Informatique

CCP PSI 2016: Corrigé

Q1 Comme un octet correspond à 8 bits, on a une taille de

$$\overbrace{64 \times 64 \times 352}^{\text{nb de pixels}} \times 12 = 17301504 \text{ bits} = 2162688 \text{ octets} = 2{,}16 \text{ Mo}$$

NB: si on considère que 1 ko = 1024 octets (alors qu'en fait c'est un kibioctet noté Kio, voir la page wikipédia sur l'octet), on trouverait 2,06 Mo, ce qui n'est pas très éloigné.

Q2 Le taux de compression est alors d'environ 50% ou plus exactement

$$\tau = \frac{2{,}16 - 1}{2{,}16} = 53{,}8\%$$

Q3 Prenons en compte le fait que plusieurs caractères apparaissent avec la même probabilité. Alors

$$\mathrm{H} = -2 \times 0{,}3 \log_2(0{,}3) - 4 \times 0{,}1 \log_2(0{,}1) = 2{,}37 \text{ bits/caractère}$$

Connaissant le nombre de bits pour un caractère (4 ici), le taux de compression vaut donc

$$\tau = \frac{4 - \mathrm{H}}{4} = 40{,}7\%$$

Le codage présenté (à 2,4 bits/caractère) se rapproche bien de l'estimation entropique.

Q4, 5 et 6 Le code complet demandé peut être implémenté de la façon suivante

```
def entropie(S):
    # Q4: On veut récupérer la liste des valeurs distinctes rencontrées dans S
    valeurs = list(set(S))
    # Q5: Calcul des différentes probabilités. Il faut d'abord compter.
    # Initialisation de la liste des proba (liée à la liste 'valeurs')
    proba = [0]*len(valeurs)
    for data in S:# Boucle sur chaque données
        # On veut assigner le poids à la valeur correspondante
        for i in range(len(valeurs)):
            if data == valeurs[i]:    # Si on trouve le gagnant,
                proba[i] += 1         # on incrémente le compteur.
    # On divise tout par le nombre de données pour avoir la proba de chaque valeur.
    proba = [c/len(S) for c in proba]
    # Q6: reste à appliquer la formule pour trouver l'entropie
```

```
    H = 0 # Initialisation
    for p in proba: # On itère sur les différentes probas
        H = H - p * log2(p) # et on rajoute la contribution à l'entropie
    return H
```

```
>>> entropie([4,5,7,0,7,8,4,1,7,4]) # Vérification
2.37095059445
```

Q7 Les entiers étant codés sur 12 bits, la suite d'instruction à donner est

```
H = entropie(bloc_image)
tau = (12-H)/12
```

Q8 Ici, il faut bien faire attention aux objets qu'on manipule: les différentes longueurs d'ondes correspondent aux différentes lignes de la matrice manipulée d'après l'énoncé.

```
def pretraitement12(donnees_brutes):
    colonnes = len(donnees_brutes[0])
    # Fabrication de la ligne moyenne
    luminance_moy = [0]*colonnes
    selection = [1,11,21,31] # Sélection de longueur d'ondes
    for j in range(colonnes):
        for i in selection:
            luminance_moy[j] += donnees_brutes[i][j]/4
    # Détection de la valeur et de la position du maximum
    j_max = 0
    for j in range(colonnes):
        if luminance_moy[j] > luminance_moy[j_max]:
            j_max = j
    luminance_max = luminance_moy[j_max]
    # On renvoie les infos demandées
    return luminance_moy,luminance_max,j_max
```

Q9 On présente deux versions, l'une « manuelle » et l'autre utilisant les facilités de Numpy (la matrices donnée en entrée étant supposée être un `np.array` à deux dimensions)

```
def pretraitement34(donnees_brutes,luminance_max,j_max):
    """ Version 'manuelle'. """
    # Extraction du spectre et normalisation en une fois
    spectre = [0] * len(donnees_brutes)
    for i in range(len(donnees_brutes)):
        spectre[i] = donnees_brutes[i][j_max] / luminance_max
    return spectre

def pretraitement34(donnees_brutes,luminance_max,j_max):
    """ Version Numpy. """
    return donnees_brutes[:,j_max] / luminance_max
```

