**A2021 – INFO**

**ÉCOLE DES PONTS PARISTECH,**
**ISAE-SUPAERO, ENSTA PARIS,**
**TÉLÉCOM PARIS, MINES PARIS,**
**MINES SAINT-ÉTIENNE, MINES NANCY,**
**IMT ATLANTIQUE, ENSAE PARIS, CHIMIE PARISTECH.**

**Concours Centrale-Supélec (Cycle International),**
**Concours Mines-Télécom, Concours Commun TPE/EIVP.**

**CONCOURS 2021**

**ÉPREUVE D'INFORMATIQUE COMMUNE**

**Durée de l'épreuve : 2 heures**

**L'usage de la calculatrice et de tout dispositif électronique est interdit.**

*Cette épreuve est commune aux candidats des filières MP, PC et PSI*

*Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :*

*INFORMATIQUE COMMUNE*

*L'énoncé de cette épreuve comporte 10 pages de texte.*

*Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.*

# Marchons, marchons, marchons...

Ce sujet propose d'appliquer des techniques informatiques à l'étude de trois marches de natures différentes :

— Une marche concrète (partie I - Randonnée)
— Une marche stochastique (partie II - Mouvement brownien d'une petite particule)
— Une marche auto-évitante (partie III - Marches auto-évitantes)

Les trois parties sont indépendantes. L'annexe à la fin du sujet fournit les canevas de code que vous devrez reprendre dans votre copie pour répondre aux questions de programmation Python.

## Partie I. Randonnée

Lors de la préparation d'une randonnée, une accompagnatrice doit prendre en compte les exigences des participants. Elle dispose d'informations rassemblées dans deux tables d'une base de données :

— La table `Rando` décrit les randonnées possibles – la clef primaire entière `rid`, son nom, le niveau de difficulté du parcours (entier entre 1 et 5), le dénivelé en mètres, la durée moyenne en minutes :

| **rid** | **rnom** | **diff** | **deniv** | **duree** |
|---|---|---|---|---|
| 1 | La belle des champs | 1 | 20 | 30 |
| 2 | Lac de Castellagne | 4 | 650 | 150 |
| 3 | Le tour du mont | 2 | 200 | 120 |
| 4 | Les crêtes de la mort | 5 | 1200 | 360 |
| 5 | Yukon Ho ! | 3 | 700 | 210 |
| ... | ... | ... | ... | ... |

— La table `Participant` décrit les randonneurs – la clef primaire entière `pid`, le nom du randonneur, son année de naissance, le niveau de difficulté maximum de ses randonnées :

| **pid** | **pnom** | **ne** | **diff_max** |
|---|---|---|---|
| 1 | Calvin | 2014 | 2 |
| 2 | Hobbes | 2015 | 2 |
| 3 | Susie | 2014 | 2 |
| 4 | Rosalyn | 2001 | 4 |
| ... | ... | ... | ... |

Donner une requête SQL sur cette base pour :

❏ **Q1** – Compter le nombre de participants nés entre 1999 et 2003 inclus.

❏ **Q2** – Calculer la durée moyenne des randonnées pour chaque niveau de difficulté.

❏ **Q3** – Extraire le nom des participants pour lesquels la randonnée n°42 est trop difficile.

❏ **Q4** – Extraire les clés primaires des randonnées qui ont un ou des homonymes (nom identique et clé primaire distincte), sans redondance.

L'accompagnatrice a activé le suivi d'une randonnée par géolocalisation satellitaire et souhaite obtenir quelques propriétés de cette randonnée une fois celle-ci effectuée. Elle a exporté les données au format texte CSV (*comma-separated values* – valeurs séparées par des virgules) dans un fichier nommé `suivi_rando.csv` : la première ligne annonce le format, les suivantes donnent les positions dans l'ordre chronologique.

