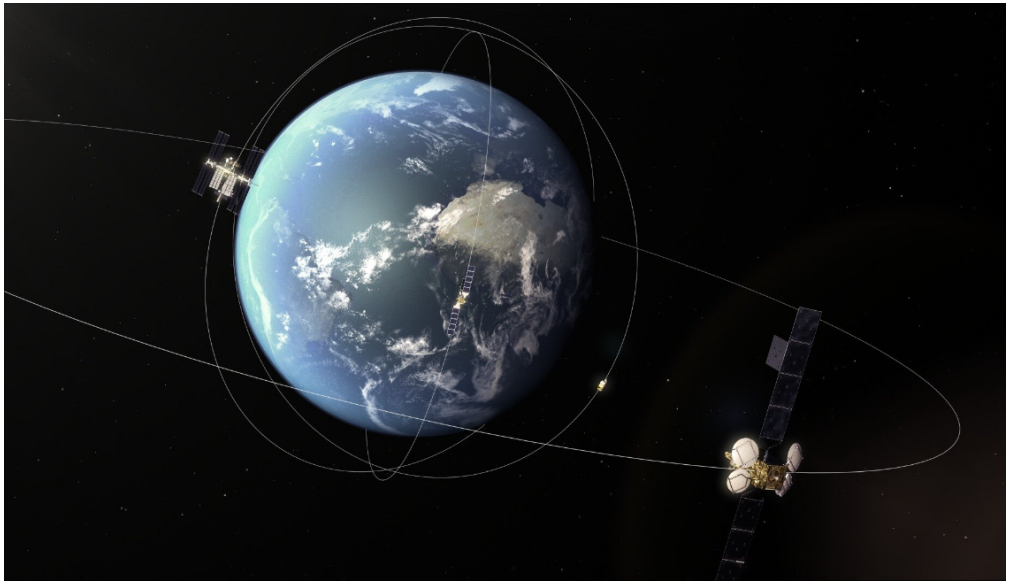
Projet MMSN 9
Trajectoire d'un satellite



Langolff Clément / Kessler Aymeric

Introduction

L'objectif de ce projet et de modéliser la trajectoire d'un satellite en dimension trois. Pour cela, une première partie est dédiée à la recherche des équations régissants le satellite en coordonnées sphériques.

Ensuite, nous implémenterons une méthode numérique pour simuler la trajectoire du satellite.

Enfin, nous effecturons diffèrentes simulations et comparerons nos résultats avec une bibliothèque d'intégration de python.

Entrée [1]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import odeint
from tabulate import tabulate

import time
```

Partie 1 théorie

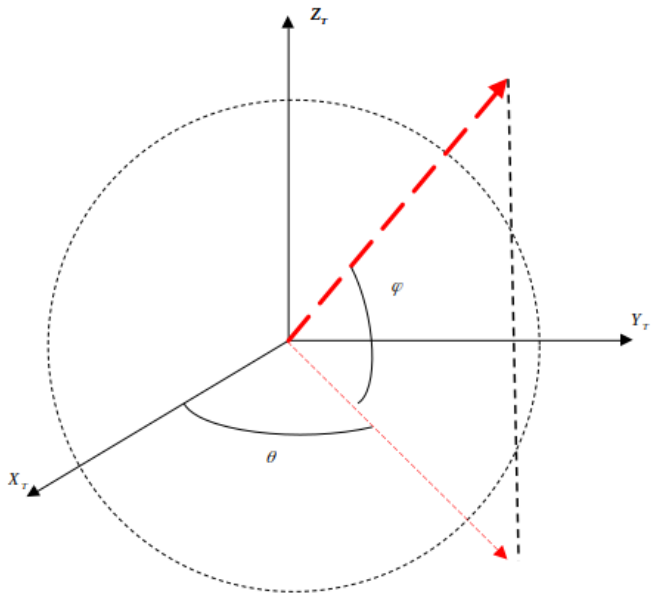
Recherche de la dynamique

Dans ce projet, nous travaillons avec les dimensions suivantes :

- Longueur [L] en km
- Masse [M] en kg
- Temps [T] en secondes

Pour établir les équations diffèrentielles, il nous faut choisir des référentiels où celles-ci seront simples à exprimer.

Notons par $I_T = (O, X_T, Y_T, Z_T)$ le repère cartésien associé à la rotation de la Terre.



Repère géocentrique I_t

Ce repère tourne à une vitesse angulaire Ω_T correspondant à la vitesse de rotation de la Terre supposée constante.

$$\Omega_T = \frac{2\pi}{jour\ sidérale\ (en\ sec)} = \frac{2\pi}{23,96 \times 3600} = 7,295.10^{-5}\ rad/s$$

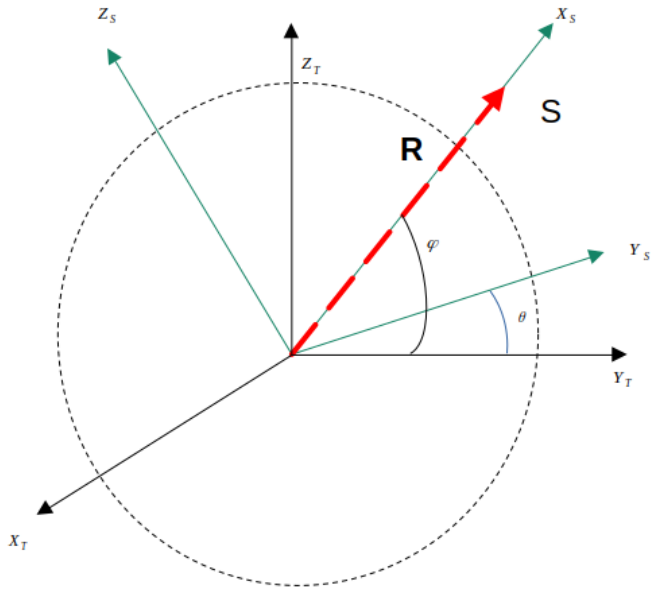
Dans tous ce projet, on note mu le paramètre de gravitation lié à un système Terre-satellite.

$$mu = GM_T$$

où G désigne la constante gravitationelle et M_T la masse de la Terre exprimé en kilogrammes.

Entrée [2]:
```
mu = 398600.4418    # G * Mt  en km³/ s²

