Simulations sur Python séance 1/2

Le but de ce TP est de réaliser des simulations sur diverses situations physiques en employant le logiciel de programmation Python. L'environnement de travail choisi est Spyder. Il offre une présentation commode, avec une console de saisie, des icônes d'exécution, etc...

Des indications sont fournies pour guider la rédaction des programmes répondant aux questions traitées. Des fichiers de correction sont disponibles sur le réseau dans le dossier PCSI.

Introduction :

Au-delà de la programmation fondamentale, il est possible d'employer avec Python des fonctions correspondant à des procédures compilées dans des bibliothèques téléchargeables (nommées aussi modules). On retiendra parmi ceux-ci :

matplotlib, utile pour le tracé de graphes, en particulier la sous-bibliothèque matplotlib.pyplot...

numpy, permettant le calcul numérique,

scipy, dévolue au calcul scientifique (intégration d'équation différentielles, traitement de signal...)

Les modules doivent être chargés préalablement à l'appel d'une fonction correspondante. Il est d'usage de renommer un module, par exemple pour le module numpy, selon la commande : import numpy as np. L'appel d'une fonction contenue dans le module numpy se fera alors par : np.fonction

On peut importer toutes les fonctions d'un module donné (par exemple scipy) par la commande : from scipy import *

Il peut être plus léger de n'importer qu'une seule fonction, si elle est la seule employée, selon la commande :

from scipy.integrate import odeint

La fonction odeint est ainsi chargée à partir du sous module scipy.integrate qui est dévolu à l'intégration numérique.

L'environnement de travail spyder charge par défaut les modules numpy, scipy et matplotlib.

1. Rappel : listes et tableaux .

Les programmes envisagés utiliseront des tableaux numériques (array), en employant le bibliothèque numpy.
Il importe de distinguer les listes des tableaux.

Une **liste** est une suite ordonnée d'éléments. Une liste peut être créée par saisie directe, ou constituée en parcourant une boucle (boucle for ou boucle while).

La ligne de code liste = Liste1 + liste2 amène non pas une addition, mais une concaténation des deux listes.

Le calcul à partir des éléments d'une liste constituée, ou entre des éléments de deux listes, ne peut s'envisager qu'en écrivant une boucle dans laquelle on appellera successivement les éléments de la liste, les résultats du calcul étant accumulés dans une liste dédiée, par la commande de forme : liste_resultat.append(resultat).

Au contraire, le calcul à partir des éléments d'un **tableau** (en anglais : **array**), ou en combinant les éléments de deux tableaux, pourra se réaliser directement sur les tableaux (pourvu qu'ils comportent le même nombre d'éléments).

Un exemple simple est proposé dans le fichier « distinction listes_tableaux ».

La création d'un tableau à partir d'une liste s'obtient par la commande : A = np.array([#éléments]).

On peut créer un tableau vide, c'est-à-dire dont les éléments sont nuls, par A = np.zeros(n) où n est le nombre d'éléments, pour un tableau d'une seule ligne. Un tableau de p lignes et q colonnes est appelé par np.zeros((p,q)). Un tableau peut bien sûr être rempli, ou modifié, à partir d'un processus en boucle « for ».

2. Tracé de graphes :

Le tracé du graphe d'une fonction f(x) sur un intervalle [$x_{min,}$ x_{max}] de sa variable est aisément réalisable à partir de la commande plot, présente dans le module matplotlib.pyplot.
Le tracé n'est obtenu qu'à partir d'un échantillonnage de valeurs numériques, défini à partir de la commande linspace, ou de la commande arange, présentes dans le module numpy.
D'une façon plus générale, l'étude de fonctions continues se fera toujours par des processus de discrétisation qui vont mettre en jeu un certain pas d'incrémentation, susceptible de poser des difficultés en cas de choix erroné.

Les valeurs incrémentées de la variable x seront produites sous forme d'un tableau (array) à partir de la commande : x = np.arange(x_initial, x_finale, pas) en choisissant des valeurs pertinentes pour les trois paramètres.

Il est aussi possible d'employer la commande np.linspace(x_initial, x_final, nombre) où le dernier paramètre désigne le nombre d'intervalles à séquencer entre x_initial et x_final.

