Informatique

Centrale MP PC PSI TSI 2015: Corrigé

I.A.1 Le + appliqué sur deux listes a pour action de les concaténer l'une à l'autre en créant une troisième liste[1]. Il ne faut donc pas confondre avec l'addition terme à terme comme on peut l'imaginer lors de la sommation de deux vecteurs.

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

I.A.2 Suivant la même sémantique, la multiplication d'un entier avec une liste revient à concaténer la liste avec elle-même autant de fois que demandé.

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

I.B Présentons trois versions possibles, l'une en créant une liste remplie de zéros, l'autre en itérant sur les éléments et en construisant la liste au fur et à mesure par ajouts successifs, la dernière en utilisant la Pythonnerie de contruction de liste par compréhension.

```python
def smul(nombre,liste):
    L = [0]*len(liste)            # Initialisation à une liste de 0
    for i in range(len(liste)):   # Autant de fois que d'éléments
        L[i] = nombre * liste[i] # On remplit avec la valeur idoine
    return L                      # On n'oublie pas de renvoyer le résultat

def smul(nombre,liste):
    L = []                        # Initialisation à une liste vide
    for element in liste:         # On itère sur les éléments
        L.append(nombre*element) # On rajoute la valeur idoine
    return L                      # On n'oublie pas de renvoyer le résultat

def smul(nombre,liste):
    return [nombre*element for element in liste] # One-liner !
```

[1] Ce n'est donc pas une bonne idée de l'utiliser pour remplir une liste élément par élément ! La recopie est en effet coûteuse en règle générale.

I.C.1 Sur l'exemple précédent, on peut définir l'addition[2]

```
def vsom(L1,L2):
    """ Fait l'addition vectorielle L1+L2 de deux listes. """
    L = [0] * len(L1)         # Inialisation à une liste de 0
    for i in range(len(L1)): # On regarde toutes les positions
        L[i] = L1[i] + L2[i] # Addition
    return L                  # Renvoi du résultat
```

I.C.2 ou la différence (c'est exactement la même chose, mais on change l'initialisation pour varier...)

```
def vdif(L1,L2):
    """ Fait la différence vectorielle L1-L2 de deux listes. """
    L = []                          # Initialisation à une liste vide
    for i in range(len(L1)):     # On regarde toutes les positions
        L.append(L1[i] - L2[i]) # Rajout de la soustraction
    return L                        # Renvoi du résultat
```

II.A.1 Si l'on note $z = y'$, alors $z' = y''$ et l'on obtient le système différentiel du premier ordre suivant

$$\boxed{\forall t \in \mathrm{I} \qquad \begin{cases} y'(t) = z(t) \\ z'(t) = f(y(t)) \end{cases}} \tag{S}$$

II.A.2 Prenons $i \in [\![0\,;\, n-2]\!]$. Alors, on a

$$\int_{t_i}^{t_{i+1}} z(t)\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} y'(t)\,\mathrm{d}t = y(t_{i+1}) - y(t_i) \qquad \text{soit} \qquad \boxed{y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} z(t)\,\mathrm{d}t}$$

et $$\int_{t_i}^{t_{i+1}} f(y(t))\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} z'(t)\,\mathrm{d}t = z(t_{i+1}) - z(t_i) \qquad \text{soit} \qquad \boxed{z(t_{i+1}) = z(t_i) + \int_{t_i}^{t_{i+1}} f(y(t))\,\mathrm{d}t}$$

II.B.1 Prenons $i \in [\![0\,;\, n-2]\!]$. En assimilant les fonctions intégrées à leurs bornes inférieures, les intégrales s'écrivent respectivement

$$\int_{t_i}^{t_{i+1}} z(t_i)\,\mathrm{d}t = z_i\,(t_{i+1} - t_i) = z_i\,h \qquad \text{et} \qquad \int_{t_i}^{t_{i+1}} f(y(t_i))\,\mathrm{d}t = f(y_i)\,h$$

Ainsi, les relations de récurrence s'écrivent

$$\boxed{y_{i+1} = y_i + z_i\,h \qquad \text{et} \qquad z_{i+1} = z_i + f(y_i)\,h}$$

La connaissance du couple (y_0, z_0) permet donc de proche en proche de déterminer les valeurs successives de y_i et z_i.

