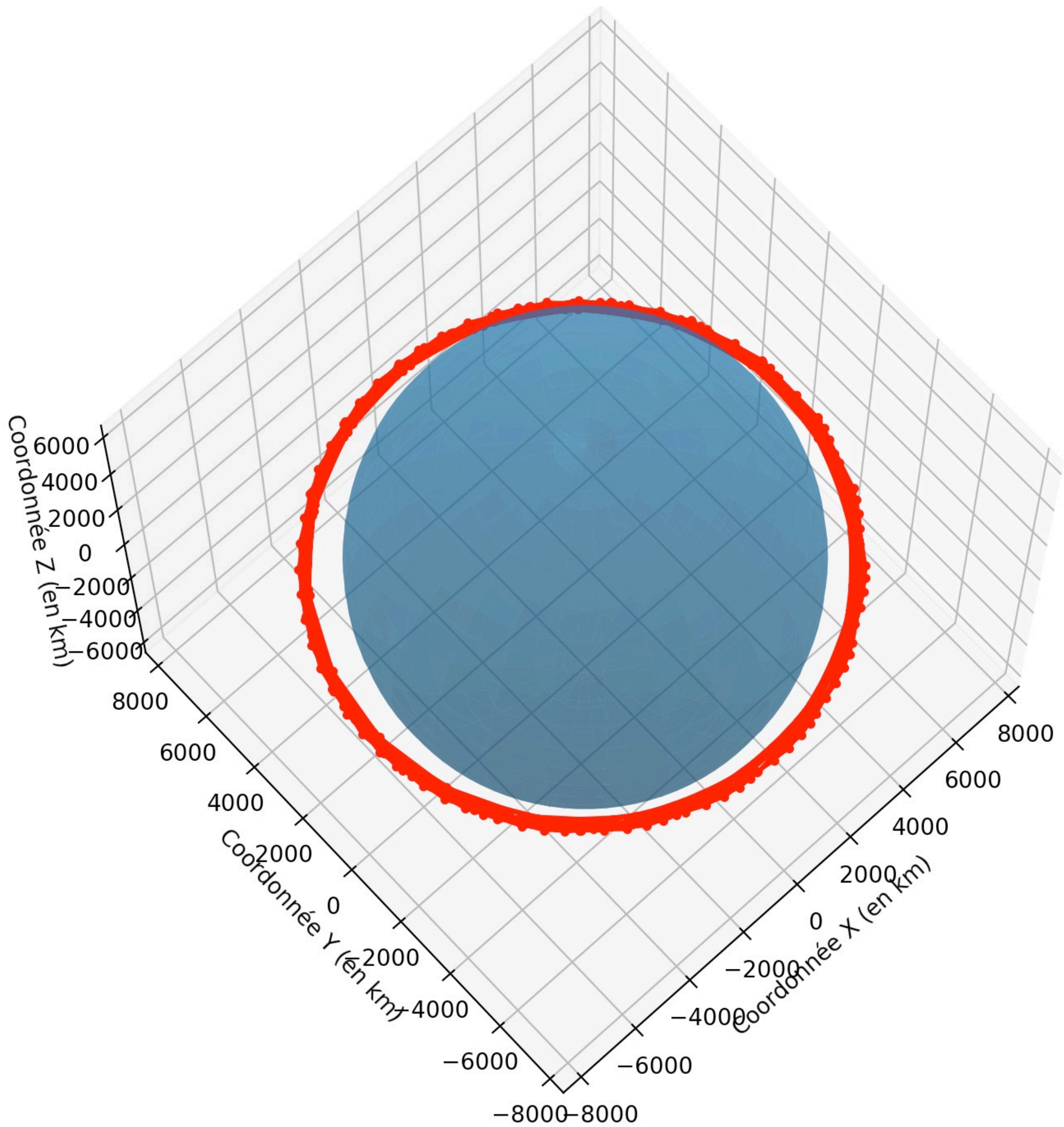
Rapport de Projet 1A

Code KESSLER - Simulateur Orbital



Introduction :

Le code KESSLER est un simulateur de mécanique spatiale que j'ai développé sous Python au cours de mon projet d'ingénieur de première année. Il se place en bout de chaîne de réception du Cubesat et permet de déterminer, à partir d'un vecteur d'état (**r**, **v**), la trajectoire du débris durant le prochain mois afin d'anticiper les manoeuvres d'évitement des satellites opérationnels (il calcule aussi le temps avant écrasement du débris au sol). Il s'intègre également au début de notre réflexion, en tant que contrainte de dimensionnement pour les facteurs d'échange relatifs à la résolution nécessaire sur le vecteur d'état : afin que les données Cubesat soient exploitables par les agences jusqu'à J+*1 semaine* après la mesure, j'en ai conclu, grâce à des simulations sur KESSLER, que le système de mesure ne devait pas excéder 100 m d'incertitude sur la position du débris lors de sa détection initiale au niveau du Cubesat. Pour limiter la propagation des erreurs au cours du temps, j'ai dû intégrer de nombreux phénomènes physiques décrits plus précisément ci-dessous. Enfin, et en vue d'élargir l'utilisation du code KESSLER à des cas d'utilisation plus larges, j'ai par ailleurs codé les principales fonctions de conversion entre paramètres orbitaux.

Objectif : déterminer l'orbite des débris rencontrés, prédire leur trajectoire et leur Time-to-Earth. À partir de ces fonctions, on estimera l'incertitude initiale acceptable pour permettre d'anticiper raisonnablement les manœuvres d'évitement de débris spatiaux.

Architecture : La routine principale du code KESSLER, s'exécutant automatiquement au lancement du fichier `Main.py`, prédit la trajectoire ultérieure du débris sur la fenêtre de temps fournie en paramètre dans la fonction principale `Plot`, et alerte l'utilisateur, le cas échéant, que le débris s'écrasera sur la Terre sur la fenêtre de temps donnée.

`Main.py` coordonne la propagation des équations de la mécanique sur le débris. Il met en oeuvre la méthode d'intégration numérique choisie (Euler, RK2 ou RK4) et met à jour un tableau X contenant les positions (3 premières coordonnées) et les vitesses (3 dernières coordonnées) du débris étudié. Il affiche ensuite une vue 3D figée de la Terre ainsi que de la trajectoire du débris autour de cette dernière, tracée en pointillés. Il affiche ensuite une animation de sa trajectoire au cours du temps, illustrant le mouvement du débris autour de la Terre. Enfin, il calcule les principaux paramètres d'intérêt (énergie cinétique, éléments képlériens, etc ...) et affiche leur évolution temporelle.

`Propagation.py` est appelé par `Main.py` à chaque étape (ou 'pas') de la méthode de résolution numérique de l'équation différentielle du mouvement pour calculer le vecteur d'accélération instantanée qui sera utilisé pour mettre à jour le vecteur vitesse, puis le vecteur position du débris. À ce jour, il prend en compte :

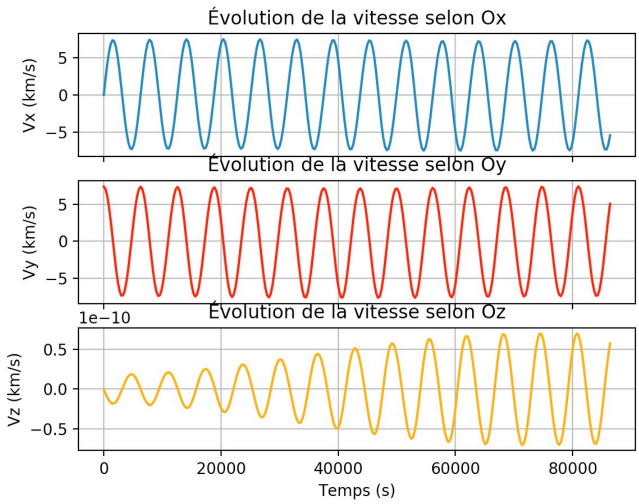
- La pression de radiation (ou albédo) dû au flux de photons en provenance du Soleil
- Le potentiel de gravitation irrégulier de la Terre, décomposé selon la base des harmoniques sphériques
- Les corrections relativistes (négligeables)
- La traînée atmosphérique
- Un terme dit "de marée", qui prend en compte l'attraction débris / Lune et débris / Soleil

