# CubeSat Attitude Control Simulation

1. Objective
2. Process Flowchart
3. Key Equations
4. Code Overview
5. Preliminary Results
6. Next Steps

**Yu Jun**
YUAA

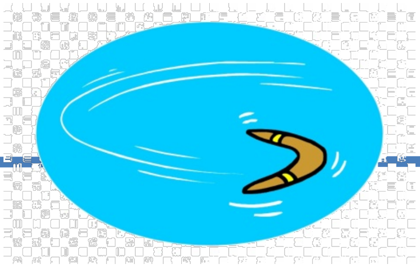
# Objective

Simulate effectiveness of **various control strategies** to point CubeSat correctly

- *For a given initial spin, how fast can it stop?*
- **Tailored** to our orbit and mechanical design
- Not flight software

Currently assumes perfect sensor, no lag in controller execution (can model B-dot implementation .etc if required)

# Problem Description

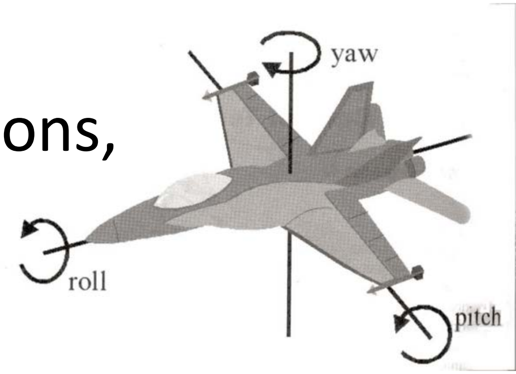


We need to consider both

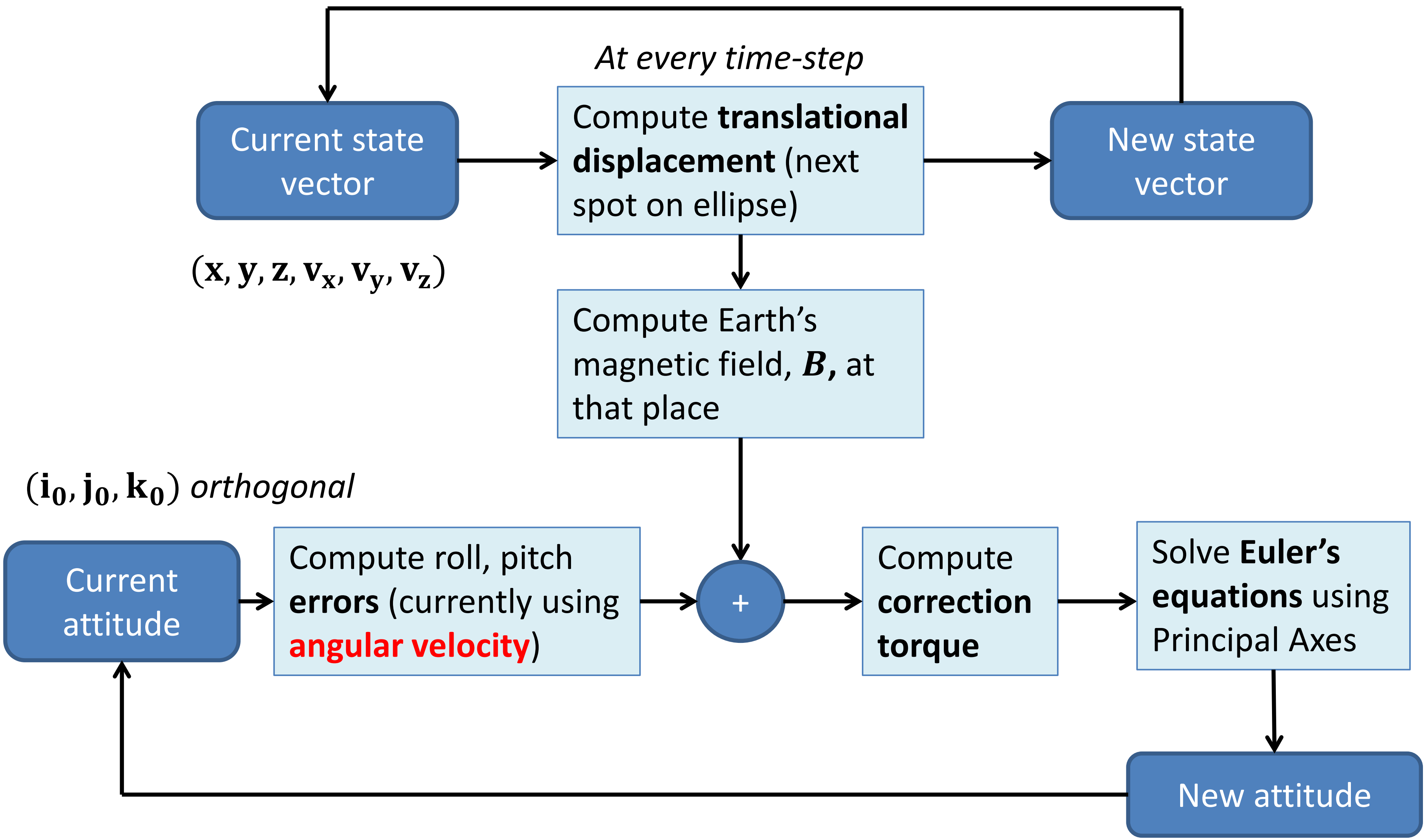
- **Translational** motion along the orbit (ellipse)
- **Rotational** motion (tumbling in space)