II.B.2 Pour écrire la fonction demandée, on n'a besoin que des valeurs initiales y_0 et z_0, de la fonction f définissant l'équation différentielle ainsi que du pas h d'intégration et du nombre n de points voulus (en comptant le point de départ). On peut alors proposer l'implémentation suivante

```
def euler(f,y0,z0,h,n):
    """ Intégration de l'équation différentielle y'' = f(y(t)) avec un pas h
    d'intégration. La fonction renvoie deux listes contenant respectivement
    les valeurs successives prises par y et par z=y'. """
    # Initialisations diverses
    y,z   = y0,z0        # Les valeurs initiales
```

[2] À noter qu'ici on ne peut pas itérer sur les éléments car on a besoin des éléments de chaque liste.

```
    Ly,Lz = [y],[z]        # Les listes de stockage
    for i in range(n-1): # n points au total mais on a déjà le départ
        fy= f(y)           # Calcul de f(y_i) (avant de mettre y à jour)
        y = y +  z*h       # Calcul de y_{i+1} à partir de y_i et z_i
        z = z + fy*h       # Calcul de z_{i+1} à partir de z_i et f(y_i)
        Ly.append(y)       # Ajout de la nouvelle valeur de y à la liste
        Lz.append(z)       # pareil pour z
    return Ly,Lz           # Renvoi des deux listes demandées
```

II.B.3.a Il s'agit d'une équation d'oscillateur harmonique. Le cours de physique peut donc aider à réaliser que $\frac{1}{2}\,y'^2 + \frac{1}{2}\,\omega^2\,y^2 = \mathrm{C^{te}}$. Pour le prouver, multiplions de chaque côté par y' de manière à reconnaître certaines dérivées via la dérivée des fonctions composées:

$$\forall t \in \mathrm{I} \qquad y'(t)\,y''(t) = -\omega^2\,y(t)\,y'(t)$$

soit

$$\forall t \in \mathrm{I} \qquad \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{y'(t)^2}{2}\right) = \frac{\mathrm{d}}{\mathrm{d}t}\left(-\frac{\omega^2\,y(t)^2}{2}\right)$$

ou encore

$$\boxed{\forall t \in \mathrm{I} \qquad \frac{y'(t)^2}{2} + \frac{\omega^2\,y(t)^2}{2} = \mathrm{E} = \mathrm{C^{te}}}$$

II.B.3.b Prenons $i \in [\![\,0\,;\,n-2\,]\!]$. On peut alors calculer

$$\begin{aligned} 2(\mathrm{E}_{i+1} - \mathrm{E}_i) &= z_{i+1}^2 + \omega^2\,y_{i+1}^2 - (z_i^2 + \omega^2\,y_i^2) \\ &= (z_i - \omega^2\,y_i\,h)^2 + \omega^2\,(y_i + z_i\,h)^2 - (z_i^2 + \omega^2\,y_i^2) \\ &= -2\omega^2\,z_i\,y_i\,h + h^2\omega^2 \times \omega^2 y_i^2 + 2\omega^2\,y_i\,z_i\,h + h^2\omega^2 \times z_i^2 \\ 2(\mathrm{E}_{i+1} - \mathrm{E}_i) &= h^2\omega^2\,(\omega^2 y_i^2 + z_i^2) \end{aligned}$$

On trouve bien $\boxed{\mathrm{E}_{i+1} - \mathrm{E}_i = h^2\omega^2 \times \mathrm{E}_i}$

II.B.3.c Un schéma numérique satisfaisant la conservation de l'énergie devrait donner

$$\boxed{\mathrm{E}_{i+1} - \mathrm{E}_i = 0}$$

II.B.3.d L'équation $z^2/2 + \omega^2 y^2/2 = \mathrm{C^{te}}$ est l'équation cartésienne d'une ellipse[3].