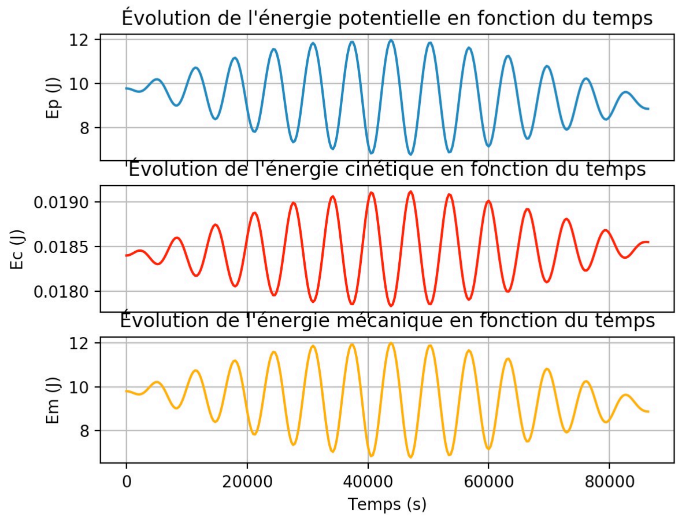
- La force due aux marées terrestres (que l'on modélise comme des excédents de masse d'eau tournant à la surface de la Terre)
- Un bruit thermique simulant toutes les autres sources d'incertitude / phénomènes physiques négligés, ayant pour effet une hausse plus rapide de l'incertitude sur le vecteur d'état du débris au cours du temps

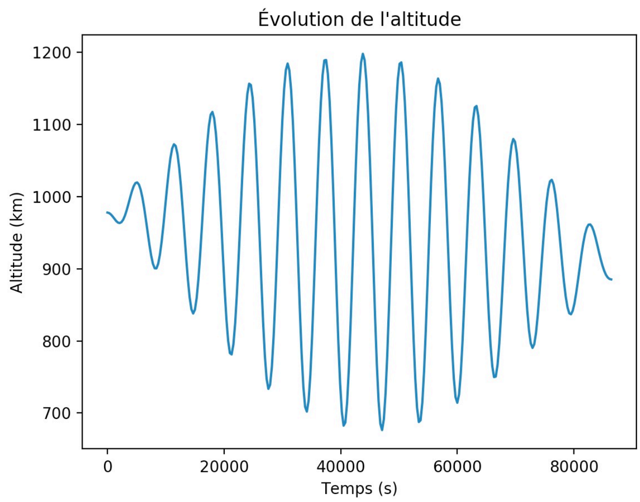
`Objects.py` définit deux classes distinctes relatives aux planètes et aux satellites. Au stade actuel d'avancement du code, elles n'ont pas trouvé d'autre intérêt que celui d'introduire en tant que méthodes les fonctions de conversion entre éléments orbitaux. Elles seront cependant incontournables afin d'assurer la maintenabilité du code au fur et à mesure que les phénomènes physiques pris en compte seront nombreux (les classes permettront de définir de la même manière tous les corps célestes orbitant dans le système Solaire, dont les paramètres, définis une fois pour toutes, pourront servir autant pour calcuelr un phénomène relativiste qu'une pression de radiation ou une attraction gravitationnelle simple).

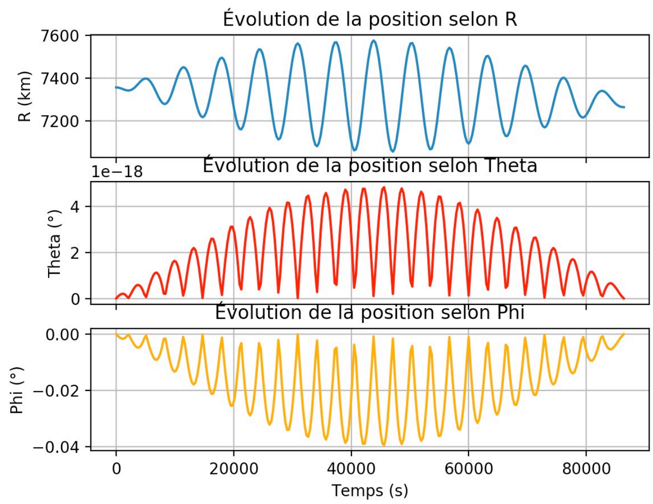
`Const.py` regroupe les principales constantes physiques impliquées dans les phénomènes étudiés et, surtout, les paramètres du débris étudié (surface efficace exposée aux frottements atmosphériques, surface efficace exposée à l'albédo, masse, etc ...).

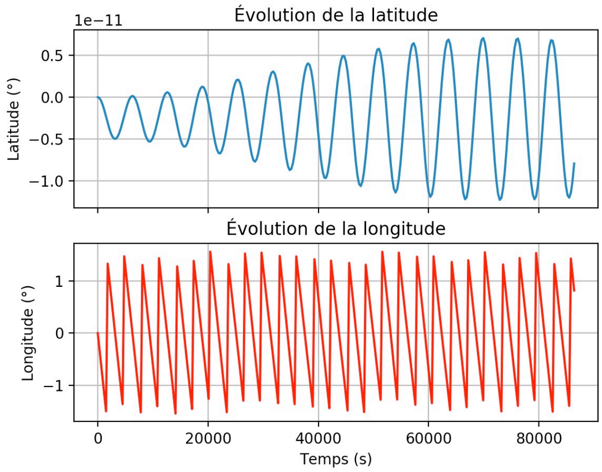
Pour les paramètres initiaux choisis dans le code présentés dans l'annexe, on obtient les graphes suivants :