Le ***théorème de Shannon*** stipule que pour discrétiser une fonction f(x) de période T, il faut la découper avec des intervalles Δx de discrétisation de longueur (nettement) inférieure à T/2 ; soit encore une fréquence d'échantillonnage au moins égale au double de la fréquence fondamentale du signal f(x). Si cette condition n'est pas respectée, le tracé sera aberrant quel que soit l'algorithme employé.

Taper la séquence suivante, observer le résultat, puis recommencer pour des valeurs de numpoints de 100 ; 200, et 1000. Conclure.

```
from numpy import *
import matplotlib.pyplot as plt

numpoints = 25
x = linspace(0,100*pi,numpoints)
plt.clf()#efface une figure tracée précédemment
plt.plot(x,sin(x))
plt.title("sin(x) ; nombre de points = "+str(numpoints))
plt.ylim(-1,1)
plt.show()
```

3. Equation différentielle du premier ordre.

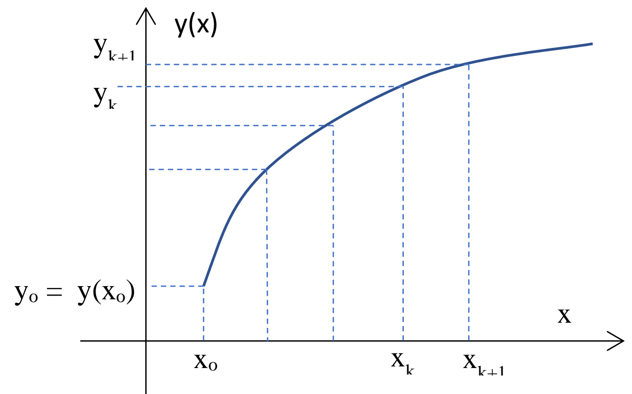
La résolution de ce type d'équation va être abordée selon deux procédés.

3.1 La méthode d'Euler.

La méthode d'Euler est une méthode numérique simple permettant l'intégration numérique d'équation différentielles par un processus itératif.
Une équation différentielle d'ordre 1 s'écrit formellement : y' = f(y(x), x) où y' = dy/dx est la dérivée de la grandeur y par rapport à la variable x et f est une fonction portant sur y ou x.

La méthode d'Euler consiste à remplacer la relation différentielle portant sur des fonctions continues de la variable x par une relation itérative sur des grandeurs discrètes ; la variable x n'est plus continue, mais incrémentée selon un pas Δx bien choisi.

A partir de valeurs initiales connues, de valeurs x_o et y_o = $y(x_o)$, elle permettra de déduire de proche en proche les valeurs successives prises par y pour les valeurs incrémentées $x_k = x_o + k.\Delta x$.



L'équation différentielle donne une expression de la variation différentielle :

$$dy = f(y(x),x).dx \qquad (1)$$

La discrétisation du problème consiste à remplacer dx par le pas Δx.
La variation infinitésimale $dy = y(x + dx) - y(x)$
devient alors une petite variation entre deux valeurs successives de x : $y(x_{k+1}) - y(x_k)$

La relation (1) se transcrit alors par : $y(x_{k+1}) - y(x_k) = f(y(x_k), x_k).\Delta x$

On peut donc déduire $y(x_{k+1})$ à partir de la connaissance des valeurs antérieures (y_k, x_k) : $y(x_{k+1}) = y(x_k) + f(y(x_k), x_k).\Delta x$

Application : Ski de vitesse (Kilomètre Lancé)

Dans cette épreuve, le skieur, doté d'un équipement spécial, prend de la vitesse en descendant une pente forte selon une trajectoire rectiligne. Les vitesses atteintes sont considérables… Heureusement, la descente est réalisée sur une piste parfaitement dégagée.

L'application de la RFD, prenant en compte le poids du skieur, la force de réaction de la piste et le frottement de l'air conduit, en projection sur l'axe correspondant à la ligne de pente, à l'équation différentielle :

$$\frac{dv(t)}{dt} = g.\sin\alpha - \frac{k}{m}.v(t)^2$$

(1) avec $g = 9{,}81\ m.s^{-2}$.

