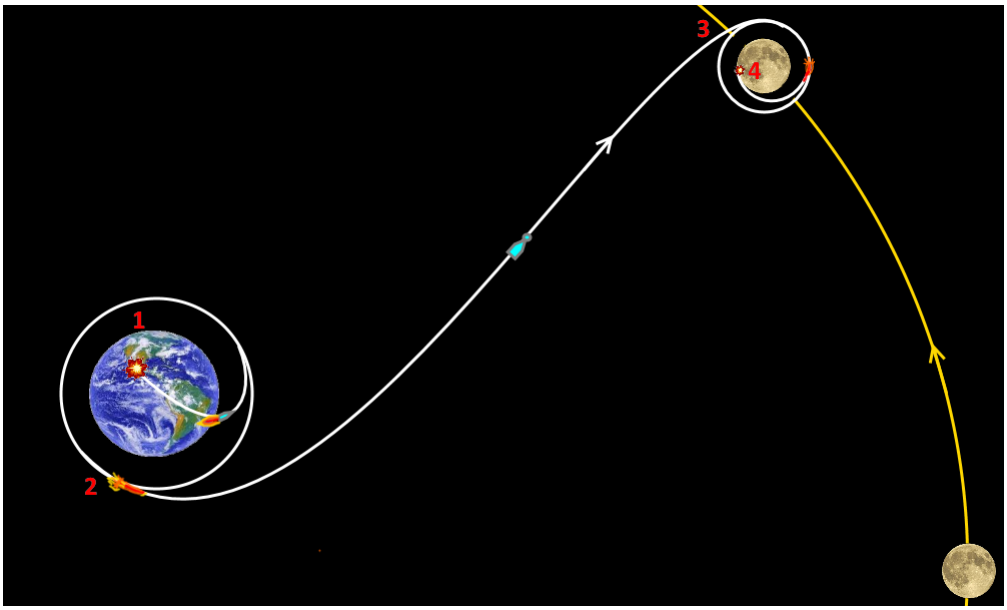
UNIVERSITÉ DE FRANCHE-COMTÉ

UFR SCIENCES & TECHNIQUES

Modélisation de la trajectoire d'un objet naturel ou artificiel au voisinage d'une planète

Maxime Sabbadini

Projet intégrateur numérique
CMI PICS, 2ème année
Année universitaire 2018-2019



Enseignant référent : **Benoît Noyelles**

Institut UTINAM

Introduction

Le projet intégrateur numérique est un projet proposé aux étudiants de Licence 2 et visant à les familiariser avec la programmation, mais également aux algorithmes d'intégration numérique, qui sont maintenant prépondérants et il est indispensable de savoir les maitriser pour leur carrière professionnelle.

Étant tout particulièrement intéressé par les lanceurs spatiaux et leur façon de mettre en orbite des satellites, j'ai donc décidé de modéliser la trajectoire d'objets aux alentours d'une planète. La totalité du code sera présentée en Annexe. Pour toute question ou remarque, vous pouvez me contacter à l'adresse suivante : maxime.sabbadini@edu.univ-fcomte.fr.

1 Projet

1.1 Naissance de l'idée

Au tout début du projet, j'avais en tête de modéliser la trajectoire d'une fusée, de son départ sur Terre jusqu'à sa mise en orbite. Cependant, ceci aurait été trop difficile à mettre en équation du fait des nombreuses poussées nécessaires à la bonne mise en orbite, mais également à cause des frottements de l'air sur l'engin, non négligeables dans ce cas. J'ai donc décidé de tenter de modéliser la trajectoire d'un objet, que celui-ci soit naturel ou artificiel, aux alentours d'une planète, de masse nettement supérieure à celle de l'objet.