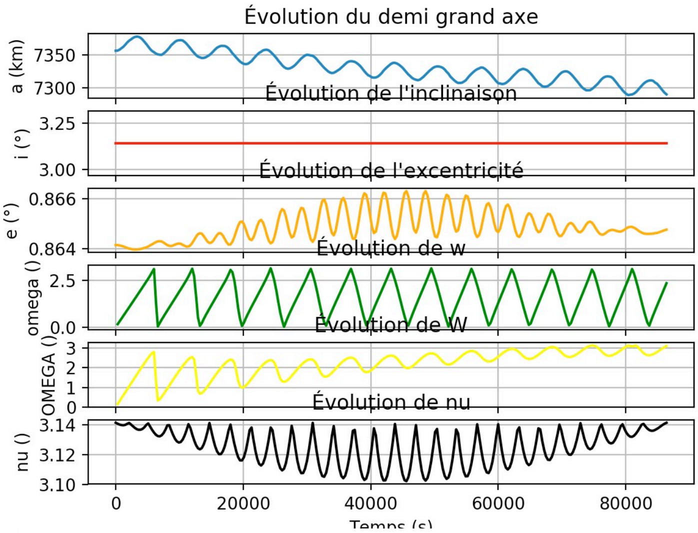


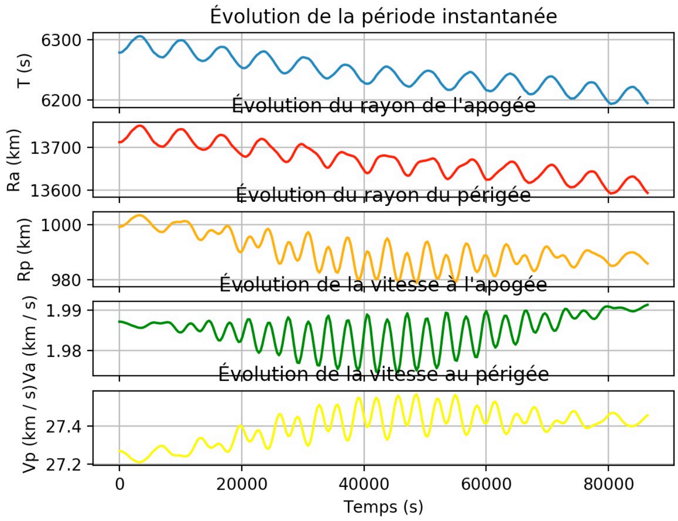


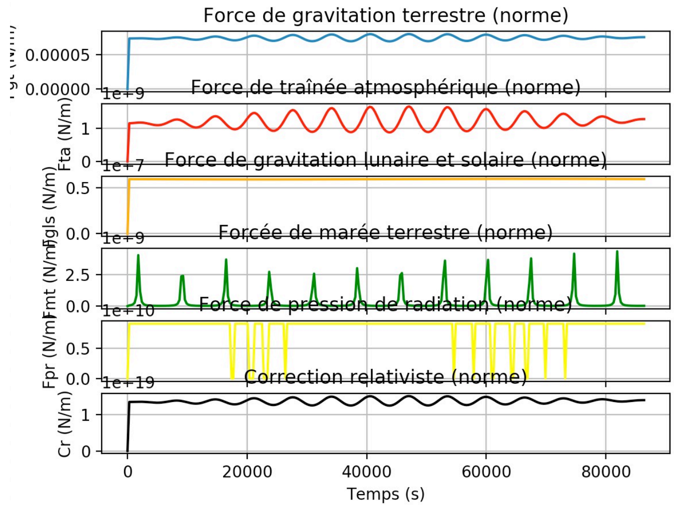




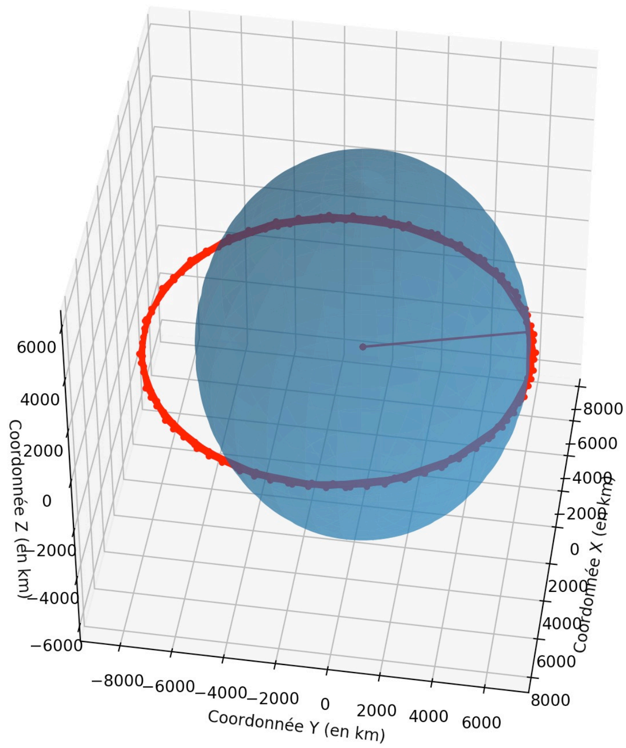




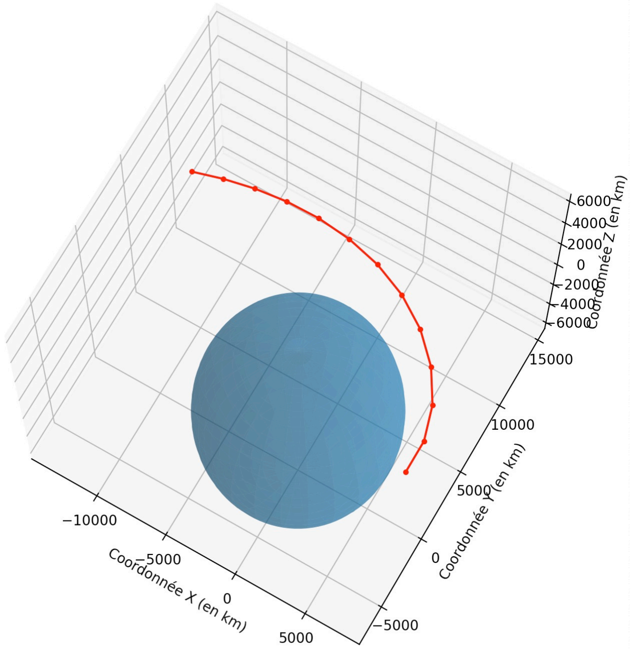


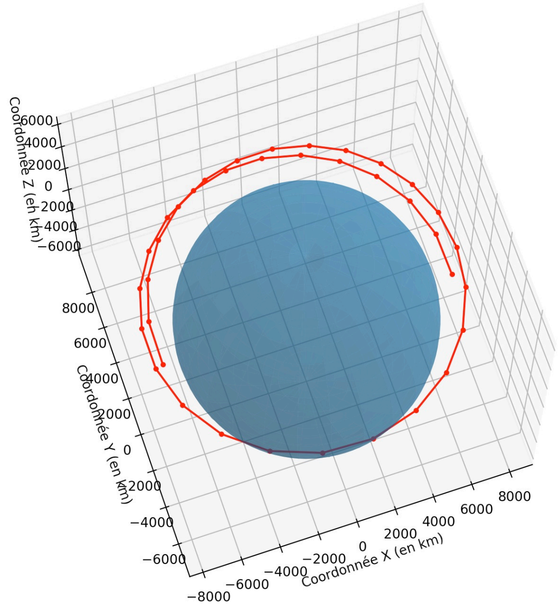


Voici la figure obtenue dans le cas où la vitesse du débris l'amène à s'écraser à la surface de la Terre :



Note : il est essentiel d'utiliser la méthode RK4, comme on peut le voir sur les exemples ci-dessous, utilisant respectivement une méthode numérique de premier ordre (Euler) et de second ordre (RK2) :





On constate expérimentalement que pour une incertitude d'environ 100 m sur la position initiale du débris, on a une incertitude d'environ 1 km sur sa position à J+1 (en fonction de paramètres macro tels que la densité de l'atmosphère, l'albédo, etc ...). Étant donné qu'une agence spatiale peut raisonnablement exploiter une connaissance de la position du débris dans une sphère de 10 km (ordre de grandeur à partir duquel l'ISS déclenche des manoeuvres d'évitement), les données fournies par le Cubesat semblent utilisables jusqu'à une semaine (en pratique, plus longtemps (1 mois généralement) car les codes de calcul professionnels ont un taux de propagation d'erreur beaucoup plus faible que KESSLER).

Les limites du modèle adopté ici sont nombreuses, parmi lesquelles (liste non exhaustive) :

- Nous avons négligé l'interaction gravitationnelle des astres autres que la Lune, la Terre et le Soleil
- La pression de radiation a des périodicités d'ordre 2 (éruptions solaires, éclipses)
- Modèle imprécis de la densité de l'atmosphère (qui n'est pas constante, mais exponentielle par morceaux). Par ailleurs, la densité (et donc la force de traînée atmosphérique) varie grandement en fonction de l'expositon au Soleil (et donc de l'heure de la journée)
- Effets de marée Terre / Lune / Soleil (interaction mutuelle à trois corps, négligée ici)
- Harmoniques gravitationnelles d'ordre > 1 (déterminantes : le modèle adopté ici ne rend absolument pas compte de la morphologie particulière de la Terre)
- Micrométéorites (particules très fines provoquant la décompositon progressive du débris : essentiellement, leur présence se modélise par une force de trainée atmosphérique aditionnelle avec une diminution de la masse du débris au cours du temps)
- Eu égard à la rotation du débris, les surfaces efficaces soumises à l'albédo et à la trainée atmosphérique varient au cours du temps

- Pression de radiation de second ordre (réfléchies par la Terre et la Lune sur le Soleil)
- Propagation des erreurs numériques (pour palier cela, il faut travailler avec des long int)