L'équation précédente traduit un modèle avec force de frottement quadratique : $F = k.v^2$ avec $k = (1/2).\rho.S.C_x$
On rassemble ci-dessous les paramètres pris en compte pour le calcul du coefficient de frottement k.

pente (°)	pente (rad)	masse (kg)	Cx	masse vol. de l'air ρ ($kg.m^{-3}$)*	surface S (m^2)
30	0,523	70	0,35	1,008	0,55

* conditions : P = 0,8 bar ; T = 273 K = 0°C.

Ces valeurs conduisent à $k = 9{,}7 \times 10^{-2}$ usi et $g.\sin\alpha = 4{,}9\ m.s^{-2}$. La vitesse atteinte en une vingtaine de seconde sera de l'ordre de $200\ km.h^{-1}$.

En utilisant la méthode d'Euler, Intégrer numériquement l'équation (1) sur une durée pertinente et avec un choix convenable pour le pas d'incrémentation Δt. Tracer les graphes traduisant l'évolution de la vitesse v(t).

3.2 La commande odeint sur la bibliothèque scipy.integrate.

La bibliothèque scipy comporte de nombreuses commandes exploitables pour le traitement numérique. En particulier la sous-bibliothèques scipy.integrate, dévolue à l'intégration, comporte la commande odeint qui permet l'intégration numérique d'équations différentielles.

La commande odeint fait appel à un algorithme d'intégration numérique par itération, relativement analogue à la méthode d'Euler, mais offrant une meilleure convergence vers la solution exacte. Pour un même nombre de point de calcul, les résultats seront donc meilleurs, et ne présenteront pas de divergence comme cela peut être le cas avec la méthode d'Euler si le pas d'incrémentation est trop grand.

Pour pouvoir utiliser la commande odeint, les équations doivent pouvoir s'expliciter comme un système d'équations différentielles du premier ordre, afin de relier le taux de variation des grandeurs à une expression portant sur les variables du problème. Elles doivent être assorties d'un jeu de conditions initiales.

La syntaxe de cette commande est relativement simple. Envisageons d'abord le cas d'une équation différentielle du premier ordre portant sur une fonction y(t), de forme : dy/dt = F(y, t)

Où F(y, t) est une fonction de y(t) et de t exprimant la relation entre la dérivée dy/dt et la fonction y(t) ainsi que la variable t.

Définir d'abord la fonction F :

```
def F(y,t) ;
        dy_dt = …
        return dy_dt
```

Déterminer un intervalle de temps t, avec un pas d'incrémentation pertinent commandé par le nombre n d'intervalles, en créant un tableau de valeurs pour t : t = np.linspace($t_{initial}$, t_{final}, n)

Appeler la commande odeint selon la syntaxe :

```
Y = odeint(F, y0, t)
```

F fait référence à la fonction définie plus haut, t est la liste des instants pour lesquels le calcul de y(t) est demandé et Y sera la liste des valeurs solution. La valeur initiale de y(t) est y0.

La liste des instants t est fournie sous la forme d'un tableau (array) obtenu par la commande

t = np.linspace($t_{initial}$, t_{final}, n)

où $t_{initial}$ et t_{final} déterminent l'intervalle de temps étudié et n le nombre de points calculés sur cet intervalle.

Le graphe de la solution s'obtient sans difficultés :

```
plt.clf()  #efface un graphe antérieur
plt.plot(t,Y)
```

En utilisant la commande odeint, reprendre l'exemple précédemment étudié et comparer les résultats obtenus.

4. Résolution d'une équation différentielle du second ordre.

Le principe d'une telle résolution est de réécrire l'équation sous la forme d'un système d'équations différentielles du premier ordre.

L'exemple ci-dessous porte sur la situation très simple d'une chute verticale sans frottement dans le champ de pesanteur uniforme. Elle est décrite par l'équation différentielle : $\ddot{z} = -g$ que l'on va réécrire comme le **système d'équations différentielles du premier ordre** : $\{\dot{v} = -g\ ,\ v = \dot{z}\}$.

