Gautier Ben Aïm

La Brachistochrone Réelle

Informatique et Physique

Numéro d'inscription : 2163

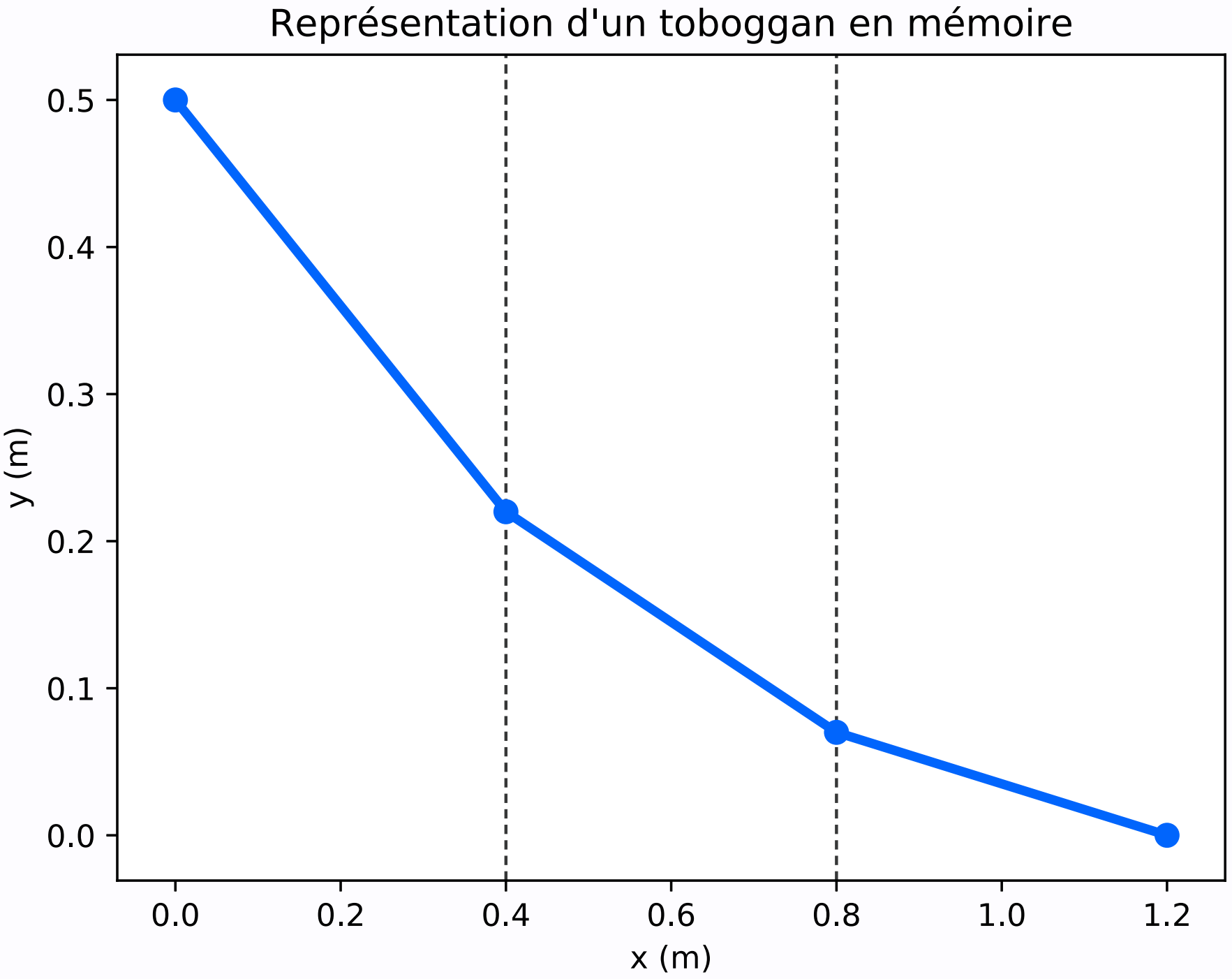
Contexte historique

- **1638**, Galilée
- **1696**, Jean Bernoulli
- **1975**, Ashby, Brittin, Love et Wyss
- **1998**, Aleksey Parnovski

La Brachistochrone Réelle

Comment construire le meilleur toboggan à l'aide d'un algorithme ?

Sommaire


Représentation d'un toboggan en mémoire
0.5
0.4
0.3
0.2
0.1
0.0
y (m)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x (m)

Ligne 25

Méthode d'Euler

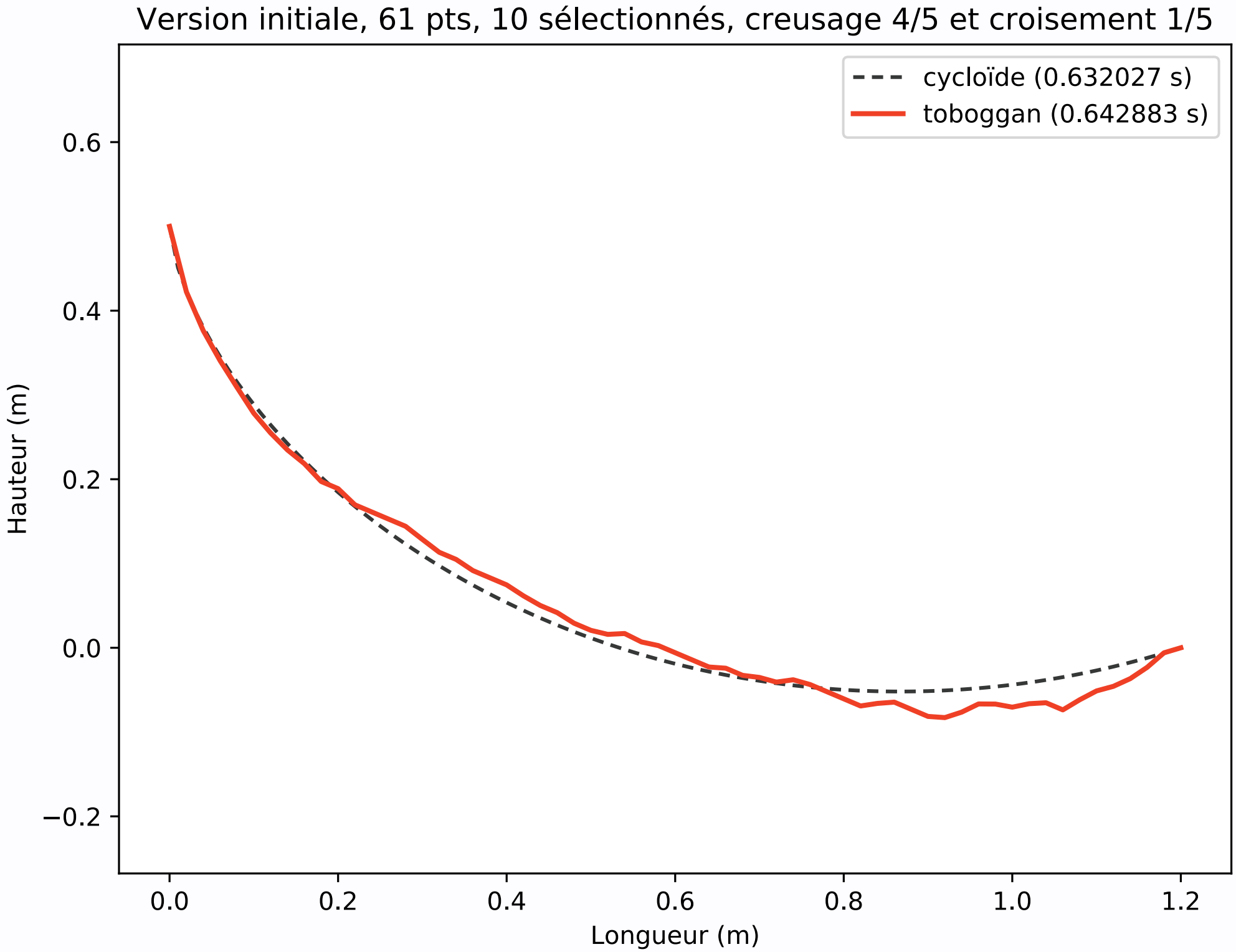
$$\begin{pmatrix} x \\ \dot{x} \end{pmatrix}_{t+\mathrm{d}t} \approx \begin{pmatrix} x \\ \dot{x} \end{pmatrix}_{t} + \begin{pmatrix} \dot{x} \\ \frac{\sum F_{ext}}{m} \end{pmatrix}_{t} \mathrm{d}t$$