Les pistes d'amélioration que j'ai envisagées jusqu'à présent (liste, à nouveau, non exhaustive) sont :
- Intégrer une analyse de la diminution d'altitude du débris au cours du temps pour remonter à sa masse / au matériau principal le constituant / la surface efficace exposée à l'albédo et à la trainée atmosphérique / le moment cinétique.
- Prédire l'instant où l'on recroisera un débris pour confirmer / valider sa reconnaissance par signature SER et émettre avec un grand gain dans la zone de passage prédite par le code de calcul
- Créer conjointement une base de données des "signatures orbitales" (par analogie avec les signatures SER) afin d'y inscrire les objets rencontrés et de mettre à jour les nouveaux objets détectés (on pourrait par ailleurs supprimer, en raison de l'incertitude croissante, les débris observés il y a plus d'un mois par notre flotte de Cubesats)
- Dans l'hyopthèse où le Cubesat a une portée suffisante, on peut "entraîner" le code sur des satellites aux trajectoires bien connues afin d'ajuster les constantes de la chaîne d'acquisition (notamment, afin d'affiner par machine learning la fonction mathématique faisant le lien entre puissance reçue et distance Cubesat / corps observé, si l'on suppose connue parfaitement la position du satellite observé et du Cubesat)
- Entraîner l'algorithme (ajuster ses paramètres macro : densité atmosphérique, albédo, etc ...) sur des bases de données de satellites bien connues (méthode des moindres carrés par exemple)
- Interpoler les éléments orbitaux d'un débris grâce à plusieurs points de mesure : faire d'abord une moyenne des estimations de ses éléments orbitaux, on pourra ensuite faire appel à la méthode des moindres carrés pour déterminer l'orbite qui "fitte" le mieux les mesures