While orbit is a "textbook" **2-body problem**, we need to determine how much torque to apply – given Earth's varying **magnetic field vector** at different points

2-axis stabilization -> stop **roll and pitch** rotations, more efficient



# Process Flowchart




*Files: Attitude_Simulator_v1.1.py, Propagator.py, Controller.py, MagneticField.py, Solver.py*

# Key Equations

- 2-Body problem $\boldsymbol{F} = m\boldsymbol{a} \Rightarrow \dfrac{d^2\boldsymbol{r}}{dt^2} = \dfrac{GM}{\boldsymbol{r}^2}$
- Earth's magnetic field (tilted dipole model) from MIT notes

$$\begin{bmatrix} B_{north} \\ B_{east} \\ B_{down} \end{bmatrix} = \left(\frac{6378}{r_{km}}\right)^3 \begin{bmatrix} -C_\varphi & S_\varphi C_\lambda & S_\varphi S_\lambda \\ 0 & S_\lambda & -C_\lambda \\ -2S_\varphi & -2C_\varphi C_\lambda & -2C_\varphi S_\lambda \end{bmatrix} \begin{bmatrix} -29900 \\ -1900 \\ 5530 \end{bmatrix}$$

Where: C=cos , S=sin, ϕ=latitude, λ=longitude
Units: nTesla


- Torque $\boldsymbol{T} = NI\boldsymbol{A} \times \boldsymbol{B}$ with N turns of coils and current I
- In principal axes (frame of satellite), Euler's equations of motion

$$\begin{aligned} M_x &= I_{xx}\dot{\omega}_x - (I_{yy} - I_{zz})\omega_y\omega_z \\ M_y &= I_{yy}\dot{\omega}_y - (I_{zz} - I_{xx})\omega_z\omega_x \\ M_z &= I_{zz}\dot{\omega}_z - (I_{xx} - I_{yy})\omega_x\omega_y \end{aligned}$$

*Files: Attitude_Simulator_v1.1.py, Propagator.py, Controller.py, MagneticField.py, Solver.py*

Attitude_Simulator_v1.1.py | Propagator.py | Controller.py | MagneticField.py | Solver.py

## *Main code*

```python
"""
CubeSat
Created on Fri Dec 27 18:07:13 2019
@author: Yu Jun
"""

import math
import numpy as np
import csv

import Propagator
import MagneticField
import Controller
import Solver

iniPos = np.array([2315480.356,6240325.846,1359050.958,\
                   -7259.977,2459.492,1284.609])
# based on ISS orbit
Pos = iniPos

Ix = 1    # moment of inertia along *principal* axes
Iy = 1
Iz = 1
turns = 10
area = 0.001
Kp = 0.2
dt = 0.001 # unified throughout
q = 1000   # data record rate (every 100th frame)
Time = 500 # seconds
totalSteps = int(Time/dt)

i0 = np.array([0,1,0])    # initial attitude of spacecraft, in inertial coord
j0 = np.array([-1,0,0])   # these '0' vectors must be orthogonal and unit mag.
k0 = np.cross(i0,j0)
omegaX = 4     # Starting test values
omegaY = 3
omegaZ = 1
TestData = np.array([omegaX,omegaY,omegaZ,Kp])

History = []   # initialize records
i0data = []
Time = 0       # seconds
```

