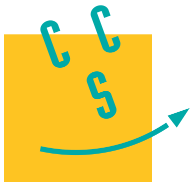

CONCOURS CENTRALE•SUPÉLEC

Informatique

MP, PC, PSI, TSI

3 heures — Calculatrices autorisées

2017

Mars Exploration Rovers
Mission d'exploration martienne

Mars Exploration Rovers (MER) est une mission de la NASA qui cherche à étudier le rôle joué par l'eau dans l'histoire de la planète Mars. Deux robots géologues, Spirit et Opportunity (figure 1), se sont posés sur cette planète, sur deux sites opposés, en janvier 2004. Leur mission est de rechercher et d'analyser différents types de roches et de sols qui peuvent contenir des indices sur la présence d'eau. Ils sont équipés de six roues et d'une suspension spécialement conçue pour leur permettre de se déplacer quelle que soit la nature du terrain rencontré. Leur cahier des charges prévoyait une durée de vie de 90 jours martiens (le jour martien est environ 40 minutes plus long que le jour terrestre). Spirit a cessé d'émettre le 22 mars 2010, soit 2210 jours martiens après son arrivée sur la planète. Début 2017, Opportunity est toujours en activité et il a parcouru plus de 44 km sur Mars.

Figure 1 Vue d'artiste d'un robot géologue de la mission *Mars Exploration Rovers* (NASA/JPL – Caltech/Cornell)

Chaque robot est équipé de plusieurs instruments d'analyse (caméra, microscope, spectromètres) et d'un bras qui permet d'amener les instruments au plus près des roches et sols dignes d'intérêt. À partir de photographies de la surface de la planète, prises à plusieurs longueurs d'ondes par différents satellites et par le robot lui-même, les scientifiques de la NASA définissent une liste d'emplacements (*points d'intérêt* ou PI) où effectuer des analyses. Cette liste est transmise au robot qui doit se rendre à chaque emplacement indiqué et y effectuer les analyses prévues. Chaque robot est capable d'effectuer un certain nombre de types d'analyses géologiques correspondant aux différents instruments dont il dispose. Une fois tous les points d'intérêts visités et les résultats des analyses

cc BY-NC-SA

transmis à la Terre, le robot reçoit une nouvelle liste de points d'intérêts et démarre une nouvelle *exploration*. Compte-tenu des contraintes de transmission entre la Terre et les robots (latence, périodes d'ombre, faible débit, etc.) il est prévu que les robots travaillent en autonomie pour planifier le parcours de chaque exploration. Ainsi, une fois la liste des points d'intérêt reçue, le robot analyse le terrain afin de détecter d'éventuels obstacles et détermine le meilleur chemin lui permettant de visiter l'ensemble de ces points en dépensant le moins d'énergie possible.

Après s'être intéressé à l'enregistrement des explorations, des points d'intérêts correspondants et des analyses à y mener, ce sujet aborde trois algorithmes qui peuvent être utilisés par le robot pour déterminer le meilleur parcours lui permettant de visiter chaque point d'intérêt une et une seule fois. Pour cela nous faisons quelques hypothèses simplificatrices.

- La zone d'exploration est dépourvue d'obstacle : le robot peut rejoindre directement en ligne droite n'importe quel point d'intérêt.
- Le sol est horizontal et de nature constante : l'énergie utilisé pour se déplacer entre deux points ne dépend que de leur distance, autrement dit le meilleur chemin est le plus court.
- La courbure de la planète est négligée compte tenu de la dimension réduite de la zone d'exploration : nous travaillerons en géométrie euclidienne et les points d'intérêts seront repérés par leurs coordonnées cartésiennes à l'intérieur de la zone d'exploration.

Les seuls langages de programmation autorisés dans cette épreuve sont Python et SQL. Toutes les questions sont indépendantes. Néanmoins, il est possible de faire appel à des fonctions ou procédures créées dans d'autres questions. Dans tout le sujet on suppose que les bibliothèques `math`, `numpy` et `random` ont été importées grâce aux instructions

```
import math
import numpy as np
import random
```

Si les candidats font appel à des fonctions d'autres bibliothèques ils doivent préciser les instructions d'importation correspondantes.