Main.py # Calcul et affichage des éléments orbitaux du débris, calcul du TimeToEarth

```
from math import *
import numpy as np
import matplotlib.pyplot as plt
from const import *
from objects import *
from matplotlib import cm
from propagation import *
from datetime import datetime



def checkIn(S): # Vérifie si le débris ne s'est pas "crashé" (auquel cas on arrête la simulation)
    r = np.linalg.norm(S.xyz)
    if r < Earth.R:
        return True



def timeToEarth(h = 300, tMax = [1, 0, 0], ordre = 4, X0=[[7356],[0],[0],[0],[sqrt(Earth.mu / Earth.R)],[0]]): # On néglige le
TTE si le débris n'est toujours pas redescendu au bout d'un an
    Tf = (tMax[0] * 24 + tMax[1]) * 3600 + tMax[2] * 60 # date de fin de la simulation (en secondes)
    print("La simulation s'étend de T = 0s à T = {0}s".format(Tf))
    steps = int(Tf / h) + 1
    print("La simulation effectuera {0} pas de valeur unitaire {1} s.".format(steps, h))
    temps = np.linspace(0, Tf, steps)
    print("Vecteur d'état initial choisi : pos = {0}, vit = {1}".format(X0[0:3], X0[3:6]))
    X = np.zeros((6, steps)) # Les trois premières composantes sont S.xyz la position du débris, les trois dernières sont
    # son vecteur vitesse
    X[:, [0]] = X0
    for i in range(1, steps):
        x = X[:, [i - 1]]

        if ordre == 4: # Méthode RK4
            k1 = propagation(x, h, temps[i]) * h # Calcul des paramètres intermédiaires (order 4)
            k2 = propagation(x + k1 / 2, h, temps[i]) * h
            k3 = propagation(x + k2 / 2, h, temps[i]) * h
            k4 = propagation(x + k3, h, temps[i]) * h

            X[:, [i]] = x + (k1 + 2 * (k2 + k3) + k4) / 6

        if ordre == 2: # Méthode RK2
            k1 = propagation(x, h, temps[i]) * h # Calcul des paramètres intermédiaires (ordre 2)
            k2 = propagation(x + k1 / 2, h, temps[i]) * h

            X[:, [i]] = x + k2

        if ordre == 1: # Méthode d'Euler
            k1 = propagation(x, h, temps[i])
            X[:, [i]] = x + k1 * h

        if checkIn(S): # Vérifiée si le débris s'est "crashé"
            return True
    return False



def plot(h = 300, fin = [0, 3, 0], ordre = 2, X0=[[7356],[0],[0],[1],[sqrt(Earth.mu / 7356)],[0]]): # Propage les équations de
la mécanique et détermine les éléments orbitaux du débris
    # h est le pas de la méthode numérique en secondes
    # fin est la date de fin sous la forme jours / heures / minutes
    # ordre correspond au degré de la méthode de Runge Kutta utilisée
    # X0 est le vecteur d'état du débris spatial

    Tf = (fin[0] * 24 + fin[1]) * 3600 + fin[2] * 60 # date de fin de la simulation (en secondes)
    print("La simulation s'étend de T = 0s à T = {0}s".format(Tf))
    steps = int(Tf / h) + 1
    print("La simulation effectuera {0} pas de valeur unitaire {1} s.".format(steps, h))
    temps = np.linspace(0, Tf, steps)
    print("Vecteur d'état initial choisi : pos = {0}, vit = {1}".format(X0[0:3], X0[3:6]))



    X = np.zeros((6, steps)) # Les trois premières composantes sont S.xyz la position du débris, les trois dernières sont
    # son vecteur vitesse
    X[:, [0]] = X0
    acceleration = [np.zeros((3, steps)) for i in range(6)]
    for i in range(1, steps):
        x = X[:, [i - 1]]

        if ordre == 4: # Méthode RK4
            k1 = propagation(x, h, temps[i], acceleration, i) * h # Calcul des paramètres intermédiaires (order 4)
            k2 = propagation(x + k1 / 2, h, temps[i], acceleration, i) * h
            k3 = propagation(x + k2 / 2, h, temps[i], acceleration, i) * h
            k4 = propagation(x + k3, h, temps[i], acceleration, i) * h

            X[:, [i]] = x + (k1 + 2 * (k2 + k3) + k4) / 6

        if ordre == 2: # Méthode RK2
            k1 = propagation(x, h, temps[i], acceleration, i) * h # Calcul des paramètres intermédiaires (ordre 2)
            k2 = propagation(x + k1 / 2, h, temps[i], acceleration, i) * h

            X[:, [i]] = x + k2
```

```python
        if ordre == 1: # Méthode d'Euler
            k1 = propagation(x, h, temps[i], acceleration, i)
            X[:, [i]] = x + k1 * h

        if checkIn(S): # Vérifiée si le débris s'est "crashé"
            steps = i - 2
            print("Le débris s'est écrasé sur la Terre")
            break



    fig = plt.figure(1, figsize=(7, 7), dpi=100, facecolor='black', edgecolor='k')
    ax = Axes3D(fig)
    ax.plot(X[0, :], X[1, :], X[2, :], '.-r')

    def drawCelestialBody(name): # Fonction dessinant la Terre sur le graphe final
        if name == "Earth":
            offset = 0
            radius = Earth.R
        elif name == "Sun":
            offset = Sun.de
            radius = Sun.R
        elif name == "Moon":
            offset = Moon.de
            radius = Moon.R

        phi, th = np.mgrid[0.0 : np.pi : 100j, 0.0 : 2.0 * np.pi : 100j]
        x = radius * np.sin(phi) * np.cos(th) + offset
        y = radius * np.sin(phi) * np.sin(th)
        z = radius * np.cos(phi)

        return x, y, z

    x, y, z = drawCelestialBody("Earth")

    ax.plot_surface(x, y, z, rstride = 4, cstride = 4, alpha = 0.5) # On trace la Terre

    ax.set_xlabel('Coordonnée X (en km)')
    ax.set_ylabel('Coordonnée Y (en km)')
    ax.set_zlabel('Coordonnée Z (en km)')

    plt.axis('square')
    plt.gca().set_aspect('equal', adjustable='box')

    plt.show() # On affiche la trajectoire complète du satellite



    # Animation de la trajectoire au cours du temps

    fig2 = plt.figure(2, figsize=(7, 7), dpi=100, facecolor='black', edgecolor='k')
    ax2 = Axes3D(fig2)
    energy_text = ax2.text2D(0.02, 0.90, 'qd', transform=ax.transAxes)

    def actualiser(num, dataLines, lines): # Animation permettant de visualiser la progression de la trajectoire
        for line, data in zip(lines, dataLines):
            line.set_data(data[0:2,:num])
            line.set_3d_properties(data[2,:num])
            d = datetime.fromtimestamp(temps[num])
            energy_text.set_text("Temps écoulé : %d jours, %d heures, %d minutes, %d secondes" % (d.day-1, d.hour, d.minute, 
d.second))
        return lines

    data = [X[0:3, :] for i in range(steps)] # On "dézippe" les données selon les coordonnées X, Y et Z

    lines = [ax2.plot(dat[0, 0:1], dat[1, 0:1], dat[2, 0:1], '-sr')[0] for dat in data] # Construction des lignes de trajectoire

    ax2.set_xlim3d([-10000, 10000])
    ax2.set_xlabel('Coordonnée X (en km)')

    ax2.set_ylim3d([-10000, 10000])
    ax2.set_ylabel('Coordonnée Y (en km)')

    ax2.set_zlim3d([-10000, 10000])
    ax2.set_zlabel('Coordonnée Z (en km)')

    ax2.set_title('Animation de la trajectoire orbitale du débris')

    ax2.plot_surface(x, y, z,  rstride=4, cstride=4, alpha=0.5)

    # Lance l'animation de la trajectoire du satellite en fonction du temps
    line_ani = animation.FuncAnimation(fig2, actualiser, int(steps / 2), fargs=(data, lines), interval=200, repeat_delay=100, 
blit=False)

    plt.show() # On affiche l'animation de la trajectoire du satellite au cours du temps



    # Calcul des éléments orbitaux tout au long de la trajectoire
    pos = X[0:3, :]
    vit = X[3:6, :]
    x, y, z = pos # Positions cartésiennes du débris
    vx, vy, vz = vit # Vitesses cartésiennes du débris
    v = (np.power(vx, 2) + np.power(vy, 2) + np.power(vz, 2)) ** (1/2) # Vitesse instantanée
    Ec = S.m * (v ** 1/2) / 2 # Énergie cinétique
    r = (np.power(x, 2) + np.power(y, 2) + np.power(z, 2)) ** (0.5)
    h = r - Earth.R # Altitude du débris par rapport au niveau de la Mer
```

```python
Ep = S.m * h # Énergie potentielle du débris
Em = Ep + Ec # Énergie mécanique du débris
a = 1 / (2 / r - (v ** 2) / Earth.mu) # Radius of the apogee

# Initialization of the parameter time-evolution arrays
hvec = np.zeros((3, steps))
nvec = np.zeros((3, steps))
evec = np.zeros((3, steps))
e = np.zeros(steps)
inc = np.zeros(steps)
OMEGA = np.zeros(steps)
omega = np.zeros(steps)
nu = np.zeros(steps)
latitude = np.zeros(steps)
longitude = np.zeros(steps)
T = np.zeros(steps)
n = np.zeros(steps)
Va = np.zeros(steps)
Vp = np.zeros(steps)
theta = np.zeros(steps)
phi = np.zeros(steps)
E = np.zeros(steps)

for i in range(steps):
    hvec[:, i] = np.cross(pos[:, i], vit[:, i]) # Inertia vector
    nvec[:, i] = np.cross([0, 0, 1], hvec[:, i]) # Node vector
    terme1 = pos[:, i] * (v[i] - Earth.mu / r[i])
    terme2 = vit[:, i] * np.dot(pos[:, i], vit[:, i])
    evec[:, i] = (1 / Earth.mu) * (terme1 - terme2) # Excentricity vector
    e[i] = np.linalg.norm(evec[:, i]) # Excentricity
    inc[i] = acos(hvec[2, i] / np.linalg.norm(hvec[:, i])) # Inclination of the orbital plane
    OMEGA[i] = acos(nvec[0, i] / np.linalg.norm(nvec[:, i]))
    omega[i] = acos(np.dot(nvec[:, i], evec[:, i]) / (np.linalg.norm(nvec[:, i]) * np.linalg.norm(evec[:, i])))
    nu[i] = acos(np.dot(evec[:, i], pos[:, i]) / (np.linalg.norm(evec[:, i]) * r[i])) # Real anomaly
    T[i] = 2 * pi * sqrt((a[i] ** 3) / Earth.mu) # Instantaneous period of the orbit
    n[i] = sqrt(Earth.mu / (a[i] ** 3)) # Mean motion

Ra = a * (1 + e) # Apogee radius
Rp = a * (1 - e) # Perigee radius

for i in range(steps):
    Va[i] = sqrt(2 * Earth.mu * Rp[i] / (Ra[i] * (Ra[i] + Rp[i]))) # Apogee instantaneous velocity
    Vp[i] = sqrt(2 * Earth.mu * Ra[i] / (Rp[i] * (Ra[i] + Rp[i]))) # Perigee instantaneous velocity
    theta[i] = asin(sin(inc[i]) * sin(nu[i])) # Tehta (spherical coordinates)
    phi[i] = asin(tan(theta[i] / tan(inc[i]))) # Phi (spherical coordinates)
    latitude[i] = asin(z[i] / r[i]) # Latitude
    longitude[i] = atan(y[i] / x[i]) # Longitude

# Calcul des normes des vecteurs d'accélération instantanée
aGravitationPotential = np.linalg.norm(acceleration[0], axis=0)
aAtmosphericDrag = np.linalg.norm(acceleration[1], axis=0)
aSunAndMoonAttraction = np.linalg.norm(acceleration[2], axis=0)
aTidalForce = np.linalg.norm(acceleration[3], axis=0)
aRadiationPressure = np.linalg.norm(acceleration[4], axis=0)
aRelativisticCorrection = np.linalg.norm(acceleration[5], axis=0)

# ------------------------------------------------------------------------
# Affichage de tous les éléments calculés précédemment en fonction du temps
# ------------------------------------------------------------------------

plt.plot(temps, h, marker=',') # Affichage de l'altitude
plt.title("Évolution de l'altitude")
plt.xlabel('Temps (s)')
plt.ylabel('Altitude (km)')
plt.show()

fig, axes = plt.subplots(nrows=3, ncols=1, sharex=True) # Étude énergétique du débris
plt.gca().set_aspect('auto', adjustable='box')
axes[0].plot(temps, Ep)
axes[1].plot(temps, Ec, color="red")
axes[2].plot(temps, Em, color="orange")
axes[0].title.set_text("Évolution de l'énergie potentielle en fonction du temps")
axes[1].title.set_text("Évolution de l'énergie cinétique en fonction du temps")
axes[2].title.set_text("Évolution de l'énergie mécanique en fonction du temps")
axes[2].set_xlabel('Temps (s)')
axes[0].set_ylabel('Ep (J)')
axes[1].set_ylabel('Ec (J)')
axes[2].set_ylabel('Em (J)')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
plt.show()

fig, axes = plt.subplots(nrows=3, ncols=1, sharex=True) # Vitesse en coordonnées cartésiennes
axes[0].plot(temps, vx)
axes[1].plot(temps, vy, color="red")
axes[2].plot(temps, vz, color="orange")
axes[0].title.set_text("Évolution de la vitesse selon Ox")
axes[1].title.set_text("Évolution de la vitesse selon Oy")
axes[2].title.set_text("Évolution de la vitesse selon Oz")
axes[2].set_xlabel('Temps (s)')
axes[0].set_ylabel('Vx (km/s)')
axes[1].set_ylabel('Vy (km/s)')
axes[2].set_ylabel('Vz (km/s)')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
```

```python
plt.show()

fig, axes = plt.subplots(nrows=3, ncols=1, sharex=True) # Positon en coordonnées cartésiennes
axes[0].plot(temps, x)
axes[1].plot(temps, y, color="red")
axes[2].plot(temps, z, color="orange")
axes[0].title.set_text("Évolution de la position selon Ox")
axes[1].title.set_text("Évolution de la position selon Oy")
axes[2].title.set_text("Évolution de la position selon Oz")
axes[2].set_xlabel('Temps (s)')
axes[0].set_ylabel('X (km)')
axes[1].set_ylabel('Y (km)')
axes[2].set_ylabel('Z (km)')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
plt.show()

fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True) # Position du projeté à la surface du globe (lat, lon)
axes[0].plot(temps, latitude)
axes[1].plot(temps, longitude, color="red")
axes[0].title.set_text("Évolution de la latitude")
axes[1].title.set_text("Évolution de la longitude")
axes[1].set_xlabel('Temps (s)')
axes[0].set_ylabel('Latitude (°)')
axes[1].set_ylabel('Longitude (°)')
axes[0].grid(True)
axes[1].grid(True)
plt.show()

fig, axes = plt.subplots(nrows=3, ncols=1, sharex=True) # Position en coordonnées sphériques
axes[0].plot(temps, r)
axes[1].plot(temps, theta, color="red")
axes[2].plot(temps, phi, color="orange")
axes[0].title.set_text("Évolution de la position selon R")
axes[1].title.set_text("Évolution de la position selon Theta")
axes[2].title.set_text("Évolution de la position selon Phi")
axes[2].set_xlabel('Temps (s)')
axes[0].set_ylabel('R (km)')
axes[1].set_ylabel('Theta (°)')
axes[2].set_ylabel('Phi (°)')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
plt.show()

fig, axes = plt.subplots(nrows=5, ncols=1, sharex=True) # Paramètres périodiques caractéristiques
axes[0].plot(temps, T)
axes[1].plot(temps, Ra, color="red")
axes[2].plot(temps, Rp, color="orange")
axes[3].plot(temps, Va, color="green")
axes[4].plot(temps, Vp, color="yellow")
axes[0].title.set_text("Évolution de la période instantanée")
axes[1].title.set_text("Évolution du rayon de l'apogée")
axes[2].title.set_text("Évolution du rayon du périgée")
axes[3].title.set_text("Évolution de la vitesse à l'apogée")
axes[4].title.set_text("Évolution de la vitesse au périgée")
axes[4].set_xlabel('Temps (s)')
axes[0].set_ylabel('T (s)')
axes[1].set_ylabel('Ra (km)')
axes[2].set_ylabel('Rp (km)')
axes[3].set_ylabel('Va (km / s)')
axes[4].set_ylabel('Vp (km / s)')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
axes[3].grid(True)
axes[4].grid(True)
plt.show()

fig, axes = plt.subplots(nrows=6, ncols=1, sharex=True) # Les six éléments képlériens
axes[0].plot(temps, a)
axes[1].plot(temps, inc, color="red")
axes[2].plot(temps, e, color="orange")
axes[3].plot(temps, omega, color="green")
axes[4].plot(temps, OMEGA, color="yellow")
axes[5].plot(temps, nu, color="black")
axes[0].title.set_text("Évolution du demi grand axe")
axes[1].title.set_text("Évolution de l'inclinaison")
axes[2].title.set_text("Évolution de l'excentricité")
axes[3].title.set_text("Évolution de w")
axes[4].title.set_text("Évolution de W")
axes[5].title.set_text("Évolution de nu")
axes[5].set_xlabel('Temps (s)')
axes[0].set_ylabel('a (km)')
axes[1].set_ylabel('i (°)')
axes[2].set_ylabel('e (°)')
axes[3].set_ylabel('omega ()')
axes[4].set_ylabel('OMEGA ()')
axes[5].set_ylabel('nu ()')
axes[0].grid(True)
axes[1].grid(True)
axes[2].grid(True)
axes[3].grid(True)
axes[4].grid(True)
axes[5].grid(True)
plt.show()
```

```python
    fig, axes = plt.subplots(nrows=6, ncols=1, sharex=True) # Affichage des accélérations instantanées au cours du temps
    axes[0].plot(temps, aGravitationPotential * S.m)
    axes[1].plot(temps, aAtmosphericDrag * S.m, color="red")
    axes[2].plot(temps, aSunAndMoonAttraction * S.m, color="orange")
    axes[3].plot(temps, aTidalForce * S.m, color="green")
    axes[4].plot(temps, aRadiationPressure * S.m, color="yellow")
    axes[5].plot(temps, aRelativisticCorrection * S.m, color="black")
    axes[0].title.set_text("Force de gravitation terrestre (norme)")
    axes[1].title.set_text("Force de traînée atmosphérique (norme)")
    axes[2].title.set_text("Force de gravitation lunaire et solaire (norme)")
    axes[3].title.set_text("Forcée de marée terrestre (norme)")
    axes[4].title.set_text("Force de pression de radiation (norme)")
    axes[5].title.set_text("Correction relativiste (norme)")
    axes[5].set_xlabel('Temps (s)')
    axes[0].set_ylabel('Fgt (N/m)')
    axes[1].set_ylabel('Fta (N/m)')
    axes[2].set_ylabel('Fgls (N/m)')
    axes[3].set_ylabel('Fmt (N/m)')
    axes[4].set_ylabel('Fpr (N/m)')
    axes[5].set_ylabel('Cr (N/m)')
    axes[0].grid(True)
    axes[1].grid(True)
    axes[2].grid(True)
    axes[3].grid(True)
    axes[4].grid(True)
    axes[5].grid(True)
    plt.show()

plot()
```