ms = 5.10e3         # masse satellite en kg
```

Soit $I_S = (O, X_s, Y_s, Z_s)$ le repère sphérique associé au repère I_T qui resulte de la rotation positive de θ par rapport à Z_T et d'une rotation négative de φ par rapport à l'axe Y_S tel que l'axe x_s coïncide avec le vecteur position noté $\boldsymbol{R}$. Ce repère suit la trajectoire du satellite.



Repère Sphérique I_S

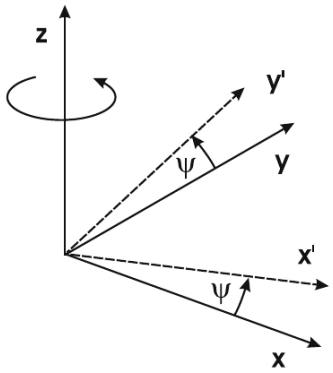
On peut alors exprimer le vecteur position dans le repère I_T:

$$\boldsymbol{R}_{|I_T} = r \begin{pmatrix} cos\varphi cos\theta \\ cos\varphi sin\theta \\ sin\varphi \end{pmatrix}$$

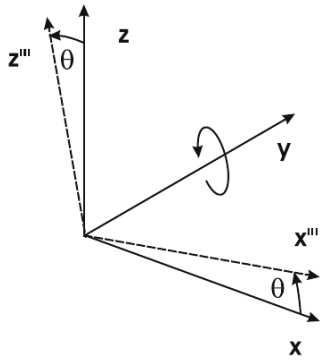
et la vitesse angulaire de la Terre

$$\boldsymbol{\Omega}_{T|I_T} = \Omega_T \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

On peut exprimer les matrice de passage du repère I_T au repère sphérique I_S qui résulte du produit de deux matrices de rotation ; une première rotation suivant l'axe Z_T de θ et une deuxième matrice de rotation suivant l'axe Y_S



$$M_z^{+\psi} = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



$$M_y^{-\theta} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

Notons ce produit des deux matrices de rotation M_{TS} permettant le passage du repère géocentrique au repère sphérique.

$$\begin{aligned} M_{TS} &= M_{TS}^{\varphi} M_{TS}^{\theta} \\ &= \begin{pmatrix} cos\varphi & 0 & sin\varphi \\ 0 & 1 & 0 \\ -sin\varphi & 0 & cos\varphi \end{pmatrix} \begin{pmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} cos\varphi cos\theta & cos\varphi sin\theta & sin\varphi \\ -sin\theta & cos\theta & 0 \\ -sin\varphi cos\theta & -sin\varphi sin\theta & cos\varphi \end{pmatrix} \end{aligned}$$

On retrouve bien évidemment

$$
\begin{aligned}
\boldsymbol{R}_{|I_S} &= M_{TS}\,\boldsymbol{R}_{|I_T} \\
&= \begin{pmatrix} cos\varphi cos\theta & cos\varphi sin\theta & sin\varphi \\ -sin\theta & cos\theta & 0 \\ -sin\varphi cos\theta & -sin\varphi sin\theta & cos\varphi \end{pmatrix} r \begin{pmatrix} cos\varphi sin\theta \\ cos\varphi cos\theta \\ sin\varphi \end{pmatrix} \\
&= r \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}
\end{aligned}
$$

L'expression de la vitesse angulaire de la Terre dans le repère sphérique est donnée par

$$
\begin{aligned}
\boldsymbol{\Omega}_{T|I_S} &= M_{TS}\,\Omega_T \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \\
&= \Omega_T \begin{pmatrix} sin\varphi \\ 0 \\ cos\varphi \end{pmatrix}
\end{aligned}
$$

On note $\boldsymbol{\omega}_{|I_S}$ la vitesse angulaire du repère I_S par rapport au repère I_T.

$$
\boldsymbol{\omega}_{|I_S} = M_{TS}^{\varphi} \begin{bmatrix} 0 \\ 0 \\ \dot{\theta} \end{bmatrix} + M_{TS}^{\theta} \begin{bmatrix} 0 \\ -\dot{\varphi} \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{\theta} sin\varphi \\ -\dot{\varphi} \\ \dot{\theta} cos\varphi \end{bmatrix}
$$

On détermine ensuite l'accélération du mobile dans le repère I_S avec le théorème du transport (wikipedia Transport theorem) (https://en.wikipedia.org/wiki/Transport_theorem)

$$
\begin{aligned}
\boldsymbol{A}_{|I_S} &= M_{TS}\,\boldsymbol{A}_{|I_T} \\
&= M_{TS}\left(\frac{d\boldsymbol{V}_{|I_T}}{dt} + 2\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{V}_{|I_T} + \boldsymbol{\Omega}_{T|I_T}\times(\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{R}_{|I_T})\right) \\
&= \underbrace{M_{TS}\frac{d\boldsymbol{V}_{|I_T}}{dt}}_{1} + \underbrace{2M_{TS}\,\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{V}_{|I_T}}_{2} + \underbrace{M_{TS}\,\boldsymbol{\Omega}_{T|I_T}\times(\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{R}_{|I_T})}_{3})
\end{aligned}
$$

- 1

$$
M_{TS}\frac{d\boldsymbol{V}_{|I_T}}{dt} = \frac{d\boldsymbol{V}_{|I_S}}{dt} + \boldsymbol{\omega}_{|I_S}\times\boldsymbol{V}_{|I_S}
$$

- 2

$$
2M_{TS}\,\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{V}_{|I_T} = 2\boldsymbol{\Omega}_{T|I_S}\times\boldsymbol{V}_{|I_S}
$$

- 3

$$
M_{TS}\,\boldsymbol{\Omega}_{T|I_T}\times(\boldsymbol{\Omega}_{T|I_T}\times\boldsymbol{R}_{|I_T}) = \boldsymbol{\Omega}_{T|I_S}\times(\boldsymbol{\Omega}_{T|I_S}\times\boldsymbol{R}_{|I_S})
$$

Il nous faut déterminer les composantes de la vitesse dans le repère sphérique.

$$
\begin{aligned}
\boldsymbol{V}_{|I_S} &= M_{TS}\frac{d\boldsymbol{R}_{|I_T}}{dt} \\
&= \frac{d\boldsymbol{R}_{|I_S}}{dt} + \boldsymbol{\omega}_{|I_S}\times\boldsymbol{R}_{|I_S} \\
&= \frac{dr}{dt}\begin{pmatrix}1\\0\\0\end{pmatrix} + \begin{pmatrix}0\\r\dot{\theta}cos\varphi\\r\dot{\varphi}\end{pmatrix} \\
&= \begin{pmatrix}\dot{r}\\r\dot{\theta}cos\varphi\\r\dot{\varphi}\end{pmatrix}
\end{aligned}
$$

On calcule les éléments du premier terme

$$\frac{d\boldsymbol{V}_{|I_S}}{dt} = \frac{d^2r}{dt^2}\begin{pmatrix}1\\0\\0\end{pmatrix} + \begin{pmatrix}\dot{\theta}sin\varphi\\-\dot{\varphi}\\\dot{\theta}cos\varphi\end{pmatrix} \times \begin{pmatrix}\dot{r}\\r\dot{\theta}cos\varphi\\r\dot{\varphi}\end{pmatrix}$$

$$= \begin{pmatrix}\ddot{r}\\r\ddot{\theta}cos\varphi + \dot{r}\dot{\theta}cos\varphi - r\dot{\varphi}\dot{\theta}sin\varphi\\r\ddot{\varphi} + \dot{r}\dot{\varphi}\end{pmatrix}$$

$$\boldsymbol{\omega}_{|I_S} \times \boldsymbol{V}_{|I_S} = \begin{pmatrix}\dot{\theta}sin\varphi\\-\dot{\varphi}\\\dot{\theta}cos\varphi\end{pmatrix} \times \begin{pmatrix}\dot{r}\\r\dot{\theta}cos\varphi\\r\dot{\varphi}\end{pmatrix}$$

$$= \begin{pmatrix}-r\dot{\varphi} - r\dot{\theta}^2cos^2\varphi\\\dot{r}\dot{\theta}cos\varphi - r\dot{\varphi}\dot{\theta}sin\varphi\\r\dot{\theta}^2cos\varphi sin\varphi + \dot{r}\dot{\varphi}\end{pmatrix}$$

Puis ceux du deuxième

$$2\boldsymbol{\Omega}_{T|I_S} \times \boldsymbol{V}_{|I_S} = 2\Omega_T \begin{pmatrix}sin\varphi\\0\\cos\varphi\end{pmatrix} \times \begin{pmatrix}\dot{r}\\r\dot{\theta}cos\varphi\\r\dot{\varphi}\end{pmatrix}$$

$$= \begin{pmatrix}-2r\Omega_T\dot{\theta}cos^2\varphi\\-2r\dot{\varphi}\Omega_T sin\varphi + 2\dot{r}\Omega_T cos\varphi\\2r\Omega_T\dot{\theta}cos\varphi sin\varphi\end{pmatrix}$$

Et enfin, le troisième terme

$$\boldsymbol{\Omega}_{T|I_S} \times (\boldsymbol{\Omega}_{T|I_S} \times \boldsymbol{R}_{|I_S}) = \begin{pmatrix} sin\varphi \\ 0 \\ cos\varphi \end{pmatrix} \times \left(\begin{pmatrix} sin\varphi \\ 0 \\ cos\varphi \end{pmatrix} \times r \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right)$$
$$= \begin{pmatrix} -2r\Omega_T^2 cos^2\varphi \\ 0 \\ r\Omega_T^2 cos\varphi sin\varphi \end{pmatrix}$$

En ajoutant tous les termes, nous obtenons l'expressions des composantes de l'accélération du mobile dans le repère sphérique

$$\boldsymbol{A}_{|I_S} = \begin{pmatrix} \ddot{r} - r\dot{\varphi}^2 - rcos^2\varphi(\dot{\theta} + \Omega_T)^2 \\ 2(\dot{\theta} + \Omega_T)(\dot{r}cos\varphi - r\dot{\varphi}sin\varphi) + r\ddot{\theta}cos\varphi \\ 2\dot{r}\dot{\varphi} + r\ddot{\varphi} + rcos\varphi sin\varphi(\Omega_T + \dot{\theta})^2 \end{pmatrix}$$

On peut alors appliquer la deuxième loi de Newton dans le repère sphérique I_S, le mobile étant soumis seulement à la force de gravitation

$$\boldsymbol{F_g} = -G\frac{M_T m_S}{r^2} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{cases} \ddot{r} & = r\dot{\varphi}^2 + rcos^2\varphi(\dot{\theta} + \Omega_T)^2 - G\frac{M_T}{r^2} \\ \ddot{\theta} & = -2(\dot{\theta} + \Omega_T)\left(\frac{\dot{r}}{r} - \dot{\varphi}tan\varphi\right) \\ \ddot{\varphi} & = -2\frac{\dot{r}}{r}\dot{\varphi} - cos\varphi sin\varphi(\Omega_T + \dot{\theta})^2 \end{cases}$$

La vitesse de rotation de la Terre étant faible par rapport à la rotation du satellite, on néglige celle-ci.

$$\begin{cases} \ddot{r} & = r\dot{\varphi}^2 + rcos^2\varphi\dot{\theta}^2 - G\frac{M_T}{r^2} \\ \ddot{\theta} & = -2\dot{\theta}\left(\frac{\dot{r}}{r} - \dot{\varphi}tan\varphi\right) \\ \ddot{\varphi} & = -2\frac{\dot{r}}{r}\dot{\varphi} - cos\varphi sin\varphi\dot{\theta}^2 \end{cases}$$

On transforme le système pour avoir une fonction de $\mathbb{R}^6 \rightarrow \mathbb{R}^6$ définie par

$$f : \begin{pmatrix} r \\ \theta \\ \varphi \\ \dot{r} \\ \dot{\theta} \\ \dot{\varphi} \end{pmatrix} \rightarrow \begin{pmatrix} \dot{r} \\ \dot{\theta} \\ \dot{\varphi} \\ r\dot{\varphi}^2 + rcos^2\varphi\dot{\theta}^2 - G\frac{M_T}{r^2} \\ -2\dot{\theta}\left(\frac{\dot{r}}{r} - \dot{\varphi}tan\varphi\right) \\ -2\frac{\dot{r}}{r}\dot{\varphi} - cos\varphi sin\varphi\dot{\theta}^2 \end{pmatrix}$$

Entrée [3]:
```
def dynamique(Y,t):