Voici le début de ce fichier pour une randonnée partant de Valmorel, en Savoie, un bel après-midi d'été :

```
lat(°),long(°),height(m),time(s)
45.461516,6.44461,1315.221,1597496966
45.461448,6.444426,1315.702,1597496970
45.461383,6.444239,1316.182,1597496973
45.461641,6.444035,1316.663,1597496979
45.461534,6.443879,1317.144,1597496984
45.461595,6.4437,1317.634,1597496989
45.461562,6.443521,1318.105,1597496994
...
```

Le module `math` de Python fournit les fonctions `asin`, `sin`, `cos`, `sqrt` et `radians`. Cette dernière convertit des degrés en radians, unité des fonctions trigonométriques. La documentation donne aussi des éléments de manipulation de fichiers textuels :

`fichier = open(NOM_FICHIER, MODE)` ouvre le fichier, en lecture si mode est `"r"`, en écriture si `"w"`.

`ligne = fichier.readline()` récupère la ligne suivante de `fichier` ouvert en lecture avec `open`.

`lignes = fichier.readlines()` donne la liste des lignes suivantes.

`fichier.close()` ferme `fichier`, ouvert avec `open`, après son utilisation.

`ligne.split(SEP)` découpe la chaîne de caractères `ligne` selon le séparateur `SEP` : si `ligne` vaut `"42,43,44"`, alors `ligne.split(",")` renvoie la liste `["42", "43", "44"]`.

On souhaite exploiter le fichier de suivi d'une randonnée – supposé préalablement placé dans le répertoire de travail – pour obtenir une liste `coords` des listes de 4 flottants (latitude, longitude, altitude, temps) représentant les points de passage collectés lors de la randonnée.

À partir du canevas fourni en annexe, et en ajoutant les `import` nécessaires :

❏ **Q5** – Implémenter la fonction `importe_rando` qui réalise cette importation en retournant la liste souhaitée, par exemple en utilisant certaines des fonctions ci-dessus, ou une autre approche de votre choix.

On suppose maintenant l'importation effectuée dans `coords`, avec au moins deux points d'instants distincts.

❏ **Q6** – Implémenter la fonction `plus_haut` qui renvoie la liste (latitude, longitude) du point le plus haut de la randonnée.

❏ **Q7** – Implémenter la fonction `deniveles` qui calcule les denivelés cumulés positif et négatif en mètres de la randonnée, sous forme d'une liste de deux flottants. Le dénivelé positif est la somme des variations d'altitude positives sur le chemin, et inversement pour le dénivelé négatif.

On souhaite évaluer de manière approchée la distance parcourue lors d'une randonnée. On suppose la Terre parfaitement sphérique de rayon $R_T = 6371$ km au niveau de la mer. On utilise la formule de haversine pour calculer la distance $d$ du grand cercle sur une sphère de rayon $r$ entre deux points de coordonnées (latitude, longitude) $(\varphi_1, \lambda_1)$ et $(\varphi_2, \lambda_2)$ :

$$d = 2\,r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

On prendra en compte l'altitude moyenne de l'arc, que l'on complètera pour la variation d'altitude par la formule de Pythagore, en assimilant la portion de cercle à un segment droit perpendiculaire à la verticale.

❏ **Q8** – Implémenter la fonction `distance` qui calcule avec cette approche la distance en mètres entre deux points de passage. On décomposera obligatoirement les formules pour en améliorer la lisibilité.

❏ **Q9** – Implémenter la fonction `distance_totale` qui calcule la distance en mètres parcourue au cours d'une randonnée.

## Partie II. Mouvement brownien d'une petite particule

De petites particules en suspension dans un liquide se déplacent spontanément sous l'effet de l'agitation thermique du milieu environnant, donnant ainsi naissance à des trajectoires apparemment chaotiques et peu régulières. Ce phénomène est à la base de la vie : l'agitation incessante des protéines permet aux réactions biochimiques de se produire à l'intérieur de nos cellules. Il a été observé expérimentalement en 1827 sur des grains de pollen en suspension dans l'eau par le botaniste Robert Brown, d'où le nom de mouvement brownien. Son étude théorique par Albert Einstein en 1905 a permis au physicien Jean Perrin d'estimer la valeur du nombre d'Avogadro dans une série d'expériences menées entre 1907 et 1909.