Propagation.py # Calcul des différentes forces s'exerçant sur le débris

```
def radiationPressure(S, t): # Je n'ai pas trouvé de modèle simple pour la pression de radiation)
    periodRevolSun = 365 * 24 * 3600
    periodRevolEarth = 24 * 3600

    Sun.xyz = np.array([Sun.R * cos(2 * pi * t * (1 / periodRevolSun - 1 / periodRevolEarth)), Sun.R * sin(2 * pi * t * (1 / 
periodRevolSun - 1 / periodRevolEarth)), 0]) # en x, y, z = 0, 0, 0

    rDebrisSun = Sun.xyz - S.xyz
    rDebrisSun /= np.linalg.norm(rDebrisSun)

    longitudeSun = atan(Sun.xyz[1] / Sun.xyz[0]) # On calcule la longitude du Soleil et celle du Débris pour déterminer si le
    longitudeDebris = atan(S.xyz[1] / S.xyz[0]) # Satellite se situe dans une zone d'ensoleillement (et donc subit ou non
    # la pression de radiation exercée par le flux de photons en provenance du Soleil)
    # Critère : si abs(longitudeSun - longitudeDebris) > 135 °, le débris ne subit plus la pression de radiation

    if abs(longitudeSun - longitudeDebris) > 2 * pi * 135 / 360:
        return 0
    print(longitudeSun, longitudeDebris)
    return radPressureDensity * rDebrisSun * S.suncs / S.m # Computes the radiation pressure force and normalizes it


def gravitationPotential(S): # On prend en compte le potentiel non parfaitement sphérique de la Terre (le potentiel 
gravitationnel
                             # ayant plutôt l'aspect d'une "patatoïde", on le décompose en harmoniques sphériques jusqu'au
                             # troisième ordre)
    a = [0, 0, 0]

    # Termes intermédiaires de calcul
    r = np.linalg.norm(S.xyz) # Distance au centre de gravité (rayon)
    sq = (S.xyz[2] / r) ** 2
    mr = Earth.R / r

    # Accélération de pesanteur selon x
    a[0] -= J[2] * (mr ** 2) * (3 / 2) * (1 - 5 * sq)
    a[0] += J[3] * (mr ** 3) * (5 / 2) * (3 - 7 * sq * (S.xyz[2] / r))
    a[0] -= J[4] * (mr ** 4) * (5 / 8) * (3 - 42 * (sq + 63 * ((S.xyz[2] / r) ** 4)))
    a[0] *= S.xyz[0]

    # Accélération de pesanteur selon y
    a[1] -= J[2] * (mr ** 2) * (3 / 2) * (1 - 5 * sq)
    a[1] += J[3] * (mr ** 3) * (5 / 2) * (3 - 7 * sq) * (S.xyz[2] / r)
    a[1] -= J[4] * (mr ** 4) * (5 / 8) * (3 - 42 * sq + 63 * (sq ** 2))
    a[1] *= S.xyz[1]

    # Accélération de pesanteur selon z
    a[2] -= J[2] * (mr ** 2) * (3 / 2) * (3 - 5 * sq)
    a[2] += J[3] * (mr ** 3) * (5 / 2) * (6 - 7 * sq) * (S.xyz[2] / r)
    a[2] *= S.xyz[2]

    a[2] += (Earth.mu / (r ** 2)) * J[3] * (mr ** 3) * 3 / 2
    a[2] += (Earth.mu * S.xyz[2] / (r ** 3)) * J[4] * (mr ** 4) * (5 / 8) * (15 - 70 * sq + 63 * (sq ** 2))

    # Dimensionnalisation de l'accélération de pesanteur
    a = np.array(a) * Earth.mu / (np.linalg.norm(S.xyz) ** 3)
    a += np.array(S.xyz) * (-Earth.mu / (r ** 3))

    return a



def relativistCorrection(S): # Correction minimes
    p = S.vxyz * S.m # Vecteur momentum
    J = np.cross(S.xyz, p) # Vecteur moment cinétique
    r = np.linalg.norm(S.xyz) # Distance au centre de la Terre

    # Correction relativiste de premier ordre
    fac1 = Earth.mu / ((c ** 2) * (r ** 3))
    terme1 = ((4 * Earth.mu / r) - np.dot(S.vxyz, S.vxyz)) * S.xyz + 4 * np.dot(S.xyz, S.vxyz) * S.vxyz

    # Correction relativiste de second ordre
    fac2 = 2 * fac1
    terme2 = (3 / (r ** 2)) * np.dot(S.xyz, J) * np.cross(S.xyz, S.vxyz) + np.cross(S.vxyz, J)

    # Correction relativiste due au soleil : non pris en compte
    fac3 = 0
    terme3 = 0

    return np.array(fac1 * terme1 + fac2 * terme2 + fac3 * terme3)


def atmosphericNoise(a, h):
    variance = 0.000001 * h
    noise = 1 + np.random.normal(0, variance, 3)
    return np.multiply(a, noise) # Bruit gaussien uniforme sur les trois coordonnées du débris. Modélise l'incertitude due
                                 # aux approximations de premier ou second ordre sur les autres corrections, ainsi que d'autres
                                 # phénomênes physiques (dilatation thermique du matériau, interactions avec les nuages de poussières
                                 # spatiales (< 1 mm, très présentes), les variations locales de densité de l'atmosphère, etc ...)
```

```python
def atmosphericDrag(S):
    r = np.linalg.norm(S.xyz)

    def density(r): # Modèle de densité atmosphérique, valide entre 100 et 1000 km environ (le modèle
                    # réel est une succession de coubes de décroissance exponentielles)
        h = r - Earth.R

        H = 1000
        return 3e-7 * exp(-(h - 1000) / H) # Modèle de décroissance exponentielle de la densité atmosphérique

    w = [0, 0, 7.2921159e-5]

    vrel = S.vxyz - np.cross(w, S.xyz)
    aDrag = (-1 / 2) * S.Cd * (S.afcs / S.m) * density(r) * np.linalg.norm(vrel) * vrel # Vecteur d'accélération de
                                                                                        # frottement atmosphérique
    return aDrag



def sunAndMoonAttraction(S, t):
    # On prend maintenant en compte la rotation de la Terre autour du Soleil, et la rotation de la Lune autour de la Terre
    periodRevolSun = 365 * 24 * 3600
    periodRevolMoon = 28 * 24 * 3600
    periodRevolEarth = 24 * 3600

    Moon.xyz = np.array([Moon.de * sin(2 * pi * t / periodRevolMoon), Moon.de * cos(2 * pi * t / periodRevolMoon), 0]) # Calcul
des positions de la Lune et de la Terre (par rapport à la Terre, centrée et fixe
    Sun.xyz = np.array([Sun.de * cos(2 * pi * t * (1 / periodRevolSun - 1 / periodRevolEarth)), Sun.de * sin(2 * pi * t * (1 /
periodRevolSun - 1 / periodRevolEarth)), 0]) # en x, y, z = 0, 0, 0

    rMoon = np.linalg.norm(S.xyz - Moon.xyz) # On calcule la distance de la Lune au débris
    rSun = np.linalg.norm(S.xyz - Sun.xyz) # On calcule la distance du Soleil au débris

    aMoon = np.array(S.xyz - Moon.xyz) * (-Moon.mu / (rMoon ** 3)) # Calcul des accélérations de pesanteur dues aux attractions
respectives
    aSun = np.array(S.xyz - Sun.xyz) * (-Sun.mu / (rSun ** 3)) # de la Lune et du Soleil

    aTot = aMoon + aSun # Calcul de l'accélération de pesanteur totale
    return aTot



def tidalForce(S, t): # Modèle TRES élémentaire
    mWater = 5.972e18 # Masse d'eau déplacée (environ 1 millionième de la masse de la Terre)
    tidePeriod = 12 * 3600 # Marée haute toute les 12 heures

    # Modèle choisi:  un excès de masse mWater se déplace à la période 12 h tout autour de la Terre dans le plan de l'Équateur
    # modifiant donc le champ gravitationnel ressenti par le débris
    highTidePosition = np.array([Earth.R * sin(2 * pi * t / tidePeriod), Earth.R * cos(2 * pi * t / tidePeriod), 0]) # Vecteur
position
    # de l'excèse de masse (marée haute)

    rHighTideDebris = np.array(S.xyz - highTidePosition)

    aTide = rHighTideDebris * (-mWater * G / (np.linalg.norm(rHighTideDebris) ** 3))

    return aTide



def propagation(x, h, t, acceleration, i):
    a = np.array([0, 0, 0])

    S.xyz = x[0:3, 0]
    S.vxyz = x[3:6, 0]

    acceleration[0][:, i] = gravitationPotential(S) # We compute the accelerations
    acceleration[1][:, i] = atmosphericDrag(S)
    acceleration[2][:, i] = sunAndMoonAttraction(S, t)
    acceleration[3][:, i] = tidalForce(S, t)
    acceleration[4][:, i] = radiationPressure(S, t)
    acceleration[5][:, i] = relativistCorrection(S)

    a = np.add(a, acceleration[0][:, i], casting="unsafe") # We sum all the accelerations
    a = np.add(a, acceleration[1][:, i], casting="unsafe")
    a = np.add(a, acceleration[2][:, i], casting="unsafe")
    a = np.add(a, acceleration[3][:, i], casting="unsafe") # Non pris en compte jusqu'à maintenant
    a = np.add(a, acceleration[4][:, i], casting="unsafe") # Non pris en compte jusqu'à maintenant
    a = np.add(a, acceleration[5][:, i], casting="unsafe")

    a = atmosphericNoise(a, h) # We apply a very subtle noise to the total acceleration (cf. atmosphericNoise)

    va = [np.zeros((6, 1))] # Initialisation du vecteur vitesse / accélération renvoyé à RK pour calculer la position du débris
                            # au pas suivant
    va = np.reshape(np.array([[S.vxyz[0]], [S.vxyz[1]], [S.vxyz[2]], [a[0]], [a[1]], [a[2]]]), (6,1))

    return va
```

