CCP Informatique PSI 2017 — Corrigé

Ce corrigé est proposé par Jean-Julien Fleck (professeur en CPGE) ; il a été relu par Josselin Giet (ENS Ulm) et Julien Dumont (professeur en CPGE).

Ce sujet bien articulé et progressif étudie la circulation automobile et l'évolution de bouchons aux heures de pointe sur l'autoroute, depuis l'étude expérimentale via le suivi de la circulation jusqu'aux simulations numériques en utilisant soit la mécanique des fluides, soit des automates cellulaires.

- La première partie se concentre sur les aspects « expérimentaux » en interrogeant une base de données (questions 1 et 2) qui stocke les informations autoroutières avant de les représenter graphiquement (question 3) et de les traiter automatiquement pour détecter un ralentissement via un algorithme de tri par insertion destiné à calculer la médiane des vitesses mesurées (questions 4 et 5).
- Vient alors une première proposition de simulation (questions 6 à 15) qui traite le trafic routier comme un flux hydrodynamique, ce qui revient à résoudre une équation aux dérivées partielles par différentes méthodes afin d'analyser leurs points forts ainsi que leurs motifs de divergence.
- Une seconde simulation à base d'automates cellulaires (questions 16 à 19) clôt le sujet en se rapprochant légèrement du sujet d'informatique des Mines de la même année, sans pour autant faire doublon avec lui. L'avantage est que l'on peut très rapidement comprendre comment créer ses propres bouchons à partir de coups de freins intempestifs de conducteurs peu attentifs.

Le sujet balaie une grande partie du programme de première année avec une brève incursion en deuxième année concernant l'algorithme de tri présenté en question 4. Il permet de se familiariser avec une structure de données matricielle qui stocke les évolutions temporelle et spatiale des objets considérés en une seule entité, ce qui peut certainement être utilisé dans un TIPE lors de simulations similaires. D'ailleurs, l'ensemble des codes permettant de reproduire certaines des figures de ce corrigé sont disponibles à l'adresse

`https://github.com/jjfPCSI1/codes_concours_CPGE`

Pour ceux qui voudraient aller plus avant dans la partie physique, le sujet PSI Centrale 2005, disponible sur `Doc-Solus.fr`, traite assez complètement de la modélisation physique abordée par ce sujet.

Indications

Partie II

1 La difficulté est dans la jointure.

2 Penser à l'instruction `GROUP BY`.

3 L'usage de Numpy trivialise quelque peu la fonction, mais on peut tout à fait faire explicitement les boucles.

4 Se demander comment on fait pour trier « naturellement » ses cartes.

5 Que se passe-t-il pendant au moins la moitié du temps ?

Partie III

7 Là encore, Numpy simplifie les choses.

8 La dépendance temporelle n'influe que sur l'intervalle de concentrations explorées.

9 Penser à jeter un œil en question 11 pour savoir les arguments a priori attendus par le concepteur du sujet.

10 Il s'agit, comme d'habitude en physique, d'approximer les dérivées par leurs taux d'accroissement, tantôt selon l'axe temporel, tantôt selon l'axe spatial, selon la dérivée considérée.

11 Pour éviter de traîner de multiples vérifications à droite des listes, penser à utiliser l'opération modulo (via l'opérateur `%`) pour mettre en place les conditions aux limites périodiques.

13 Le schéma d'intégration est adapté pour un déplacement dans le sens des flèches des représentations de la question 12.

14 L'introduction d'une fonction annexe pour la dérivation peut permettre de simplifier le code.

Partie IV

16 Se rappeler qu'on désire modéliser la réalité et que, dans celle-ci, une voiture a une certaine longueur. Pour le calcul de la vitesse, ne pas oublier les changements d'unités adéquats.

17 Traduire les trois premières étapes de l'algorithme proposé à chaque fois que l'on croise une voiture dans une case.

18 Ne pas oublier qu'il ne faut prédire un déplacement que si une voiture est effectivement présente dans la case considérée.

19 Les bouchons correspondent aux agrégats noirs. Pour améliorer la simulation, penser à tous les paramètres des autoroutes réelles que l'on a été amené à simplifier dans cette étude.

II. Traitement des données expérimentales

1 La requête SQL demandée nécessite de faire une jointure avec la table des stations pour trouver la station voulue.

```
SELECT id_comptage,date,voie,q_exp,v_exp
    FROM COMPTAGES JOIN STATIONS
    ON STATIONS.id_station = COMPTAGES.id_station
    WHERE nom = 'M8B'
```

Notons que la table de provenance de l'attribut `id_station` doit être précisée lors de la jointure puisqu'il porte le même nom dans les deux tables. En revanche, il n'y a pas d'ambiguïté concernant les autres attributs, il n'est donc pas nécessaire de les préciser.

2 La requête demandée nécessite maintenant de rassembler (à l'aide de l'instruction `GROUP BY`) les mesures faites à un même instant sur toutes les voies pour sommer les voitures qui y passent.

```
SELECT date,SUM(q_exp) FROM COMPTAGES_M8B GROUP BY date
```

3 Il s'agit là de calculer le vecteur des concentrations correspondant aux différentes données en effectuant le rapport, à chaque fois, du débit par la vitesse. On peut le faire de manière naturelle en itérant sur les différentes composantes des vecteurs avant de faire le graphique.