Dans cette partie, on s'intéresse à un modèle du mouvement brownien proposé par Paul Langevin en 1908. Dans ce modèle, la particule étudiée est supposée soumise à deux actions de la part du fluide :

— Une force de frottement fluide $\overrightarrow{f_F} = -\alpha\,\overrightarrow{v}$

— Une force $\overrightarrow{f_B}$ aléatoire simulant l'action désordonnée des molécules d'eau sur la particule.

L'équation différentielle à laquelle est soumise cette particule est alors :

$$\frac{d\overrightarrow{v}}{dt} = -\frac{\alpha\,\overrightarrow{v}}{m} + \frac{\overrightarrow{f_B}}{m}$$

Pour faire une simulation en 2 dimensions, on prend une particule de masse $m = 10^{-6}$ kg, on attribue à $\alpha$ la valeur $10^{-5}$ kg/s, on suppose enfin que $\overrightarrow{f_B}$ change à chaque pas d'intégration, avec une direction isotrope aléatoire (l'angle suit une loi de probabilité uniforme) et une norme qui est la valeur absolue d'une variable aléatoire qui suit une loi de probabilité gaussienne (loi normale) d'espérance $\mu$ nulle et d'écart-type $\sigma = 10^{-8}$ N.

On simule le vecteur d'état $E = (x, y, \dot{x}, \dot{y})$ de la particule en intégrant $\dot{E} = (\dot{x}, \dot{y}, \ddot{x}, \ddot{y})$ selon la méthode d'Euler. $E$ et $\dot{E}$ seront chacun représentés par une liste de 4 flottants.

L'instruction `assert` *`expression`* de Python vérifie la véracité d'une expression booléenne et interrompt brutalement l'exécution du programme si ce n'est pas le cas. Elle permet de vérifier très simplement une précondition ou un invariant. Voir l'exemple à la ligne 10 du canevas.

Le module `random` de Python fournit la fonction `uniform(bi, bs)`, qui renvoie un flottant aléatoire entre les valeurs `bi` et `bs` incluses en utilisant une densité de probabilité uniforme, et la fonction `gauss(mu, sigma)`, qui renvoie un flottant aléatoire en utilisant une densité de probabilité gaussienne d'espérance `mu` et d'écart-type `sigma`.

Le module `math` fournit enfin les fonctions `cos` et `sin` (en radians), la fonction `sqrt` et la constante `pi`. La fonction `abs` est directement disponible.

En utilisant le canevas fourni en annexe, et en ajoutant les `import` nécessaires :

❏ **Q10** – Implémenter la fonction `vma(v1, a, v2)` (multiplication-addition sur des vecteurs) avec `v1` et `v2` des listes de flottants et `a` un scalaire flottant, qui renvoie une nouvelle liste de flottants correspondant à :

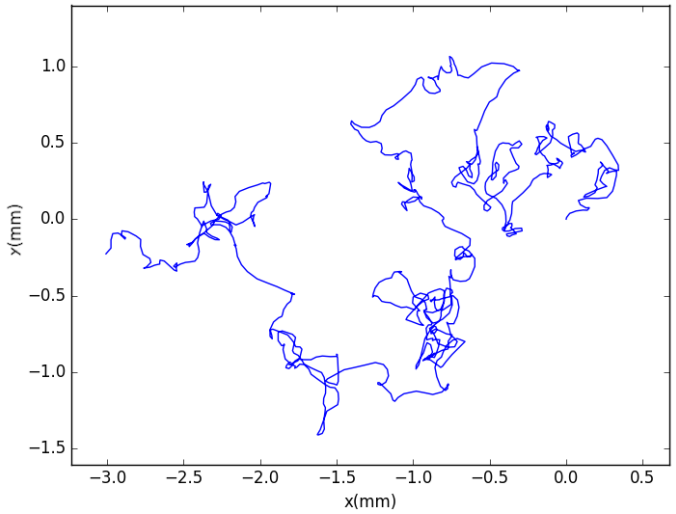
$$\overrightarrow{v} = \overrightarrow{v_1} + a\,\overrightarrow{v_2}$$

La fonction vérifiera que les deux listes en entrée sont de même longueur avec `assert`.

