TP n° 13 : Correction

A. Quelques fonctions utilitaires

1. Pour les listes, l'opérateur `+` est l'opérateur de concaténation. Ainsi

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

La multiplication d'une liste par un entier n revient à la concaténer avec elle-même n fois. En particulier

```
>>> 2 * [1, 2, 3]
[1, 2, 3, 1, 2, 3]
```

2. Plusieurs versions possibles. Par exemple

```
def smul(n, L):
    M = [0]*len(L)
    for i in range(len(L)):
        M[i] = n*L[i]
    return M
```

```
def smul(n, L):
    M = []
    for element in L:
        M.append(n*element)
    return M
```

3.

```
def vsom(L1, L2):
    L = [0]*len(L1)
    for i in range(len(L1)):
        L[i] = L1[i] + L2[i]
    return L
```

On pourrait aussi comme précédemment initialiser `L` en une liste vide puis utiliser la méthode `append`.

4. Soit on adapte `vsom`, soit on l'utilise ainsi que `smul`.

```
def vdif(L1, L2):
    L = []
    for i in range(len(L1)):
        L.append(L1[i]-L2[i])
    return L
```

```
def vdif(L1, L2):
    return vsom(L1, smul(-1,L2))
```

B. Étude de schémas numériques

5. *Mise en forme du problème*

5.a) Comme $y'' = f(y)$ et $z = y'$, on a $z' = y'' = f(y)$. Ainsi l'équation (1) se réécrit $(S) : \begin{cases} z = y' \\ z' = f(y(t)) \end{cases}$.

5.b) On utilise le fait que $z(t) = y'(t)$ et ce que l'on appelle parfois le « théorème fondamental du calcul intégral » :

$$\int_{t_i}^{t_{i+1}} z(t)\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} y'(t)\,\mathrm{d}t = [y(t)]_{t_i}^{t_{i+1}} = y(t_{i+1}) - y(t_i)$$

d'où la première relation en ajoutant $y(t_i)$ à chaque membre de cette égalité.

On obtient de manière analogue l'autre relation en utilisant le fait que $f(y(t)) = z'(t)$ d'après la question précédente.

6. *Schéma d'Euler explicite*

6.a) Pour tout $i \in [\![0\,;n-2]\!]$, on a $y(t_i) \approx y_i$, $y(t_{i+1}) \approx y_{i+1}$, de même pour z et, pour tout $t \in [t_i\,;t_{i+1}]$, $y(t) \approx y_i$ et $z(t) \approx z_i$.
En remplaçant alors dans les relations obtenues à la question précédente, on obtient :

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} z_i \,\mathrm{d}t = y_i + [z_i t]_{t_i}^{t_{i+1}} = y_i + z_i\,(t_{i+1} - t_i)\,.$$

Comme $t_{i+1} - t_i = t_{\min} + (i+1)h - (t_{\min} + ih) = h$, on a bien $y_{i+1} = y_i + hz_i$.
En approximant $f(y(t))$ par la constante $f(y_i)$ sur l'intervalle $[t_i\,;t_{i+1}]$ et en procédant de même que ci-dessus, on obtient $z_{i+1} = z_i + hf(y_i)$.

6.b) C'est une question très classique à maîtriser !

```
def euler(f,y0,z0,h,n):
    Ly = [y0]    #Liste de stockage des valeurs de y_i
    Lz = [z0]    #Idem pour z_i
    for i in range(n-1):
        Ly.append(Ly[i] + h*Lz[i])      #On ajoute à la fin de Ly
                                        #la valeur de y_{i+1}
        Lz.append(Lz[i] + h*f(Ly[i]))   #Idem pour Lz avec z_{i+1}
    return Ly, Lz
```

7. *Schéma de Verlet*
Il s'agit simplement d'adapter la fonction précédente aux équations de Verlet.

```
def verlet(f,y0,z0,h,n):
    Ly = [y0]
    Lz = [z0]
    for i in range(n-1):
        Ly.append(Ly[i] + h*Lz[i] + h**2/2*f(Ly[i]))
        Lz.append(Lz[i] + h/2*(f(Ly[i]) + f(Ly[i+1])))
    return Ly, Lz
```

Remarque : le calcul de z_{i+1} nécessite la valeur de y_{i+1} donc il est nécessaire de faire les calculs pour y avant ceux pour z contrairement à la méthode d'Euler pour laquelle l'ordre n'importait pas.

C. Problème à N corps

8. *Les forces en jeu dans ce problème*

8.a) On a

$$\overrightarrow{F_j} = \sum_{\substack{k=0\\k\neq j}}^{N} \overrightarrow{F_{k/j}} = \sum_{\substack{k=0\\k\neq j}}^{N} G\frac{m_j m_k}{r_{jk}^3}\overrightarrow{P_jP_k}.$$

8.b) Il s'agit de la distance usuelle entre deux points de l'espace repérés par leurs coordonnées cartésiennes. On peut utiliser la fonction `vdif` définie dans la partie A si l'on veut.

```
def distance(p1,p2):
    d = 0
    vecteur = vdif(p2,p1)    #Le vecteur P1P2
    for i in range(3):
        d += vecteur[i]**2   #On somme les carrés de
                             #chaque composante
    return d**0.5    #Le résultat est la racine carré de la somme
```

8.c) On utilise la fonction précédente pour la distance et la fonction `vdif` définie au début du sujet pour obtenir les coordonnées du vecteur $\overrightarrow{P_1P_2}$.

```
def force2(m1,p1,m2,p2):
    G = 6.67*10**(-11)
    k = G*m1*m2/distance(p1,p2)**3
    return smul(k, vdif(p2,p1))
```

Remarque : on aurait aussi pu définir la constante G en dehors de la fonction.

