Informatique

CCP PSI 2017: Corrigé

Q1 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 = COMPTAGE.id_station
    WHERE nom = 'M8B'
```

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

Q2 La requête demandée nécessite maintenant de rassembler (via l'instruction `GROUP BY`) les mesures faites à un même instant sur toute les voies pour sommer les voitures qui y passent.

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

Q3 Il s'agit là de calculer le vecteur des concentrations correspondant aux différentes données par 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 (en supposant les modules `numpy` et `matplotlib.pyplot` déjà importés comme indiqué plus bas[1])

```
1  from numpy import *                # Imports présupposés par le sujet
2  from matplotlib.pyplot import *   # (mais mauvaise bonne idée en pratique)
3
4  def trace(q_exp,v_exp):
5      n = len(v_exp)           # Taille des vecteurs utilisés
6      c_exp = zeros(n)         # Initialisation du tableau des concentrations
7      for i in range(n):       # pour un calcul séquentiel.
8          c_exp[i] = q_exp[i] / v_exp[i]
9      plot(c_exp,q_exp,'o')    # Tracé du graphe (attention 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:

[1] Ce qui est 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, mais qui peut se comprendre pour une épreuve écrite où le nombre de fonctions disponibles est réduit.

```python
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 automatiquement des boucles associées « derrière le rideau ».

Q4 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 » et donc décaler chaque élément (depuis la position `j-1`) d'un cran vers la droite (donc vers la position `j` soit en utilisant `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

```python
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.
        while 0 < j and v < v_exp[j-1]:# Tant qu'on n'arrive pas tout à gauche
                                         # ou qu'on ne trouve 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]           # et on renvoie l'élément 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 complexité quadratique, le choix 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.

Q5 Une valeur renvoyée de 30 signifie que durant au moins la moitié du temps d'observation, la circulation à cet endroit était congestionnée.

Q6 Il y a un total de $n = \lfloor \mathrm{T}emps/\mathrm{d}t \rfloor$ pas de temps et $p = \lfloor \mathrm{L}a/\mathrm{d}x \rfloor$ pas d'espace, soit un tableau C à deux dimensions de tailles (n, p).

Q7 Il s'agit comme à la question 3 de construire un tableau à partir d'un autre (en utilisant l'expression donnée de $v(t,x)$ et en en déduisant $q(t,x) = c(t,x) \times v(t,x)$), ce qu'on peut faire en faisant explicitement apparaître la boucle d'affection:

```python
def debit(v_max,c_max,C_ligne):
    n = len(C_lignes)
    q = zeros(n)                           # Initialisation du tableau
    for i in range(n):
        v = v_max * (1 - C_lignes[i]/c_max) # On calcule d'abord la vitesse
        q[i] = C_lignes[i] * v             # puis le débit associé.
```

```
7       return q
```

ou alors en utilisant les facilités de Numpy qui trivialisent quelque peu la question:

```
1   def debit(v_max,c_max,C_ligne):
2       v = v_max * (1 - C_ligne/c_max)
3       return C_ligne * v
```

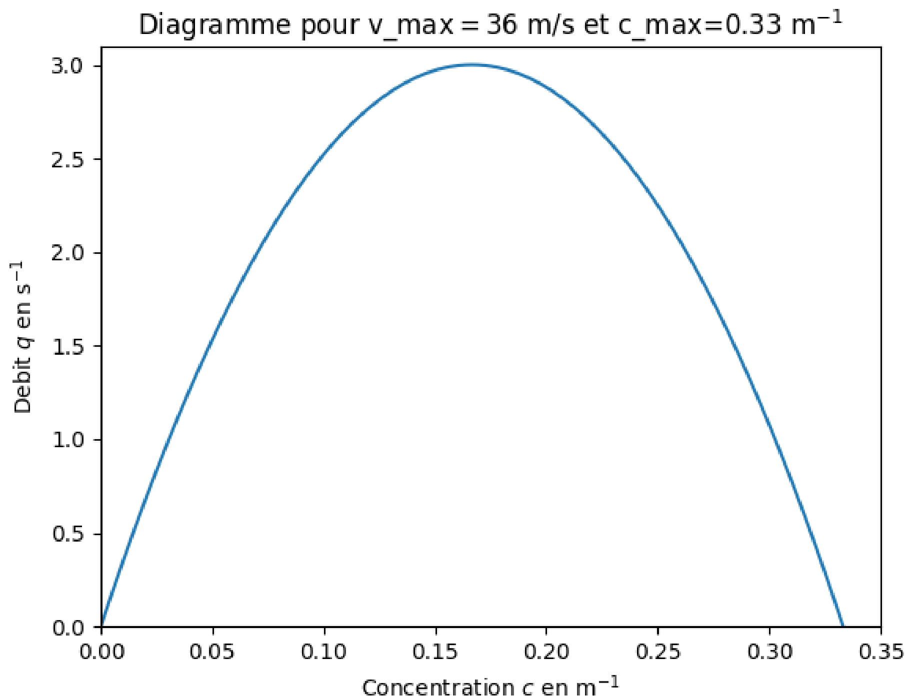
Q8 La fonction `diagramme` ne nécessite la connaissance que des mêmes arguments de la fonction `debit`, à savoir `v_max` (un flottant exprimé en m.s^{-1}), `c_max` (un flottant exprimé en m.$^{-1}$) et `C_ligne` (un vecteur aussi en m^{-1}). Le code n'est pas demandé mais pourrait ressembler à cela

```
1   def diagramme(v_max,c_max,C_ligne):
2       Q = debit(v_max,c_max,C_ligne)
3       plot(C_ligne,Q)
```

