Correction du devoir de vacances

```
import numpy as np

#Q1
"""
Chaque coordonnée va de -10^13 à 10^13.
On cherche n tel que 2^n>10^13, on trouve n = 44.
Avec le signe, chaque coordonnée doit être codée sur 45 bits.
Pour les 3 coordonnées des 556 172 objets, cela donne 75 083 220 bits, soit environ 9 Mio.
Ce n'est pas une contrainte forte puisque les ordinateurs ont plusieurs Gio de mémoire.
Remarque : il s'agit de calculs théorique. D'après sys.getsizeof(), Python utilise au moins 28 octets pour stocker les entiers.
"""

#Q2
"""
Un nombre à virgule flottante est stocké sur 64 bits. Il faut donc multiplier le résultat précédent par 64/45.
On obtient environ 13 Mio.
"""

#Q3
"""
Python calcule avec les entiers en valeurs exactes mais les distances seront données au mètre près.
On peut considérer qu'une distance dans le système solaire est de l'ordre du milliard de kilomètres. Les entiers fourniront donc 12 chiffres significatifs (parfois 9, parfois 13 selon les objets considérés).
En flottant, on dispose toujours de 15 chiffres significatifs. Les résultats seront donc plus précis.
"""

#Q4
masses = [6000,500,300,299,250,100,4]

def insere_masse(masse: float):
    a,b = 0,len(masses)-1
    if masse>masses[0]:
        masses.insert(0,masse)
        return
    elif masse<masses[b]:
        masses.append(masse)
        return
    while b-a>1:
        c = (a+b)//2
        if masse > masses[c]: b = c
        else: a = c
    masses.insert(b,masse)

#Q5
"""
positions[0] correspond à la position du soleil et est donc égal à np.array([0,0,0]).
"""
```

#Q6

"""

D'après la loi de l'attraction universelle, la force qu'exerce un corps j sur un corps i est F = G * Mi * Mj / d**2, suivant le vecteur Fij w(que l'on divise par sa norme pour obtenir un vecteur unitaire)
On doit ensuite sommer toutes ces forces.

"""

```
def force(i):
    G = 6.67408 * 10**(-11)
    f = [0,0,0]
    n = len(masses)
    for j in range(n):
        if j!=i:
            vectij = positions[j]-positions[i]
            forceij = (G*masses[i]*masses[j]/(sum((vectij)**2))**(3/2))*vectij
            f += forceij
    return np.array(f)

# Remarque : on peut multiplier par G*masses[i] à la fin plutôt que dans la boucle
```

#Q7

"""

Principe fondamental de la dynamique : a = 1/m * F
Or, a = v' donc en appliquant la méthode d'Euler, on remplace à chaque étape v par v + 1/m * F * dt

"""

```
def calcule_vitesse(vitesses,dt):
    n = len(vitesses)
    nouvelles_vitesses = []
    for i in range(n):
        nouv_vitessei = vitesses[i] + force(i)*dt/masses[i] # opération vectorielle puisque vitesses[i] et force(i)
                                                                sont des arrays à trois coordonnées
        nouvelles_vitesses.append(nouv_vitessei)
    return np.array(nouvelles_vitesses)
```

#Q8

"""

Même principe : pos = pos + v*dt

"""

```
def calcule_position(positions,dt):
    return positions + dt*vitesses  # en utilisant les propriétés vectorielles des tableaux numpy
```

#Q9

"""