II.B.3.e Une trajectoire de phase où l'énergie se conserve devrait être fermée, ce qui n'est pas le cas ici. D'après le calcul mené précédemment, on voit que l'énergie devrait augmenter au cours du temps. Cela correspond à des valeurs de la vitesse z ou de la position y de plus en plus grande, d'où la forme en spirale qui part vers l'extérieur. On peut aussi justifier la rotation en sens horaire puisque dans les zones où z' est positif, on doit avoir y qui augmente (on se déplace donc vers la droite) alors que dans les zones où z' est négatif, on doit avoir y qui diminue (on se déplace vers la gauche).

II.C.1 Procédons de la même manière que pour l'écriture de la fonction `euler` précédente:

```
def verlet(f,y0,z0,h,n):
    """ Intégration de l'équation différentielle y'' = f(y(t)) avec un pas h
    d'intégration. La fonction renvoie deux listes contenant respectivement
    les valeurs successives prises par y et par z=y'. """
    # Initialisations diverses
    y,z   = y0,z0          # Les valeurs initiales
    Ly,Lz = [y],[z]        # Les listes de stockage
```

[3] Voir le cours de physique sur les portrait de phase des oscillateurs à un degré de liberté

```
    for i in range(n-1): # n points au total mais on a déjà le départ
        fi= f(y)         # Calcul de f(y_i) (avant de mettre y à jour)
        y = y + z*h + h*h/2*fi  # Mise à jour de y
        fip1 = f(y)      # Calcul de f(y_{i+1})
        z = z + h/2*(fi + fip1) # Mise à jour de z
        Ly.append(y)     # Ajout de la nouvelle valeur de y à la liste
        Lz.append(z)     # pareil pour z
    return Ly,Lz         # Renvoi des deux listes demandées
```

II.C.2.a Prenons $i \in [\![0\,;\, n-2]\!]$. On peut alors recalculer $\mathrm{E}_{i+1} - \mathrm{E}_i$ comme à la question II.B.3.b. Néanmoins, notons pour démarrer que

$$y_{i+1} = y_i + h z_i - \frac{h^2\omega^2}{2}\, y_i = y_i \left(1 - \frac{h^2\omega^2}{2}\right) + h z_i$$

et
$$\begin{aligned} z_{i+1} &= z_i - \omega^2\, \frac{h}{2}\, (y_i + y_{i+1}) \\ &= z_i - \omega^2\, \frac{h}{2} \left(y_i \left(2 - \frac{h^2\omega^2}{2}\right) + h z_i \right) \end{aligned}$$

soit
$$z_{i+1} = z_i \left(1 - \frac{h^2\omega^2}{2}\right) - \frac{h\omega^2}{2} \left(2 - \frac{h^2\omega^2}{2}\right) y_i = z_i \left(1 - \frac{h^2\omega^2}{2}\right) - h\omega^2\, y_i + \mathrm{O}(h^3)$$

Reprenons à présent le calcul de la différence d'énergie en utilisant le fait que $a^2 - b^2 = (a+b)(a-b)$

$$\begin{aligned} 2(\mathrm{E}_{i+1} - \mathrm{E}_i) &= {z_{i+1}}^2 + \omega^2\, {y_{i+1}}^2 - ({z_i}^2 + \omega^2\, {y_i}^2) \\ &= (z_{i+1} + z_i)(z_{i+1} - z_i) + \omega^2\, (y_{i+1} + y_i)(y_{i+1} - y_i) \\ &= \left[z_i \left(2 - \frac{h^2\omega^2}{2}\right) - h\omega^2\, y_i + \mathrm{O}(h^3) \right] \times (-\omega^2) \left(y_i\, h + z_i\, \frac{h^2}{2} + \mathrm{O}(h^3) \right) \\ &\quad + \omega^2 \left[y_i \left(2 - \frac{h^2\omega^2}{2}\right) + h z_i \right] \times \left(h z_i - y_i\, \frac{h^2\omega^2}{2} \right) \\ &= \left[-h\omega^2\, (2 z_i - h\omega^2\, y_i) \left(y_i + z_i\, \frac{h}{2} \right) + \mathrm{O}(h^3) \right] + \left[h\omega^2\, (2 y_i + h\, z_i) \left(z_i - y_i\, \frac{h\omega^2}{2} \right) + \mathrm{O}(h^3) \right] \end{aligned}$$

On trouve bien (même s'il a fallu bosser un peu...) que

$$\boxed{\mathrm{E}_{i+1} - \mathrm{E}_i = \mathrm{O}(h^3)}$$

II.C.2.b On observe une trajectoire fermée (à la précision de la représentation graphique), on peut donc affirmer que l'énergie du système correspondant est bien conservée.