8.d) Pour obtenir la force totale, il suffit d'additionner toutes les forces comme montré à la question 8.a. Pour additionner les vecteurs forces, on utilise la fonction `vsom` du début du sujet.

```
def forceN(j, masse, pos):
    force = [0]*3
    for k in range(len(masse)):
        if k != j:
            force_kj = force2(masse[j],pos[j],masse[k],pos[k])
            force = vsom(force, force_kj)
    return force
```

9. *Approche numérique*

9.a) `position[i]` et `vitesse[i]` sont des listes de N listes (où N est le nombre de particules) de 3 éléments chacune. Elles contiennent respectivement les positions et les vitesses (sous forme de vecteur) de toutes les particules à l'instant t_i.

9.b) D'après le principe fondamental de la dynamique appliqué au corps j, on a $m_j\overrightarrow{a_j} = \overrightarrow{F_j}$ d'où la relation souhaitée en divisant par la masse qui est non nulle.

9.c)

```
1  def fct(masse, pos, vit, h):
2      N = len(masse)
3      L = []
4      for j in range(N):
5          next_pj = smul(0, pos[j])
6          mj, pj, vj = masse[j], pos[j], vit[j]
7          force = forceN(j, masse, pos)
8          for k in range(3):
9              next_pj[k] = pj[k] + h*vj[k] + h**2/2*force[k]/mj
10         L.append(next_pj)
11     return L
```

i. La variable `force` est une liste de trois flottants (la réponse est écrite dans l'énoncé de la question 8.d). Elle représente la force subie par le corps j.

ii. Le nombre 3 représente les trois dimensions de l'espace.

iii. La ligne 9 consiste à calculer la k-ème composante du vecteur position à l'instant t_{i+1}. Comme on procède via l'algorithme de Verlet :
- la position $y_i \approx y(t_i)$ est représentée par `pj`,
- la vitesse $z_i \approx z(t_i) = y'(t_i)$ est représentée par `vj`,
- le terme $f(y_i) \approx f(y(t_i)) = z'(t_i) = y''(t_i)$ est l'accélération qui vaut donc la force divisée par la masse d'après la question 9.b.

iv. Cette fonction renvoie une liste de N listes à 3 éléments, chacune de ses listes représentant le vecteur position à l'instant t_{i+1}. Autrement dit, cette fonction permet de connaître la position à l'instant suivant de chaque particule lorsqu'on connaît la position et la vitesse de chaque particule à un instant donné.

9.d)

```
1  def etat_suiv(masse, pos, vit, h):
2      N = len(masse)
3      next_pos = fct(masse, pos, vit, h):
```

```
    next_vit = []
    for j in range(N):
        next_vj = smul(0, vit[j])
        mj, vj = masse[j], vit[j]
        fi = smul(1/mj, forceN(j,masse,pos))
        fiplus1 = smul(1/mj, forceN(j,masse,next_pos))
        for k in range(3):
            next_vj[k] = vj[k] + h/2*(fi[k] + fiplus1[k])
        next_vit.append(next_vj)
    return next_pos, next_vit
```

i. On aurait pu aussi utiliser `next_vj = [0]*len(vit[j])`

ii. On utilise l'expression de l'accélération obtenue à la question 9.b et on se sert de la fonction `smul` du début du sujet pour diviser le vecteur `force` par le scalaire `masse[j]`.

iii. Il suffit de reprendre l'équation concernant la vitesse $z = y'$ qui définit la méthode de Verlet.

iv. La fonction `etat_suiv(masse, pos, vit, h)` renvoie un couple de deux listes, chacune contenant N listes à 3 éléments. Elles représentent respectivement les vecteurs position et vitesse de chaque particule à l'instant t_{i+1}.

10. *Complexité*

10.a) La fonction `force2` fait simplement appel aux fonctions `distance` et `smul` sur des listes de taille 3, elle ne dépend pas du nombre de corps donc sa complexité est $O(1)$.

La fonction `forceN` contient une boucle de taille N, chaque tour comportant un appel à `force2` et un à `vsom` sur des vecteurs de taille 3. Ces deux dernières étant de complexité $O(1)$, la fonction `forceN` est de complexité $O(N)$.

10.b) Tout d'abord, la fonction `fct` est de complexité $O(N^2)$ car elle contient une boucle de taille N dans laquelle l'opération la plus coûteuse est l'appel de `forceN` qui est $O(N)$.

Ainsi, la fonction `etat_suiv` commence par un appel à `fct` qui est $O(N^2)$. Ensuite, il y a une boucle de taille N dans laquelle est appelée deux fois la fonction `forceN` qui est de complexité $O(N)$, les autres calculs étant $O(1)$. Cette boucle est donc de complexité $O(N^2)$.

Finalement, la fonction `etat_suiv` est de complexité $O(N^2) + O(N^2) = O(N^2)$.

10.c) i. On voit que les points sont quasiment alignés. La relation est donc de type affine : il existe deux réels a et b tels que $\ln(\tau_N) = a\ln(N) + b$. En utilisant les valeurs des points extrêmes, on obtient $a \approx 2$ et $b \approx -9{,}5$ d'où $\ln(\tau_N) = 2\ln(N) - 9{,}5$.

ii. On compose la relation précédente par l'exponentielle et on obtient $\tau_N = \mathrm{e}^{2\ln N - 9{,}5} = \dfrac{1}{\mathrm{e}^{9{,}5}} N^2$.

iii. On a obtenu à la question précédente que le temps de calcul τ_N est proportionnel à N^2, ce qui est en accord avec l'étude effectuée à la question 10.b.

Ce TP est une adaptation du sujet de Centrale 2015 intitulé « Autour de la dynamique gravitationnelle ».