> On va ici procéder comme le suggère l'énoncé, c'est-à-dire importer `numpy` et `matplotlib.pyplot` « à la sauvage » par les commandes `from numpy import *` et `from matplotlib.pyplot import *`, mais il s'agit a priori d'un mauvais réflexe en pratique puisque cela pollue l'espace des noms et peut mener à des conflits si les deux modules définissent une fonction de même nom. Cela peut néanmoins se comprendre pour une épreuve écrite où le nombre de fonctions disponibles est réduit et cela permet donc d'appeler `zeros` ou `plot` sans préciser les habituels `np.zeros` ou `plt.plot`.
>
> En outre, on utilise par la suite une autre forme de `zeros` que celle rappelée en annexe, à savoir `zeros(n)` pour créer un vecteur contenant `n` zéros. Le rappel de l'annexe concerne la création de tableaux multidimensionnels pour lesquels il faut spécifier la taille de chaque dimension et qui servirait si on avait à initialiser le tableau `C` des concentrations, utilisé à partir de la question 9, mais l'énoncé ne demande finalement jamais de le faire.

```
from numpy import *                 # Imports présupposés par le sujet
from matplotlib.pyplot import *     # (mais mauvaise bonne idée en pratique)

def trace(q_exp,v_exp):
    n = len(v_exp)          # Taille des vecteurs utilisés
    c_exp = zeros(n)        # Initialisation du tableau des concentrations
    for i in range(n):      # pour un calcul séquentiel.
        c_exp[i] = q_exp[i] / v_exp[i]
    plot(c_exp,q_exp,'o') # Tracé du graphe (Gare à l'ordre des arguments !)
```

Néanmoins, comme l'énoncé signale que les arguments sont des tableaux Numpy, on peut aussi calculer `c_exp` « au vol » en utilisant les facilités de Numpy, qui s'occupe tout seul de la boucle sous-jacente sur toutes les composantes des deux vecteurs. Ainsi, l'écriture de la fonction s'en trouve grandement simplifiée :

```
def trace(q_exp,v_exp):
    c_exp = q_exp / v_exp    # Divisions composante par composante
    plot(c_exp,q_exp,'o')    # Tracé du graphe
```

> Logiquement, une telle solution devrait être acceptée sans sourciller par le correcteur, mais il peut être bon de rappeler dans sa copie que l'on sait que Numpy s'occupe des boucles associées « derrière le rideau ».

4 Il s'agit ici d'un algorithme de **tri par insertion**. On prend un à un chaque élément (de gauche à droite) et on le laisse « couler » dans la partie gauche de la liste (qui est celle qui est triée à ce stade), jusqu'à ce qu'il trouve sa place, c'est-à-dire jusqu'à ce qu'il ne soit plus le plus petit considéré jusqu'ici dans les comparaisons. Pour qu'il fonctionne, il faut « faire de la place », donc décaler chaque élément (depuis la position `j-1`) d'un cran vers la droite (donc vers la position `j` en utilisant l'instruction `v_exp[j] = v_exp[j-1]`) tout en continuant de déplacer l'élément `v` vers la gauche via `j = j-1`. **C'est donc la deuxième proposition qui est la bonne** et la fonction s'écrit totalement :

```
def congestion(v_exp):
    nbmesures = len(v_exp)
    for i in range(nbmesures):       # Pour chaque point de mesure,
        v = v_exp[i]                 # on note la valeur à classer
        j = i                        # et d'où l'on part.
        # Tant qu'on n'arrive pas tout à gauche ou qu'on ne trouve
        while 0<j and v<v_exp[j-1]:# pas la place de v,
            v_exp[j] = v_exp[j-1]    # on pousse à droite
            j = j-1                  # et on regarde plus à gauche.
        v_exp[j] = v                 # On place finalement v au bon endroit.
    return v_exp[nbmesures//2]       # Renvoi de l'élément au milieu après tri.
```

Dans le meilleur des cas (liste triée), la complexité est linéaire : la boucle `while` ne s'exécute jamais puisqu'on n'itère qu'une seule fois sur tous les éléments. Dans le pire des cas (liste décroissante), chaque boucle `while` s'exécute i fois au i^{e} tour de la boucle `for`, donc $n(n-1)/2$ fois au total, ce qui mène à une complexité quadratique.

Malgré cette dernière éventualité, le choix d'un tri par insertion reste pertinent puisqu'on peut penser que l'on ne veuille guère prendre une médiane sur plus d'une heure de temps (on s'intéresse souvent à l'état quasi-instantané du trafic) donc pour seulement une dizaine de mesures (deux mesures étant séparées de 6 minutes environ), ce qui n'est pas critique en terme de complexité.

5 Une valeur renvoyée de 30 signifie que la circulation à cet endroit était congestionnée puisque la vitesse était inférieure ou égale à 30 km/h pendant la moitié du temps d'observation et que l'énoncé signale que la circulation est considérée congestionnée en dessous d'une vitesse de 40 km/h.

> Il est assez clair que 30 km/h n'est pas tout à fait la vitesse à laquelle on espère progresser sur une autoroute.