Import functions

Approx. ISS orbit parameters; in **SI units**, refer to **Two Line Elements**

Tailor to our CubeSat inertia, coil geometry
**Time-step 1 ms, Proportional gain factor Kp**

Initial **disturbance** (goal is to damp this)

x,y,z for roll, pitch, yaw

## *Main code (contd)*

```python
# in the loop
for n in range(totalSteps):
    """I. 2 Body Forward Propagation"""
    newPos = Propagator.RK4(Pos[0],Pos[1],Pos[2],Pos[3],Pos[4],Pos[5],dt)
    # using classical 2-body only (good enough for now)
    x  = newPos[0]
    y  = newPos[1]
    z  = newPos[2]
    vx = newPos[3]
    vy = newPos[4]
    vz = newPos[5]
    Pos = newPos                          # close loop

    r_vectorMag = (x**2 + y**2 + z**2)**0.5 # magnitude of radius vect
    lat = np.arcsin(z/r_vectorMag)          # latitude, radian
    long = np.arctan2(y,x)                  # longitude; y=0 is Greenw

    """II. Magnetic Field Calculation"""
    Bfield = MagneticField.TiltedDipole(lat, long, r_vectorMag)

    """III. Magnetorquer Output"""
    # sub 0's: actual spacecraft orientation, unit vectors
    Signal = Controller.RateBasedControl(omegaX,omegaY,omegaZ,Kp)
    #Signal = Controller.RollPitchControl(i0,j0,k0,x,y,z,vx,vy,vz, Kp)
    RollTorque  = Signal[0]*turns*area # scalar
    PitchTorque = Signal[1]*turns*area # exerted torque mag, in principal axes

    """IV. Numerical Integration for omegas"""
    nextOmega = Solver.EulerEqnSolver(omegaX, omegaY, omegaZ, RollTorque, \
                                      PitchTorque,0,Ix,Iy,Iz,dt)
    # set yaw torque (Mz) = 0 in two axis control
    omegaX = nextOmega[0]
    omegaY = nextOmega[1]
    omegaZ = nextOmega[2]
```

Compute translational displacement in Geocentric Inertial Frame

Convert x,y,z to latitude/ longitude

Can calculate electrical power demanded, and scale accordingly

Solve Euler's equations for angular velocity

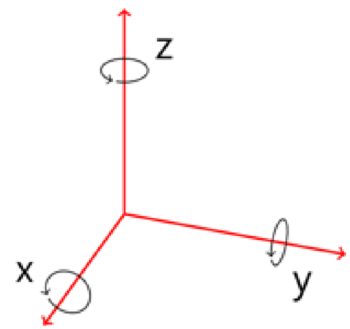
## *Main code (contd)*

```
    """V. Effecting omega"""  # ensure unit vectors remain unit magnitude, orthogonal
    # omegaX. rotate the principal axes accordingly. omegaX = Roll. x0 supposed to be forward facing
    d0X = omegaX*dt    # radian, small angle
    i0new = i0
    j0new = j0*math.cos(d0X) + k0*math.sin(d0X)
    k0new = k0*math.cos(d0X) - j0*math.sin(d0X)

    i0 = i0new
    j0 = j0new
    k0 = k0new   # put in the new values

    # omegaY = Pitch
    d0Y = omegaY*dt    # radian, small angle
    j0new = j0
    i0new = i0*math.cos(d0Y) - k0*math.sin(d0Y)
    k0new = k0*math.cos(d0Y) + i0*math.sin(d0Y)

    i0 = i0new
    j0 = j0new
    k0 = k0new   # put in the new values

    # omegaZ = Yaw
    d0Z = omegaZ*dt    # radian, small angle
    k0new = k0
    i0new = i0*math.cos(d0Z) + j0*math.sin(d0Z)
    j0new = j0*math.cos(d0Z) - i0*math.sin(d0Z)

    i0 = i0new
    j0 = j0new
    k0 = k0new   # put in the new values

    if (n//q)*q == n:
        Time = Time + q*dt
        Data = [Time,Pos[0],Pos[1],Pos[2],Pos[3],Pos[4],Pos[5],Bfield[0],\
            Bfield[1],Bfield[2],RollTorque,PitchTorque,omegaX, omegaY, omegaZ]
        History.append(Data)     # B vectors experienced

print(omegaX)
print("Simulation done. Timestep", dt, "sec. Data recorded every", q, "frames.")
print("Total time:", dt*totalSteps, "sec")
print("[omegaX, omegaY, omegaZ, Kp]:", TestData)

with open('SimulationData.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(["Time (s)","x (m)", "y (m)", "z (m)", "vx (m/s)" ,"vy (m/s)",\
                    "vz (m/s)","B_x (T)","B_y (T)","B_z (T)",\
                    "RollTorque (Nm)","PitchTorque (Nm)","OmegaX (rad/s)",\
                    "OmegaY (rad/s)", "OmegaZ (rad/s)",])
    for row in History:
        writer.writerow(row)
```