Du point de vue syntaxique, ceci est réalisé en créant des listes faisant intervenir les coordonnées de position et de vitesse du mobile, et en faisant jouer la commande odeint sur une équation portant sur ces listes selon les modalités précédentes.

```
#résoudre une ED du second ordre, exemple : chute libre verticale d'une hauteur de 10 mètres
from numpy import *
import matplotlib.pyplot as plt
from scipy.integrate import odeint
plt.clf()

# création d'un vecteur d'état [position,vitesse],  coords = [z,vz] et expression de la relation
# différentielle associant dz/dt à vz et dvz/dt à az (ici az = cste = -9.81).
def equadiff(coords,t):
    z = coords[0]
    vz = coords[1]
    az = -9.81
    return[vz,az]

#définition des conditions initiales
z0 = 10
vz0 = 0
coords0 = [z0,vz0]

t = linspace(0,2,100)

#résolution du système et tracé de la solution
sol = odeint(equadiff,coords0,t)
z = sol[:,0]
vz = sol [:,1]
plt.clf()  #efface un éventuel tracé précédent
plt.plot(t,z)
plt.show()
```

5. Tir d'un projectile dans le champ de pesanteur avec frottement ou non.

Le propos de cette activité est de procéder à l'intégration numérique conduisant au tracé de la trajectoire.

Le champ de pesanteur est supposé uniforme, de module g = 9,81 m.s^{-2}.

5.1 Tir sans frottement.

Considérons d'abord le cas d'un tir « dans le vide », c'est à dire en négligeant tout frottement.

On envisage un volant de badminton, envoyée du fond du court avec une vitesse initiale de 30 m.s^{-1} avec un angle de tir α = 25° = 0,44 rad par rapport à l'horizontale (vers le haut !), à partir d'une hauteur de 1,0 m.

L'équation du mouvement est obtenue à partir de la Relation Fondamentale de la Dynamique et s'écrit en projection sur les axes (Ox), axe horizontal et (Oz), axe vertical ascendant :

$$\{\ddot{x} = 0 \,;\ \ddot{z} = -9{,}81\}$$

Les conditions initiales sont déterminées par une position initiale à l'origine du repère :

$$\{x(0) = 0 \,; z(0) = 0\}$$

et une vitesse initiale dont les coordonnées sont :

$$\{\dot{x}(0) = v_o.cos\alpha \,; \dot{z}(0) = v_o.sin\alpha\}$$

Ce système est relativement simple à intégrer, puisqu'il se présente sous forme de deux équations différentielles d'ordre deux, indépendantes, assorties de leurs conditions initiales.

L'outils d'intégration employé sera la fonction odeint, présente dans le sous-module scipy.integrate. En s'inspirant du programme proposé au chapitre précédent, intégrer numériquement les équations du mouvement et procéder au tracé de la trajectoire.

5.2 Tir avec frottement quadratique.

Le projectile est lancé avec des conditions initiales de position et vitesse identiques. On veut prendre en maintenant en compte le frottement selon un modèle quadratique.

Ce problème mène à un système d'équations différentielles n'admettant pas de solution analytique. Il ne pourra être intégré que numériquement.

L'équation du mouvement obtenue par projection de la Relation Fondamentale de la Dynamique sur les axes (Ox) et (Oz), s'écrit :

$$\left\{\ddot{x} = -\frac{k}{m}.v.\dot{x};\ \ddot{z} = -9{,}81 - \frac{k}{m}.v.\dot{z}\right\}$$

Vues les conditions, la valeur numérique du rapport k/m sera : k/m = 0,15 usi.

C'est un système de deux équations différentielles non linéaires, liées entre elles puisque le module v de la vitesse intervient dans chacune d'elles, ce module dépendant des dérivées premières de x(t) et de z(t) :

$$v = \sqrt{\dot{x}^2 + \dot{z}^2}$$

Les conditions initiales, identiques au cas précédent, sont déterminées par une position initiale à l'origine du repère :

$$\{x(0) = 0 \,; z(0) = 0\}$$

et une vitesse initiale dont les coordonnées sont :

$$\{\dot{x}(0) = v_o.cos\alpha \,; \dot{z}(0) = v_o.sin\alpha\}$$

Le processus d'intégration doit prendre en compte le caractère lié des deux équations différentielles, en travaillant sur un vecteur à quatre coordonnées, coords = [x, vx, z, vz].

Une solution complètement rédigée est proposée en annexe ; s'y reporter en cas de besoin. Il est évidemment préférable de rédiger soit même le programme souhaité, le report à la solution n'étant à exercer qu'en dernier recours !

Il sera pertinent, pour faire comparaison, de tracer sur un même graphe les trajectoires calculées en présence ou en absence de frottement.