1.2 Connaissances acquises

Tout d'abord, au delà des techniques d'intégration numérique, que nous aborderons plus tard, nous devons nous intéresser aux équations qui régissent le mouvement d'un objet aux alentours d'une planète. Nous savons que la force d'attraction gravitationnelle qu'exerce un corps sur l'autre s'écrit :

$$\vec{F}_{1\to 2} = -G\frac{m_1 m_2}{r^2}\vec{u}_{1\to 2}. \tag{1}$$

Avec m_1, m_2 les masses des corps, r la distance les séparant, G la constante de gravitation universelle et $\vec{u}_{1\to 2}$ un vecteur unitaire dans la direction de la droite reliant les centres de gravité des deux corps.
Pour établir les équations du mouvement, imaginons le système suivant :

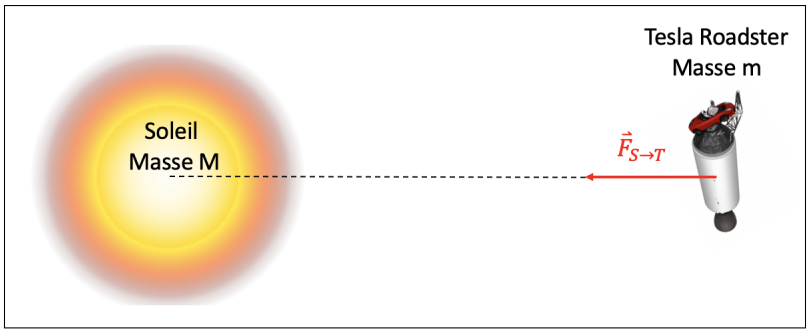


FIGURE 1 – Système considéré pour l'établissement de l'équation du mouvement

Application du PFD à la Tesla Roadster :

$$\sum \vec{F}_{ext} = m\vec{a}$$

$$\vec{F}_{S\to T} = m\vec{a}$$

Avec $\vec{a}$ le vecteur accélération.
Avec l'équation (1), on obtient :

$$\vec{a} = -\frac{GM}{r^3}\vec{r}. \tag{2}$$

Nous voyons avec l'équation (2) que pour obtenir la position, nous devrons l'intégrer deux fois. Or on sait que l'intégration se fait à une constante près, il nous faut deux conditions initiales : la vitesse et la position.

Maintenant que nous avons connaissance de l'équation que nous devons intégrer, nous devons nous intéresser aux techniques d'intégration numérique. Au cours du projet, j'ai étudié principalement deux techniques d'intégration : les méthodes d'Euler et de Runge-Kutta.

— **Méthode d'Euler**

La méthode d'intégration d'Euler est une méthode dite d'ordre 1, c'est-à-dire qu'elle approxime la courbe entre deux points aux temps t et $t + dt$ par la tangente à la courbe. Cette méthode repose sur les équations (3) :

$$\begin{cases} \vec{r}(t + \Delta t) &= \vec{r}(t) + \vec{v}(t)\Delta t \\ \vec{v}(t + \Delta t) &= \vec{v}(t) + \vec{a}(t)\Delta t \end{cases} \tag{3}$$

Pour mieux comprendre, on peut se référer à la figure 2.

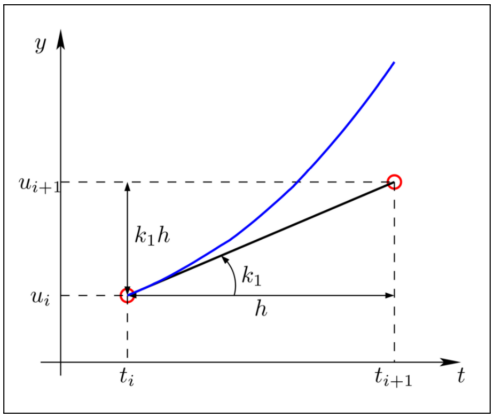


FIGURE 2 – Illustration de la méthode d'Euler (Référence [1])

Nous voyons ci-dessus (figure 2) que la courbe bleue est approximée par la courbe noire entre les temps t_i et t_{i+1}. Cette méthode montre ses limites : si l'intervalle entre deux points est trop grand, la solution numérique diverge beaucoup de la solution analytique. Au contraire, plus cet intervalle est petit, plus la solution numérique se rapprochera de la solution analytique. Cependant, cela demandera plus de ressources de calcul à l'ordinateur. Il faut donc trouver un compromis pour avoir la trajectoire la plus proche de la réalité possible sans surcharger de calculs notre machine.

— **Méthode de Runge-Kutta d'ordre 4 (RK4)**

La méthode de RK4, comme son nom l'indique, est une méthode d'intégration numérique d'ordre 4, très utilisée en informatique. Celle-ci, contrairement à la méthode d'Euler, ne calcule pas uniquement la tangente (ou dérivée) à la trajectoire au temps t_i pour en déduire la position au temps t_{i+1}. Cette méthode calcule la dérivée en 4 points de l'intervalle pour avoir une plus grande précision. Pour illustrer ceci, nous trouverons ci-dessous (figure 3) le schéma montrant les corrections effectuées pour RK2 (j'ai choisi de montrer le fonctionnement de RK2 pour que cela soit plus facile à comprendre, le fonctionnement de RK4 étant similaire).