Ligne 44

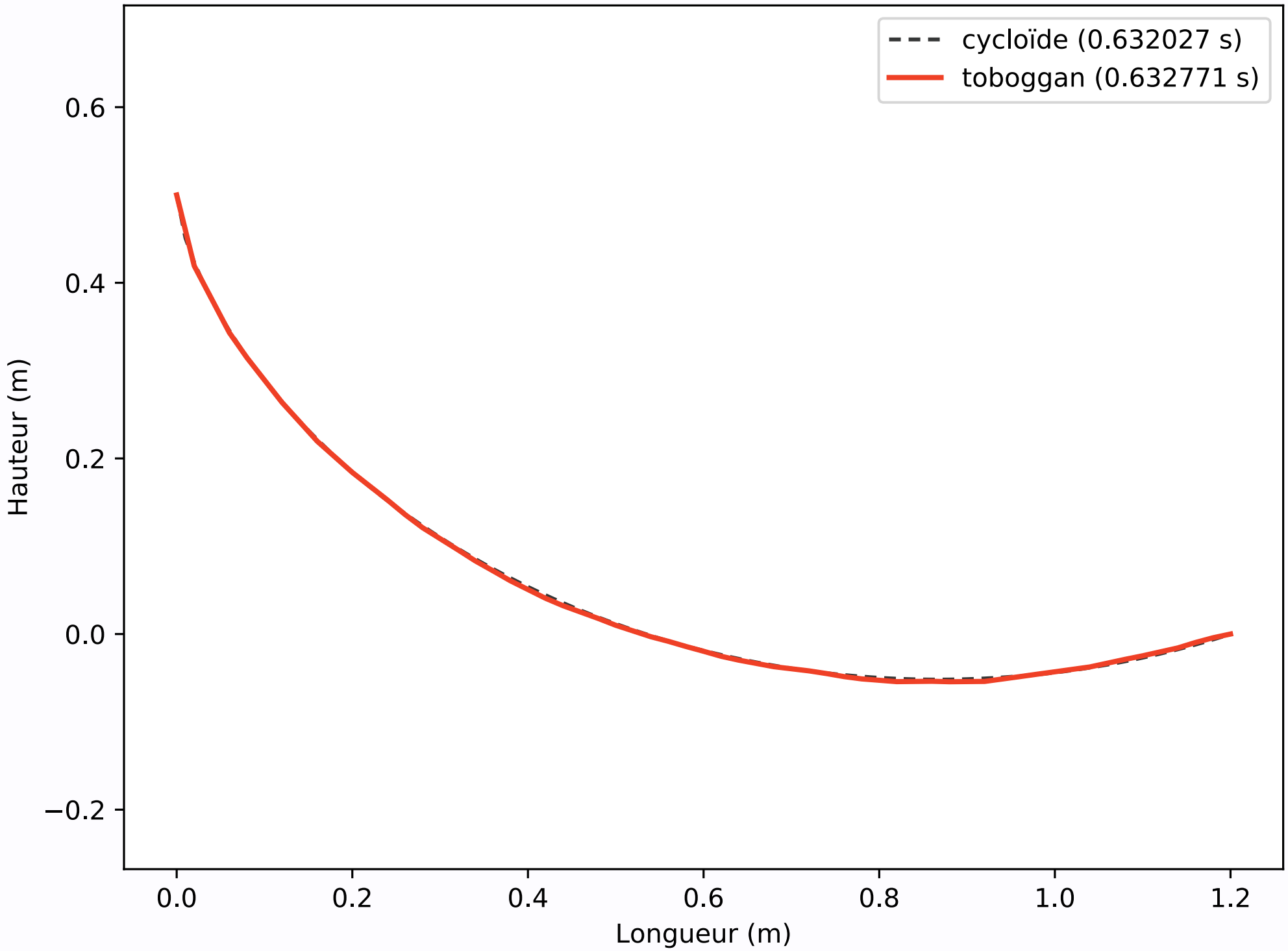
Algorithme génétique

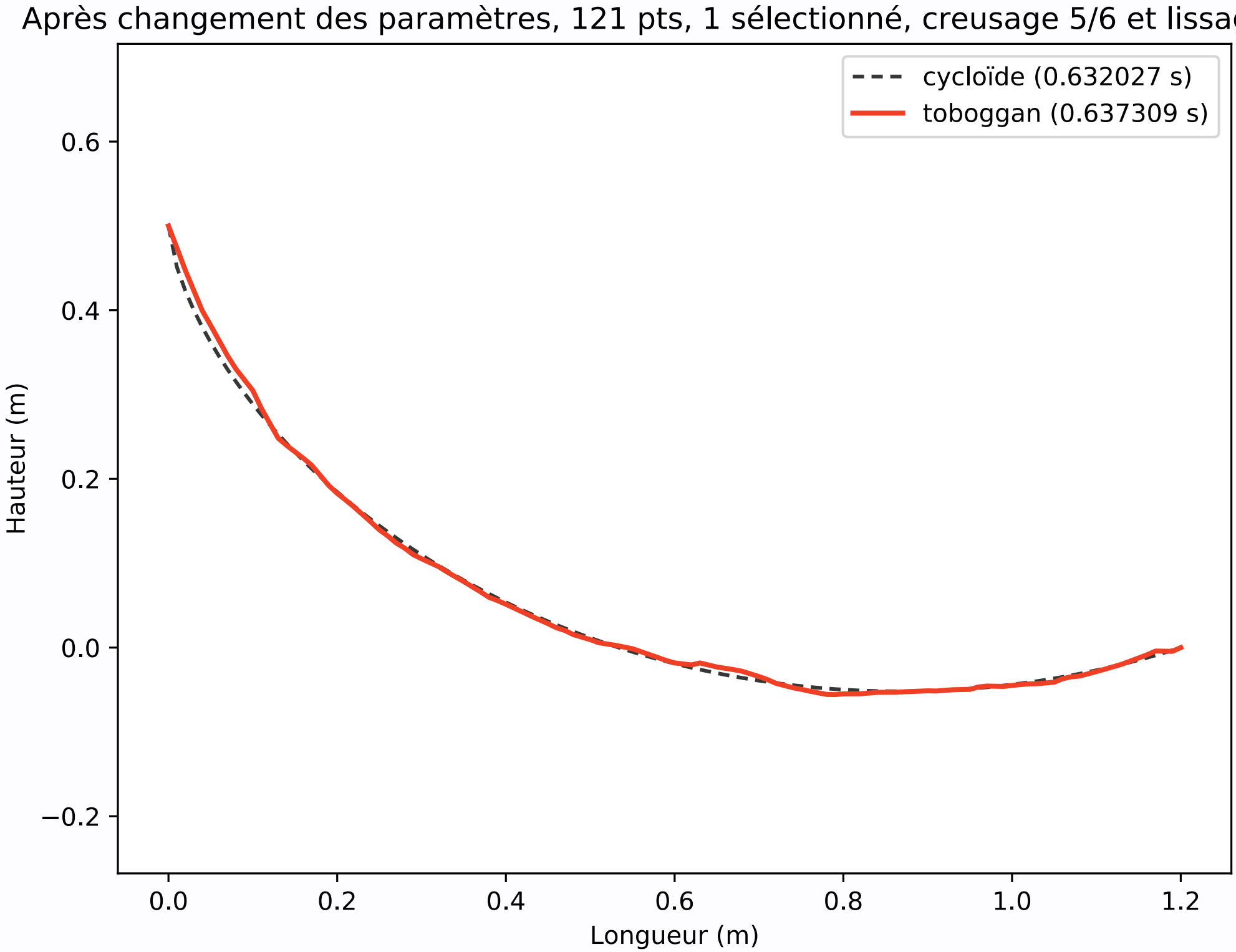
- Évolution — mutation et croisement
- Sélection