Ce sujet utilise la syntaxe des annotations pour préciser le types des arguments et du résultat des fonctions à écrire. Ainsi

```
def maFonction(n:int, x:float, d:str) -> np.ndarray:
```

signifie que la fonction `maFonction` prend trois arguments, le premier est un entier, le deuxième un nombre à virgule flottante et le troisième une chaine de caractères et qu'elle renvoie un tableau numpy.

Une liste de fonctions utiles est donnée à la fin du sujet.

I Création d'une exploration et gestion des points d'intérêt

Une exploration est un ensemble de points d'intérêts à l'intérieur d'une zone géographique limitée, une série d'analyses étant associée à chaque point d'intérêt. Chaque type d'analyse que le robot peut effectuer est codifié et référencé par un nombre entier. Un point d'intérêt est repéré par deux entiers, positifs ou nuls, correspondant à ses coordonnées cartésiennes en millimètres à l'intérieur de la zone d'exploration. L'ensemble des points d'intérêts d'une exploration qui en contient n est représenté par un objet de type `numpy.ndarray`, à éléments entiers, à 2 colonnes et n lignes, l'élément d'indice `i,0` correspondant à l'abscisse du point d'intérêt i et l'élément d'indice `i,1` à son ordonnée.

	x	y
0	345	635
1	1076	415
2	38	859
3	121	582

Figure 2 Exemple d'exploration avec quatre points d'intérêt

I.A – Génération d'une exploration d'essai

I.A.1) Choix de points au hasard

a) Afin de disposer de données pour tester les différents algorithmes de calcul de chemin qui seront développés plus tard, écrire une fonction qui construit une exploration au hasard. Cette fonction d'entête

```
def générer_PI(n:int, cmax:int) -> np.ndarray:
```


(cc) BY-NC-SA

prend en paramètres le nombre de points d'intérêts à générer et la largeur de la zone d'exploration (supposée carrée) et renvoie un objet de type `numpy.ndarray` contenant les coordonnées de n points **deux à deux distincts** choisis au hasard dans la zone d'exploration (figure 2).

b) Quelles contraintes doivent vérifier les arguments de la fonction `générer_PI` ?

I.A.2) Calcul des distances

On dispose de la fonction d'entête

```
def position_robot() -> tuple:
```

qui renvoie un couple donnant les coordonnées actuelles du robot dans le système de coordonnées de l'exploration à planifier. Ainsi l'instruction `x, y = position_robot()` permet de récupérer les coordonnées courantes du robot.

Afin de faciliter l'application des différents algorithmes de recherche de chemin, on souhaite construire un tableau des distances entre les différents points d'intérêt d'une exploration et entre ceux-ci et la position courante du robot au moment du calcul. Écrire une fonction d'entête

```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
```

qui prend en paramètre un tableau de n points d'intérêt tel que décrit précédemment et renvoie un tableau de nombres flottants, de dimension $(n+1) \times (n+1)$, tel que l'élément d'indice `i,j` fournit la distance entre les points d'intérêt i et j, l'indice `n` désignant le point de départ du robot.

I.B – Traitement d'image

On dispose de photographies d'une zone d'exploration effectuées à différentes longueur d'onde. Chaque photographie a été mises à l'échelle de la zone d'exploration puis stockée dans un tableau numpy (`np.ndarray`) à deux dimensions. Les dimensions correspondent aux coordonnées géographiques du point photographié, chaque élément est un entier, compris entre 0 et 255, donnant l'intensité du point considéré sur l'image. Ainsi l'élément d'indice `x,y` contient un entier, compris entre 0 et 255, correspondant à l'intensité sur la photographie considérée du point de coordonnées (x, y). À partir de ces photographies, les géologues déterminent les endroits à analyser en filtrant ceux qui ont un profil d'émission caractéristique de certaines roches intéressantes.

I.B.1) Analyse d'une image

La fonction `F1` ci-dessous prend en paramètre une photographie représentée comme décrit plus haut. Expliquer ce que fait cette fonction et décrire son résultat.

```
1  def F1(photo:np.ndarray) -> np.ndarray:
2      n = photo.min()
3      b = photo.max()
4      h = np.zeros(b - n + 1, np.int64)
5      for p in photo.flat:
6          h[p - n] += 1
7      return h
```

I.B.2) Sélection de points d'intérêts

Écrire une fonction d'entête

```
def sélectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
```

où `photo` est un tableau représentant une photographie. Le résultat de la fonction `sélectionner_PI` est un tableau à deux dimensions, de structure similaire à celui décrit figure 2, contenant les coordonnées des points dont l'intensité sur la photographie est comprise entre `imin` et `imax`.