Updating the effect of omegas; watch out for the **rotating** axes!

Based on attitude knowledge, can work out solar flux

Save data at preset rate



# Function: B field

Attitude_Simulator_v1.1.py | Propagator.py | Controller.py | MagneticField.py | Solver.py

```python
@author: user
"""
import numpy as np

def TiltedDipole(lat,long, r_vectorMag):
    """Analytical Model of Earth's Magnetic Field as a Tilted Dipole.
    Input: latitude and longitude radians, radial distance from Earth core
    Output: Magnetic field in inertial Earth x, y, z component
    (pick z is geographic North pole, y=0 at Greenwich)"""

    B_row1 = np.array([-np.cos(lat),np.sin(lat)*np.cos(long),np.sin(lat)*np.sin(long)])
    B_row2 = np.array([0,np.sin(long),-np.cos(long)])
    B_row3 = np.array([-2*np.sin(lat),-2*np.cos(lat)*np.cos(long),-2*np.cos(lat)*np.sin(long)])
    B_column = np.array([-29900,-1900,5530])   # from physics

    Mat1 = np.multiply(B_row1, B_column)
    B_north = Mat1[0] + Mat1[1] + Mat1[2]   # matrix multiplication for B_north
    Mat2 = np.multiply(B_row2, B_column)
    B_east = Mat2[0] + Mat2[1] + Mat2[2]    # B_east
    Mat3 = np.multiply(B_row3, B_column)
    B_down = Mat3[0] + Mat3[1] + Mat3[2]    # B_down
    BfieldRot = 10**(-9)*np.array([B_north,B_east,B_down])*(6378000/r_vectorMag)**3
    # this outputs North, East, Down vectors. in Teslas

    # Now, in inertial frame
    Bfieldx = BfieldRot[0]*(-np.sin(lat)*np.cos(long)) + \
              BfieldRot[1]*np.cos(long) + BfieldRot[2]*(-np.cos(lat)*np.cos(long))
    Bfieldy = BfieldRot[0]*(-np.sin(lat)*np.sin(long)) + BfieldRot[1]*np.sin(long) +\
              BfieldRot[2]*(-np.cos(lat)*np.sin(long))
    Bfieldz = BfieldRot[0]*np.cos(lat) - BfieldRot[2]*np.sin(lat)

    return [Bfieldx,Bfieldy,Bfieldz]
```