    r, theta, phi, vr, vtheta, vphi = Y

    cp = np.cos(phi)
    vtheta2 = vtheta**2


    ar = r * (vphi**2 + cp**2 * vtheta2) - mu / r**2
    atheta = - 2 * vtheta * (vr / r - vphi * np.tan(phi))
    aphi = - 2 * vr * vphi / r - cp * np.sin(phi) * vtheta2

    return  np.array([vr,vtheta,vphi,ar,atheta,aphi])
```

Conversion des paramètres sphériques dans le repère cartèsien

On sait que (Wikipedia Spherical coordinate system) (https://en.wikipedia.org/wiki/Spherical_coordinate_system)

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = sign(y) arccos\left(\frac{x}{\sqrt{x^2 + y^2}}\right)$$

$$\varphi = arcsin\left(\frac{z}{r}\right)$$

en dérivant, nous trouvons

$$\dot{r} = \frac{x\dot{x} + y\dot{y} + z\dot{z}}{\sqrt{x^2 + y^2 + z^2}}$$

$$\dot{\theta} = sign(y)\frac{\dot{x}y - x\dot{y}}{x^2 + y^2}$$

$$\dot{\varphi} = \frac{\dot{z}r - z\dot{r}}{r\sqrt{r^2 - z^2}}$$

Entrée [4]:
```
def XtoR(X,vX) :
    x,y,z = X
    vx,vy,vz = vX

    rxy = np.sqrt(x**2 + y**2)

    r = np.sqrt(x**2 + y**2 + z**2)
    theta =  np.sign(y) * np.arccos(x / rxy)
    phi = np.arcsin(z / r)

    vr = (x * vx + y * vy + z * vz) / r
    vtheta = np.sign(y) * (vx * y - x * vy) / rxy**2
    vphi = (vz * r - z * vr) / (r*np.sqrt(r**2-z**2))


    return np.array([r,theta,phi]),np.array([vr,vtheta,vphi])
```

Inversement, on peut exprimer les coordonnées du repère cartésien en fonction des coordonnées sphériques

$$x = rcos\varphi sin\theta$$

$$y = rcos\varphi cos\theta$$

$$z = rsin\varphi$$

et en dérivant

$$\dot{x} = \dot{r}cos\varphi cos\theta - r\dot{\theta}cos\varphi sin\theta - r\dot{\varphi}cos\theta sin\varphi$$
$$\dot{y} = \dot{r}cos\varphi cos\theta + r\dot{\theta}cos\varphi sin\theta - r\dot{\varphi}sin\theta sin\varphi$$
$$\dot{z} = \dot{r}sin\varphi + r\dot{\varphi}cos\varphi$$

Entrée [5]:
```
def RtoX(R,vR):
    r,theta,phi = R
    vr,vtheta,vphi = vR

    ct = np.cos(theta)
    cp = np.cos(phi)

    st = np.sin(theta)
    sp = np.sin(phi)

    x = r * ct * cp
    y = r * st * cp
    z = r * sp

    vx = vr*ct*cp-r*vtheta*st*cp-r*vphi*ct*sp
    vy = vr*st*cp+r*vtheta*ct*cp-r*vphi*st*sp
    vz = vr*sp + r*vphi*cp

    return np.array([x,y,z]), np.array([vx,vy,vz])
```

Méthode de simulation

Dans ce projet, nous souhaitons comparer les méthodes d'Euler Explicite et de Runge Kutta pour la résolution du mouvement du satellite.

fonction d'Euler Explicite

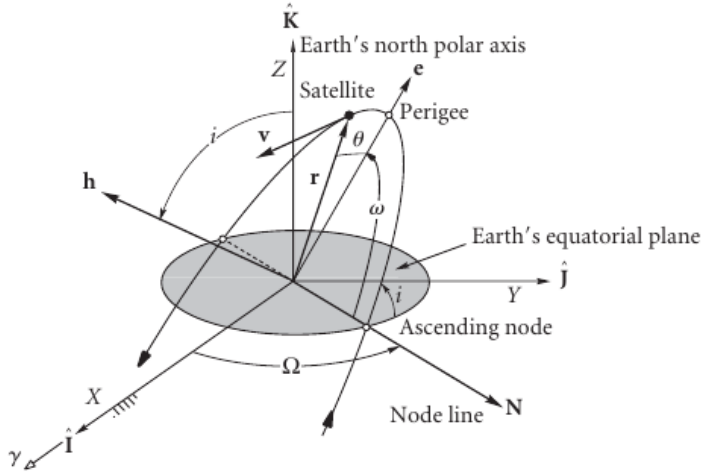
Entrée [6]:
```
def euler_explicite(f, Y0, t):
        d = len(Y0)
        n = len(t)
        Y = np.zeros((d,n))
        Y[:,0] = Y0
        dt = t[1] - t[0]

        for i in range(n-1):
            Y[:,i+1] = Y[:,i] + dt * f(Y[:,i],t[i])

        return Y
```

fonction de Runge Kutta 4

Entrée [7]:
```
def RK4(f, Y0, t) :

    d = len(Y0)
    n = len(t)
    Y = np.zeros((d,n))
    Y[:,0] = Y0
    dt = t[1] - t[0]
    dt_div2 = 0.5 * dt
    dt_div6 = dt / 6
    for i in range(n-1) :
        K1 = f(Y[:,i], t[i])
        K2 = f(Y[:,i] + dt_div2 * K1, t[i] + dt_div2)
        K3 = f(Y[:,i] + dt_div2 * K2, t[i] + dt_div2)
        K4 = f(Y[:,i] + dt * K3, t[i] + dt)

        Y[:,i+1] = Y[:,i] + dt_div6 * (K1 + 2 * K2 + 2 * K3 + K4)

    return Y
```

calcul des paramètres de la trajectoire

Pour se donner une idée de l'allure de la trajectoire, nous implementons une fonction permettant de trouver les paramètres de la trajectoire suivant la position et la vitesse initiale exprimées dans le repère géocentrique. Cette fonction nous permet surtout de déterminer la période T de la trajectoire qui va nous permettre d'integrer sur un intervalle de temps qui correspond exactement à un tour complet autour de la Terre. Les raisonnements pour déterminer ces paramètres n'ont pas été développés, l'intégralité de leurs développements est disponible dans l'ouvrage "Orbital mechanics for engineering students" de Howard Curtis (Page 158).



Entrée [8]:

```
def param(R,V,mu):

    I = np.array([1,0,0])
    J = np.array([0,1,0])
    K = np.array([0,0,1])

    r = np.linalg.norm(R)
    v = np.linalg.norm(V)
    vr = np.dot(R,V) / r

    H = np.cross(R,V)
    h = np.linalg.norm(H)

    i = np.arccos(H[2] / h)

    N = np.cross(K,H)
    n = np.linalg.norm(N)

    if (N[2] >= 0) :
        omega = np.arccos(N[0]/n)
    else :
        omega = 360 - np.arccos(N[0]/n)

    E = ((v**2-mu/r)*R-r*vr*V) / mu
    e = np.linalg.norm(E)

    if (E[2] >= 0) :
        w = np.arccos(np.dot(N,E) / (n*e))
    else :
        w = 360 - np.arccos(np.dot(N,E) / (n*e))

    if (vr >= 0) :
        theta = np.arccos((h**2/(mu*r) - 1) / e)
    else :
        theta = 360 - np.arccos((h**2/(mu*r) - 1) / e)

    return h,e,i*180/np.pi,omega*180/np.pi,w*180/np.pi,theta*180/np.pi
```

Partie 2 Simulations

Pour commencer nos simulations, nous reprenons les conditions initiales d'un exemple de l'ouvrage d'Howard Curtis (Page 184) pour vérifier notre implémentation.

Conditions initiales

$$\begin{cases} X_{0|I_T} & = \begin{pmatrix} -3670 \\ -3870 \\ 4400 \end{pmatrix} \\ VX_{0|I_T} & = \begin{pmatrix} 4,7 \\ -7,4 \\ 1 \end{pmatrix} \end{cases}$$

Entrée [9]:
```
X0 = np.array([-3670,-3870,4400])
VX0 = np.array([4.7,-7.4,1])
R0,VR0 = XtoR(X0,VX0)

r0,theta0,phi0 = R0
vr0,vtheta0,vphi0 = VR0

Y0 = np.array([r0,theta0,phi0,vr0,vtheta0,vphi0])
```

On recherche ensuite les paramètres spécifiques à cette trajectoire.

Entrée [10]:

```
h,e,i,omega,w,theta11 = param(X0,VX0,mu)

vit  = np.linalg.norm(VX0)

rp = h**2 / (mu*(1 + e * np.cos(0)))
ra = h**2 / (mu*(1 + e * np.cos(180 * np.pi /180)))
a =(ra  + rp) / 2
T = 2* np.pi * a**(3/2) / np.sqrt(mu)

resHC = np.array([6914,8.823,58930,0.42607,39.687,130.32,42.373,52.404,15178,6109,10640,10928])


titreY = np.array(["rayon r (km)","vitesse v (km)","moment angulaire (constant) (km²/s)"
                   ,"exentricité e","inclinaison i (deg)","position angulaire noeud ascendant \u03A9 (deg)"
                   "argument de périgée \u03C9 (deg)","angle d'anomalie \u0398 (deg)","rayon apogée ra (km)
                   "rayon périgée rp (km)","axe semi majeur a (km)","période T (sec)"])

titreX = np.array(["/","Simulation","Résultat de Howard Curtis"])

donnée = np.array([r0,vit,h,e,i,omega,w,theta11,ra,rp,a,T])
res = np.array([titreY,donnée,resHC]).T
res = np.concatenate(([titreX],res))
print(tabulate(res,tablefmt='fancy_grid'))
```

/	Simulation	Résultat de Howard Curtis
rayon r (km)	6914.173847973451	6914.0
vitesse v (km)	8.823264701911645	8.823
moment angulaire (constant) (km²/s)	58926.98031462328	58930.0
exentricité e	0.4260716897917769	0.42607
inclinaison i (deg)	39.686895948496385	39.687
position angulaire noeud ascendant Ω (deg)	130.32219193991136	130.32
argument de périgée ω (deg)	42.3725162312213	42.373
angle d'anomalie θ (deg)	52.404125991709506	52.404
rayon apogée ra (km)	15178.643100897327	15178.0
rayon périgée rp (km)	6108.706209169387	6109.0
axe semi majeur a (km)	10643.674655033357	10640.0
période T (sec)	10928.193267863448	10928.0

calcul de la solution et récupération des coordonnées

On commence notre simulation avec un pas de $\frac{T}{N}$ avec $N = 5000$. Pour distinguer l'efficacité des méthodes, nous calculons leur temps d'exécution.

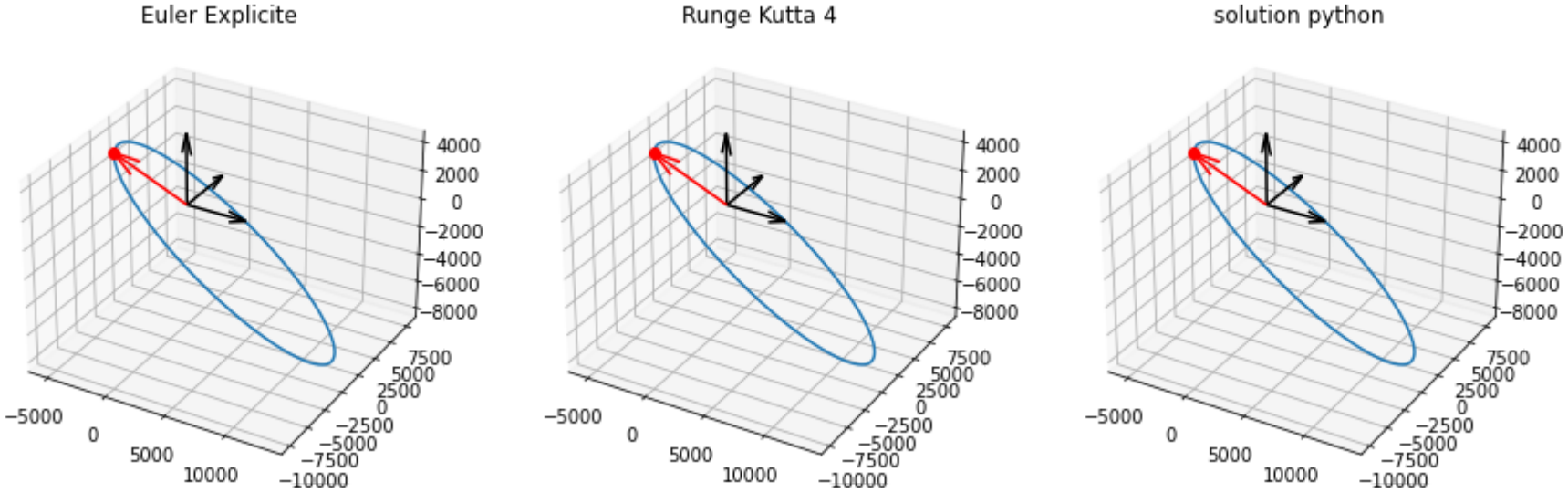
Entrée [11]:
```
N = 5000
temps = np.linspace(0,T,N)

tps1 = time.time()
EE = euler_explicite(dynamique,Y0,temps)
tps2 = time.time()
tpsExecEE = tps2 - tps1


tps1 = time.time()
rk4 = RK4(dynamique,Y0,temps)
tps2 = time.time()
tpsExecRK4 = tps2 - tps1


tps1 = time.time()
solPython = odeint(dynamique,Y0,temps)
tps2 = time.time()
tpsExecSolPython = tps2 - tps1
```

Entrée [12]:
```
# récupération des composantes
rEE = EE[0,:]
rrk4 = rk4[0,:]
rsolPython = solPython[:,0]

thetaEE = EE[1,:]
thetark4 = rk4[1,:]
thetasolPython = solPython[:,1]

phiEE = EE[2,:]
phirk4 = rk4[2,:]
phisolPython = solPython[:,2]

vrEE = EE[3,:]
vrrk4 = rk4[3,:]
vrsolPython = solPython[:,3]

vthetaEE = EE[4,:]
vthetark4 = rk4[4,:]
vthetasolPython = solPython[:,4]

vphiEE = EE[5,:]
vphirk4 = rk4[5,:]
vphisolPython = solPython[:,5]
```

On transforme les coordonnées sphériques dans le repère cartésien

Entrée [13]:
```
XEE,VXEE = RtoX(np.array([rEE,thetaEE,phiEE]),np.array([vrEE,vthetaEE,vphiEE]))

Xrk4,VXrk4 = RtoX(np.array([rrk4,thetark4,phirk4]),np.array([vrrk4,vthetark4,vphirk4]))

XsolPython,VXsolPython = RtoX(np.array([rsolPython,thetasolPython,phisolPython]),
                              np.array([vrsolPython,vthetasolPython,vphisolPython]))
```

Entrée [14]:
```
fig = plt.figure(figsize=(15,15))

ax1 = fig.add_subplot(1,3,1,projection='3d')

ax1.plot(XEE[0,:],XEE[1,:],XEE[2,:])
ax1.plot(XEE[0,0],XEE[1,0],XEE[2,0],'ro')

ax1.quiver(0,0,0,XEE[0,0],XEE[1,0],XEE[2,0],color='r')
ax1.quiver(0,0,0,5000,0,0,color='black')
ax1.quiver(0,0,0,0,5000,0,color='black')
ax1.quiver(0,0,0,0,0,5000,color='black')
ax1.set_title("Euler Explicite")
ax2 = fig.add_subplot(1,3,2,projection='3d')

ax2.plot(Xrk4[0,:],Xrk4[1,:],Xrk4[2,:])
ax2.plot(Xrk4[0,0],Xrk4[1,0],Xrk4[2,0],'ro')

ax2.quiver(0,0,0,Xrk4[0,0],Xrk4[1,0],Xrk4[2,0],color='r')
ax2.quiver(0,0,0,5000,0,0,color='black')
ax2.quiver(0,0,0,0,5000,0,color='black')
ax2.quiver(0,0,0,0,0,5000,color='black')
ax2.set_title("Runge Kutta 4")

ax3 = fig.add_subplot(1,3,3,projection='3d')

ax3.plot(XsolPython[0,:],XsolPython[1,:],XsolPython[2,:])
ax3.plot(XsolPython[0,0],XsolPython[1,0],XsolPython[2,0],'ro')