Objects.py # Méthodes relatives aux objets célestes considérés (planètes / débris)

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

from const import *



class CelestialBody: # Structure defining the planets and the stars
    def __init__(self, name, m, R, de):
        # Class used to instanciate the parameters of the Earth, the Sun, the Moon
        self.name = name # Name of the celestial body (string)
        self.m = m # Mass of the celestial body (in kg)
        self.R = R # Radius of the celestial body (in m)
        self.xyz = np.array([0, 0, 0]) # Position wrt Earth (Earth.xyz = 0, 0, 0)
        self.de = de # Distance to Earth (in m), -1 if not relevant
        self.mu = self.m * G # Gravitational mass parameter
        self.vlib = sqrt(self.mu / self.R) # Liberation velocity



class Satellite: # Defines a space junk / a satellite wrt to an Earth-related referential
    def __init__(self, name, a, e, i, omega, OMEGA, v):
        # Class used to instanciate and keep track of the parameters of a space debris
        self.name = name

        # Master parameters (they fully describe the motion of the satellite)
        self.a = a # Semi-major axis (in m)
        self.e = e # Excentricité (without unit)
        self.i = i # Tilt (in degrees)
        self.omega = omega # Argument of the periapse (in degrees)
        self.OMEGA = OMEGA # Longitude of the ascending node (in degrees)
        self.v = v # True anomaly (in degrees)