Implement matrix

# Function: Propagator

```python
GM = 3.986*(10**14)    # SI units
rE = 6378137           # meters
u = array([2315480.356,6240325.846,1359050.958,-7259.977,2459.492,1284.609],float)
#    test case^
J2 = 0.0010826266835531513
J3 = -0.0000025
J4 = -0.0000016
J5 = -0.00000015
J6 = 0.00000057
totalTime=860
dt=1.

def grad(p0,p1,p2,p3,p4,p5):
    # faster version
    r = sqrt(p0**2+p1**2+p2**2)
    thetaP = 0.00007292115
    v = sqrt((p3+thetaP*p1)**2+(p4-thetaP*p0)**2+p5**2)
    return [p3,p4,p5,-GM*(p0)/r**3,-GM*(p1)/r**3,-GM*(p2)/r**3]

def RK4(u0,u1,u2,u3,u4,u5,dt):   # standard RK4 implementation
    k1 = grad(u0,u1,u2,u3,u4,u5)
    k2 = grad(u0+k1[0]*dt/2, u1+k1[1]*dt/2, u2+k1[2]*dt/2, \
              u3+k1[3]*dt/2, u4+k1[4]*dt/2, u5+k1[5]*dt/2)
    k3 = grad(u0+k2[0]*dt/2, u1+k2[1]*dt/2, u2+k2[2]*dt/2,\
              u3+k2[3]*dt/2, u4+k2[4]*dt/2, u5+k2[5]*dt/2)
    k4 = grad(u0+k3[0]*dt, u1+k3[1]*dt, u2+k3[2]*dt, \
              u3+k3[3]*dt, u4+k3[4]*dt,u5+k3[5]*dt)
    res = [u0 + dt/6*(k1[0]+2*k2[0]+2*k3[0]+k4[0]), \
           u1 + dt/6*(k1[1]+2*k2[1]+2*k3[1]+k4[1]), \
           u2 + dt/6*(k1[2]+2*k2[2]+2*k3[2]+k4[2]), \
           u3 + dt/6*(k1[3]+2*k2[3]+2*k3[3]+k4[3]), \
           u4 + dt/6*(k1[4]+2*k2[4]+2*k3[4]+k4[4]), \
           u5 + dt/6*(k1[5]+2*k2[5]+2*k3[5]+k4[5])]
    return res
```

Gradient function for Runge Kutta method

# Function: Controller, Solver

Attitude_Simulator_v1.1.py | Propagator.py | Controller.py | MagneticField.py | Solver.py

```python
"""
Controller
Calculates deviation from target attitude and outputs control signal
Proposed Two Axis (pitch and roll) stabilization code assuming circular orbit
@author: Yu Jun
"""
import numpy as np
import math

def RateBasedControl(omegaX,omegaY,omegaZ, Kp):
    """Simple Control Method for Two Axis Stablisation (Roll and Pitch)
    Inputs: Spacecraft Attitude and State Vector, Kp Gain
    Outputs: Roll and Pitch Control Currents

    Assume current omegas known, either from B-dot or accelerometer IRL"""
    Roll = - omegaX*Kp       # proportional gain. counter error
    Pitch = - omegaY*Kp
    return Roll,Pitch
```

Only proportional angular velocity based control

```python
def EulerEqnSolver(omegaX,omegaY,omegaZ, MomentX, MomentY, MomentZ, Ix, Iy, Iz, dt):
    """Numerically solves Euler equations of motion
    Inputs: current ang. velocity, torques, & moments of inertia in Principal Axes
    Outputs: next timestep's angular velocities in Principal Axes frame
    Direct forward march numerical integration
    Linearize omega dot across one timestep; expect some error over time"""

    omegaX0 = omegaX    # old variable, nth step
    omegaY0 = omegaY
    omegaZ0 = omegaZ

    omegaX = omegaX0 + (MomentX/Ix + (Iy-Iz)/Ix*omegaY0*omegaZ0)*dt #(n+1)th step
    omegaY = omegaY0 + (MomentY/Iy + (Iz-Ix)/Iy*omegaZ0*omegaX0)*dt
    omegaZ = omegaZ0 + (MomentZ/Iz + (Ix-Iy)/Iz*omegaX0*omegaY0)*dt
    return omegaX,omegaY,omegaZ
```