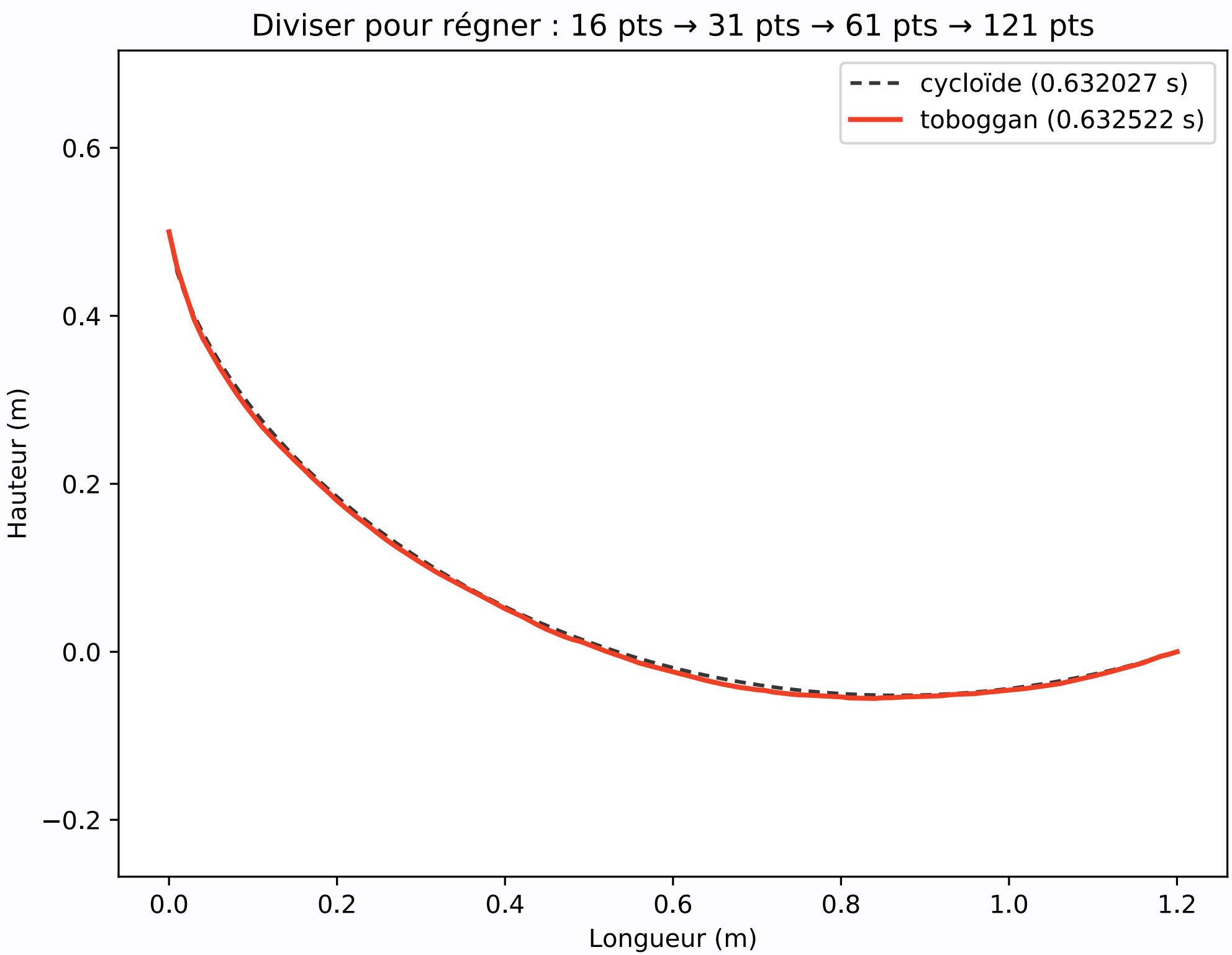
Ligne 111

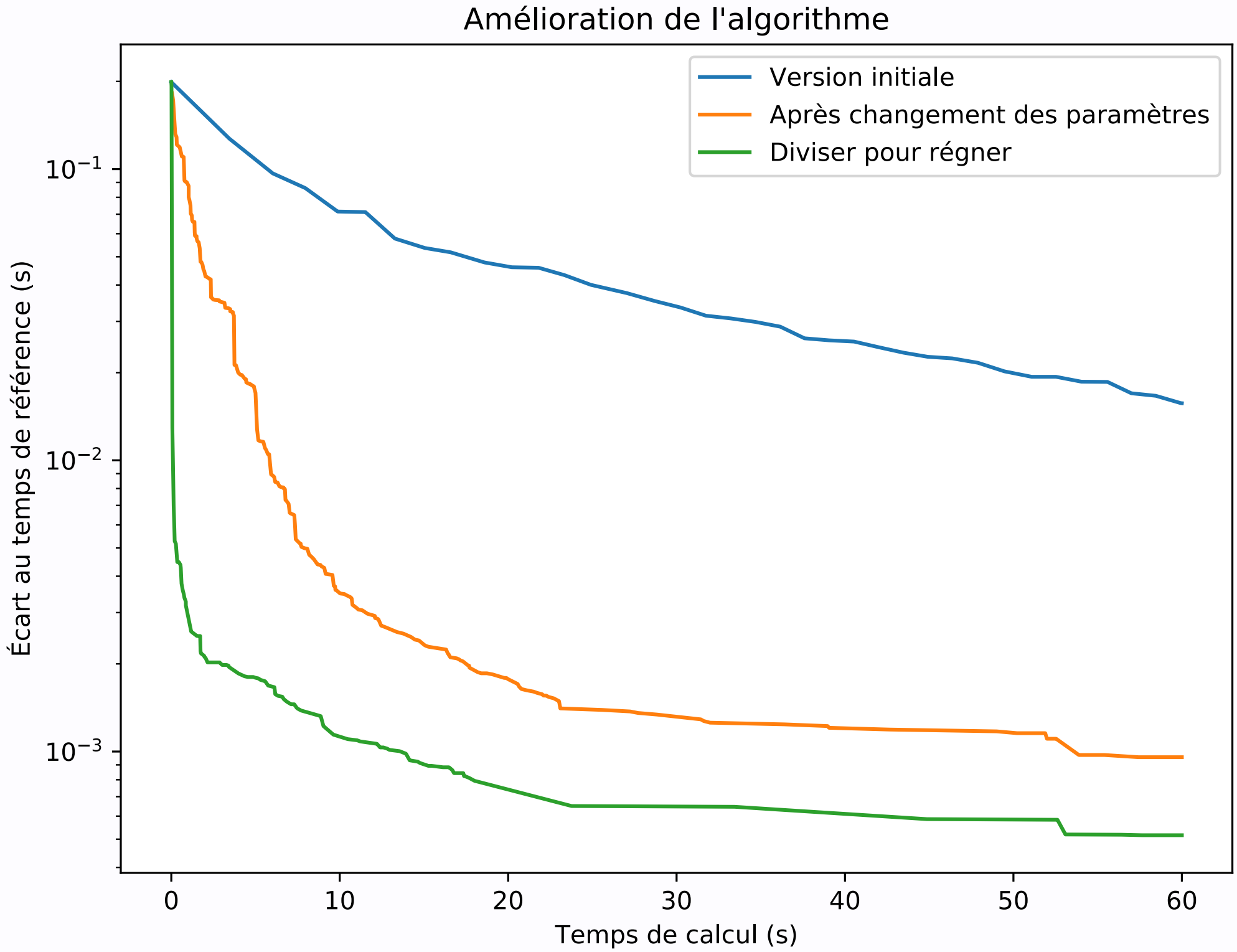

Version initiale, 61 pts, 10 sélectionnés, creusage 4/5 et croisement 1/5
cycloïde (0.632027 s)
toboggan (0.642883 s)
0.6
0.4
0.2
0.0
−0.2
Hauteur (m)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Longueur (m)

Après changement des paramètres, 61 pts, 1 sélectionné, creusage 5/6 et lissage 1/6




Après changement des paramètres, 121 pts, 1 sélectionné, creusage 5/6 et lissage 1/6
cycloïde (0.632027 s)
toboggan (0.637309 s)
Hauteur (m)
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Longueur (m)


Diviser pour régner : 16 pts → 31 pts → 61 pts → 121 pts
cycloïde (0.632027 s)
toboggan (0.632522 s)
Hauteur (m)
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Longueur (m)


Amélioration de l'algorithme
Version initiale
Après changement des paramètres
Diviser pour régner
10^{-1}
10^{-2}
10^{-3}
Écart au temps de référence (s)
0
10
20
30
40
50
60
Temps de calcul (s)