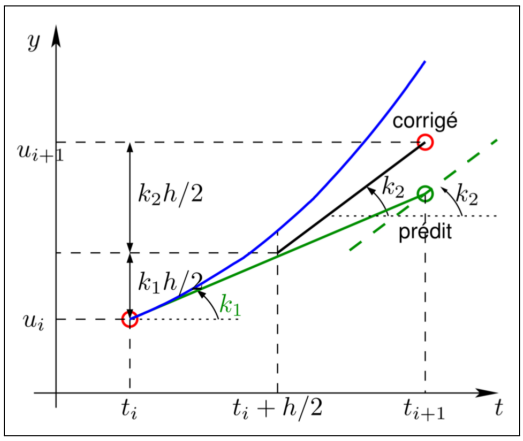


FIGURE 3 – Illustration de la méthode de RK2 (Référence [1])

1.3 Le code

Dans cette partie, on s'intéressera aux parties les plus importantes du code. La première partie est la fonction permettant de calculer la trajectoire par le biais de la méthode d'Euler. Il est important de noter qu'ici, je présente la méthode d'Euler. Cependant, celle-ci ne sera pas utilisée pour la suite, j'ai décidé de la présenter tout de même car son fonctionnement simple permet de mieux appréhender le principe d'intégration numérique :

```
1  r=np.zeros((n, 3))        #Tableau pour la methode d'Euler
2  v=np.zeros((n, 3))
3
4  ##Conditions Initiales
5
6  v[0,0]=v0x          #m/s
7  v[0,1]=v0y
8  v[0,2]=v0z
9  r[0,0]=x0             #m
10 r[0,1]=y0
11 r[0,2]=z0
12
13 def trajectoire_euler(r, v, ti, tf, dt, n):
14
15     v_x=v[0,0]
16     v_y=v[0,1]
17     v_z=v[0,2]
18     x=r[0,0]
19     y=r[0,1]
20     z=r[0,2]
21
22
23     for i in range (1,n):
24         re=np.sqrt(x**2+y**2+z**2)
25
26         x+=dt*v_x
27         y+=dt*v_y
28         z+=dt*v_z
29
30         r[i,0]=x
31         r[i,1]=y
32         r[i,2]=z
33
34         v_x+=dt*(-G*M/re**3*x)
35         v_y+=dt*(-G*M/re**3*y)
36         v_z+=dt*(-G*M/re**3*z)
37
38         v[i,0]=v_x
39         v[i,1]=v_y
40         v[i,2]=v_z
41
42     return v, r
```

Listing 1 – Fonction pour la méthode d'Euler

Les lignes 1 et 2 servent à créer des tableaux vides (de dimensions (n, 3)) dans lesquels on stockera nos vitesses et positions. Les lignes 6 à 11 nous permettent de déclarer nos conditions initiales sur la vitesse et la position qui sont nécessaires pour l'intégration (cf partie 1.2). Les lignes suivantes sont une simple implémentation des équations (3) en 3 dimensions.

On peut maintenant s'intéresser à la partie la plus importante : l'intégration par la méthode de RK4.

```python
def f(r, v):
    x = r[0]
    y = r[1]
    z = r[2]
    re=np.sqrt(x**2+y**2+z**2)          #Definition du vecteur a 6 composantes
    vect = np.zeros(6)
    vect[0] = v[0]
    vect[1] = v[1]
    vect[2] = v[2]
    vect[3] = -G*M/re**3*x
    vect[4] = -G*M/re**3*y
    vect[5] = -G*M/re**3*z
    return vect

def rk4(r, v, dt, n):
    for i in range(0,n-1):

        k1 = f(r[:,i],v[:,i])                                #Integration par RK4
        k2 = f(r[:,i]+ k1[0:3]*dt*0.5,v[:,i]+ k1[3:6]*dt*0.5)
        k3 = f(r[:,i]+ k2[0:3]*dt*0.5,v[:,i]+ k2[3:6]*dt*0.5)
        k4 = f(r[:,i]+ k3[0:3]*dt,v[:,i]+ k3[3:6]*dt)
        k = (k1 + 2*k2 + 2*k3 + k4)/6
        r[:,i+1] = r[:,i] + dt*k[0:3]
        v[:,i+1] = v[:,i] + dt*k[3:6]

    return r
```

Listing 2 – Fonction pour la méthode de RK4

On voit que l'intégration avec RK4 se divise en deux parties :

— On définit tout d'abord un vecteur à 6 composantes, à savoir les vitesses et accélérations selon les 3 axes, ce qui a pour but de faciliter l'intégration.