ax3.quiver(0,0,0,XsolPython[0,0],XsolPython[1,0],XsolPython[2,0],color='r')
ax3.quiver(0,0,0,5000,0,0,color='black')
ax3.quiver(0,0,0,0,5000,0,color='black')
ax3.quiver(0,0,0,0,0,5000,color='black')
ax3.set_title("solution python")

plt.show()
```



Étant donné que l'on calcule nos trajectoires sur une période équivalente à un tour complet, nous allons évaluer l'erreur de chaque méthode en calculant la distance entre le dernier point calculé et le point initial.

Entrée [15]:
```
errDistEE = np.linalg.norm(X0-XEE[:,-1])
errDistrk4 = np.linalg.norm(X0-Xrk4[:,-1])
errDistSolPython = np.linalg.norm(X0-XsolPython[:,-1])


tabErr = np.array(["erreur en mètre",errDistEE,errDistrk4,errDistSolPython])
tabTpsExec = np.array(["temps d'exécution",tpsExecEE,tpsExecRK4,tpsExecSolPython])
titre = np.array(["/","Euler EXplicite","Runge Kutta 4","solution Python"])
print(tabulate(([titre,tabErr,tabTpsExec]),tablefmt='fancy_grid'))
```

/	Euler EXplicite	Runge Kutta 4	solution Python
erreur en mètre	407.643685769243	7.71403550795853e-08	0.031239497932548935
temps d'exécution	0.31121182441711426	1.1629483699798584	0.023441791534423828

L'erreur de distance n'est peut être pas visible dans les graphes entre les diffèrentes méthodes mais elle est clairement distinguable dans le tableau ci-dessus. Sans Surprise, la méthode d'Euler Explicite est la moins précise mais on peut noter qu'elle ne s'en sort pas trop mal en comparaison de la dimension des distances (407 mètres à comparer avec le rayon de périgée de l'ordre de 6000km).
Les méthodes de Runge-Kutta et de la librairie python sont en revanche extrêmement précises, de l'ordre du nanomètre pour la méthode rk4 et au centimètre près pour la librairie python. Cependant, il est intéressant de noter que la méthode utilisée par python est 100 fois plus rapide à l'exécution. La méthode utilisée par la fonction odeint provient de la fonction lsoda codée en FORTRAN qui utilise la méthode d'Adam ou de Gears suivant la stabilité du problème.

Analyse de l'erreur

Pour tester la précision numérique des méthodes, nous effectuons plusieurs tests en changeant le nombre de pas.

Entrée [16]:
```
def calculErreur(listN,Y0) :

    h,e,i,omega,w,theta11 = param(X0,VX0,mu)
    rp = h**2 / (mu*(1 + e * np.cos(0)))
    ra = h**2 / (mu*(1 + e * np.cos(180 * np.pi /180)))
    a =(ra  + rp) / 2
    T = 2* np.pi * a**(3/2) / np.sqrt(mu)
    tabTpsExec = np.zeros((len(listN),3))
    tabErreur =  np.zeros((len(listN),3))
    i = 0
    for N in (listN) :
        temps = np.linspace(0,T,N)

        # calcul des solutions
        tps1 = time.time()
        EE = euler_explicite(dynamique,Y0,temps)
        tps2 = time.time()
        tpsExecEE = tps2 - tps1

        tps1 = time.time()
        rk4 = RK4(dynamique,Y0,temps)
        tps2 = time.time()
        tpsExecRK4 = tps2 - tps1

        tps1 = time.time()
        solPython = odeint(dynamique,Y0,temps)
        tps2 = time.time()
        tpsExecSolPython = tps2 - tps1

        tabTpsExec[i,:] = tpsExecEE,tpsExecRK4,tpsExecSolPython

        #recupération du dernier point pour chaques méthodes
        XEE,VXEE = RtoX(np.array([EE[0,-1],EE[1,-1],EE[2,-1]]),np.array([EE[3,-1],EE[4,-1],EE[5,-1]]))
        Xrk4,VXrk4 = RtoX(np.array([rk4[0,-1],rk4[1,-1],rk4[2,-1]]),np.array([rk4[3,-1],rk4[4,-1],rk4[5,-1]
        XsolPython,VXsolPython = RtoX(np.array([solPython[-1,0],solPython[-1,1],solPython[-1,2]]),
                              np.array([solPython[-1,3],solPython[-1,4],solPython[-1,5]]))

        #calcul de l'erreur
        errDistEE = np.linalg.norm(X0-XEE)
        errDistrk4 = np.linalg.norm(X0-Xrk4)
        errDistSolPython = np.linalg.norm(X0-XsolPython)
```

```
        tabErreur[i,:] = errDistEE,errDistrk4,errDistSolPython

        i+=1

    return tabErreur,tabTpsExec
```

calcul des solutions pour diffèrents N

Entrée [17]:
```
X0 = np.array([-3670,-3870,4400])
VX0 = np.array([4.7,-7.4,1])
R0,VR0 = XtoR(X0,VX0)

r0,theta0,phi0 = R0
vr0,vtheta0,vphi0 = VR0

Y0 = np.array([r0,theta0,phi0,vr0,vtheta0,vphi0])
listN = (500,1000)

# récupération des erreurs et du temps d'exécution
tabErr,tabTps = calculErreur(listN,Y0)
```

Entrée [18]:
```
titreErr = ("N","errEE","errRK4","errSolPython")
titreTps = ("N","temps EE","temps RK4","temps SolPython")

tabErr = np.concatenate(([listN],tabErr.T))
tabTps = np.concatenate(([listN],tabTps.T))

print(tabulate(np.concatenate(([titreErr],tabErr.T)),tablefmt='fancy_grid'))
print(tabulate(np.concatenate(([titreTps],tabTps.T)),tablefmt='fancy_grid'))
```

N	errEE	errRK4	errSolPython
500.0	3895.465418834584	0.000534381313921392	0.07215232881833882
1000.0	1999.5934635538908	4.1425295505037226e-05	0.03496322866861213

N	temps EE	temps RK4	temps SolPython
500.0	0.03168296813964844	0.1276836395263672	0.023282766342163086
1000.0	0.0625307559967041	0.21140694618225098	0.016016721725463867

Nous pouvons remarquer que l'erreur de la méthode d'Euler Explicite est divisée par 2 lorsque l'on multiplie le nombre de pas par 2. Le temps, quand à lui, n'est pas doublé. En revanche, il est doublé pour la méthode de Runge Kutta 4. Étonnemment, il diminue pour la méthode de la librairie Python.

Voici un capture d'écran pour un pas doublé jusqu'à une valeur de 64000.

N	errEE	errRK4	errSolPython
500.0	3895.465418834584	0.000534381313921392	0.07215232881833882
1000.0	1999.5934635538908	4.1425295505037226e-05	0.03496322866861213
2000.0	1012.017547860257	2.837312642207394e-06	0.03480577837181398
8000.0	255.21324607606593	1.1383749691801447e-08	0.04109241903951886
16000.0	127.78710999097255	2.204522526643112e-09	0.07403919493440667
32000.0	63.9384757717751	6.839586422818309e-09	0.38530558720266816
64000.0	31.980442594936772	3.5038989255176917e-09	0.342606525281317

N	temps EE	temps RK4	temps SolPython
500.0	0.04796957969665527	0.13486909866333008	0.0181119441986084
1000.0	0.055347442626953125	0.22100138664245605	0.01662611961364746
2000.0	0.10001754760742188	0.47533535957336426	0.017800569534301758
8000.0	0.4309980869293213	1.7721710205078125	0.017560958862304688
16000.0	0.7866976261138916	3.557140350341797	0.018518686294555664
32000.0	1.5784447193145752	6.978183746337891	0.021877765655517578
64000.0	3.166468381881714	13.989160060882568	0.025667428970336914

L'erreur faite par la méthode d'Euler Explicite est bien divisée par 2 à chaques fois. Paradoxalement, l'erreur de la méthode de la librairie Python ne fait que d'augmenter lorsque l'on augmente le pas. L'erreur de la méthode de Runge Kutta 4 à l'air de converger mais il n'y a aucun intêret à augmenter le nombre de pas puisque le temps d'exécution ne fait que de doubler pour une précision déjà excellente.

Repère périfocale

Par la suite, nous souhaitons pouvoir éxercer plus de force sur le satellite que la force de gravitation. Une force de poussée moteur par exemple. Pour cela nous allons faciliter la tache en procédant à une projection de la trajectoire en 3 dimensions sur le plan parallèle à la trajectoire. Ce plan dispose d'un nouveau repère P (p_x et p_y) que l'on nomme repère périfocale.

On utilise pour cela une matrice de rotation que l'on calcule grâce aux paramètres calculés précédéments. On effectue d'abord une rotation suivant Ω,puis une rotation suivant l'inclinaison i et on aligne les vecteurs unitaires avec le semi-axe majeur en effectuant une rotation de ω. Nous obtenons alors comme matrice (ref Pages 174, Howard Curtis)

$$Q_{P\to T} = \begin{pmatrix} cos(\Omega)cos(\omega) - sin(\Omega)sin(\omega)cos(i) & -cos(\Omega)sin(\omega) - sin(\Omega)cos(i)cos(\omega) & sin(\Omega)sin(i) \\ sin(\Omega)cos(\omega) + cos(\Omega)sin(\omega)cos(i) & -sin(\Omega)sin(\omega) + cos(\Omega)cos(\omega)cos(i) & -cos(\Omega)sin(i) \\ sin(i)sin(\omega) & sin(i)cos(\omega) & cos(i) \end{pmatrix}$$

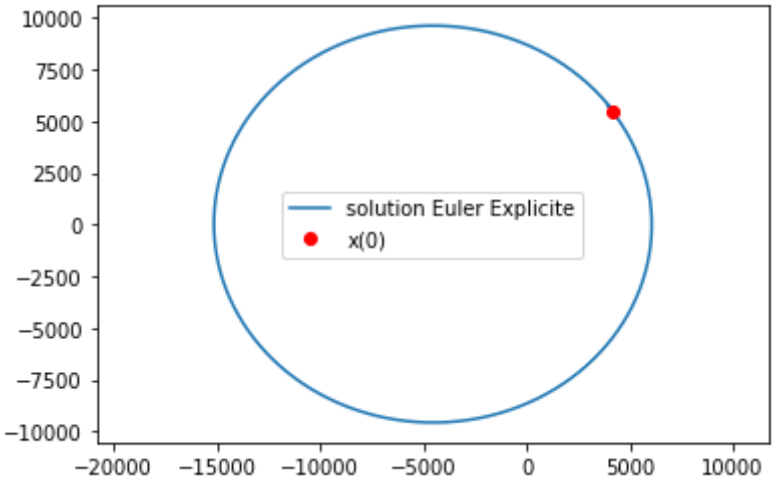
Puisque la matrice Q est unitaire, on peut faire le retour du repère périfocale vers le repère terrestre en prenant sa transposé.

Entrée [19]:
```
def RotationMat(omega,i,w):   # en degrès
    i = i * np.pi / 180
    omega = omega * np.pi / 180
    w = w * np.pi / 180