Q10 La matrice M produit d'une colonne C par une ligne L est telle que $M_{ij} = C_i \times L_j$. Ainsi, on a

```python
from numpy import * # Pour générer le tableau de 0

def pretraitement5(luminance_moy,spectre_max):
    matrice_modele = zeros((len(spectre_max),len(luminance_moy)))
    for i in range(len(spectre_max)):
        for j in range(len(luminance_moy)):
            matrice_modele[i][j] = spectre_max[i] * luminance_moy[j]
    return matrice_modele
```

Q11 En supposant que le moyennage se fera toujours uniquement sur 4 lignes, `pretraitement12` ne fait que des actions dont le nombre est proportionnel au nombre m de colonnes (voir les boules des lignes 6 et 11), donc en $\mathrm{O}(m)$.

À l'inverse, `pretraitement34` ne fait qu'une boucle sur le nombre n de ligne Sa complexité calculatoire est donc en $\mathrm{O}(n)$.

Enfin, la construction de la matrice modèle nécessite de passer une fois par chaque case de la matrice, donc une complexité en $\mathrm{O}(m \times n)$ qui prédomine sur les deux précédentes.

Au total, on a une complexité en $\mathrm{O}(m \times n)$.

Q12 D'après le texte, la « prédiction » revient juste à faire la différence entre deux valeurs consécutives de la liste x. On peut alors écrire

```python
def prediction(x):
    return x[1:] - x[:-1]
```

ou plus pédestrement (et certainement plus proche des attendus du programme)

```python
def prediction(x):
    erreur = [0] * (len(x)-1)
    for i in range(len(erreur)):
        erreur[i] = x[i+1] - x[i]
    return erreur
```

Q13 Pour celle-ci, qui vise à vérifier la bonne utilisation des branchements conditionnels, on va s'autoriser la primitive `min` de Python.

```python
def mappage(erreur,x):
    delta = [0] * len(erreur)
    for k in range(len(delta)):
        thetak = min(x[k], 4095-x[k])
        if 0 <= erreur[k] <= thetak:
            delta[k] = 2*erreur[k]
        elif -thetak <= erreur[k] < 0:
            delta[k] = 2*abs(erreur[k])-1
        else:
            delta[k] = thetak + abs(erreur[k])
    return delta
```

Q14 Pour $p = 3$, on a $2^p = 8$. Il faut donc calculer les divisions euclidiennes de chaque δ_k. On a alors

$4 = 0\times8+4$	soit	0 100
$10 = 1\times8+2$	soit	10 010
$18 = 2\times8+2$	soit	110 010
$18 = 2\times8+2$	soit	110 010
$6 = 0\times8+6$	soit	0 110
$1 = 0\times8+1$	soit	0 001
$3 = 0\times8+3$	soit	0 011

Q15 et 16 D'après ce qu'on lit dans la queston 16, il semblerait que `code1` et `code2` doivent être des chaînes de caractère. On peut alors écrire

```
def codage(delta_k,p_opt):
    diviseur = 2**p_opt
    code1 = []
    code2 = []
    for k in range(len(delta_k)):
        # Calcul du quotient et du reste par division euclidienne
        quotient = delta_k[k] // diviseur
        reste = delta_k[k] % diviseur
        # Autant de '1' que le quotient et on rajoute un '0'
        code1.append('1'*quotient + '0')
        # conversion en binaire de l'entier correspondant
        code2.append(bin(reste))
    return code1,code2 # Pas clair de savoir quoi renvoyer...
```

Q17 Il s'agit d'une application du théorème de la quantité de mouvement (ou « loi de la quantité de mouvement », « Principe Fondamental de la Dynamique » ou encore « Relation Fondamentale de la Dynamique ») au satellite dans le référentiel héliocentrique, le tout divisé par la masse du satellite qui se simplifie gentiment au passage. Le terme de gauche est l'accélération du satellite. Le premier terme à droite correspond au champ gravitationnel du Soleil et le second correspond à celui de Vénus.