I.C – Base de données

Afin d'assurer son autonomie opérationnelle, le robot dispose localement des informations nécessaires à son fonctionnement quotidien. Ainsi, il enregistre la durée d'utilisation de ses différents instruments embarqués. Il connait également les différents types d'analyses qu'il peut effectuer et, pour chacun de ces types, les instruments à utiliser. Il enregistre la prochaine exploration, c'est-à-dire les différents points d'intérêts qu'il doit visiter et pour chacun la ou les analyses qu'il doit effectuer. D'autre part, il conserve les résultats d'analyses effectuées lors de ses explorations passées. Ces résultats ne sont effacés qu'après confirmation de leur bonne transmission sur Terre.

Ces différentes informations sont stockées dans une base de données relationnelle dont le modèle physique est schématisé figure 3.

BY-NC-SA

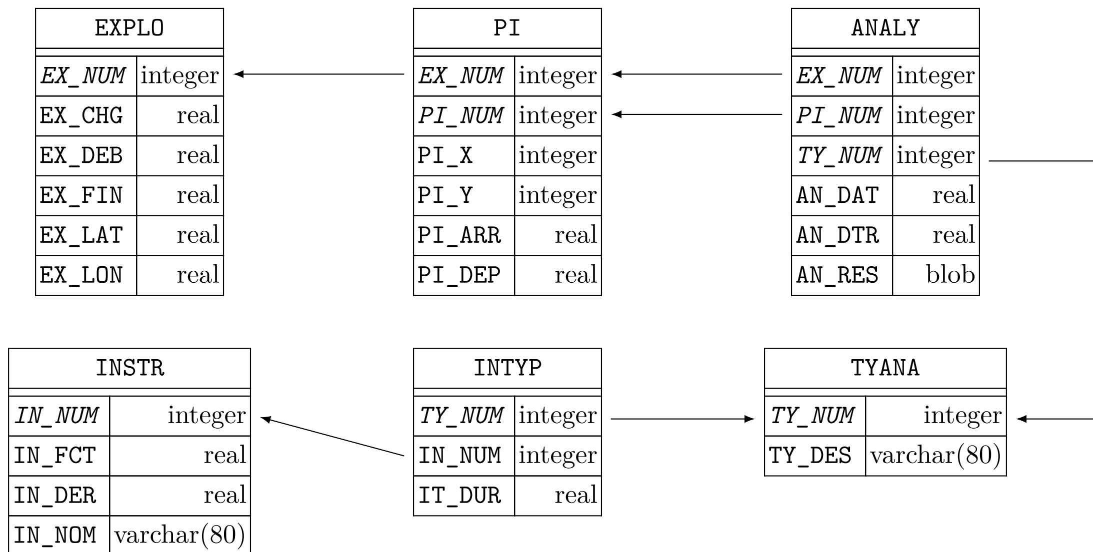

Figure 3 Structure physique de la base de données d'un robot

Cette base comporte les six tables suivantes :

— la table EXPLO des explorations, avec les colonnes
  - EX_NUM numéro (entier) de l'exploration (clef primaire)
  - EX_CHG date de transmission des points d'intérêts de cette exploration
  - EX_DEB date de début de l'exploration (NULL si l'exploration n'est pas encore commencée)
  - EX_FIN date de fin de l'exploration (NULL si l'exploration n'est pas encore terminée)
  - EX_LAT latitude (en degrés décimaux) du point de coordonnées $(0, 0)$ de la zone d'exploration
  - EX_LON longitude (en degrés décimaux) du point de coordonnées $(0, 0)$ de la zone d'exploration

— la table PI des points d'intérêts, de clef primaire (EX_NUM, PI_NUM), avec les colonnes
  - EX_NUM numéro de l'exploration à laquelle appartient le point d'intérêt
  - PI_NUM numéro du point d'intérêt dans l'exploration (au sein d'une exploration les PI sont numérotés en séquence en commençant à 0, ce numéro n'a pas de rapport avec l'ordre dans lequel les PI sont explorés par le robot)
  - PI_X l'abscisse du point d'intérêt dans la zone d'exploration (entier positif en millimètres)
  - PI_Y l'ordonnée du point d'intérêt dans la zone d'exploration (entier positif en millimètres)
  - PI_ARR date d'arrivée du robot au point d'intérêt (NULL si ce point n'a pas encore été visité)
  - PI_DEP date à laquelle le robot a quitté le point d'intérêt (NULL si ce point n'a pas encore été exploré ou si la visite est en cours)