❏ **Q11** – Implémenter la fonction `derive(E)` qui renvoie la dérivée du vecteur d'état passé en paramètre d'après l'équation différentielle décrite en introduction de la partie.

❏ **Q12** – Implémenter la fonction `euler(E0, dt, n)` pour résoudre numériquement l'équation différentielle par la méthode d'Euler avec `E0` le vecteur d'état initial, `dt` le pas d'intégration et `n` le nombre de pas de la simulation. La fonction renvoie la liste des $n + 1$ vecteurs d'états.

À titre d'exemple, voici le tracé d'une trajectoire obtenue par cette méthode :




## Partie III. Marche auto-évitante

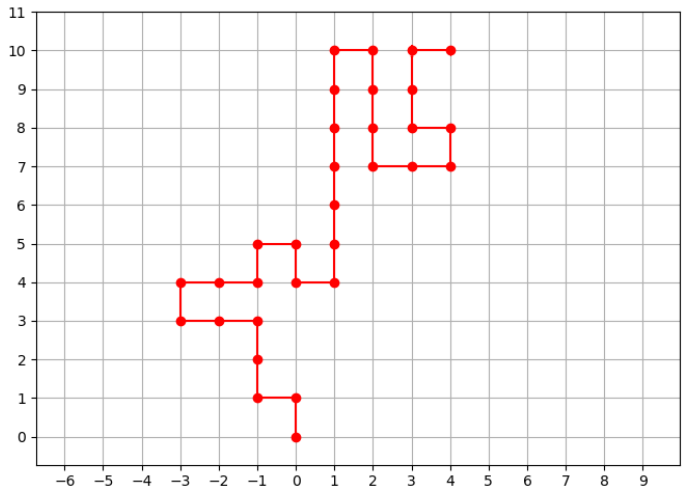
Une marche auto-évitante est un processus au cours duquel un point décrit un chemin auto-évitant, c'est à dire qui ne se recoupe jamais lui-même. Ce modèle peut servir lors de la modélisation d'une chaîne polymère : En effet, deux monomères distincts de la chaîne ne peuvent pas se trouver à deux positions identiques pour des raisons d'encombrement stérique.

Dans ce sujet, on appellera chemin auto-évitant (CAE) de longueur $n$ tout ensemble de $n+1$ points $P_i \in \mathbb{Z}^2$ pour $0 \leq i \leq n$ tels que :

- $\forall i \qquad \|P_{i+1} - P_i\| = 1$
- $\forall (i,j) \quad i \neq j \Rightarrow P_i \neq P_j$

Chaque point du chemin sera représenté par une liste à deux éléments entiers précisant les deux coordonnées (par exemple $P_i = (5, -4)$ est représenté par la liste `[5,-4]`). Le chemin lui-même est constitué de la liste des points, dans l'ordre dans lequel ils sont atteints. Les codes Python produits dans cette partie devront manipuler exclusivement des coordonnées entières.

Voici un exemple de représentation graphique de CAE de longueur 30 à partir de $(0, 0)$ :



On s'intéresse dans un premier temps à une méthode naïve pour générer un chemin auto-évitant de longueur $n$ sur une grille carrée. La méthode adoptée est une approche gloutonne :

1. Le premier point est choisi à l'origine : $P_0 = (0, 0)$
2. En chaque position atteinte par le chemin, on recense les positions voisines accessibles pour le pas suivant et on en sélectionne une au hasard. En l'absence de positions accessibles, l'algorithme échoue.
3. On itère sur l'étape 2 jusqu'à ce que le chemin possède la longueur désirée, ou échoue.

Le module Python `random` fournit la fonction `randrange(n)`, qui renvoie un entier compris entre 0 et $n-1$ inclus, et la fonction `choice(L)`, qui renvoie l'un des éléments de la liste `L`, dans les deux cas choisis aléatoirement avec une probabilité uniforme.

L'expression Python `x in L` est une expression booléenne qui vaut `True` si `x` est l'un des éléments de `L` et `False` sinon. On supposera que la méthode employée pour évaluer cette expression sur une liste est une recherche séquentielle.