    co = np.cos(omega)
    cw = np.cos(w)
    ci = np.cos(i)

    so = np.sin(omega)
    sw = np.sin(w)
    si = np.sin(i)

    Q11 =   co * cw - so * sw * ci
    Q12 = - co * sw - so * ci * cw
    Q13 =   so * si
    Q21 =   so * cw + co * ci * sw
    Q22 = - so * sw + co * ci * cw
    Q23 = - co * si
    Q31 =   si * sw
    Q32 =   si * cw
    Q33 =   ci

    Q = np.array([[Q11,Q12,Q13],
                  [Q21,Q22,Q23],
                  [Q31,Q32,Q33]])

    return Q  #Qp→T
```

Entrée [20]:
```
Q = RotationMat(omega,i,w)
```

On effectue une projection de la solution calculée par la méthode d'Euler Explicite sur le plan périfocale.

Entrée [21]:
```
per = np.matmul(Q.T,XEE).T
plt.plot(per[:,0],per[:,1],label="solution Euler Explicite")
plt.plot(per[0,0],per[0,1],'ro',label="x(0)")
plt.axis('equal')
plt.legend()
plt.show()
```



Théoriquement, cette matrice annule la composante z pour chaques vecteurs calculés. On définit alors un nouveau type d'erreur en sommant la dernière composante de tous les vecteurs pour chaques méthodes.

Entrée [22]:
```
# projection sur le plan orbital
perEE = np.matmul(Q.T,XEE).T
perRK4 = np.matmul(Q.T,Xrk4).T
perSolPython = np.matmul(Q.T,XsolPython).T
```

Entrée [23]:
```
# calcul de l'erreur
ErrPerEE = np.abs(np.sum(perEE[:,2]))
ErrPerRK4 = np.abs(np.sum(perRK4[:,2]))
ErrPerSolPython = np.abs(np.sum(perSolPython[:,2]))
```

Entrée [24]:
```
tabErrPer = np.array([ErrPerEE,ErrPerRK4,ErrPerSolPython])
titre = ("erreur EE","erreur RK4","erreur sol Python")
print(tabulate([titre,tabErrPer],tablefmt='fancy_grid'))
```

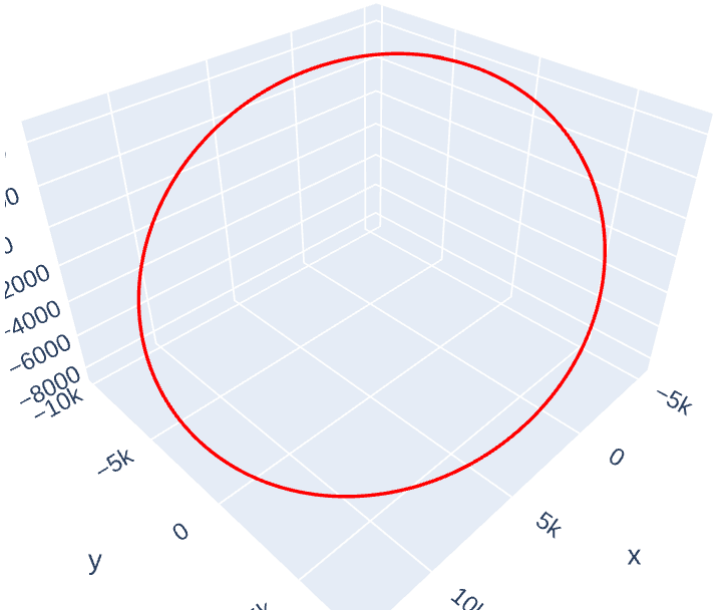
erreur EE	erreur RK4	erreur sol Python
66638.81296558293	2.849568226870325e-05	12.248313436602475

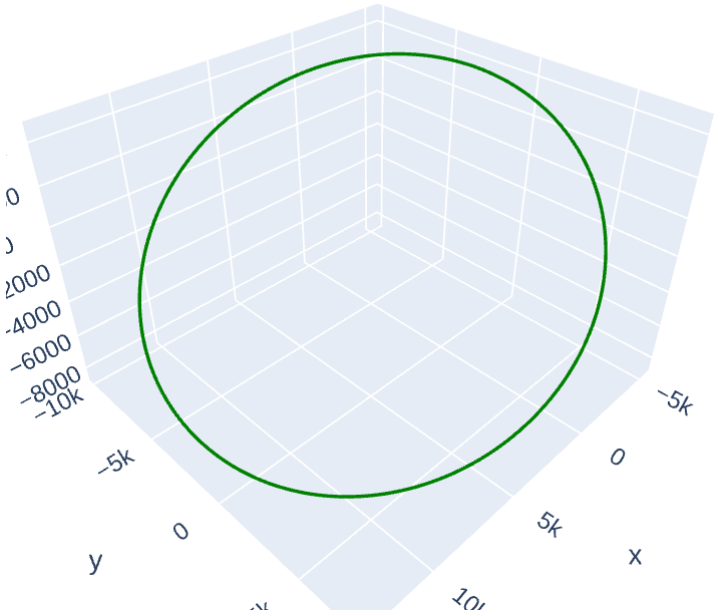
Nous tirons les mêmes conclusions que précédement, la méthode de Runge Kutta 4 est bien la plus précise, aux prix du coût de calcul.

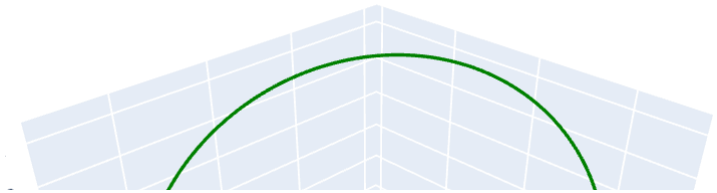
Ci dessous, une visualisation interactive de la trajectoire pour chaques méthodes.

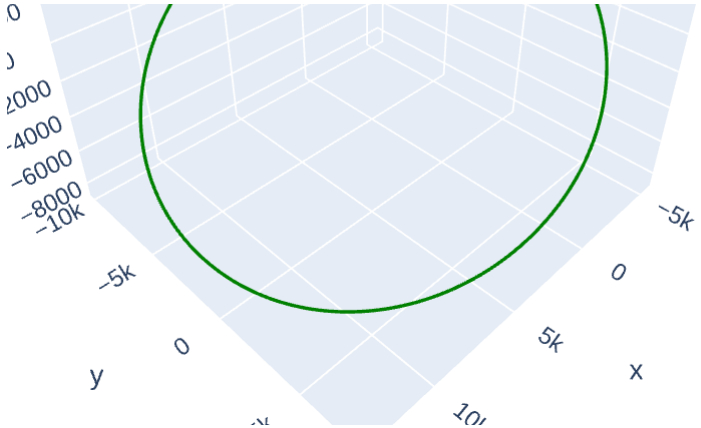
Entrée [25]:
```
import plotly.graph_objects as go
from plotly.subplots import make_subplots