— Ensuite, on définit nos quatre coefficients, comme décrit dans la partie 1.2, ce qui nous permet de construire notre vecteur position.

Enfin le code est bâti dans un système de coordonnées cartésiennes, centrées sur le corps central, comme on peut le voir dans le schéma suivant (figure 4) :

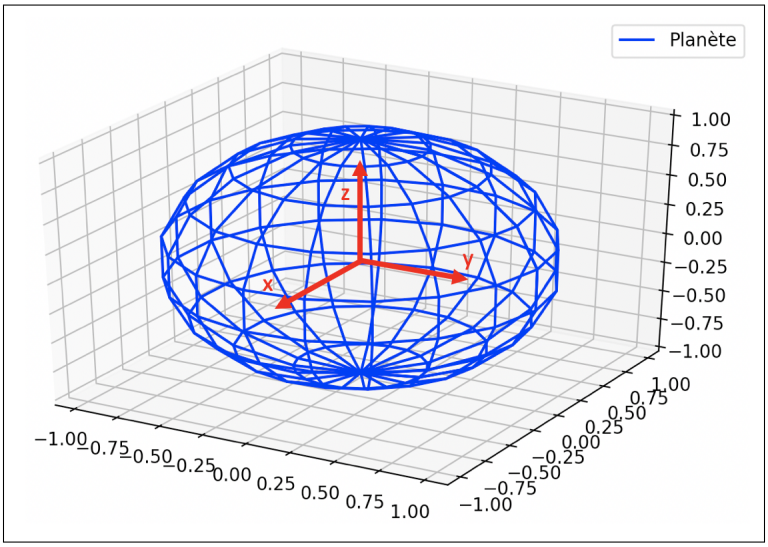
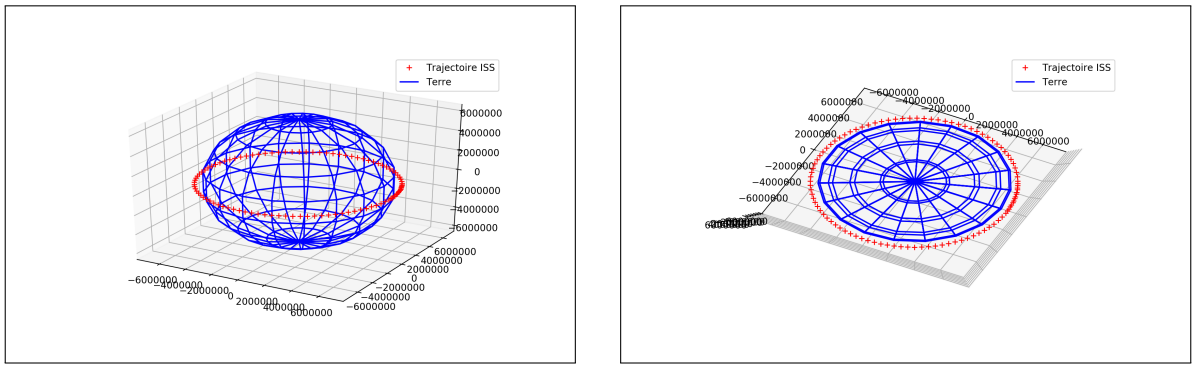


