, deriver_v, limite, pas
82         )
83
84         if temps is None:
85             return None
86
87         temps_total += temps
88         limite -= temps
89         depart = arrivee
90
91     return temps_total
92
93
94 #
95 # II. Algorithme hybride
96 # =====
97 #
98
99 def generer_evaluateur(appliquer_pfd):
100     """
101     Renvoie une fonction qui calcule le score (le temps de parcours)
102     d'un toboggan.
103
104     appliquer_pfd : fonction, renvoie deriver_v
105     """
106     return lambda toboggan, limite, pas: (
107         calculer_temps_toboggan(toboggan, appliquer_pfd, limite, pas)
108     )
109
110
111 def muter_creuser(toboggan, n):
112     """ Creuse un intervalle choisi au hasard d'une profondeur au hasard. """
113     _, hauteur, points = toboggan
114     i = np.random.randint(len(points))
115     j = np.random.randint(len(points))
116     if i > j:
117         i, j = j, i
118     h = hauteur / (1. + 0.05 * n)
119     v = np.random.uniform(-h, h)
120     for k in range(i, j + 1):

```

```

121         points[k] += v
122
123
124 def muter_lisser(toboggan, n):
125     """ Prend un point au hasard et en fait la moyenne de ses voisins. """
126     _, _, points = toboggan
127     i = np.random.randint(len(points) - 2)
128     points[i + 1] = (points[i] + points[i + 2]) / 2.
129
130
131 def diviser(toboggan, nb_points):
132     """ Coupe chaque segment pour augmenter le nombre de points. """
133     longueur, hauteur, anciens_points = toboggan
134     anciens_points = [hauteur] + anciens_points + [0.]
135     ancien_nb_points = len(anciens_points)
136     points = []
137
138     for i in range(1, nb_points - 1):
139         x = i * (ancien_nb_points - 1) / (nb_points - 1)
140         j = int(x)
141         t = x % 1
142         points.append((1 - t) * anciens_points[j] + t * anciens_points[j + 1])
143
144     return longueur, hauteur, points
145
146
147 def generer_incrementeur(evaluateur, nb_points, facteur_nb_points, pas, facteur_pas):
148     """
149     Renvoie une fonction qui permet de passer à la génération suivante.
150
151     evaluateur      : fonction, renvoyée par generer_evaluateur
152     nb_points       : entier, nombre de points initial
153     facteur_nb_points : flottant, coefficient multiplicateur
154     pas             : flottant, pas initial
155     facteur_pas      : flottant, coefficient multiplicateur
156     """
157
158     def premiere_generation(meilleur_candidat):
159         """ Lorsque incrementer_generation est appelée pour la première fois. """
160
161         def calculer_score(toboggan, limite):
162             return evaluateur(toboggan, limite, pas)
163
164         meilleur_score = calculer_score(meilleur_candidat, 10.)
165         if meilleur_score is None:
166             raise Exception("Le candidat proposé ne fonctionne pas")
167         return meilleur_candidat, meilleur_score, calculer_score
168
169     def incrementer_generation(generation, meilleur_candidat, meilleur_score):
170         """ Passe à la génération suivante. """
171         if generation == 0:
172             return premiere_generation(meilleur_candidat)
173
174         nouveau_pas = pas * facteur_pas ** generation
175
176         def calculer_score(toboggan, limite):
177             return evaluateur(toboggan, limite, nouveau_pas)
178
179         meilleur_candidat = diviser(
180             meilleur_candidat, (nb_points - 1) * facteur_nb_points ** generation + 1

```