La méthode d'Euler explicite est la méthode habituelle : $y(t+dt) \approx y(t) + y'(t)*dt$
La méthode d'Euler implicite est : $y(t+dt) \approx y(t) + y'(t+dt)*dt$
On applique la méthode implicite pour le calcul des positions puisqu'on met à jour les vitesses avant les positions.
On pourrait sans mal conserver les anciennes vitesses pour le calcul des positions si l'énoncé ne permettait pas la méthode implicite.

```
"""
def simulation(T,N):
    dt = T/N
    res_positions = [positions]
    nouv_vitesses = vitesses
    nouv_positions = positions
    for i in range(N):
        nouv_vitesses = calcule_vitesse(nouv_vitesses,dt)
        nouv_positions = calcule_position(nouv_positions,dt)
        res_positions.append(nouv_positions)
    return res_positions

#Q10
"""
simulation appelle N fois calcule_vitesse et calcule_position.
calcule_vitesse appelle n fois force, qui est en O(n), donc calcule_vitesse est en O(n²).
On peut négliger calcule_position (linéaire) devant calcule_vitesse.
Ainsi, simulation est en O(Nn²).
"""

#Q11
"""
La variable res_positions contient N+1 lignes avec dans chaque ligne les 3 coordonnées des 556 172 objets.
Une année de 365 jours contient 31 536 000 secondes.
res_positions contient donc 52 618 322 244 516 flottants codés sur 64 bits, donc nécessite 420946577956128 octets, soit 382,8 Tio.
"""

#Q12
"""
Au niveau mémoire, cette simulation n'est pas réalisable en l'état.
Même en stockant sur disque les positions au fur et à mesure des calculs, il faudrait 192 disques durs de 1 To !

Au niveau temps, avec n = 556 172 et N = 31 536 001, la simulation exécute de l'ordre de Nn² = 10**19 opérations.
Un processeur ayant une fréquence de 1 Ghz effectue 10**9 opérations par seconde.
La simulation nécessitera 10**10 secondes, soit 317 ans, et même probablement beaucoup plus car nous n'avons pas compté toutes les opérations élémentaires dans le calcul de complexité.

La simulation est clairement irréalisable.
"""

#Q13
"""
On peut accélérer le programme en négligeant les objets de petites masses qui n'influeront pas vraiment sur les trajectoires dans le calcul des forces.
On pourrait aussi négliger les objets éloignés mais cela est plus complexe à gérer.
```

Au niveau mémoire, on peut stocker les positions des corps toutes les heures par exemple au lieu de toutes les secondes. Cela semble raisonnable à l'échelle du système solaire.
On divise alors l'espace nécessaire par 3600 et cela rentre sur un disque dur ordinaire (mais toujours pas dans la mémoire vive).
"""

#Q14
"""
La méthode des trapèzes consiste à approcher l'intégrale S de f sur [a,b] la somme des (f(x_i)+f(x_i+1))/2 * dt avec (x0, x1, ... ,xn) subdivision de [a,b].
Donc S = dt*(f(a)/2+f(b)/2+f(x1) + ... + f(x_n-1)
"""

```
def norme(v):
    return (sum(v**2))**0.5

def distance_parcourue(i,res_vitesses,T):
    N = len(res_vitesses)
    S = (norme(res_vitesses[0][i])+norme(res_vitesses[-1][i]))/2
    for j in range(1,N-1):
        S += norme(res_vitesses[j][i])
    return S*T/N
```

#Q15
"""
Soit f définie sur R par f(E) = E-e*sin(E)-M. f est dérivable sur R et pour tout réel x, f'(E) = 1 - e*cos(E).
On construit une suite (En) convergeant vers la racine cherchée avec E0 = M et pour tout n, E_n+1 = En-f(En)/f'(En)
"""

```
def resolution_Kepler(e,M):
    E = M                                #E(n)
    E2 = E-(E-e*np.sin(E)-M)/(1-e*np.cos(E))    #E(n+1)
    while abs(E2-E)>epsilon:
        E,E2 = E2,E2-(E2-e*np.sin(E2)-M)/(1-e*np.cos(E2))
    return E2
```

#Q16
"""
```
SELECT date, COUNT(*)
FROM corps_celestes
GROUP BY date
```
"""

#Q17
"""
```
SELECT *
FROM corps_celestes
WHERE date = d
  AND x^2+y^2+z^2 < 10^13
```
"""

#Q18

"""