L'algorithme final

- Hybride
- Progresse continuellement

Ligne 227

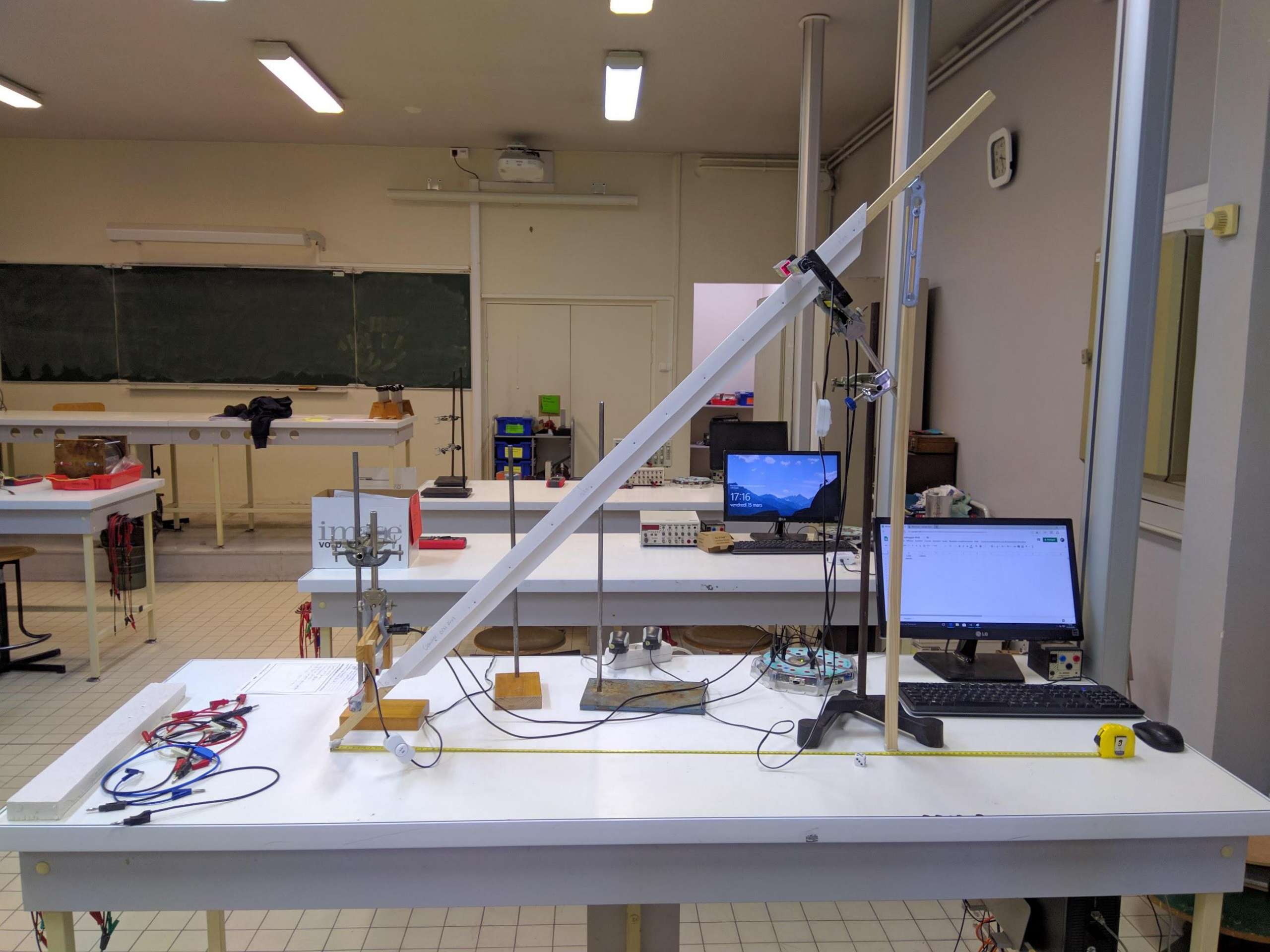
La brachistochrone *réelle*

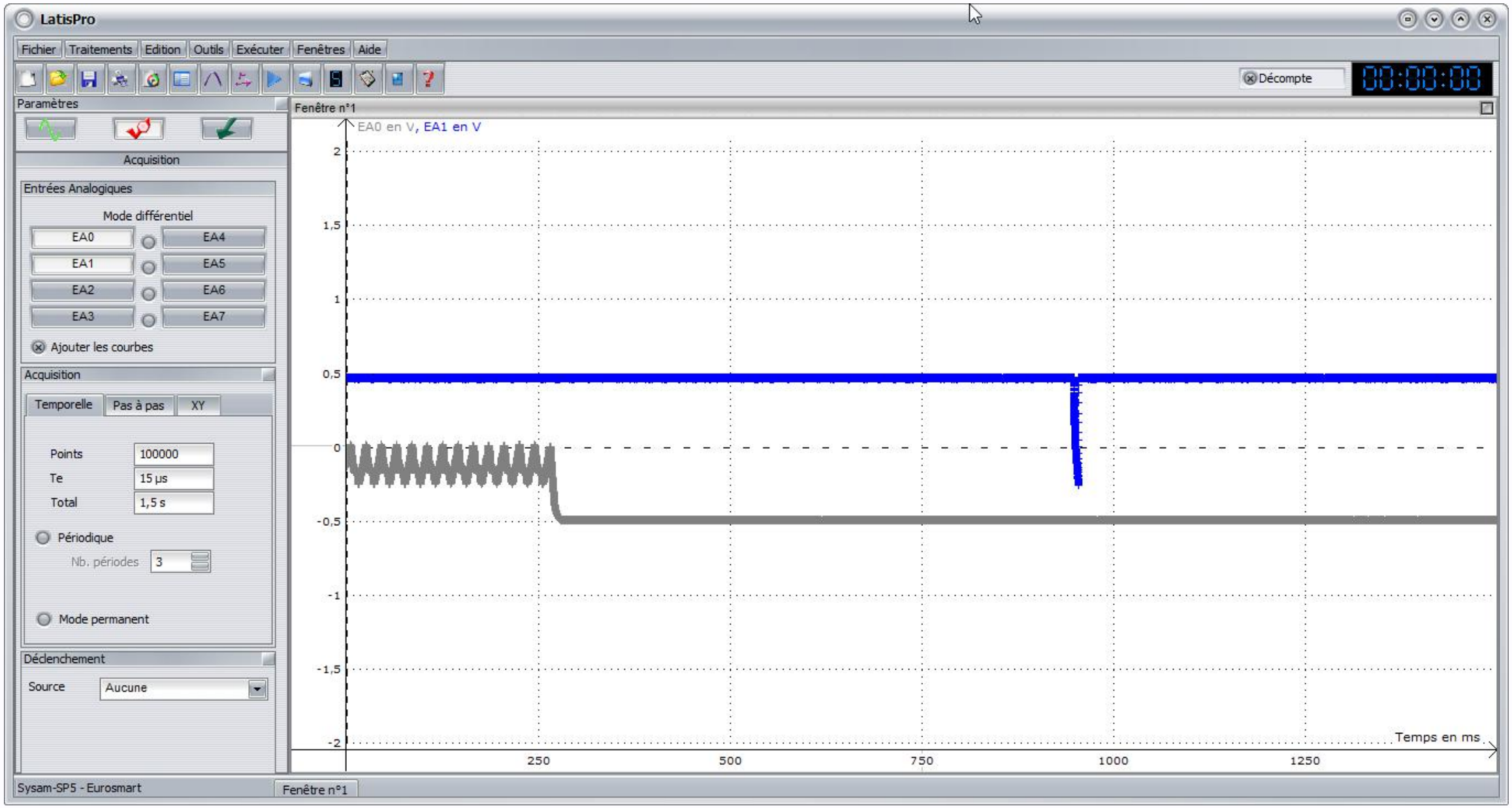
```
def deriver_v():
    return g*sin(θ)
```

```
def deriver_v(v):
    return g*sin(θ) - f*g*cos(θ) - α*v - β*v*v
```