        # Slave parameters (calculated from the major parameters)
        self.t = 0 # Date since the reference epoch J200 (in s)
        self.T = 0 # Period (in s)
        self.Ra = 0 # Distance between the Earth and the apogee (in m)
        self.Rp = 0 # Distance between the Earth and the perigee (in m)
        self.Va = 0 # Velocity at the apogee (in m/s)
        self.Vp = 0 # Velocity at the perigee (in m/s)
        self.V = 0 # Velocity (in m /s)
        self.mu = 0 # Mass-Grav constant ()
        self.E = 0 # Mechanical energy (in J)
        self.m = 0 # Mass of the satellite (in kg)

        # Other coordinates
        self.h = 0 # Altitude (wrt the mean sea level) (in m)
        self.R = 0 # Distance between the satellite and the center of mass (in m)
        self.M = 0 # Mean anomaly (in degrees)
        self.n = 0 # Mean motion ()
        self.b = 0 # Semi-minor axis (in m)
        self.E = 0 # Excentric anomaly (in degrees)
        self.omega_bar = 0 # Longitude of the periapse (in degrees)
        self.L = 0 # Mean longitude (in degrees)
        self.latlon = np.array([0, 0, 0]) # Geodesic coordinates (in °, °)

        # Positions
        self.xyz = np.array([0, 0, 0])
        self.sph = np.array([0, 0, 0])

        # Velocities
        self.vxyz = np.array([0, 0, 0])
        self.vsph = np.array([0, 0, 0])

        # Intermediary propagation parameters
        self.evec = np.array([0, 0, 0]) # Excentricity vector
        self.hvec = np.array([0, 0, 0]) # Specific momentum
        self.nvec = np.array([0, 0, 0]) # Node vector

        # Propagation-related parameters
        self.suncs = 0.01 # Sun-exposed cross-section (in m^2)
        self.afcs = 0.01 # Atmospheric flow exposed cross-section (in m^2)
        self.Cd = 2 # Drag coefficient (without unit)

        # Computed parameters
        self.TTE = 0 # Time to Earth (in seconds)
        self.orbitName = "Default" # As a function of the excentricity
        self.orbitBand = "Default" # As a function of the altitude

    def _h_(self): # Computes the current altitude of the space junk
        self.h = _R_() - Earth.R
        return self.h

    def _T_(self): # Computes the orbit period
        self.T = 2 * pi * sqrt(self.a**3 / Earth.mu)
        return self.T

    def _n_(self): # Computes the mean motion
        self.n = sqrt(Earth.mu / (a ** 3))
        return self.n

    def _V_(self): # Computes the velocity vector amplitude
```

```
        self.V = Earth.mu / self.R
        return self.V

    def _R_(self): # ...
        self.R = self.a * (1 - self.e**2) / (1 + self.e * cos(self.v))
        return self.R

    def _Ra_(self): # Computes the apogee radius
        self.Ra = self.a * (1 + self.e)
        return self.Ra

    def _Rp_(self): # Computes the perigee radius
        self.Rp = self.a * (1 - self.e)
        return self.Rp

    def _Vp_(self): # Computes the perigee velocity of the celestial body
        _Ra_()
        _Rp_()

        self.Vp = sqrt(2 * Earth.mu * self.Ra / (self.Rp * (self.Ra + self.Rp)))
        return self.Vp

    def _Va_(self): # Computes the apogee velocity of the celestial body
        _Ra_()
        _Rp_()

        self.Va = sqrt(2 * Earth.mu * self.Rp / (self.Ra * (self.Ra + self.Rp)))
        return self.Va

    def _sph_from_kep_(self): # Computes spherical coordinates based on keplerian elements
        self.sph[0] = _R_() # Radius between the satellite and the Earth's centre
        self.sph[1] = asin(sin(self.i) * sin(self.v)) # Tehta
        self.sph[2] = asin(tan(self.sph[1] / tan(self.i))) # Phi
        return self.sph

    def _xyz_from_sph_(self): # Computes cartesian coordinates based on spherical coordinates
        _sph_()

        self.xyz[0] = self.sph[0] * cos(self.sph[1]) * cos(self.sph[2])
        self.xyz[1] = self.sph[0] * cos(self.sph[1]) * sin(self.sph[2])
        self.xyz[2] = self.sph[0] * sin(self.sph[2])
        return self.xyz

    def _E_(self):
        self.E = self.m * (0.5 * (_V_() ** 2) - Earth.mu / _sph_()[0])
        return self.E

    def _R_(self):
        self.R = _sph_()[0]
        return self.R

    def _E_(self): # Computes E
        self.E = acos((self.e * cos(self.v)) / (1 + self.e * cos(self.v)))
        return self.E

    def _M_(self): # Computes mean anomaly
        _E_()        self.M = self.E - self.e * self.E
        return self.M

    def _lalo_from_xyz_(self): # Computes latitude / longitude from cartesian coordinates
        _xyz_from_sph_()

        self.latlon[0] = math.asin(self.xyz[2] / self.r)
        self.latlon[1] = math.atan(self.xyz[1] / self.xyz[0])
        if self.latlon[1] < 0:
            self.latlon[0] += 2 * pi
        return self.latlon

    def _hvec_(self):
        self.hvec = np.cross(self.xyz, self.vxyz)
        return self.hvec

    def _nvec_(self):
        self.nvec = np.cross([0, 0, 1], _hvec_())
        return self.nvec

    def _evec_(self) :
        terme1 = (np.dot(self.vxyz, self.vxyz) - Earth.mu / np.norm(self.xyz)) * self.xyz
        terme2 = np.dot(self.xyz, self.vxyz) * self.vxyz
        self.evec = (1 / Earth.mu) * (terme1 - terme2)
        return self.evec

    def _kep_(self): # Returns a dictionary giving the Keplerian elements of the celestial body
        self.kep = {name : "Keplerian elements", a : self.a, i : self.i, e : self.e, omega : self.omega, OMEGA : self.OMEGA, v : 
self.v}
        return self.kep

    def _masterParameters_(self): # Computes the six Keplerian elements of the celestial body based on traditional parameters
        _nvec_()
        _evec_()
        self.a = 1 / (2 / np.norm(self.xyz) - np.dot(self.vxyz, self.vxyz) / Earth.mu)
        self.e = np.norm(self.evec)
        self.i = acos(self.hvec[2] / np.norm(self.hvec))
        self.OMEGA = acos(self.nvec[0] / np.norm(self.nvec))
        self.omega = acos(np.dot(self.nvec, self.evec) / (np.norm(self.nvec) * np.norm(self.evec)))
        self.v = acos(np.dot(self.evec, self.xyz) / (np.norm(self.evec) * np.norm(selv.xyz)))
        return _kep_()
```

Const.py # Constantes physiques et définition des principaux corps célestes considérés

```
S = Satellite("Débris Spatial", 0, 0, 0, 0, 0, 0)
S.m = 0.01 # Mass = 10 g
S.suncs = 0.0001 # Sun-exposed cross-section : 10cm2
S.afcs = 0.0001 # Atmospheric flow exposed cross-section : 10cm2
S.Cd = 1.2 # Drag coefficient

G = 6.67428e-20 # Physical constants
sideralDay = 23 * 3600 + 56 * 60 + 4.091
c = 2.99792458e8

J = [-1, 0, 1082.645e-6, -2.546e-6, -1.649e-6] # Zonal coefficients (harmonic graviational field)
radPressureDensity = 9.08e-7

Earth = CelestialBody("Earth", 5.972e24, 6378.137, -1)

Sun = CelestialBody("Sun", 1.989e30, 695510, 1.496e8)
Sun.xyz = np.array([Sun.R, 0, 0]) # On place le Soleil sur l'axe Ox passant par le centre de la Terre

Moon = CelestialBody("Moon", 7.34767309e22, 1737.1, 405696)
Moon.xyz = np.array([0, Moon.R, 0]) # On place la Lune sur l'axe Oy passant par le cntre de la Terre
```