II.C.2.c Le schéma de Verlet[4] permet une intégration bien plus précise que le schéma d'Euler pour ce type de problème où la dérivée seconde y'' de la position ne dépend que de la position y et non de la vitesse z.

[4] Le schéma de Verlet correspond au schéma d'intégration « saute-mouton » (ou « leap-frog » en anglais) qui consiste à évaluer les vitesses aux pas de temps demi-entiers de sorte que

$$z_{i+1/2} = z_{i-1/2} + h\, f_i \qquad \text{et} \qquad y_{i+1} = y_i + h\, z_{i+1/2}$$

où l'on a $z_i = \dfrac{z_{i-1/2} + z_{i+1/2}}{2}$. En effet, en combinant l'équation donnant $z_{i+1/2}$ avec celle donnant $z_{i+3/2}$, on trouve comme annoncé que

$$z_{i+1} = z_i + h\, \frac{f_i + f_{i+1}}{2}$$

D'autre part, si $z_i = (z_{i+1/2} + z_{i-1/2})/2$ et que $z_{i-1/2} = z_{i+1/2} - h\, f_i$, on en déduit que $z_i = z_{i+1/2} - h\, f_i/2$, soit bien

$$y_{i+1} = y_i + h \left(z_i + \frac{h\, f_i}{2} \right)$$

III.A.1 La force totale s'exerçant sur la particule j est la somme de toutes les forces s'exerçant sur cette particule, soit

$$\boxed{\overrightarrow{\mathrm{F}_j} = \sum_{k\neq j} \overrightarrow{\mathrm{F}_{k/j}} = \sum_{k\neq j} \mathrm{G}\frac{m_j m_k}{{r_{jk}}^3}\overrightarrow{\mathrm{P}_j\mathrm{P}_k}}$$

III.A.2 Écrivons la fonction `force2` en utilisant les fonctions `smul`, `vsom` et `vdiff` définies précédemment. On rajoute aussi une fonction `norme` qui calcule la norme euclidienne d'un vecteur

```
def norme(v):
    """ Calcule la norme euclidienne d'un vecteur dont on donne ses
    composantes dans une base orthonormée. """
    norme2 = 0                    # Initialisation de la norme carrée
    for i in range(len(v)):       # On passe toutes les composantes
        norme2 = norme2 + v[i]**2# Ajout du carré de la composante
    return norme2**0.5            # Le résultat est la racine carrée de la somme

def force2(m1,p1,m2,p2):
    """ Renvoie une liste représentative de la force exercée par la particule
    2 sur la particule 1. """
    P1P2 = vdif(p2,p1)            # Le vecteur P1P2
    G = 6.67e-11                  # Constante universelle de gravitation
    a = G*m1*m2 / norme(P1P2)**3 # Constante multiplicative devant le vecteur
    return smul(a,P1P2)           # Renvoi du vecteur a*P1P2
```

III.A.3 Pour avoir la force totale, il suffit de boucler sur tous les autres points

```
def forceN(j,m,pos):
    """ Force gravitationnelle totale exercée par toutes les particules sur la
    particule j. """
    force = smul(0,pos[j])  # Initialisation au vecteur nul de bonne taille
    for k in range(len(m)): # On passe toutes les particules en revue
        if k != j:          # Si on n'est pas sur la particule concernée
            f_k_sur_j = force2(m[j],pos[j],m[k],pos[k]) # Force individuelle
            force = vsom(force,f_k_sur_j) # et ajout à la force totale
    return force            # Renvoi de la force totale après sommation
```

III.B.1 `position[i]` et `vitesse[i]` sont deux listes donnant à l'instant $t_i = t_0 + i \times h$ les positions et vitesses de toutes les particules. Ce sont donc des listes contenant N listes (avec N le nombre de particules) à 3 éléments (les trois coordonnées pour chaque vecteur).