— la table INSTR des instruments embarqués, avec les colonnes
  - IN_NUM le numéro (entier) de l'instrument (clef primaire)
  - IN_FCT la durée pendant lequel l'instrument a déjà été utilisé depuis l'arrivée sur la planète (nombre décimal : fraction de jour martien)
  - IN_DER la date de la dernière utilisation de l'instrument
  - IN_NOM nom de l'instrument

— la table TYANA des types d'analyses à effectuer, avec les colonnes
  - TY_NUM le numéro de référence (entier) du type d'analyse (clef primaire)
  - TY_DES le nom du type d'analyse

— la table INTYP des instruments utilisés pour un type d'analyse, de clef primaire (TY_NUM, IN_NUM), avec les colonnes
  - TY_NUM le numéro de référence (entier) du type d'analyse
  - IN_NUM le numéro (entier) de l'instrument
  - IT_DUR la durée standard d'utilisation de l'instrument dans ce type d'analyse (nombre décimal : fraction de jour martien)

— la table ANALY indiquant pour chaque point d'intérêt les types d'analyses à effectuer ou effectuées, de clef primaire (EX_NUM, PI_NUM, TY_NUM) et avec les colonnes
  - EX_NUM numéro de l'exploration à laquelle appartient le point d'intérêt
  - PI_NUM numéro du point d'intérêt dans l'exploration
  - TY_NUM type de l'analyse

BY-NC-SA

- `AN_DAT` date de l'analyse (`NULL` si l'analyse n'a pas été effectuée)
- `AN_DTR` date de transmission sur Terre des résultats de l'analyse (`NULL` si l'analyse n'a pas été transmise)
- `AN_RES` résultat de l'analyse (donnée opaque dont la signification dépend du type d'analyse)

Toutes les dates sont stockées sous forme d'un nombre décimal correspondant au nombre de jours martiens depuis l'arrivée du robot sur la planète.

I.C.1) Écrire une requête SQL qui donne le numéro de l'exploration en cours, s'il y en a une.

I.C.2) Écrire une requête SQL qui donne, pour une exploration dont on connait le numéro, la liste des points d'intérêts de cette exploration avec leurs coordonnées.

I.C.3) Écrire une requête SQL qui donne la surface, en mètres carrés, de chaque zone déjà explorée par le robot. La zone d'exploration est définie comme le plus petit rectangle qui englobe l'ensemble des points d'intérêts de l'exploration et dont les bords sont parallèles aux axes de référence (axes des abscisse et des ordonnées).

I.C.4) Quelle est la surface maximale d'une zone d'exploration que peut stocker cette base de données ?

I.C.5) Écrire une requête SQL qui donne, pour l'exploration en cours, le nombre de fois où chaque instrument doit être utilisé et sa durée d'utilisation théorique (en jours martiens) pour la totalité de l'exploration.

II Planification d'une exploration : première approche

Avant de démarrer une nouvelle exploration, le robot doit déterminer un chemin qui lui permet de passer par tous les points d'intérêts une et une seule fois. L'enjeu de l'opération est de trouver le chemin le plus court possible afin de limiter la dépense d'énergie et de limiter l'usure du robot.

Chaque point d'intérêt sera repéré par un entier positif ou nul correspondant à son indice dans le tableau des points d'intérêt. Un chemin d'exploration sera représenté par un objet de type `list` donnant les indices des points d'intérêt dans l'ordre de leur parcours.

II.A – Quelques fonctions utilitaires

II.A.1) Longueur d'un chemin

Écrire une fonction d'entête

```
def longueur_chemin(chemin:list, d:np.ndarray) -> float:
```

qui prend en paramètre un chemin à parcourir et la matrice des distances entre points d'intérêt (telle que renvoyée par la fonction `calculer_distances`) et renvoie la distance que doit effectuer le robot pour suivre ce chemin en partant de sa position courante (correspondant à la dernière ligne/colonne du tableau `d`) et en visitant tous les points d'intérêt dans l'ordre indiqué.