FIGURE 4 – Système de coordonnées du code

1.4 Quelques exemples

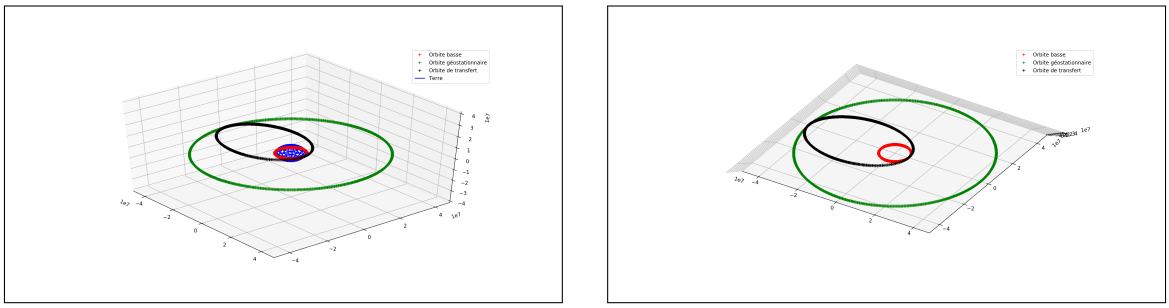
Avec le programme, nous pouvons ainsi tracer la trajectoire de l'ISS [1] autour de la Terre :



(a) Vue en perspective (b) Vue du dessus

FIGURE 5 – Trajectoire de l'ISS autour de la Terre

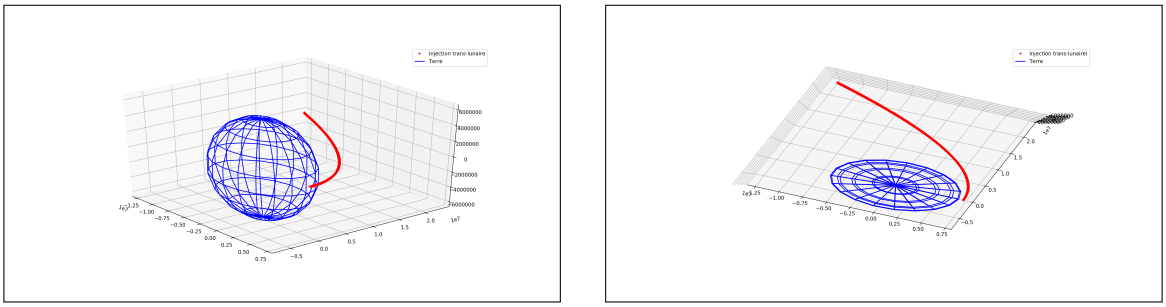
Nous pouvons également tracer les trajectoires d'un satellite mis en orbite basse par son lanceur et son changement d'orbite pour accéder à l'orbite géostationnaire, par le biais d'une orbite de transfert elliptique.



(a) Vue en perspective (b) Vue du dessus (sans la Terre)

FIGURE 6 – Trajectoire d'un satellite allant de l'orbite basse à l'orbite géostationnaire

Lors des missions Apollo, après la mise en orbite des modules lunaires et de commandes, l'étage S-IVB du lanceur SATURN V effectue une poussée afin de se placer en orbite elliptique de transfert vers la Lune, on appelle ceci l'injection trans-lunaire (étape 2 sur l'image de couverture). Nous pouvons donc nous proposer de modéliser cette orbite de transfert :



(a) Vue en perspective (b) Vue du dessus

FIGURE 7 – Trajectoire des missions APOLLO après l'injection trans-lunaire

1. International Space Station

Conclusion et perspectives d'améliorations

Ce projet m'a permis de me familiariser avec le langage de programmation Python, qui m'était encore inconnu il y a un an. De plus, ce projet m'a permis de traiter un sujet que j'apprécie tout particulièrement et dans lequel je continuerai mes études, cela m'a donc permis d'aborder certaines notions plus en détail et de manière relativement complète. Je remercie David Cornu et Fabrice Devaux pour leur aide sur certains aspects de mon programme et je remercie également Benoît Noyelles pour sa patience et son écoute, qui m'ont permises de pouvoir traiter ce sujet de la meilleure des manières. Le programme présente certains défauts, notamment le manque d'interface graphique, qui rend son utilisation compliquée. Le développement de cette interface graphique est une perspective d'amélioration, ce qui permettra à l'utilisateur d'avoir une meilleure prise en main. De plus, un autre corps massif pourrait être ajouté afin de pouvoir réaliser des modélisations de trajectoire les plus réalistes possibles.

Ressources bibliographiques

[1] : Cours de Programmation et Algorithmes numériques Licence 2, *David Cornu 2019*.

[2] : Computer Methods for Odrinary Differential Equations and Differential-Algebraic Equations, *Uri M. Ascher & Linda R. Petzold 1998*.