La valeur spéciale `None` est utilisée en Python pour représenter une valeur invalide, inconnue ou indéfinie. L'expression booléenne `v is None` indique si la valeur de `v` est cette valeur spéciale.

En utilisant le canevas fourni en annexe, et en ajoutant les `import` nécessaires :

❏ **Q13** – Implémenter la fonction `positions_possibles(p, atteints)` qui construit la liste des positions suivantes possibles à partir du point `p`. La liste `atteints` contient les points déjà atteints par le chemin.

❏ **Q14** – Mettre en évidence graphiquement un exemple de CAE le plus court possible pour lequel, à une étape donnée, la fonction `positions_possibles` va renvoyer une liste vide. En prennant en compte les symétries, combien de tels chemins distincts existent pour cette longueur minimale ?

❏ **Q15** – Implémenter la fonction `genere_chemin_naif(n)` qui construit la liste des points représentant le chemin auto-évitant de longueur `n` et renvoie le résultat, ou bien renvoie la valeur spéciale `None` si à une étape `positions_possibles` renvoie une liste vide.

❏ **Q16** – Évaluer avec soin la complexité temporelle asymptotique dans le pire des cas de la fonction `genere_chemin_naif(n)` en fonction de `n`, en supposant que la fonction ne renvoie pas `None`.

Une personne curieuse et patiente a écrit le code suivant :

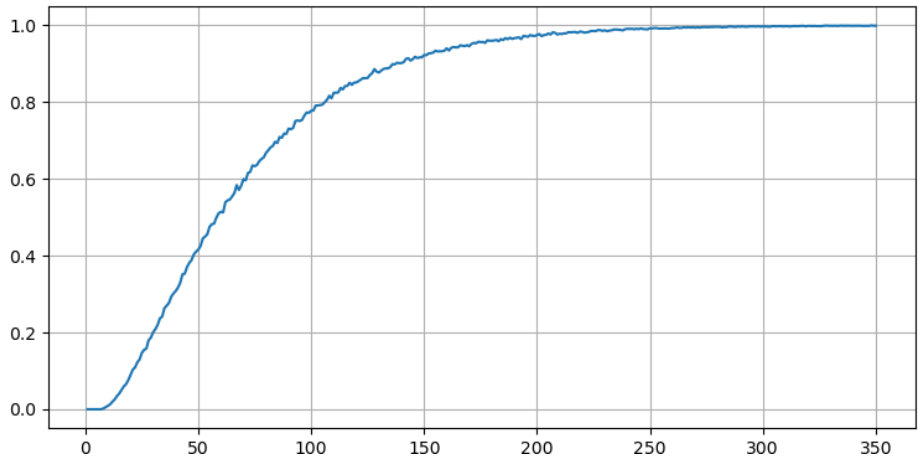
```
from chemin import genere_chemin_naif

N, M, L, P = 10000, 351, [], []

for n in range(1, M):
    nb = 0
    for i in range(N):
        chemin = genere_chemin_naif(n)
        if chemin is None:
            nb += 1
    L.append(n)
    P.append(nb / N)

import matplotlib.pyplot as plt
plt.plot(L, P)
plt.grid()
plt.show()
```

Après un long moment, elle a obtenu le graphique suivant, volontairement laissé sans étiquettes d'axes :