II.A.2) Normalisation d'un chemin

Écrire une fonction d'entête

```
def normaliser_chemin(chemin:list, n:int) -> list:
```

qui prend en paramètre une liste d'entiers et renvoie une liste correspondant à un chemin valide, c'est-à-dire contenant une seule fois tous les entiers entre 0 et `n` (exclu). Pour cela cette fonction commence par supprimer les éventuels doublons (en ne conservant que la première occurrence) et les valeurs supérieures ou égales à `n`, sans modifier l'ordre relatif des éléments conservés, puis ajoute à la fin les éventuels éléments manquants en ordre croissant de numéros.

II.B – Force brute

Pour rechercher le plus court chemin, on peut imaginer de considérer tous les chemins possibles et de calculer leur longueur. On obtiendra ainsi à coup sûr le chemin le plus court.

II.B.1) Déterminer en fonction de n, nombre de points à visiter, le nombre de chemins possibles passant exactement une fois par chacun des points.

II.B.2) Cet algorithme est-il utilisable pour une zone d'exploration contenant 20 points d'intérêts ? Justifier.

II.C – Algorithme du plus proche voisin

Une idée simple pour obtenir un algorithme utilisable est de construire un chemin en choisissant systématiquement le point, non encore visité, le plus proche de la position courante.

II.C.1) Écrire une fonction d'entête

```
def plus_proche_voisin(d:np.ndarray) -> list:
```

qui prend en paramètre le tableau des distances résultat de la fonction `calculer_distances` (question I.A.2) et fournit un chemin d'exploration en appliquant l'algorithme du plus proche voisin.

 BY-NC-SA

II.C.2) Quelle est la complexité temporelle de l'algorithme du plus proche voisin en considérant que cet algorithme est constitué des deux fonctions `calculer_distances` et `plus_proche_voisin` ?

II.C.3) En considérant les trois points de coordonnées $(0,0)$, $(0,3000)$, $(0,7000)$ et en choisissant un point de départ adéquat pour le robot, montrer que l'algorithme du plus proche voisin ne fournit pas nécessairement le plus court chemin.

Dans la pratique, on constate que, dès que le nombre de points d'intérêt devient important, l'algorithme du plus proche voisin fournit un chemin qui peut être 50% plus long que le plus court chemin.

III Deuxième approche : algorithme génétique

Les algorithmes génétiques s'inspirent de la théorie de l'évolution en simulant l'évolution d'une population. Ils font intervenir cinq traitements.

1. Initialisation
   Il s'agit de créer une population d'origine composée de m individus (ici des chemins pour l'exploration à planifier). Généralement la population de départ est produite aléatoirement.
2. Évaluation
   Cette étape consiste à attribuer à chaque individu de la population courante une note correspondant à sa capacité à répondre au problème posé. Ici la note sera simplement la longueur du chemin.
3. Sélection
   Une fois tous les individus évalués, l'algorithme ne conserve que les « meilleurs » individus. Plusieurs méthodes de sélection sont possibles : choix aléatoire, ceux qui ont obtenu la meilleure note, élimination par tournoi, etc.
4. Croisement
   Les individus sélectionnés sont croisés deux à deux pour produire de nouveaux individus et donc une nouvelle population. La fonction de croisement (ou reproduction) dépend de la nature des individus.
5. Mutation
   Une proportion d'individus est choisie (généralement aléatoirement) pour subir une mutation, c'est-à-dire une transformation aléatoire. Cette étape permet d'éviter à l'algorithme de rester bloqué sur un optimum local.

En répétant les étapes de sélection, croisement et mutation, l'algorithme fait ainsi évoluer la population, jusqu'à trouver un individu qui réponde au problème initial. Cependant dans les cas pratiques d'utilisation des algorithmes génétiques, il n'est pas possible de savoir simplement si le problème est résolu (le plus court chemin figure-t-il dans ma population ?). On utilise donc des conditions d'arrêt heuristiques basées sur un critère arbitraire.

Le but de cette partie est de construire un algorithme génétique pour rechercher un meilleur chemin d'exploration que celui obtenu par l'algorithme du plus proche voisin.