Forward time-step; modular code can be updated

# Thoughts on Function Files

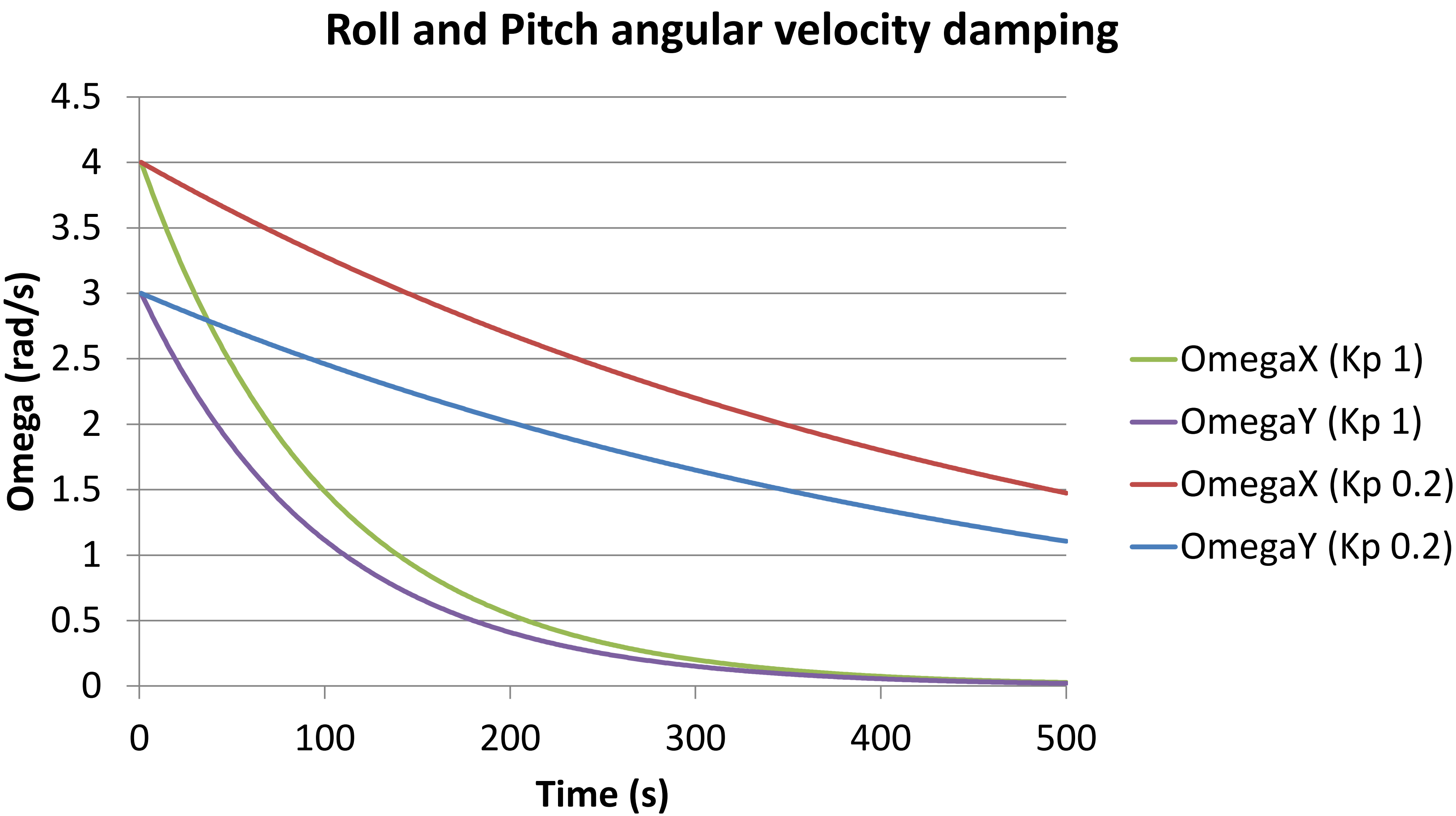
- Propagator.py: solves 2-body problem using Runge-Kutta 4 numerical method
  - We *could* consider orbit perturbations like Earth J2-6 harmonics, but I think **unnecessary** at this stage (errors from elsewhere + **different time-scale** of orbit vs tumbling)

- Controller.py: uses linear gain factor (P part of PID; this coefficient must be tested!)

?

- MagneticField.py: implements analytical tilted dipole model (see MIT AeroAstro notes)
  - Probably good enough for now; check SI **units**!