Mesure des coefficients

- $-fg\cos\theta$
- $-\alpha v$
- $-\beta v^2$


17:16
vendredi 15 mars


LatisPro
Fichier
Traitements
Edition
Outils
Exécuter
Fenêtres
Aide
Décompte
00:00:00
Paramètres
Acquisition
Entrées Analogiques
Mode différentiel
EA0
EA1
EA2
EA3
EA4
EA5
EA6
EA7
Ajouter les courbes
Acquisition
Temporelle
Pas à pas
XY
Points
100000
Te
15 µs
Total
1,5 s
Périodique
Nb. périodes
3
Mode permanent
Déclenchement
Source
Aucune
Fenêtre n°1
EA0 en V, EA1 en V
2
1,5
1
0,5
0
-0,5
-1
-1,5
-2
250
500
750
1000
1250
Temps en ms
Sysam-SP5 - Eurosmart
Fenêtre n°1

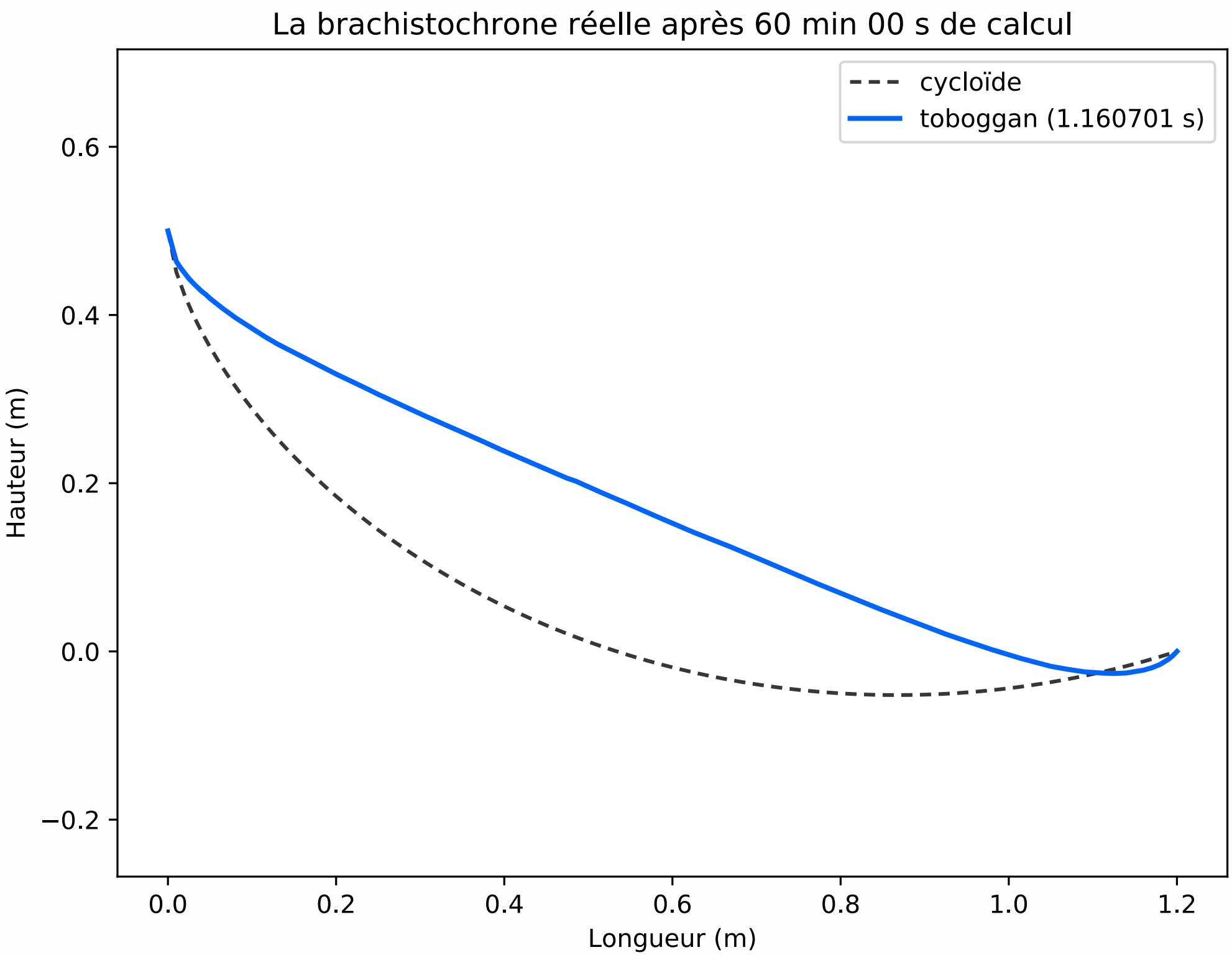
Calcul des coefficients

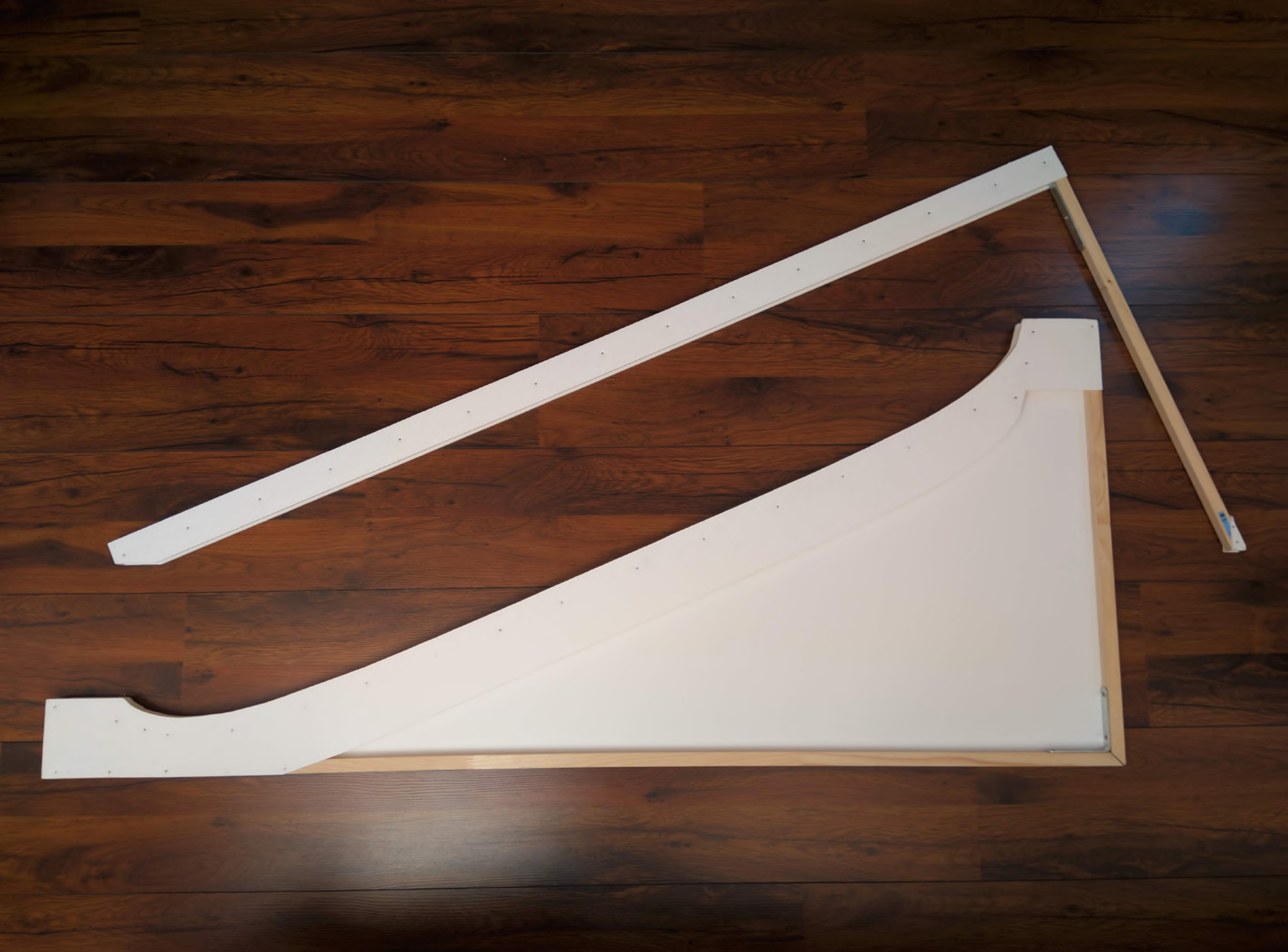
8 mesures + 1 expérience $\longrightarrow$ 3 coefficients

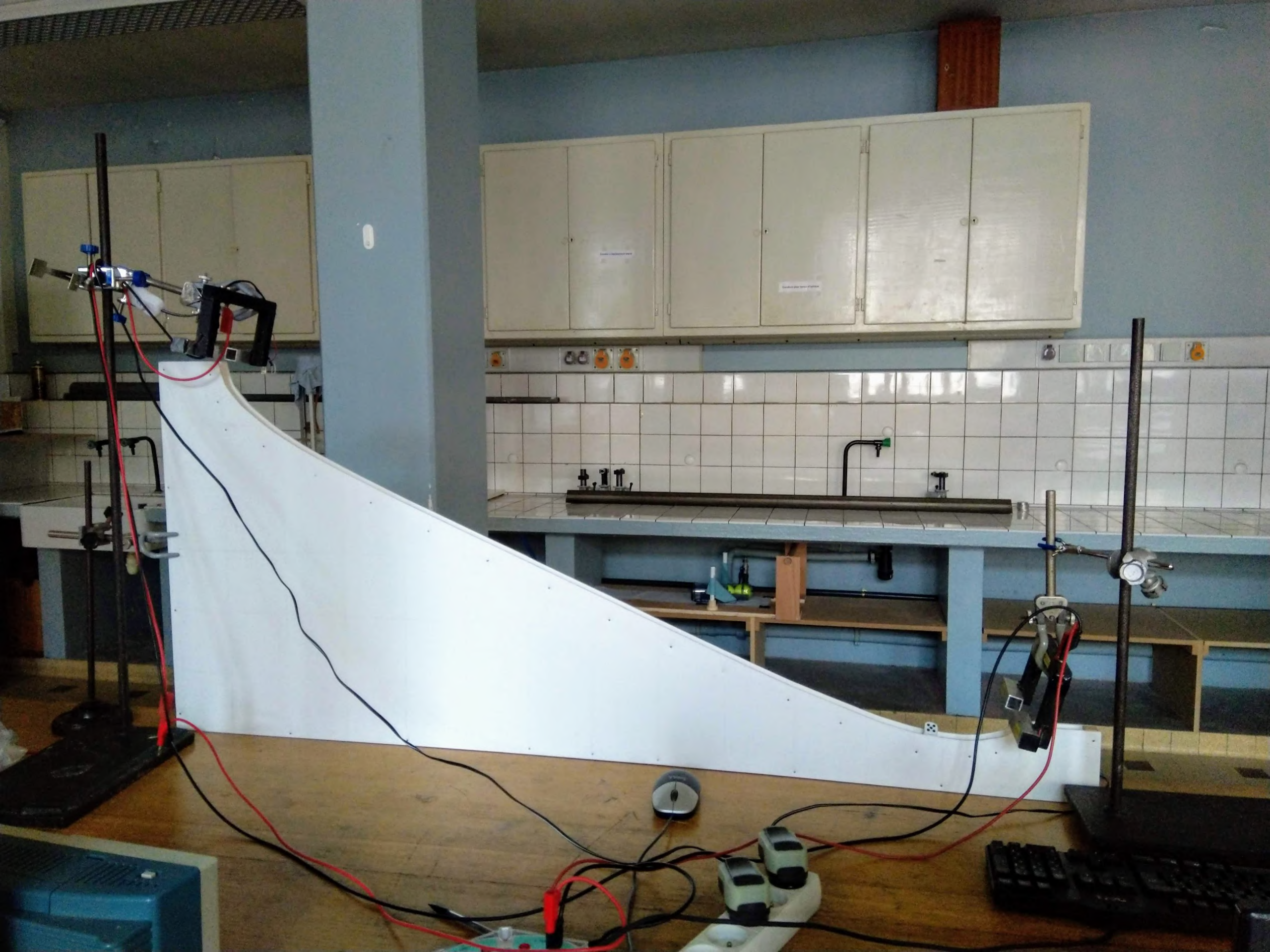
Expression des frottements

$$F = -0.3263g\cos\theta - 0.0026v - 0.4748v^2$$

Ligne 352


La brachistochrone réelle après 60 min 00 s de calcul
cycloïde
toboggan (1.160701 s)
Hauteur (m)
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Longueur (m)