III.A – Initialisation et évaluation

Une population est représentée par une liste d'individus, chaque individu étant représenté par un couple *(longueur, chemin)* dans lequel

- *chemin* désigne un chemin représenté comme précédemment par une liste d'entiers correspondant aux indices des points d'intérêt dans le tableau des distances produit par la fonction `calculer_distances` ;
- *longueur* est un entier correspondant à la longueur du chemin, en tenant compte de la position de départ du robot.

Écrire une fonction d'entête

```
def créer_population(m:int, d:np.ndarray) -> list:
```

qui crée une population de m individus aléatoires. Cette fonction prend en paramètre le nombre d'individus à engendrer et le tableau des distances entre points d'intérêt (et la position courante du robot) tel que produit par la fonction `calculer_distances`. Elle renvoie une liste d'individus, c'est-à-dire de couples *(longueur, chemin)*.

III.B – Sélection

Écrire une fonction d'entête

```
def réduire(p:list) -> None:
```

qui réduit une population de moitié en ne conservant que les individus correspondant aux chemins les plus courts. On rappelle que `p` est une liste de couples *(longueur, chemin)*. La fonction `réduire` ne renvoie pas de résultat mais modifie la liste passée en paramètre.

 BY-NC-SA

III.C – Mutation

III.C.1) Écrire une fonction d'entête

```
def muter_chemin(c:list) -> None:
```

qui prend en paramètre un chemin et le transforme en inversant aléatoirement deux de ses éléments.

III.C.2) Écrire une fonction d'entête

```
def muter_population(p:list, proba:float, d:np.ndarray) -> None:
```

qui prend en paramètre une population dont elle fait muter un certain nombre d'individus. Le paramètre `proba` (compris entre 0 et 1) désigne la probabilité de mutation d'un individu. Le paramètre `d` est la matrice des distances entre points d'intérêt.

III.D – Croisement

III.D.1) Écrire une fonction d'entête

```
def croiser(c1:list, c2:list) -> list:
```

qui crée un nouveau chemin à partir de deux chemins passés en paramètre. Ce nouveau chemin sera produit en prenant la première moitié du premier chemin suivi de la deuxième moitié du deuxième puis en « normalisant » le chemin ainsi obtenu.

III.D.2) Écrire une fonction d'entête

```
def nouvelle_génération(p:list, d:np.ndarray) -> None:
```

qui fait grossir une population en croisant ses membres pour en doubler l'effectif. Pour cela, la fonction fait se reproduire tous les couples d'individus qui se suivent dans la population `(p[i], p[i+1])` et `(p[m-1], p[0])` de façon à produire m nouveaux individus qui s'ajoutent aux m individus de la population de départ.

III.E – Algorithme complet

III.E.1) Écrire une fonction d'entête

```
def algo_génétique(PI:np.ndarray, m:int, proba:float, g:int) -> float, list:
```

qui prend en paramètre un tableau de points d'intérêts (figure 2), la taille `m` de la population, la probabilité de mutation `proba` et le nombre de générations `g`. Cette fonction implante un algorithme génétique à l'aide des différentes fonctions écrites jusqu'à présent et renvoie la longueur du plus court chemin d'exploration et le chemin lui-même obtenus au bout de `g` générations.

III.E.2) Est-il possible avec l'implantation réalisée, qu'une itération de l'algorithme dégrade le résultat : le meilleur chemin obtenu à la génération $n+1$ est plus long que celui de la génération n ?

Dans l'affirmative, comment modifier le programme pour que cette situation ne puisse plus arriver ?

III.E.3) Quelles autres conditions d'arrêt peut-on imaginer ? Établir un comparatif présentant les avantages et inconvénients de chaque condition d'arrêt envisagée.

Opérations et fonctions Python disponibles

Fonctions

- `range(n)` renvoie la séquence des `n` premiers entiers ($0 \to n-1$)
- `list(range(n))` renvoie une liste contenant les `n` premiers entiers dans l'ordre croissant :
  `list(range(5))` → `[0, 1, 2, 3, 4]`
- `random.randrange(a, b)` renvoie un entier aléatoire compris entre `a` et `b-1` inclus (`a` et `b` entiers)
- `random.random()` renvoie un nombre flottant tiré aléatoirement dans $[0,1[$ suivant une distribution uniforme
- `random.shuffle(u)` permute aléatoirement les éléments de la liste `u` (modifie `u`)
- `random.sample(u, n)` renvoie une liste de `n` éléments distincts de la liste `u` choisis aléatoirement, si `n > len(u)`, déclenche l'exception `ValueError`
- `math.sqrt(x)` calcule la racine carrée du nombre x
- `math.ceil(x)` renvoie le plus petit entier supérieur ou égal à `x`