Et l'allure obtenue correspond à une parabole puisque

$$q = c \times v = c \times \left(1 - \frac{c}{c_{\max}}\right) v_{\max}$$

qui est bien une parabole en c s'annulant en $c = 0$ et $c = c_{\max}$



L'allure du diagramme ne dépend pas explicitement du temps, seul son échantillonnage en dépendra selon l'intervalle de valeurs de concentrations qui sera exploré par `C_ligne`. Nous avons pris un intervalle $[\,0\,;c_{\max}\,]$ pour la représentation précédente, mais la fonction ne sera échantillonnée que sur l'intervalle [`min(C_ligne)` ; `max(C_ligne)`] qui lui dépendra a priori du temps.

Q9 La fonction demandée doit valoir `c1` pour les indices appartenant à l'intervalle ⟦ 0 ; `d1//dx` ⟧ puis `c2` sur l'intervalle ⟦ `d1//dx + 1` ; `d2//dx` ⟧ et enfin à nouveau $c1$ sur l'intervalle ⟦ `d2//dx + 1` ; `La//dx` ⟧. L'écriture de la fonction n'est pas demandée, mais on donne la spécification demandée dans la docString[2]. À noter que la question 11 permet de connaître les arguments attendus par le concepteur du sujet.

[2] La fonction est donnée en bonus sous forme condensée en utilisant les facilités de Numpy.

```python
def C_depart(dx,d1,d2,c1,c2,C):
    """ Renvoie le profil de concentration initiale sous forme d'un plateau
    rectangulaire de concentration c2 sur ]d1;d2] et de concentration c1
    ailleurs. Les arguments sont
    * dx (float): pas d'espace
    * d1 (float): début du plateau
    * d2 (float): fin du plateau
    * c1 (float): concentration du plancher
    * c2 (float): concentration du plateau
    * C  (array): tableau bidimensionnel de largeur supérieure à d2//dx
    dont on veut initialiser la première ligne
    La fonction ne renvoie rien mais modifie au vol la tableau
    """
    n1,n2 = int(d1//dx), int(d2//dx)
    C[0,:] = c1           # On initialise tout à c1
    C[0,n1+1:n2+1] = c2 # Et on met à c2 les points de l'intervalle [[n1+1;n2]]
```

Q10 Dans l'équation (**1**), on peut remplacer

$$\frac{\partial q}{\partial x} = \frac{Q_{j+1} - Q_j}{\mathrm{d}x} \qquad \text{et} \qquad \frac{\partial c}{\partial t} = \frac{C_{i+1,j} - C_{i,j}}{\mathrm{d}t}$$