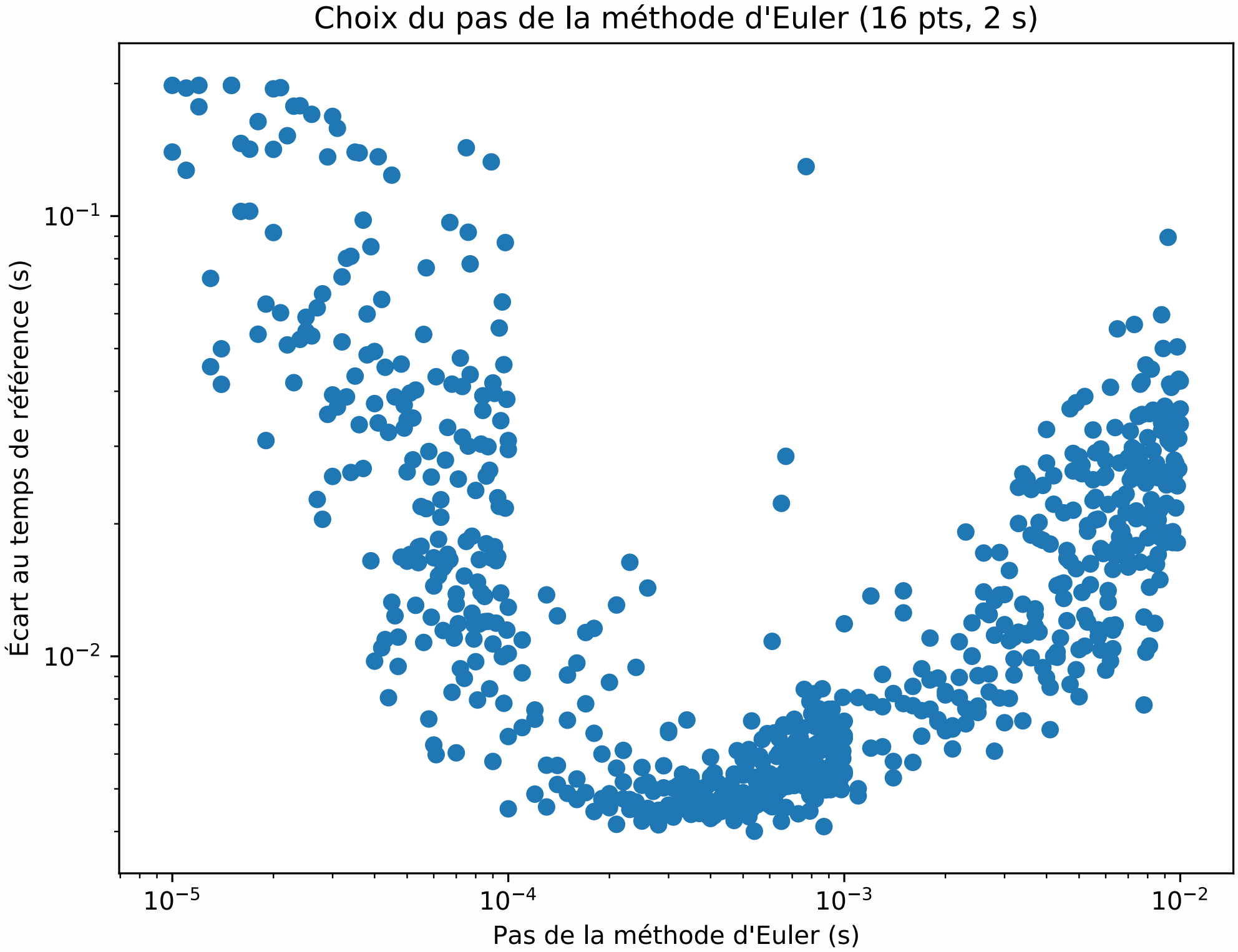
Toboggan	t_c (s)	t_m (s)	$U(t_m)$ (s)
Ligne droite	1,972	2,40	± 0,40
Optimal	1,161	1,21	± 0,02

Merci de votre attention

Vous avez des questions ?

Annexes

θ	l (cm)	t_m (ms)	t_c (ms)	Δ (ms)
45°	50	438	481	−43
	100	669	707	−39
44°	50	458	489	−31
	100	669	719	−50
40°	50	533	530	3
	100	780	779	1
36°	50	672	626	46
	100	899	920	−20


Choix du pas de la méthode d'Euler (16 pts, 2 s)
Écart au temps de référence (s)
10^{-1}
10^{-2}
10^{-5}
10^{-4}
10^{-3}
10^{-2}
Pas de la méthode d'Euler (s)

```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, facteur_nb_points, pas, facteur_pas):
    """
    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):
        """ 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,
    nb_generations,
    generation_suivante,
    incrementer_generation,
    periode_lisser,
    signaler_fin,
    rafraichir=None,
):
    """
    Améliore itérativement le toboggan donné en argument.

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

    generation = 0
    meilleur_candidat, meilleur_score, calculer_score = incrementer_generation(
        generation, toboggan, None
    )

    # 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

    # Paramètres de l'algorithme
    nb_points = 121  # Départ + intermédiaires + arrivée
    pas = 0.000001  # Intervalle de temps dt

    nb_generations = 4
    generation_suivante = 150
    periode_lisser = 8

    nb_points_initial = 16
    facteur_nb_points = 2
    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)

    # Affichage
    plt.figure("Toboggan", figsize=(8, 6), dpi=72)
    plt.plot(
        np.linspace(0., longueur, nb_points),
        [hauteur] + cycloide[2] + [0.],
        "#363737",
        dashes=[3, 2],
        label="cycloïde"
        if temps_cycloide is None
        else "cycloïde ({:f} s)".format(temps_cycloide),
    )
    graphe, = plt.plot(
        np.linspace(0., longueur, nb_points_initial),
        [hauteur] + ligne[2] + [0.],
        "#ef4026",
        linewidth=2,
        label="toboggan",
    )
    plt.title("La brachistochrone réelle")
    plt.xlabel("Longueur (m)")
    plt.ylabel("Hauteur (m)")
    plt.axis("equal")
    plt.legend()
    plt.draw()
    plt.pause(0.001)

    def generer_chronometre():
        """ Renvoie toutes les fonctions dépendantes du temps. """

        debut = time()

        def temps_ecoule():
            """ Temps écoulé. """
            return time() - debut

        def signaler_fin():
            """ Signal de fin. """
            return temps_ecoule() > temps_de_calcul

        def rafraichir(toboggan, temps):
            """ Met à jour le graphe à chaque amélioration. """
            t = temps_ecoule()
            nb_points = len(toboggan[2]) + 2
            if len(graphe.get_xdata()) != nb_points:
                graphe.set_xdata(np.linspace(0., longueur, nb_points))
            graphe.set_ydata([hauteur] + toboggan[2] + [0.])
            graphe.set_label("toboggan ({:f} s)".format(temps))
            plt.title(
                "La brachistochrone réelle après {:d} min {:0>2d} s de calcul".format(
                    int(t / 60), int(t % 60)
                )
            )
            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'algorithme 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()
```