Q18 Il s'agit de faire une projection (propre) de la RFD précédente. On obtient

$$\begin{cases} \mathrm{S}_x(x,y,t) = -\dfrac{\mathrm{G}m_s\,x}{(x^2+y^2)^{3/2}} - \dfrac{\mathrm{G}m_v\,(x-x_v)}{((x-x_v)^2+(y-y_v)^2)^{3/2}} \\ \mathrm{S}_y(x,y,t) = -\dfrac{\mathrm{G}m_s\,y}{(x^2+y^2)^{3/2}}\,y - \dfrac{\mathrm{G}m_v\,(y-y_v)}{((x-x_v)^2+(y-y_v)^2)^{3/2}} \end{cases}$$

```
def position_venus(t):
    return rv*cos(Omega*t+Phi),rv*sin(Omega*t+Phi)

def eval_sm(x,y,t):
    xv,yv = position_venus(t)
    normes= sqrt(x**2 + y**2)
    normev= sqrt((x-xv)**2 + (y-yv)**2)
    Sx = -Gms * x / normes**3 - Gmv * (x-xv) / normev**3
    Sy = -Gms * y / normes**3 - Gmv * (y-yv) / normev**3
```

Q19 Euler permet d'obtenir

$$\boxed{u_{i+1} = u_i + \mathrm{S}_x(x_i, y_i, t)\,\Delta t \qquad \text{et} \qquad v_{i+1} = v_i + \mathrm{S}_y(x_i, y_i, t)\,\Delta t}$$

Q20 De même

$$\boxed{x_{i+1} = x_i + u_i\,\Delta t \qquad \text{et} \qquad y_{i+1} = y_i + v_i\,\Delta t}$$

Q21 et 22

```
# Prédéclarations globales
n = 300      # Ou autres...
deltaT = 1

def trajectoire_sonde(x0,y0,u0,v0):
    # Q21: Initialisation (on remplira au fur et à mesure après)
    x = [x0]
    y = [y0]
    u = [u0]
    v = [v0]
    t = [0]
    # Q22: Remplissage avec relations de récurrences
    for i in range(n): # n et deltaT sont déjà déclarés
        Sx,Sy = eval_sm(x[i],y[i],t[i])
        u.append(u[i] + Sx*deltaT)
        v.append(v[i] + Sy*deltaT)
        x.append(x[i] + u[i]*deltaT)
        y.append(y[i] + v[i]*deltaT)
        t.append(t[-1]+ deltaT)
    return x,y,u,v,t
```

Q23 30 jours représentent $30 \times 86\,400 = 2\,592\,000$ s. 4 octets (soit 32 bits) suffisent à coder les entiers correspondant au temps. On a donc un total de

$$\boxed{\overbrace{30 \times 86\,400}^{\text{nb de pas}} \times (4 \times 8 + 4) = 93\,312\,000 \text{ octet} = 89{,}0 \text{ Mo}}$$

Cela commence à faire une taille appréciable mais largement gérable par les machines modernes.

Q24 On a besoin en entrée des vitesse `u` et `v` calculées précédemment ainsi que de la liste `t` des temps. Les valeurs étant données dans le cadre du système internationnal, les valeurs obtenues pour les vitesses sont en m/s qu'il convient de diviser par 1000 pour obtenir des km/s.

```
from matplotlib.pyplot import * # pour utiliser plot, xlabel and Cie directement
# Note du correcteur: je pense que c'est un mauvais choix, mais c'est pour
# suivre l'énoncé...

def vitesse_sonde(u,v,t):
    norme = []
    for i in range(len(u)):
        # On convertit au vol en km/s
        norme.append(sqrt(u[i]**2+v[i]**2) / 1000)
    plot(t,norme)
    xlabel('Temps en s')
```

```
12      ylabel('Vitesse en km/s')
13      title('Evolution de la vitesse au cours du temps dans notre simulation')
```

En voici une version légèrement plus complexe (préparée par Yohan Loquais) qui donne à la fois l'évolution de la vitesse et la trajectoire suivie par la sonde.