Ne pas oublier que les listes doivent être triées selon les masses décroissantes.
Le plus simple est d'effectuer le tri en SQL.
Dans un premier temps, j'inclue les masses dans ma sélection afin d'effectuer le tri.
Ensuite, je ne sélectionne que les colonnes qui m'intéressent.
Je me suis amusé à utiliser la méthode format qui remplace {} par la chaine de caractères donnée.
On peut très bien s'en passer :

```
   sql = '(SELECT * FROM corps_celestes WHERE date = d AND x^2+y^2+z^2 < 10^13 ORDER BY masse DESC)'
   masses = requete_BDD('SELECT masses FROM '+sql)
```

"""

```
def initialisation():
   global masses, positions, vitesses
   sql = 'SELECT {} FROM (SELECT * FROM corps_celestes WHERE date = d AND x^2+y^2+z^2 < 10^13 ORDER BY masse DESC)'
   masses = requete_BDD(sql.format('masse'))
   positions = requete_BDD(sql.format('x,y,z'))
   vitesses = requete_BDD(sql.format('vx,vy,vz'))
```

#Q19

```
def ata(objets):
   AT = [[objet[0] for objet in objets],[objet[1] for objet in objets],[1 for objet in objets]]
   res = []
   for i in range(3):
      ligne = []
      for j in range(3):
         ligne.append(sum(AT[i][k]*AT[j][k] for k in range(len(objets))))
      res.append(ligne)
   return res
```

#Q20

"""

ATB est une matrice colonne (x,y,z)

"""

```
def atb(objets):
   x,y,z = 0,0,0
   for i in range(len(objets)):
      x += objets[i][0]*objets[i][2]
      y += objets[i][1]*objets[i][2]
      z += objets[i][2]
   return [x,y,z]
```

#Q21

```
ATA = ata(objets)
ATB = atb(objets)
```

```
def resoudre(ATA,ATB):
    # On complète la matrice ATA avec les éléments de ATB
    for i in range(3):
        ATA[i].append(ATB[i])
    for i in range(3):
        # Choix du pivot
        indice_pivot = i
        for k in range(i,3):
            if abs(ATA[k][i]) > abs(ATA[indice_pivot][i]):
                indice_pivot = k
        # Permutation des lignes
        ATA[i],ATA[indice_pivot] = ATA[indice_pivot],ATA[i]
        # Division de la ligne par le pivot
        pivot = ATA[i][i]
        for j in range(i,4): ATA[i][j] = ATA[i][j]/pivot
        for k in range(0,3):
        # Enlève à la ligne k, aki * la ligne i, mettant ainsi aki à 0
            if k != i:
                aki = ATA[k][i]
                for j in range(i,4):
                    ATA[k][j] = ATA[k][j]-aki*ATA[i][j]
    return [ATA[i][3] for i in range(3)]

#Q22
def plan(objets):
    A = np.matrix(objets)          # transforme objets en matrice
    A[:,2]=np.ones((len(objets),1)) # remplace la dernière colonne par des 1
    B = np.matrix(objets)[:,2]     # récupère la dernière colonne d'objets
    return (A.T*A)**-1*A.T*B
```

Compléments pour tester :

```
# Représentation graphique
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# points
x = [objet[0] for objet in objets]
y = [objet[1] for objet in objets]
z = [objet[2] for objet in objets]
plt.figure()
ax = plt.subplot(111, projection='3d')
ax.scatter(x, y, z, color='b')

# plan
xlim = ax.get_xlim()
ylim = ax.get_ylim()
X,Y = np.meshgrid(np.arange(xlim[0], xlim[1]),
                  np.arange(ylim[0], ylim[1]))
Z = np.zeros(X.shape)
fit = resoudre(ATA,ATB)
for r in range(X.shape[0]):
    for c in range(X.shape[1]):
        Z[r,c] = fit[0] * X[r,c] + fit[1] * Y[r,c] +
fit[2]
ax.plot_wireframe(X,Y,Z, color='k')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```