```

181         )
182
183         score = calculer_score(meilleur_candidat, 2 * meilleur_score)
184         if not score is None:
185             meilleur_score = score
186
187         return meilleur_candidat, meilleur_score, calculer_score
188
189     return incrementer_generation
190
191
192 def evoluer(
193     toboggan,
194     nb_generations,
195     generation_suivante,
196     incrementer_generation,
197     periode_lisser,
198     signaler_fin,
199     rafraichir=None,
200 ):
201     """
202     Améliore itérativement le toboggan donné en argument.
203
204     toboggan            : triplet
205     nb_generations      : entier, maximum de modifications des paramètres
206     generation_suivante : entier, individus à tester avant de passer
207     incrementer_generation : fonction, appelée au changement de génération
208     periode_lisser     : entier, période entre deux lissages
209     signaler_fin        : fonction, commande l'arrêt de la fonction
210     rafraichir          : fonction, appelée à chaque amélioration
211     """
212
213     generation = 0
214     meilleur_candidat, meilleur_score, calculer_score = incrementer_generation(
215         generation, toboggan, None
216     )
217
218     # Nombre de candidats générés, dernier progrès enregistré
219     n = 0
220     dernier_progres = 0
221     nb_progres = 0
222     print("Initialisation, score : {:.f}".format(meilleur_score))
223
224     while not signaler_fin():
225         n += 1
226
227         # Si l'algorithme ne progresse plus, on augmente la finesse
228         if (
229             n - dernier_progres >= generation_suivante
230             and generation < nb_generations - 1
231         ):
232             generation += 1
233             dernier_progres = n
234             meilleur_candidat, meilleur_score, calculer_score = incrementer_generation(
235                 generation, meilleur_candidat, meilleur_score
236             )
237             print(
238                 "Génération {} ({}), score : {:.f}".format(generation, n, meilleur_score)
239             )
240

```

```

241     # On prend un nouveau candidat
242     candidat = (meilleur_candidat[0], meilleur_candidat[1], meilleur_candidat[2][:])
243
244     # On le mute
245     if n % periode_lisser == 0:
246         muter_lisser(candidat, n)
247     else:
248         muter_creuser(candidat, n)
249
250     # Et enfin on le teste
251     score = calculer_score(candidat, meilleur_score)
252     if not score is None and score < meilleur_score:
253
254         nb_progres += 1
255         dernier_progres = n
256         meilleur_candidat = candidat
257         meilleur_score = score
258
259         if not rafraichir is None:
260             rafraichir(meilleur_candidat, meilleur_score)
261
262     print("{} individus testés, {} conservés").format(n, nb_progres))
263     return meilleur_candidat
264
265
266 #
267 # III. Génération d'une cycloïde
268 # =====
269 #
270
271 def generer_cycloide(longueur, hauteur, nb_points):
272     """ Renvoie le toboggan cycloïde. """
273
274     def trouver_zero(f, a, b, precision=1e-9):
275         """ Recherche dichotomique du zéro de f entre a et b. """
276         fa = f(a)
277         while b - a > precision:
278             m = (a + b) / 2.
279             fm = f(m)
280             if fm == 0.:
281                 return m
282             elif fm * fa > 0.:
283                 a = m
284                 fa = f(a)
285             else:
286                 b = m
287         return m
288
289     # Valeur de theta du point d'arrivée
290     theta = trouver_zero(
291         lambda t: hauteur / longueur - (1. - np.cos(t)) / (t - np.sin(t)),
292         0.001,
293         2 * np.pi,
294     )
295     # Rayon de la cycloïde reliant le départ et l'arrivée
296     r = hauteur / (1. - np.cos(theta))
297
298     # Points de la courbe paramétrée
299     courbe = []
300     for i in range(2 * nb_points + 1):

```

```

301         t = theta * i / (2 * nb_points)
302         x = r * (t - np.sin(t))
303         y = r * (np.cos(t) - 1.) + hauteur
304         courbe.append((x, y))
305
306     # Points intermédiaires du toboggan
307     points = []
308     j = 0
309     for i in range(1, nb_points - 1):
310         x = longueur * i / (nb_points - 1)
311         while courbe[j][0] < x:
312             j += 1
313         a = (courbe[j][1] - courbe[j - 1][1]) / (courbe[j][0] - courbe[j - 1][0])
314         b = courbe[j][1] - a * courbe[j][0]
315         points.append(a * x + b)
316
317     return longueur, hauteur, points
318
319
320 #
321 # IV. Génération de la meilleure courbe
322 # =====
323 #
324
325 if __name__ == "__main__":
326
327     import sys
328     import matplotlib.pyplot as plt
329     from time import time
330
331     debut = time()
332
333     # Paramètres de l'expérience
334     longueur = 1.2
335     hauteur = 0.5
336
337     # Paramètres de l'algorithme
338     nb_points = 121 # Départ + intermédiaires + arrivée
339     pas = 0.000001 # Intervalle de temps dt
340
341     nb_generations = 4
342     generation_suivante = 150
343     periode_lisser = 8
344
345     nb_points_initial = 16
346     facteur_nb_points = 2
347     pas_initial = 0.0004
348     facteur_pas = 0.2
349
350     temps_de_calcul = int(sys.argv[1]) if len(sys.argv) >= 2 else 60
351
352     def appliquer_pfd(x, y):
353         """ PFD au point parcourant le toboggan. """
354         g_sin_theta = 9.81 * y / (y * y + x * x) ** 0.5
355         fg_cos_theta = 0.3263 * 9.81 * x / (y * y + x * x) ** 0.5
356         a = g_sin_theta - fg_cos_theta
357         # Renvoie la dérivée de la vitesse v exprimée en fonction d'elle-même
358         return lambda v: a - 0.0026 * v - 0.4748 * v * v
359
360     # Calcul pour la cycloïde

```