6. Mouvement newtonien.

On examine les trajectoires obtenues pour un objet céleste placé dans un potentiel newtonien, c'est à dire soumis ici à un champ de gravitation produit par un centre attractif fixe.

On considère l'étude dans le référentiel héliocentrique, le centre attractif étant constitué par le Soleil de masse M = $2{,}00.10^{30}$ kg. Le centre su Soleil placé à l'origine O (0,0) d'un repère cartésien (O, x, y). Le mobile de masse m (ce peut être la Terre) part du point P_o = (150 millions de km, 0) avec une vitesse initiale de coordonnées (0, v_o). La constante universelle de gravitation vaut G = $6{,}67.10^{-11}$ Nm²kg^{-2}. Le produit GM vaut $1{,}33.10^{20}$ usi.
On envisagera 4 cas :

- v_o = racine(GM/R) = 29,7 km/s : trajectoire circulaire ;
- v_o = 35,0 km/s : trajectoire elliptique ;
- v_o = 29,7*racine(2) = 42,0 km/s : trajectoire parabolique ;
- v_o = 45,0 km/s : trajectoire hyperbolique.

La durée d'étude est de 6.10^7 s soit environ deux ans

Mise en équation :

Le mobile de position P et de masse m est soumis uniquement à la gravitation solaire :

$$\vec{F} = \frac{-GMm}{r^2}\overrightarrow{e_r}$$

Où r = OP est la distance au soleil et $\overrightarrow{e_r}$ est l'unitaire radial : $\overrightarrow{e_r} = \overrightarrow{OP}$/r.

La deuxième loi de Newton donne accès à l'accélération $\vec{a}$ du mobile par :

$$m\vec{a} = \frac{-GMm}{r^2}\overrightarrow{e_r}$$

Soit en simplifiant par m et en explicitant la relation sur une base cartésienne où :

$$\overrightarrow{OP} = x\,\overrightarrow{e_x} + y\,\overrightarrow{e_y} \qquad et \quad r = \sqrt{x^2+y^2}$$

On obtient :

$$\ddot{x}\,\overrightarrow{e_x} + \ddot{y}\,\overrightarrow{e_y} = \frac{-GM}{r^3}\overrightarrow{OP}$$

Soit selon $\overrightarrow{e_x}$: $\ddot{x} = -GM.(x^2+y^2)^{-3/2}.x$

et selon $\overrightarrow{e_y}$: $\ddot{y} = -GM.(x^2+y^2)^{-3/2}.y$

On a donc un système de deux équations différentielles non linéaires du second ordre, liées.

On se ramène à un système d'équations différentielles du premier ordre en introduisant les coordonnées du vecteur vitesse. $\vec{v} = v_x\,\overrightarrow{e_x} + v_y\,\overrightarrow{e_y}$. D'où le système :

$$\{\, v_x = \dot{x}\ ;\ \dot{v_x} = -GM.(x^2+y^2)^{-3/2}.x\ ;\ v_y = \dot{y}\ ;\ \dot{v_y} = -GM.(x^2+y^2)^{-3/2}.x\ \}$$

Etude par simulation numérique.

En employant la méthode de votre choix (méthode d'Euler ou commande odeint), obtenir le tracé des trajectoires correspondant aux quatre cas envisagés.

Le pas d'incrémentation conseillé est de $\Delta t = 10^4$ s pour une durée totale de 6.10^7 s.

Annexe :

Tir avec frottement quadratique.

```
from math import *
from scipy import *
from pylab import *
from matplotlib import *
from scipy.integrate import odeint

t = linspace(0,2,101)
alpha = 0.15
CI = array([0,30*cos(18*pi/180),1,30*sin(18*pi/180)])

def syst(coords,t):
    x = coords[0]
    vx = coords[1]
    z = coords[2]
    vz = coords[3]
    ax = -alpha*vx*sqrt(vx*vx+vz*vz)
    az = -9.81-alpha*vz*sqrt(vx*vx+vz*vz)
    return[vx,ax,vz,az]

sols = odeint(syst,CI,t)
x = sols[:,0]
z = sols[:,2]
vx = sols[:,1]
vz = sols[:,3]

plt.clf()  #efface un éventuel tracé précédent

ylim(0,6)
plot(x,z)
show()
```