fig = make_subplots(
    rows=2, cols=2,
    specs=[[{'type': 'scene'}, {'type': 'scene'}],[{'type': 'scene'},{'type': 'scene'}]])


fig.add_trace(
    go.Scatter3d(x=XEE[0,:],y=XEE[1,:],z=XEE[2,:],marker=dict(size=1,color="red")),
    row=1, col=1)

fig.add_trace(
    go.Scatter3d(x=Xrk4[0,:],y=Xrk4[1,:],z=Xrk4[2,:],marker=dict(size=1,color="green")),
    row=1, col=2)

fig.add_trace(
    go.Scatter3d(x=XsolPython[0,:],y=XsolPython[1,:],z=XsolPython[2,:],marker=dict(size=1,color="green")),
    row=2, col=1)


fig.update_layout(
    title_text='Orbite avec les diffèrentes méthodes',
    height=1000,
    width=1000
)
```

Orbite avec les diffèrentes méthodes









Simulation 2

On souhaite simuler la trajectoire du satellite sur une orbite géostationnaire. Pour cela nous devons trouver les bonnes conditions initiales. Être sur une orbite géostationnaire signifie rester à la même position au dessus de la Terre. Ainsi le satellite doit avoir une vitesse angulaire exactement égale à celle de la Terre, sa période vaut alors un jours.

Entrée [26]:
```
Tgeo = 23.96 * 3600
```

La vitesse angulaire ω_{geo} est lié à la période par la formule

$$\omega_{geo} = \frac{2\pi}{T}$$

Entrée [27]:
```
wgeo = 2 * np.pi / Tgeo
```

On peut alors trouver l'altitude de l'orbite grâce à la 3ème loi de Kepler

$$T = \frac{2\pi}{\sqrt{\mu}} r^{\frac{3}{2}}$$

$$r_{geo} = \left(\frac{T_{geo}^2 \mu}{4\pi^2} \right)^{\frac{1}{3}} = \left(\frac{\mu}{\omega_{geo}^2} \right)^{\frac{1}{3}}$$

Entrée [28]:
```
rgeo = (mu / wgeo**2)**(1/3)
```

Conditions initiales

$$\begin{cases} X_{0|I_S} & = \begin{pmatrix} r_{geo} \\ 0 \\ 0 \end{pmatrix} \\ VX_{0|I_S} & = \begin{pmatrix} 0 \\ \omega_{geo} \\ 0 \end{pmatrix} \end{cases}$$

Entrée [29]:
```
R0 = np.array([rgeo,0,0])
VR0 = np.array([0,wgeo,0])
```

Entrée [30]:
```
X0,VX0 = RtoX(R0,VR0)

h,e,i,omega,w,theta11 = param(X0,VX0,mu)

vit  = np.linalg.norm(VX0)

rp = h**2 / (mu*(1 + e * np.cos(0)))
ra = h**2 / (mu*(1 + e * np.cos(180 * np.pi /180)))
a =(ra  + rp) / 2   #en km
T = 2* np.pi * a**(3/2) / np.sqrt(mu)    #en seconde

titreY = np.array(["rayon r (km)","vitesse v (km)","moment angulaire (constant) (km²/s)"
                   ,"exentricité e","inclinaison i (deg)","position angulaire noeud ascendant \u03A9 (deg)"
                   "argument de périgée \u03C9 (deg)","angle d'anomalie \u0398 (deg)","rayon apogée ra (km)
                   "rayon périgée rp (km)","axe semi majeur a (km)","période T (sec)"])

donnée = np.array([r0,vit,h,e,i,omega,w,theta11,ra,rp,a,T])
res = np.array([titreY,donnée]).T
print(tabulate(res,tablefmt='fancy_grid'))
```

rayon r (km)	6914.17
vitesse v (km)	3.07357
moment angulaire (constant) (km²/s)	129687
exentricité e	1.5043e-15
inclinaison i (deg)	0
position angulaire noeud ascendant Ω (deg)	nan
argument de périgée ω (deg)	nan
angle d'anomalie θ (deg)	nan
rayon apogée ra (km)	42194.1
rayon périgée rp (km)	42194.1
axe semi majeur a (km)	42194.1
période T (sec)	86256

```
/tmp/ipykernel_17671/445587266.py:20: RuntimeWarning:
invalid value encountered in scalar divide
/tmp/ipykernel_17671/445587266.py:28: RuntimeWarning:
invalid value encountered in scalar divide
/tmp/ipykernel_17671/445587266.py:33: RuntimeWarning:
invalid value encountered in arccos
```

On retrouve ici l'excentricité typique d'une orbite circulaire (e = 0) ainsi que la période de rotation de la Terre. Les conditions initiales étant posées pour que la trajectoire reste dans le plan équatoriale, la position angulaire du noeud ascendant, l'argument de périgé et l'angle d'anomalie n'ont plus de sens dans ce contexte, ils ne sont pas définis.

Entrée [31]:
```
N = 5000
temps = np.linspace(0,T,N)
Y0 = np.array([R0[0],R0[1],R0[2],VR0[0],VR0[1],VR0[2]])

# calcul des solutions
tps1 = time.time()
EE = euler_explicite(dynamique,Y0,temps)
tps2 = time.time()
tpsExecEE = tps2 - tps1


tps1 = time.time()
rk4 = RK4(dynamique,Y0,temps)
tps2 = time.time()
tpsExecRK4 = tps2 - tps1


tps1 = time.time()
solPython = odeint(dynamique,Y0,temps)
tps2 = time.time()
tpsExecSolPython = tps2 - tps1
```

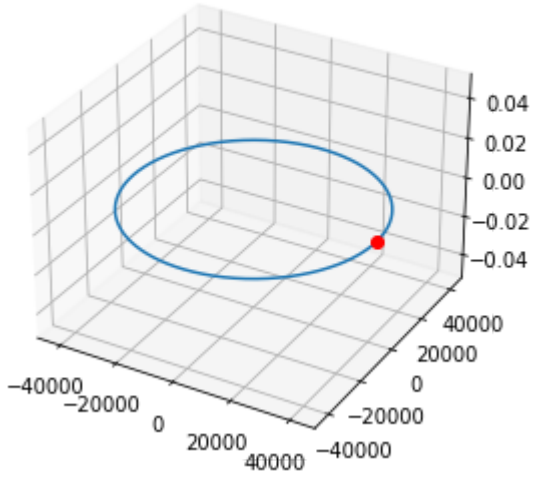
Entrée [32]:
```
# récupération des composantes
rEE = EE[0,:]
rrk4 = rk4[0,:]
rsolPython = solPython[:,0]

thetaEE = EE[1,:]
thetark4 = rk4[1,:]
thetasolPython = solPython[:,1]

phiEE = EE[2,:]
phirk4 = rk4[2,:]
phisolPython = solPython[:,2]

vrEE = EE[3,:]
vrrk4 = rk4[3,:]
vrsolPython = solPython[:,3]

vthetaEE = EE[4,:]
vthetark4 = rk4[4,:]
vthetasolPython = solPython[:,4]

vphiEE = EE[5,:]
vphirk4 = rk4[5,:]
vphisolPython = solPython[:,5]
```

Entrée [33]:
```
# conversion dans le repère géocentrique
XEE,VXEE = RtoX(np.array([rEE,thetaEE,phiEE]),np.array([vrEE,vthetaEE,vphiEE]))

Xrk4,VXrk4 = RtoX(np.array([rrk4,thetark4,phirk4]),np.array([vrrk4,vthetark4,vphirk4]))

XsolPython,VXsolPython = RtoX(np.array([rsolPython,thetasolPython,phisolPython]),
                              np.array([vrsolPython,vthetasolPython,vphisolPython]))
```

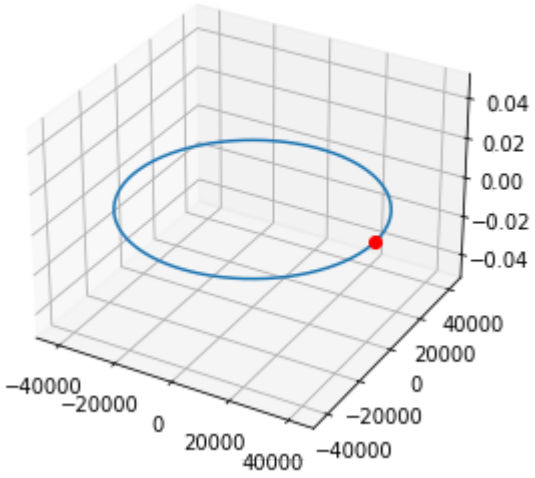
Entrée [34]:

```
fig = plt.figure(figsize=(15,15))

ax1 = fig.add_subplot(1,3,1,projection='3d')

ax1.plot(XEE[0,:],XEE[1,:],XEE[2,:])
ax1.plot(XEE[0,0],XEE[1,0],XEE[2,0],'ro')



ax1.set_title("Euler Explicite")
ax2 = fig.add_subplot(1,3,2,projection='3d')

ax2.plot(Xrk4[0,:],Xrk4[1,:],Xrk4[2,:])
ax2.plot(Xrk4[0,0],Xrk4[1,0],Xrk4[2,0],'ro')


ax2.set_title("Runge Kutta 4")

ax3 = fig.add_subplot(1,3,3,projection='3d')

ax3.plot(XsolPython[0,:],XsolPython[1,:],XsolPython[2,:])
ax3.plot(XsolPython[0,0],XsolPython[1,0],XsolPython[2,0],'ro')


ax3.set_title("solution python")

plt.show()
```

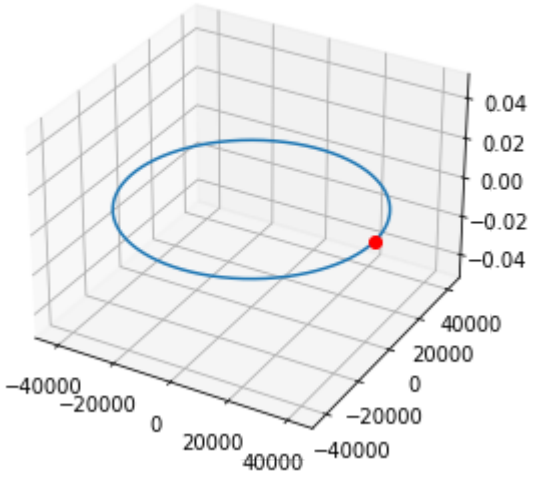
Euler Explicite



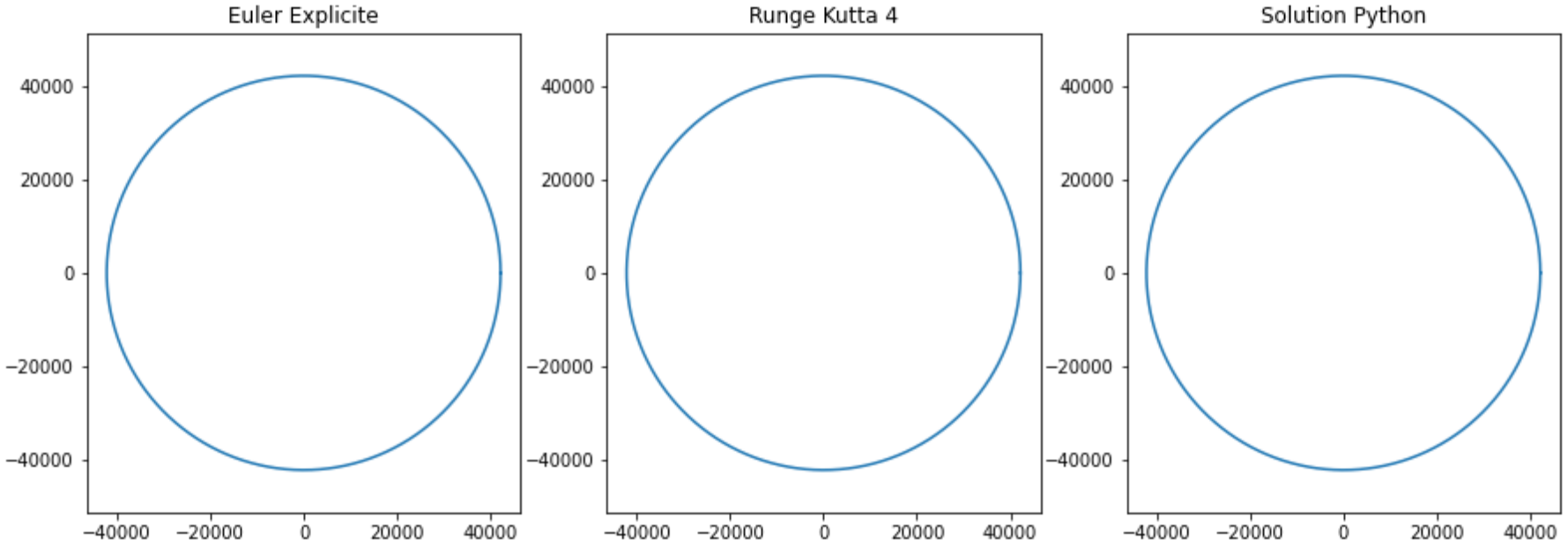
Runge Kutta 4



solution python



Entrée [35]:
```
fig = plt.figure(figsize=(15,5))
ax1 = fig.add_subplot(1,3,1)

ax1.axis('equal')
ax1.plot(XEE[0],XEE[1])
ax1.set_title("Euler Explicite")
ax2 = fig.add_subplot(1,3,2)
ax2.axis('equal')
ax2.plot(Xrk4[0],Xrk4[1])
ax2.set_title("Runge Kutta 4")

ax3 = fig.add_subplot(1,3,3)
ax3.axis('equal')
ax3.plot(XsolPython[0],XsolPython[1])
ax3.set_title("Solution Python")
plt.show()
```



Nos trajectoires sont bien circulaires, cherchons à calculer les erreurs sur cette orbite.

Entrée [36]:
```
errDistEE = np.linalg.norm(X0-XEE[:,-1])
errDistrk4 = np.linalg.norm(X0-Xrk4[:,-1])
errDistSolPython = np.linalg.norm(X0-XsolPython[:,-1])


tabErr = np.array(["erreur en mètre",errDistEE,errDistrk4,errDistSolPython])
tabTpsExec = np.array(["temps d'exécution",tpsExecEE,tpsExecRK4,tpsExecSolPython])
titre = np.array(["/","Euler EXplicite","Runge Kutta 4","solution Python"])
print(tabulate(([titre,tabErr,tabTpsExec]),tablefmt='fancy_grid'))
```

/	Euler EXplicite	Runge Kutta 4	solution Python
erreur en mètre	1.376274889411593e-09	1.376274889411593e-09	4.590292159301864e-07
temps d'exécution	0.3372986316680908	1.110645055770874	0.0020923614501953125

Cette fois ci la méthode d'Euler Explicite est la plus précise ! La solution python reste cependant bien plus rapide même si on perd en précision.

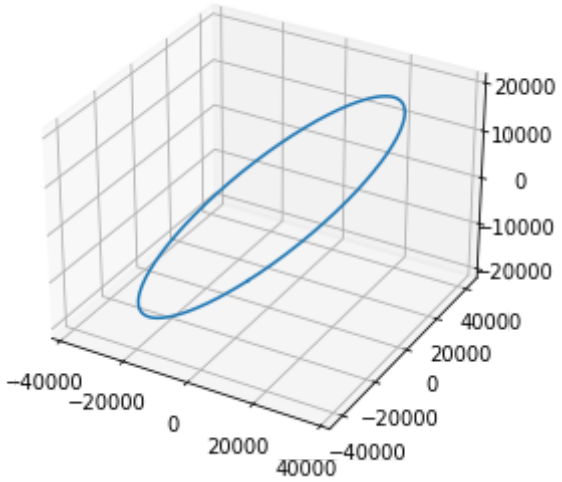
Grâce à la matrice de passage du repère périfocale vers le repère géocentrique, nous pouvons maintenant donner des inclinaisons à notre orbite géostationnaire. L'orbite étant circulaire, changer la valeur de l'argument du périgé ω n'aura aucune incidence. En revanche, nous pouvons changer l'inclinaison i ainsi que la position du noeud ascendant Ω.

Entrée [37]:
```
Q = RotationMat(-70,30,0)  # varOmega, i , w
```

On calcule les nouvelles composantes dans le repère géocentrique pour la solution de Runge Kutta 4.

Entrée [38]:
```
InvPer = np.matmul(Q,Xrk4)
```

Entrée [39]:
```
fig = plt.figure(figsize=(15,15))

ax1 = fig.add_subplot(1,3,1,projection='3d')
ax1.plot(InvPer[0,:],InvPer[1,:],InvPer[2,:])
plt.show()
```



Conclusion

Pour conclure, ce projet a été très intéressant puisqu'il a touché à diffèrents domaines d'étude comme la physique, les mathématiques et la programmation.
Nous nous somme rendu compte que l'étude de trajectoire d'un satellite en 3 dimensions est bien plus compliquée que l'étude en 2 dimensions.
Concernant l'étude numérique, nous nous sommes rendu compte qu'il n'y avait pas de bonne ou mauvaise méthode pour la résolution des équations diffèrentielles. Tout dépend du problème posé, des conditions initiales et surtout du niveau de précision éxigé. Cependant la méthode qui allie efficacité et précision pour le cadre de notre étude est la méthode implémenté par python.

Nous remercions Madame Zidani pour avoir proposé ce sujet à qui nous portons un grand interêt.