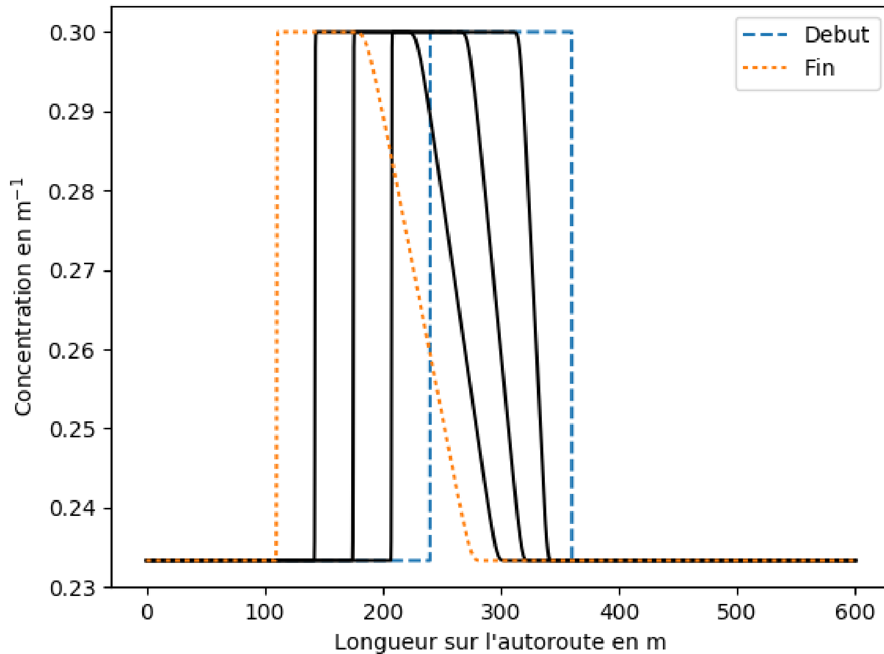
Ainsi,
$$\frac{Q_{j+1} - Q_j}{\mathrm{d}x} + \frac{C_{i+1,j} - C_{i,j}}{\mathrm{d}t} = 0 \qquad \text{soit} \qquad \boxed{C_{i+1,j} = C_{i,j} - \frac{Q_{j+1} - Q_j}{\mathrm{d}x}\,\mathrm{d}t}$$

ce qui correspond à la proposition numéro ② de l'énoncé.

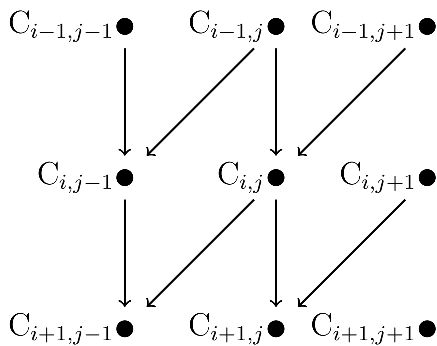
Q11 Procédons à la résolution progressive du problème posé. Notez l'utilisation de `(j+1)%n` pour éviter les `IndexError: list index out of range` lorsqu'on arrive tout à droite de l'autoroute.

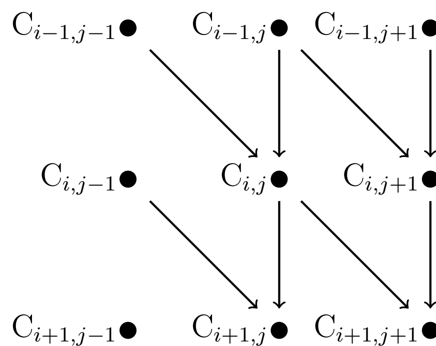
```python
def resolution(C,dt,dx,c_max,v_max):
    p,n = C.shape # nombre de pas de temps (p) et d'espace (n)
    for i in range(0,p-1):     # On regarde instant après instant
        # On commence par calculer le débit à l'instant i
        Q = debit(v_max,c_max,C[i])
        # On l'utilise alors pour calculer la concentration au temps suivant
        for j in range(n):     # en itérant sur tout l'espace
            Qjp1 = Q[(j+1)%n] # Conditions aux limites périodiques
            # Application de la formule
            C[i+1,j] = C[i,j] - (Qjp1 - Q[j]) / dx * dt
    return C # L'énoncé demande le renvoi (mais C est modifié anyway)
```

L'application du schéma précédent permet bien de retrouver les résultats annoncés par l'énoncé dans le cadre d'une forte concentration (recul progressif du bouchon)