BY-NC-SA

— `math.floor(x)` renvoie le plus grand entier inférieur ou égal à `x`
— `sorted(u)` renvoie une nouvelle liste contenant les éléments de la liste `u` triés dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes ou des tuples, l'ordre utilisé est l'ordre lexicographique)

Opérations sur les listes

— `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` → 3 ; `len([[1,2], [3,4]])` → 2
— `u + v` construit une liste constituée de la concaténation des listes `u` et `v` :
  `[1, 2] + [3, 4, 5]` → `[1, 2, 3, 4, 5]`
— `n * u` construit une liste constitué de la liste `u` concaténée `n` fois avec elle-même :
  `3 * [1, 2]` → `[1, 2, 1, 2, 1, 2]`
— `e in u` et `e not in u` déterminent si l'objet `e` figure dans la liste `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
  `2 in [1, 2, 3]` → `True` ; `2 not in [1, 2, 3]` → `False`
— `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (similaire à `u = u + [e]`)
— `del u[i]` supprime de la liste `u` son élément d'indice `i`
— `del u[i:j]` supprime de la liste `u` tous ses éléments dont les indices sont compris dans l'intervalle [`i`, `j`[
— `u.remove(e)` supprime de la liste `u` le premier élément qui a pour valeur `e`, déclenche l'exception `ValueError` si `e` ne figure pas dans `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$
— `u.insert(i, e)` insère l'élément `e` à la position d'indice `i` dans la liste `u` (en décalant les éléments suivants) ; si `i >= len(u)`, `e` est ajouté en fin de liste
— `u[i], u[j] = u[j], u[i]` permute les éléments d'indice `i` et `j` dans la liste `u`
— `u.sort()` trie la liste `u` en place, dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes ou des tuples, l'ordre utilisé est l'ordre lexicographique)

Opérations sur les tableaux (`np.ndarray`)

— `np.array(u)` crée un nouveau tableau contenant les éléments de la liste `u`. La taille et le type des éléments de ce tableau sont déduits du contenu de `u`
— `np.empty(n, dtype)`, `np.empty((n, m), dtype)` crée respectivement un vecteur à `n` éléments ou une matrice à `n` lignes et `m` colonnes dont les éléments, de valeurs indéterminées, sont de type `dtype` qui peut être un type standard (`bool`, `int`, `float`, ...) ou un type spécifique numpy (`np.int16`, `np.float32`, ...). Si le paramètres `dtype` n'est pas précisé, il prend la valeur `float` par défaut
— `np.zeros(n, dtype)`, `np.zeros((n, m), dtype)` fonctionne comme `np.empty` en initialisant chaque élément à la valeur zéro pour les types numériques ou `False` pour les types booléens
— `a.ndim` nombre de dimensions du tableau `a` (1 pour un vecteur, 2 pour une matrice, etc.)
— `a.shape` tuple donnant la taille du tableau `a` pour chacune de ses dimensions
— `len(a)` taille du tableau `a` dans sa première dimension (nombre d'éléments d'un vecteur, nombre de lignes d'une matrice, etc.) équivalent à `a.shape[0]`
— `a.size` nombre total d'éléments du tableau `a`
— `a.flat` itérateur sur tous les éléments du tableau `a`
— `a.min()`, `a.max()` renvoie la valeur du plus petit (respectivement plus grand) élément du tableau `a` ; ces opérations ont une complexité temporelle en $O($`a.size`$)$
— `b in a` détermine si `b` est un élément du tableau `a` ; si `b` est un scalaire, vérifie si `b` est un élément de `a` ; si `b` est un vecteur ou une liste et `a` une matrice, détermine si `b` est une ligne de `a`
— `np.concatenate((a1, a2))` construit un nouveau tableau en concaténant deux tableaux ; `a1` et `a2` doivent avoir le même nombre de dimensions et la même taille à l'exception de leur taille dans la première dimension (deux matrices doivent avoir le même nombre de colonnes pour pouvoir être concaténées)

• • • FIN • • •

BY-NC-SA