III.B.2 Pour calculer les positions suivantes, il faut calculer l'accélération (qui vaut $\overrightarrow{\mathrm{F}_j}/m_j$, attention à ne pas oublier la masse m_j!) et utiliser les vitesses pour l'incrément de position

```
def pos_suiv(m,pos,vit,h):
    """ Calcul de la position suivante connaissant les positions et vitesses à
    l'instant t_i ainsi que le pas de temps h voulu.
    Version où l'on parcourt manuellement les trois dimensions d'espace.
    Attention, l'accélération vaut la force divisée par la masse (on aurait
    mieux fait de calculer les accélérations directement pour économiser
    quelques calculs...). """
```

```
8       L = []                      # Initialisation des nouvelles positions
9       for j in range(len(m)):  # On parcourt les particules une à une
10          mj,pj,vj = m[j], pos[j], vit[j]  # Des raccourcis pour la lisibilité
11          force = forceN(j,m,pos)          # Vecteur force totale sur j
12          next_pj = smul(0,pj)             # Initialisation nouvelle position pour j
13          for k in range(len(pj)):         # Boucle sur les dimensions d'espace
14              next_pj[k] = pj[k] + vj[k]*h + h**2/2 * force[k]/mj  # et Verlet
15          L.append(next_pj)    # Ajout du résultat à la liste
16      return L                 # et renvoi final une fois complètement remplie
```

III.B.3 Pour calculer l'incrément de vitesse, on a besoin des positions suivantes (d'où la question précédente) et faire par deux fois la somme des forces (sur les anciennes et nouvelles positions pour calculer f_i et f_{i+1}). Attention, comme avant, ce dont on a vraiment besoin c'est de l'accélération et non de la force. On fait aussi uniquement la version où l'on parcourt directement les dimensions d'espace car cela devient pénible à faire avec `vsom` et compagnie.

```
1   def etat_suiv(m,pos,vit,h):
2       """ Calcul de l'état suivant (position et vitesse) pour toutes les
3       particules connaissant ces valeurs à la date t_i. """
4       new_pos = pos_suiv(m,pos,vit,h) # On calcule tout de suite les nouvelles positions
5       new_vit = []                    # Liste vide pour les nouvelles vitesses
6       for j in range(len(m)):         # Les particules, une à une
7           mj,vj= m[j],vit[j]          # Raccourcis
8           fi   = smul(1/mj,forceN(j,m,pos))     # Accélération à t_i
9           fip1 = smul(1/mj,forceN(j,m,new_pos)) # Accélération à t_{i+1}
10          next_vj = smul(0,vj)        # Initialisation à la vitesse nulle pour la taille
11          for k in range(len(vj)):    # Boucle sur les dimensions d'espace
12              next_vj[k] = vj[k] + h/2 * (fi[k] + fip1[k]) # Application de Verlet
13          new_vit.append(next_vj)     # Ajout à la liste des vitesses
14      return new_pos,new_vit          # Renvoi des nouvelles positions et nouvelles vites
```

III.B.4.a Visiblement, la relation entre $\ln(\tau_N)$ et $\ln(N)$ est affine du type

$$\ln(\tau_N) = \alpha \ln(N) + \beta$$

ce qui signifie que la durée des calculs suit une loi de puissance du type $\tau_N = \tau_1 N^\alpha$ où α est le coefficient directeur de la droite précédente.

III.B.4.b D'après le graphique, $\ln(\tau_N)$ évolue de 2,5 à 6,5 quand $\ln(N)$ évolue de 6 à 8, soit un coefficient directeur $\alpha = 2$. On peut donc raisonnablement penser que l'algorithme est de complexité quadratique en nombre de particules.

III.B.5.a La fonction `pos_suiv` est déjà de complexité quadratique en nombre de particules puisqu'il faut trouver la position suivante de chacune des N particules en calculant à chaque fois une somme des forces exercées par les $N-1$ autres particules. La complexité de `etat_suiv` ne peut donc être moindre puisqu'elle fait un appel à `pos_suiv`. Néanmoins, la complexité n'est pas non plus d'ordre supérieur puisque l'on fait une boucle sur toutes les particules (ligne 6) et que dans cette boucle, on appelle deux fois une fonction (`forceN`) de complexité linéaire en N. La complexité globale est donc bien quadratique en N.