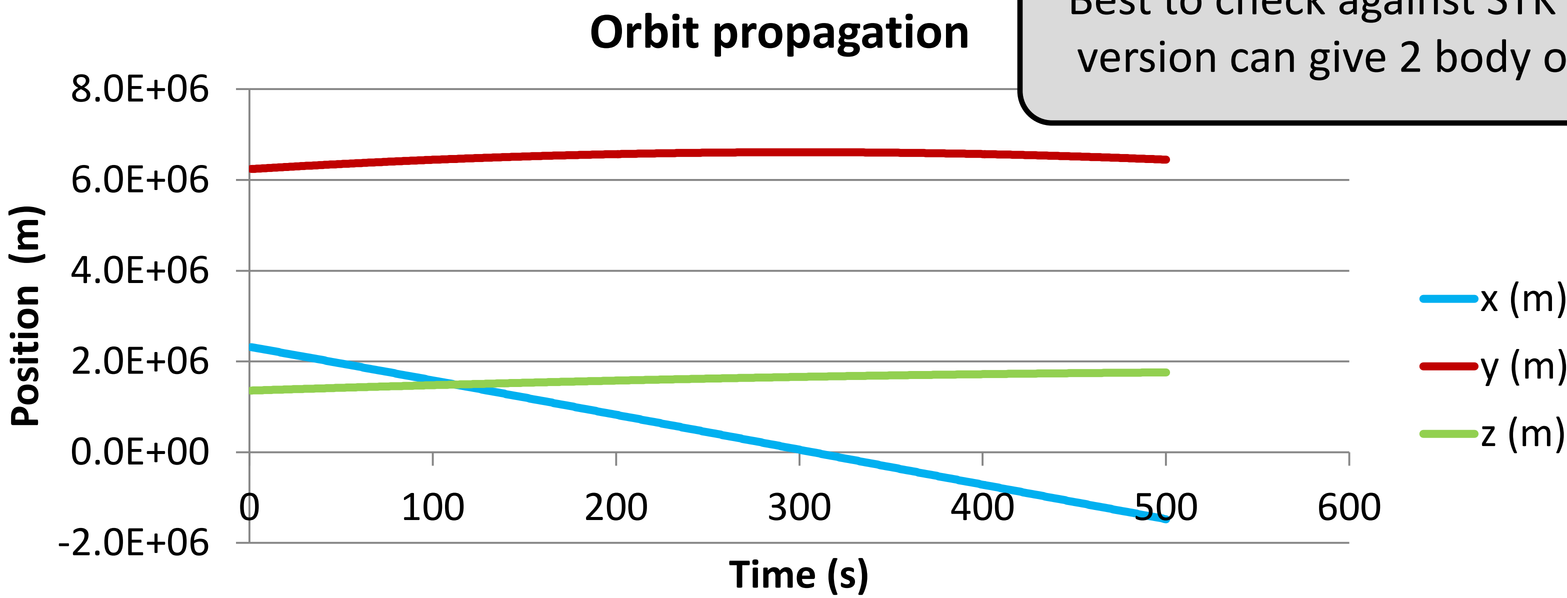
- Solver.py: uses **forward time march**
  - Suggest we start here to improve accuracy