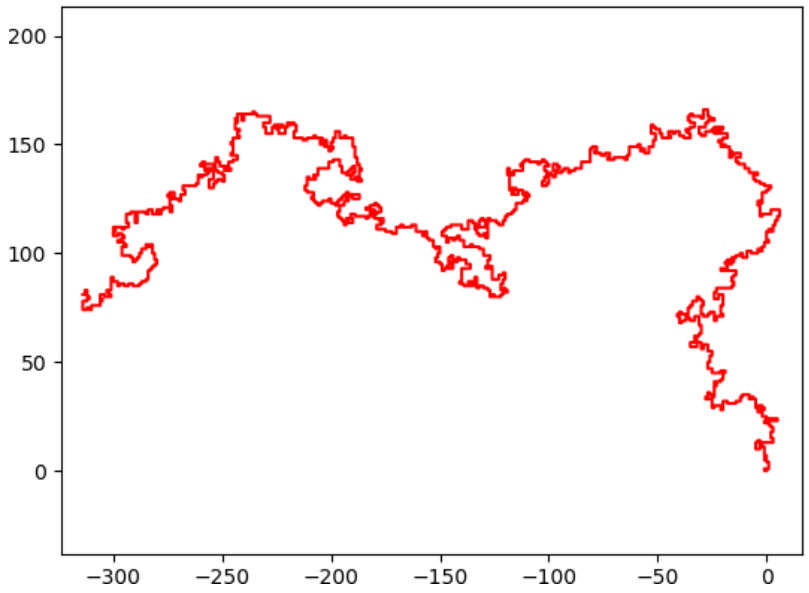



❏ **Q17** – Décrire ce que représente ce graphique, et interpréter sa signification pour la méthode naïve.

Afin d'éviter les inconvénients de la méthode précédente, on s'intéresse à une solution différente nommée *méthode du pivot*, proposée par Moti Lal en 1969. Son principe est le suivant :

1. On part d'un chemin auto-évitant arbitraire de longueur $n$. Ici, on choisira une initialisation très simple, le chemin droit `[ [0, 0], [1, 0], [2, 0], ..., [n, 0] ]`.
2. On sélectionne au hasard un point, nommé pivot, entre le second et l'avant-dernier du chemin, et un angle aléatoire de rotation parmi $\pi$, $+\frac{\pi}{2}$ et $-\frac{\pi}{2}$.
3. On laisse les points avant le pivot inchangés, et on fait subir une rotation de centre le pivot et d'angle choisi à l'ensemble des points situés strictement après le pivot.
4. Si le chemin ainsi obtenu est auto-évitant, on le garde. Sinon, on reprend à l'étape 2 la sélection d'un pivot et d'un angle, jusqu'à en trouver qui conviennent.
5. On répète les étapes 2 à 4 un certain nombre de fois. Le choix du nombre minimal de rotations à effectuer pour obtenir un chemin non corrélé au chemin initial est laissé de côté dans ce sujet.

Cette méthode permet de générer de manière efficace des marches auto-évitantes de plusieurs milliers de points, comme la marche ci-dessous de longueur 2000 :



Dans cet algorithme, une étape importante est la vérification qu'un chemin donné est auto-évitant. Pour vérifier cela on pourrait bien sûr s'y prendre comme dans la fonction `positions_possibles`, mais on adopte une méthode différente avec une fonction `est_CAE(chemin)` qui trie les points du chemin, puis met à profit ce tri pour vérifier efficacement que le chemin est auto-évitant.

On utilise pour cela la fonction `sorted` Python qui renvoie une nouvelle liste triée par ordre croissant. Elle fonctionne sur tous types d'éléments, y compris des listes pour lesquelles l'ordre lexicographique (ordre du premier élément, en cas d'égalité du second, etc.) est appliqué. On suppose de plus que la complexité temporelle asymptotique dans le pire des cas de cette fonction est la meilleure possible.

❏ **Q18** – Rappeler, sans la justifier, la complexité temporelle asymptotique dans le pire des cas attendue de `sorted` en fonction de la longueur de la liste à trier. Donner le nom d'un algorithme possible pour son implémentation.

En utilisant le canevas fourni en annexe, et en ajoutant les `import` nécessaires :

❏ **Q19** – Implémenter la fonction `est_CAE(chemin)` qui vérifie si un chemin est auto-évitant en se basant sur `sorted` et renvoie un résultat booléen. Elle devra ne pas être de complexité temporelle asymptotique dans le pire des cas supérieure à la fonction `sorted` : vous prouverez ce dernier point.

❏ **Q20** – Implémenter la fonction `rot(p, q, a)` qui renvoie le point image du point `q` par la rotation de centre `p` et d'angle définit par la valeur de `a` : `0` pour $\pi$, `1` pour $\frac{\pi}{2}$, `2` pour $-\frac{\pi}{2}$.