III.B.5.b Les déterminations empirique et théorique concordent: tout va bien !

IV.A On demande vraisemblablement les masses de tous les corps présents dans la base:

```
SELECT masse FROM corps
```

IV.B.1 Supposons, comme le suggère l'énoncé, qu'il existe une fonction SQL `tmin()` qui donne[5] la date à partir de laquelle on souhaite lancer la simulation. On veut donc compter le nombre de corps distincts qui ont une entrée dans la table avant la date `tmin()`. On peut faire ainsi

```
SELECT COUNT(DISTINCT id_corps) FROM etat WHERE datem < tmin()
```

IV.B.2 Il s'agit d'utiliser une fonction d'agrégation qui donne le plus grand (`MAX(datem)`) des temps inférieurs à `tmin()` pour chaque corps (`GROUP BY`)

```
SELECT id_corps,MAX(datem) FROM etat WHERE datem < tmin() GROUP BY id_corps
```

IV.B.3 Il va falloir passer par deux jointures puisque l'attribut `masse` se trouve seulement dans la table `corps`, que les positions et vitesses se trouvent dans la table `etat` et qu'il faut utiliser la table `date_mesure` pour trouver la date correspondante dans la table `etat`. Attention, la jointure entre `date_mesure` et `etat` doit porter sur deux colonnes car deux corps peuvent avoir été mesurés au même moment (par exemple s'il sont spatialement proches et donc sur une même prise de vue). On suppose que la distance au centre attendue est euclidienne[6].

```
SELECT masse,x,y,z,vx,vy,vz FROM corps AS c
       JOIN etat AS e ON c.id_corps = e.id_corps
       JOIN date_mesure AS d ON (datem = date_der AND e.id_corps = d.id_corps)
       WHERE masse > masse_min() AND ABS(x) < arete()/2
          AND ABS(y) < arete()/2 AND ABS(z) < arete()/2
       ORDER BY x*x+y*y+z*z
```

À noter que `ABS()` n'existe pas sur tous les systèmes de gestion de base de données. Dans le doute, on pourrait remplacer `ABS(x)<arete()/2` par `x > -arete()/2 AND x < arete()/2` et de même pour y et z.

IV.C Il ne reste plus qu'à écrire la boucle globale et ne pas oublier de convertir les données dans les bonnes unités (mètres et mètres par seconde plutôt qu'unités astronomiques et kilomètres par seconde[7])

```
1  def simulation_verlet(deltat,n):
2      """ Simulation globale avec un pas de temps deltat (il a fini par
3      sortir du bois :o) et un nombre n de pas de temps à effectuer."""
4      pos = [smul(1.5e11,p) for p in p0]  # Conversion des UA en m
5      vit = [smul(1e3,v)    for v in v0]  # Conversion des km/s en m/s
6      liste_positions = [pos]# Initialisation de la liste à renvoyer
7      for i in range(n):     # Autant de fois que demandé,
8          pos,vit = etat_suiv(m,pos,vit,deltat)      # on détermine l'état suivant
9          to_append = [smul(1/1.5e11,p) for p in pos]# Conversion inverse
10         liste_positions.append(to_append)          # et ajout à la liste
11     return liste_positions
```