Q12



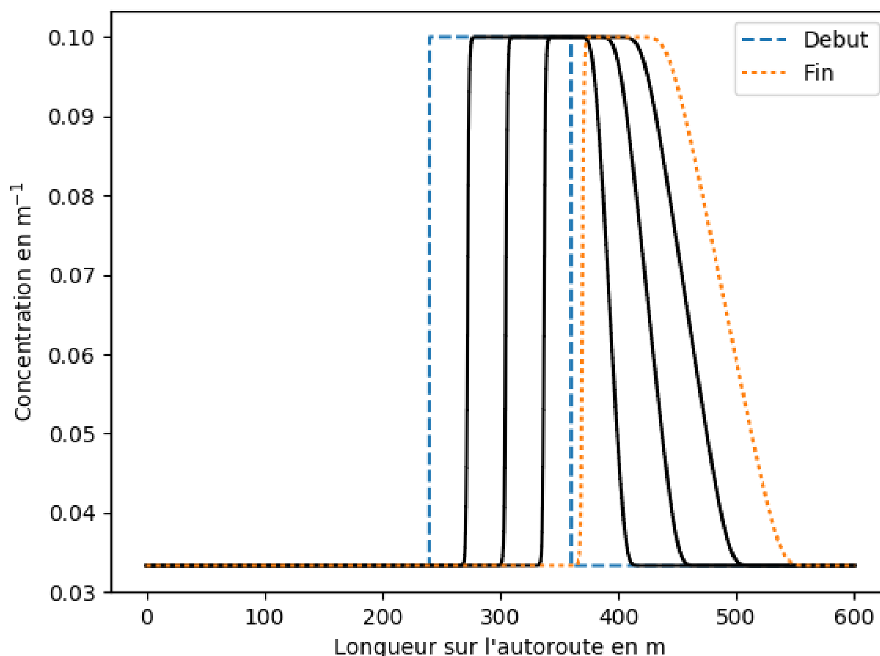


Q13 Le schéma d'Euler « avant » est adapté lorsque l'onde de densité représentant le bouchon se déplace vers l'arrière de l'autoroute alors que le schéma « arrière » est adapté lorsqu'elle se déplace vers l'avant.

Q14 Repartons du code proposé à la question 11 et modifions-le pour l'adapter aux schémas demandés. On va introduire une fonction annexe qui s'occupe de dériver proprement avec le schéma centré proposé tout en imposant les conditions aux limites périodiques.

```
1  def derive(f,dx,j):
2      n = len(f)
3      return (f[(j+1)%n] - f[(j-1)%n]) / (2*dx)
4
5  def resolution(C,dt,dx,c_max,v_max):
6      p,n = C.shape # nombre de pas de temps (p) et d'espace (n)
7      for i in range(0,p-1): # On regarde instant après instant
8          # On commence par calculer le débit à l'instant i
9          Q = debit(v_max,c_max,C[i])
10         # On l'utilise alors pour calculer la concentration au temps suivant
11         for j in range(n):     # en itérant sur tout l'espace
12             # Evolution de la concentration
```

```
13                Cij_moyen = (C[i,(j+1)%n]+C[i,(j-1)%n])/2
14                C[i+1,j] = Cij_moyen - derive(Q,dx,j) * dt
15        return C # L'énoncé demande le renvoi (mais C est modifié anyway)
```



Q15 Il suffit de changer le mode de calcul du tableau Q du code précédent (ligne 9) pour l'adapter à la regression expérimentale. Par exemple sous la forme

```
            a0,a1,a2,a3 = regression(q_exp,c_exp)
            Q = a3*C[i]**3 + a2*C[i]**2 + a1*C[i] + a0
```

Q16 La taille des cellules doit être supérieure à la taille des véhicules que l'on veut modéliser (un véhicule ne peut pas déborder sur plusieurs cellules), donc avec une valeur d'au moins 3 ou 4 mètres.

La distance parcourue à la vitesse maximale pendant un intervalle de temps dt vaut $v_{\max}\,\mathrm{d}t$, ce qui correspond à

$$\boxed{v_{\text{cell}} = \left\lceil \frac{v_{\max}\,\mathrm{d}t}{\mathrm{d}x} \right\rceil = \left\lceil \frac{(130/3{,}6)\times 1{,}2}{7{,}5} \right\rceil = 6 \text{ cellules/s}}$$

Q17 Il s'agit de traduire l'algorithme proposé en code.

```
1   def maj(Route,Vitesses,p,v_max,i):
2       Vitesses_suivantes = array(Vitesses[i]) # Copie de l'état actuel
3       N = len(Vitesses_suivantes)
4       for j in range(N):
5           if Route[i][j] == 1: # Si il y a une voiture dans la case
6               # Étape 1: Accélération
7               if Vitesses_suivantes[j] < v_max:
8                   Vitesses_suivantes[j] = Vitesses_suivantes[j] + 1
9               # Étape 2: Décélération
10              dn = distance(Route,i,j)
```

```python
            if Vitesses_suivantes[j] > dn - 1:
                Vitesses_suivantes[j] = dn - 1
            # Étape 3: Facteur aléatoire
            if rand() < p and Vitesses_suivantes[j] > 0:
                Vitesses_suivantes[j] = Vitesses_suivantes[j] - 1
    return Vitesses_suivantes
```