❏ **Q21** – Implémenter la fonction `rotation(chemin, i_piv, a)` qui renvoie un nouveau chemin identique à `chemin` jusqu'au pivot d'indice `i_piv`, et qui a subi une rotation de `a` autour du pivot (même codage que la fonction précédente) sur la partie strictement après le pivot.

❏ **Q22** – Implémenter la fonction `genere_chemin_pivot(n, n_rot)` permettant de générer un chemin auto-évitant de longueur `n` en appliquant `n_rot` rotations. L'application d'une rotation peut nécessiter plusieurs tentatives.

❏ **Q23** – On considère un pivot, son point précédent et son point suivant : Quel est l'impact prévisible sur les rotations admissibles ? Suggérer un moyen de prendre en compte cette propriété pour améliorer l'algorithme.

*Fin de l'énoncé*

*Annexe canevas de codes Python sur les pages suivantes*

Annexe : canevas de codes Python

— Partie I : Randonnée

```python
# import Python à compléter...

# importation du fichier d'une randonnée
def importe_rando(nom_fichier):
    # À compléter...

coords = importe_rando("suivi_rando.csv")

# donne le point (latitude, longitude) le plus haut de la randonnée
def plus_haut(coords):
    # À compléter...

print("point le plus haut", plus_haut(coords))
# exemple : point le plus haut [45.461451, 6.443064]

# calcul des dénivelés positif et négatif de la randonnée
def deniveles(coords):
    # À compléter...

print("denivelés", deniveles(coords), "m")
# exemple : denivelés [4.059999999999945, -1.1759999999999309] m

RT = 6371 # rayon moyen volumétrique de la Terre en km

# distance entre deux points
def distance(coord1, coord2):
    # À compléter...

print("premier intervalle", distance(coords[0], coords[1]), "m")
# exemple : premier intervalle 16.230964254992816 m

# distance totale de la randonnée
def distance_totale(coords):
    # À compléter...

print("distance parcourue", distance_totale(coords), "m")
# exemple : distance parcourue 187.9700904658368 m
```

— Partie II : Mouvement brownien

```python
# import Python à compléter...

# paramètres physiques
MU = 0.0      # N
SIGMA = 1E-8  # N
M = 1E-6      # kg
ALPHA = 1E-5  # kg/s

# vérification des hypothèses sur les paramètres
assert MU >= 0 and SIGMA > 0 and M > 0 and ALPHA > 0

# multiplication-addition vectorielle
def vma(v1, a, v2):
    # À compléter...

# dérivée du vecteur d'état
def derive(E):
    # À compléter...

# intégration par la méthode d'Euler
def euler(E0, dt, n):
    Es = [ E0 ]
    # À compléter...
    return Es

# simulation
DT = 0.002    # durée du pas en secondes
N = 5000      # nombre de pas

Es = euler([ 0.0, 0.0, 0.0, 0.0 ], DT, N)

print("trajectoire", Es)
```

— Partie III : Chemin auto-évitant - méthode naïve

```python
# import Python à compléter...

# positions auto-évitantes suivantes possibles
def positions_possibles(p, atteints):
    possibles = []
    # À compléter...
    return possibles

# génération gloutonne d'un chemin de longueur n
# renvoie None en cas d'échec
def genere_chemin_naif(n):
    chemin = [ [ 0, 0 ] ] # on part de l'origine
    # À compléter...
    return chemin

N = 10
print("chemin", genere_chemin_naif(N))
```

— Partie III : Chemin auto-évitant - méthode du pivot

```python
# import Python à compléter...

# vérifie si un chemin est CAE
def est_CAE(chemin):
    # À compléter...

# calcule la rotation de q autour de p selon a :
# Pi si a vaut 0, Pi/2 si a vaut 1, -Pi/2 si a vaut 2
def rot(p, q, a):
    # À compléter...

# renvoie le chemin dont les points après i_pivot
# ont subi une rotation a codé comme précédemment
def rotation(chemin, i_pivot, a):
    # À compléter...

# génère un chemin de longueur n (donc n+1 points)
def genere_chemin_pivot(n, n_rot):
    # À compléter...

N, A = 1000, 2.3
print("chemin", genere_chemin_pivot(N, int( A * N )))
```