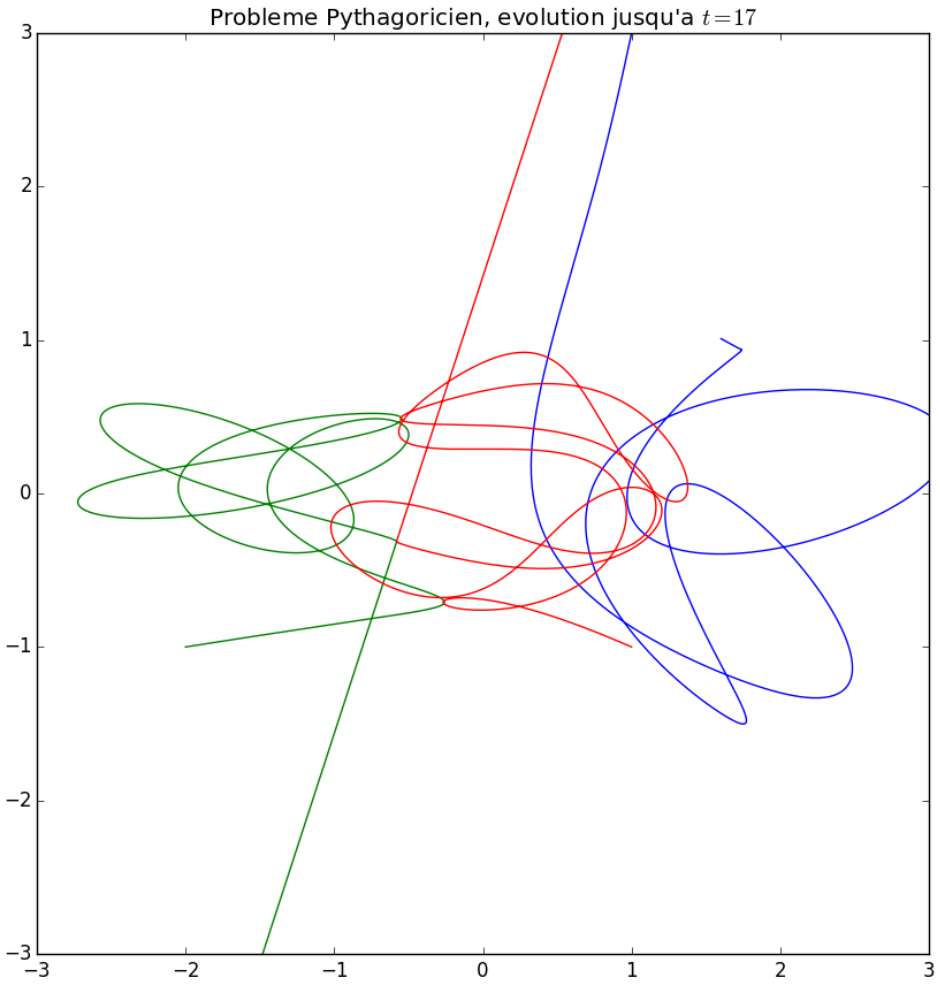
[5] Un peu comme `NOW()` donne la date actuelle avec la requête `SELECT NOW()`.
[6] Mais on se demande alors pourquoi on n'a pas choisi une sphère plutôt qu'un cube...
[7] Fort heureusement les masses sont déjà en kilogrammes et non en grammes comme c'est parfois l'usage en astro

Pour finir, vous trouverez à l'adresse

https://github.com/jjfPCSI1/py4phys/blob/master/lib/M_schema_de_verlet.py

un programme qui utilise toutes les fonctions définies dans ce corrigé pour essayer de reproduire les calculs du problème Pythagoricien[8]. Il a permis de produire l'image suivante ainsi qu'un petit film que l'on peut retrouver là

http://pcsi.kleber.free.fr/IPT/doc/M_schema_de_verlet.mp4



Le programme a été adapté dans un système d'unité où G vaut 1 pour pouvoir se comparer aux graphiques fournis dans le papier. Et mis à part l'explosion aux alentours de $t = 16$ du fait d'un approche trop serrée des étoiles rouge et verte[9], l'accord est plutôt bon pour un pas de temps raisonnable (`1e-4`).

Voici une autre version pour un affichage 3D:

https://github.com/jjfPCSI1/py4phys/blob/master/lib/M_schema_de_verlet3D.py

On y observe notamment une formation de binaire (vers $t = 27$), destruction de la binaire ($t = 31$) du fait d'un triangle amoureux et reformation de la binaire initiale ($t = 34$) qui va durer jusqu'à la fin de la simulation ($t = 57$).

[8] Voir un papier de Szebehely et Peters de 1967 sur le sujet http://articles.adsabs.harvard.edu/cgi-bin/nph-iarticle_query?1967AJ.....72..876S&data_type=PDF_HIGH&whole_paper=YES&type=PRINTER&filetype=.pdf

[9] On n'échappe pas à son destin si l'on n'a pas un pas de temps adaptatif ou une régularisation en cas d'interaction à trop courte distance.