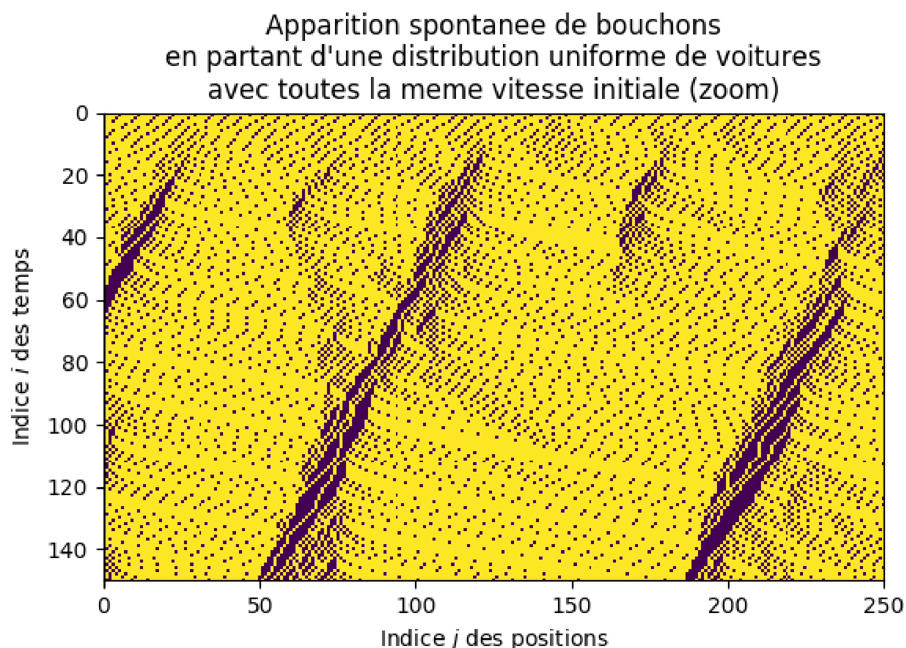
La fonction `distance` n'était pas demandée mais on peut l'écrire sous la forme suivante.

```python
def distance(Route,i,j):
    N = len(Route[i])
    d = 1 # On est au moins à une distance 1 de la prochaine case
    while Route[i][(j+d)%N] != 1: # Tant qu'on ne rencontre pas d'autre voiture,
        d = d+1                  # on continue à regarder devant soi
    return d
```

Q18 Reste à faire attention pour la mise à jour

```python
def deplacement(Vitesses,Route,Vitesses_suivantes,i):
    N = len(Vitesses_suivantes)
    for j in range(N):
        if Route[i][j] == 1: # Si il y a une voiture,
            # on prédit sa nouvelle position
            prochain = (j + Vitesses_suivantes[j])%N
            # et on met à jour
            Route[i+1][prochain] = 1
            Vitesses[i+1][prochain] = Vitesses_suivantes[j]
    return Route, Vitesses
```

L'application complète de l'algorithme donne le résultat suivant



Q19 Il s'agit de repérer des zones de la route où la concentration de voitures est importante, donc où le nombre de points noirs est localement important. Et effectivement, on observe (par exemple aux environs de la position $j = 100$ pour un temps $i = 140$ sur la figure de l'énoncé) des agrégats de points noirs qui « remontent » le flot des voitures, ce qui est un phénomène assez connu des embouteillages: ceux-ci peuvent se propager comme des ondes parfois à contre-sens du sens de circulation puisque de nouvelles voitures rentrent à chaque instant dans l'embouteillage alors que celles de devant en sortent. Il suffit alors que la sortie de l'embouteillage se fasse plus rapidement que l'entrée pour voir cet effet de remontée (le conducteur « anticipe » l'arrivée dans le bouchon en entrée en limitant sa vitesse lors de la phase de décélération).

Pour pouvoir reproduire d'autres caractéristiques des embouteillages, on peut jouer sur le nombre de voitures initialement présentes sur la route ou la taille des cellules. On peut aussi tenter de rajouter des voies (mais c'est plus subtil et demandera bien plus de travail pour que le tout soit cohérent), des entrées, des sorties (avec une certaine probabilité de sortir), des zones à vitesse réduite, etc.