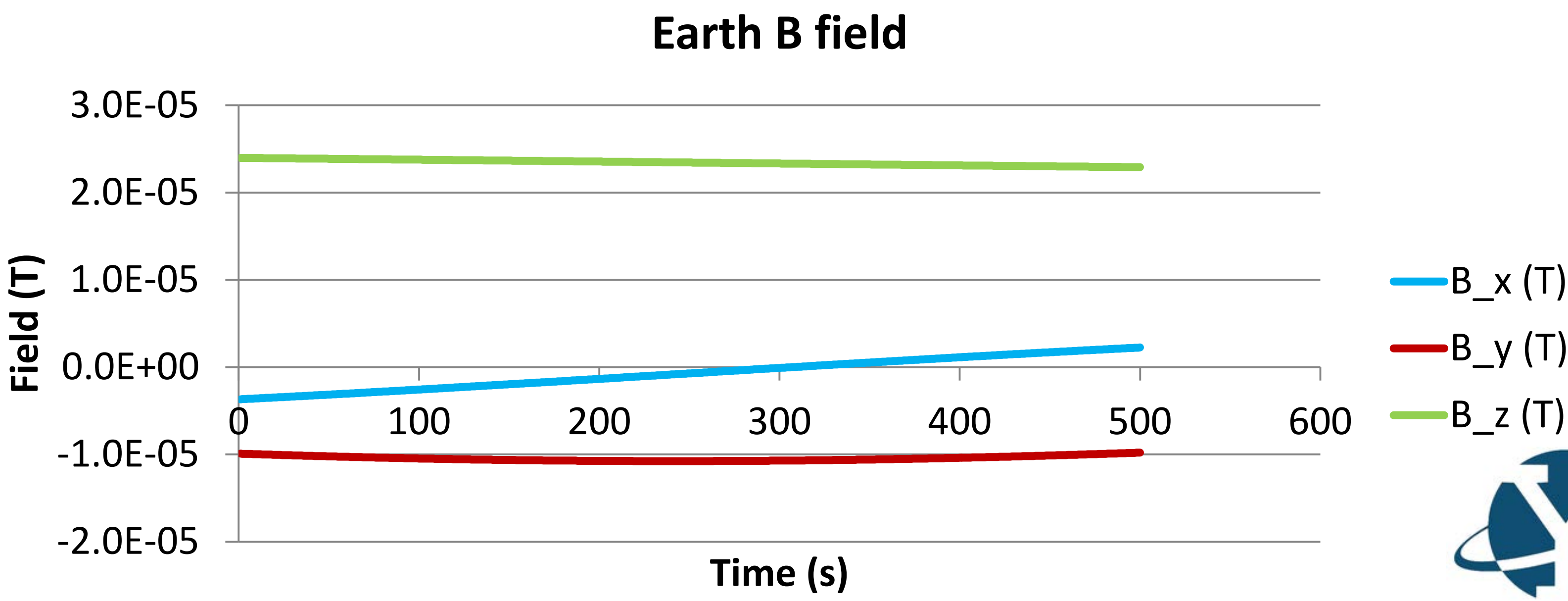



Exponential decay from **proportional** control $\frac{d\omega}{dt} = k\omega$

Greater Kp -> faster decay -> but power okay?

Best to check against STK (free version can give 2 body orbit)



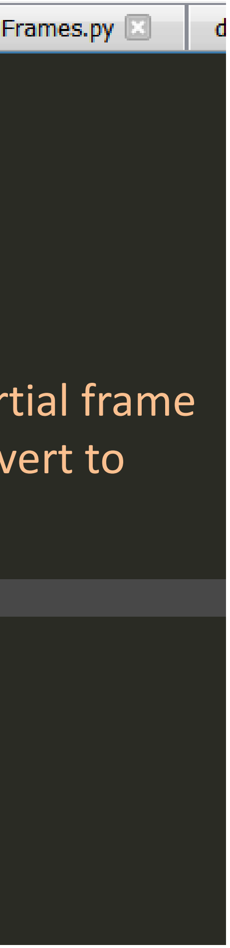



# Suggested next steps

1. Check math and accuracy of results
2. Substitute our CubeSat values
3. Nicer way to get omega from Euler’s equations?
4. If ^ are OK, determine solar flux available
5. Compare active control vs passive (gravity boom) control time needed

# References


- MIT AeroAastro notes
  - https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-851-satellite-engineering-fall-2003/lecture-notes/ (esp Lecture 9 on ADCS)
- Numerical solutions
  - https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
  - https://en.wikipedia.org/wiki/Euler%27s_equations_(rigid_body_dynamics)




From Lecture 3, we have that the transformation of a vector from a coordinate system
coordinate system $x_1', x_2' x_3'$ is given by

$$\begin{pmatrix} H_1' \\ H_2' \\ H_3' \end{pmatrix} = \begin{pmatrix} \boldsymbol{i}_1' \cdot \boldsymbol{i}_1 & \boldsymbol{i}_1' \cdot \boldsymbol{i}_2 & \boldsymbol{i}_1' \cdot \boldsymbol{i}_3 \\ \boldsymbol{i}_2' \cdot \boldsymbol{i}_1 & \boldsymbol{i}_2' \cdot \boldsymbol{i}_2 & \boldsymbol{i}_2' \cdot \boldsymbol{i}_3 \\ \boldsymbol{i}_3' \cdot \boldsymbol{i}_1 & \boldsymbol{i}_3' \cdot \boldsymbol{i}_2 & \boldsymbol{i}_3' \cdot \boldsymbol{i}_3 \end{pmatrix} \begin{pmatrix} H_1 \\ H_2 \\ H_3 \end{pmatrix} = [T] \begin{pmatrix} H_1 \\ H_2 \\ H_3 \end{pmatrix} .$$

where we have introduced the symbol [T] for the transformation matrix.