```

361     cycloide = generer_cycloide(longueur, hauteur, nb_points)
362     calculer_score = generer_evaluateur(appliquer_pfd)
363     temps_cycloide = calculer_score(cycloide, 10., pas)
364
365     # Point de départ de l'algorithme
366     ligne = generer_ligne(longueur, hauteur, nb_points_initial)
367
368     # Affichage
369     plt.figure("Toboggan", figsize=(8, 6), dpi=72)
370     plt.plot(
371         np.linspace(0., longueur, nb_points),
372         [hauteur] + cycloide[2] + [0.],
373         "#363737",
374         dashes=[3, 2],
375         label="cycloïde"
376         if temps_cycloide is None
377         else "cycloïde ({:f} s)".format(temps_cycloide),
378     )
379     graphe, = plt.plot(
380         np.linspace(0., longueur, nb_points_initial),
381         [hauteur] + ligne[2] + [0.],
382         "#ef4026",
383         linewidth=2,
384         label="toboggan",
385     )
386     plt.title("La brachistochrone réelle")
387     plt.xlabel("Longueur (m)")
388     plt.ylabel("Hauteur (m)")
389     plt.axis("equal")
390     plt.legend()
391     plt.draw()
392     plt.pause(0.001)
393
394     def generer_chronometre():
395         """ Renvoie toutes les fonctions dépendantes du temps. """
396
397         debut = time()
398
399         def temps_ecoule():
400             """ Temps écoulé. """
401             return time() - debut
402
403         def signaler_fin():
404             """ Signal de fin. """
405             return temps_ecoule() > temps_de_calcul
406
407         def rafraichir(toboggan, temps):
408             """ Met à jour le graphe à chaque amélioration. """
409             t = temps_ecoule()
410             nb_points = len(toboggan[2]) + 2
411             if len(graphe.get_xdata()) != nb_points:
412                 graphe.set_xdata(np.linspace(0., longueur, nb_points))
413             graphe.set_ydata([hauteur] + toboggan[2] + [0.])
414             graphe.set_label("toboggan ({:f} s)".format(temps))
415             plt.title(
416                 "La brachistochrone réelle après {:d} min {:0>2d} s de calcul".format(
417                     int(t / 60), int(t % 60)
418                 )
419             )
420             if temps_cycloide is None or temps <= temps_cycloide:

```



```

421         graphe.set_color("#0165fc")
422     plt.legend()
423     plt.draw()
424     plt.pause(0.001)
425
426     return signaler_fin, rafraichir
427
428     signaler_fin, rafraichir = generer_chronometre()
429
430     # Appel de l'algorithmme hybride
431     toboggan = evoluer(
432         ligne,
433         nb_generations,
434         generation_suivante,
435         generer_incrementeur(
436             calculer_score,
437             nb_points_initial,
438             facteur_nb_points,
439             pas_initial,
440             facteur_pas,
441         ),
442         periode_lisser,
443         signaler_fin,
444         rafraichir,
445     )
446     temps = calculer_score(toboggan, 10., pas)
447
448     rafraichir(toboggan, temps)
449     print("Temps sur le toboggan optimisé : {:.f} secondes".format(temps))
450
451     if not temps_cycloide is None:
452         print(
453             (
454                 "Temps sur la cycloïde ..... : {:.f} secondes\n" +
455                 "Différence de temps ..... : {:.f} secondes"
456             ).format(temps_cycloide, abs(temps_cycloide - temps))
457         )
458     else:
459         print("La cycloïde ne permet pas de rejoindre les deux points")
460
461     # Temps d'exécution
462     print("Calculé en {:.f} secondes".format(time() - debut))
463
464     if len(sys.argv) >= 3 and sys.argv[2] == "svg":
465         plt.savefig("toboggan.svg")
466     plt.show()

```