Annexe

```
""" UE Projet integrateur Numerique """



##############################################################################
# Modelisation de la trajectoire d'un objet naturel ou artificiel au      #
#                          voisinage d'une planete                          #
##############################################################################

#_____________________Maxime Sabbadini, 2019_____________________#



""" Importation des librairies """

import numpy as np
import matplotlib.pyplot as plt
import pylab as pl
from mpl_toolkits.mplot3d import Axes3D


""" Declaration des constantes """

G=6.67408e-11 #SI
M=5.972e24       #Masse du corps central
rayon= 6371e3  #Rayon du corps central (m)

""" Declaration des conditions initiales """

ti=0       #Temps initial (s)
tf=4000  #Temps final (s)
dt=10    #Temps entre chaque points (s)
n=int((tf-ti)/dt)  #Nombre de points pris dans l'intervalle de temps

v0x=0           #m/s
v0y=7667+3170
v0z=0

x0=6779e3      #m
y0=0
z0=0

""" Definition des fonctions """

def trajectoire_euler(r, v, ti, tf, dt, n):    #Fonction utilisant la methode d'Euler
    v_x=v[0,0]                                                  #Inutilisee
    v_y=v[0,1]
    v_z=v[0,2]
    x=r[0,0]
    y=r[0,1]
    z=r[0,2]


    for i in range (1,n):
        re=np.sqrt(x**2+y**2+z**2)

        x+=dt*v_x
        y+=dt*v_y
        z+=dt*v_z

        r[i,0]=x
        r[i,1]=y
        r[i,2]=z

        v_x+=dt*(-G*M/re**3*x)
        v_y+=dt*(-G*M/re**3*y)
        v_z+=dt*(-G*M/re**3*z)

        v[i,0]=v_x
        v[i,1]=v_y
        v[i,2]=v_z
```

```
73
74      return v, r
75
76
77  r2 = np.zeros((3,n),dtype = float)   #Tableaux pour la methode de RK4
78  v2 = np.zeros((3,n),dtype = float)
79
80  ##Conditions initiales
81
82  r2[0,0] = x0
83  r2[1,0] = y0
84  r2[2,0] = z0
85  v2[0,0] = v0x
86  v2[1,0] = v0y
87  v2[2,0] = v0z
88
89  """ Fonction RK4 """
90
91  def f(r, v):
92      x = r[0]
93      y = r[1]
94      z = r[2]
95      re=np.sqrt(x**2+y**2+z**2)          #Definition du vecteur a 6 composantes
96      vect = np.zeros(6)
97      vect[0] = v[0]
98      vect[1] = v[1]
99      vect[2] = v[2]
100     vect[3] = -G*M/re**3*x
101     vect[4] = -G*M/re**3*y
102     vect[5] = -G*M/re**3*z
103     return vect
104
105 def rk4(r, v, dt, n):
106     for i in range(0,n-1):
107
108         k1 = f(r[:,i],v[:,i])
109         k2 = f(r[:,i]+ k1[0:3]*dt*0.5,v[:,i]+ k1[3:6]*dt*0.5)  #Integration par RK4
110         k3 = f(r[:,i]+ k2[0:3]*dt*0.5,v[:,i]+ k2[3:6]*dt*0.5)
111         k4 = f(r[:,i]+ k3[0:3]*dt,v[:,i]+ k3[3:6]*dt)
112         k = (k1 + 2*k2 + 2*k3 + k4)/6
113         r[:,i+1] = r[:,i] + dt*k[0:3]
114         v[:,i+1] = v[:,i] + dt*k[3:6]
115
116     return r
117
118 L=rk4(r2,v2,dt,n)  #Appel de la fonction
119
120 u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]  #Forme du corps central
121 x = rayon*np.cos(u)*np.sin(v)
122 y = rayon*np.sin(u)*np.sin(v)
123 z = rayon*np.cos(v)
124
125
126 ##Trace de/des trajectoire(s)
127
128 fig=pl.figure(figsize=(8,5))
129 ax=fig.add_subplot(111, projection='3d')
130 ax.plot(L[0,:], L[1,:], L[2,:], marker='+', ls='', color='r',label='Trajectoire') #Trace de
        la trajectoire
131 ax.plot_wireframe(x, y, z, color="b", label='Planete')    #Trace du corps central
132 pl.legend()
133 pl.show()
```

Listing 3 – Code complet