# CubeSat ADCS Simulation v3

1. Introduction
2. Code and Section Overview
3. Results
4. Next steps

**Yu Jun Shen**
YUAA (Nov 2020)

# Version history

- v1 in Fall 2020, based on propagator
- v2.2 with Sun, edited main file and Power function
- v3 checked Nov 2020, functions cleaned up

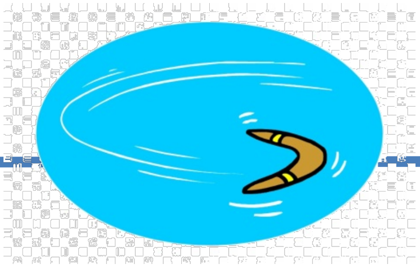
# Objective

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

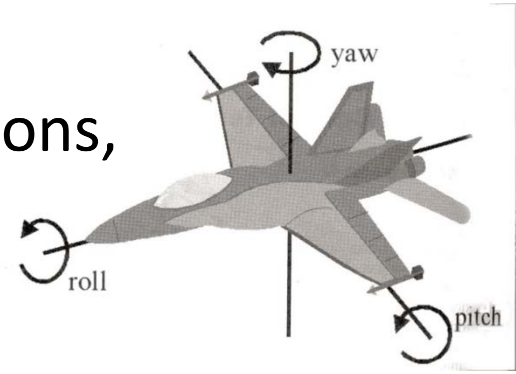


We need to consider both

- **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

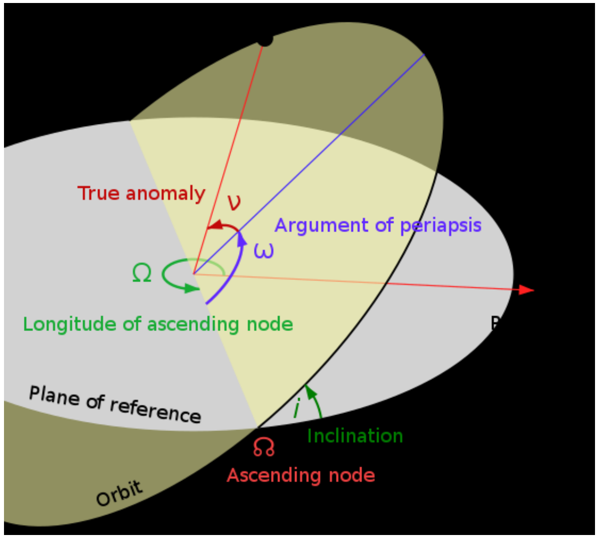


# Outline of simulation

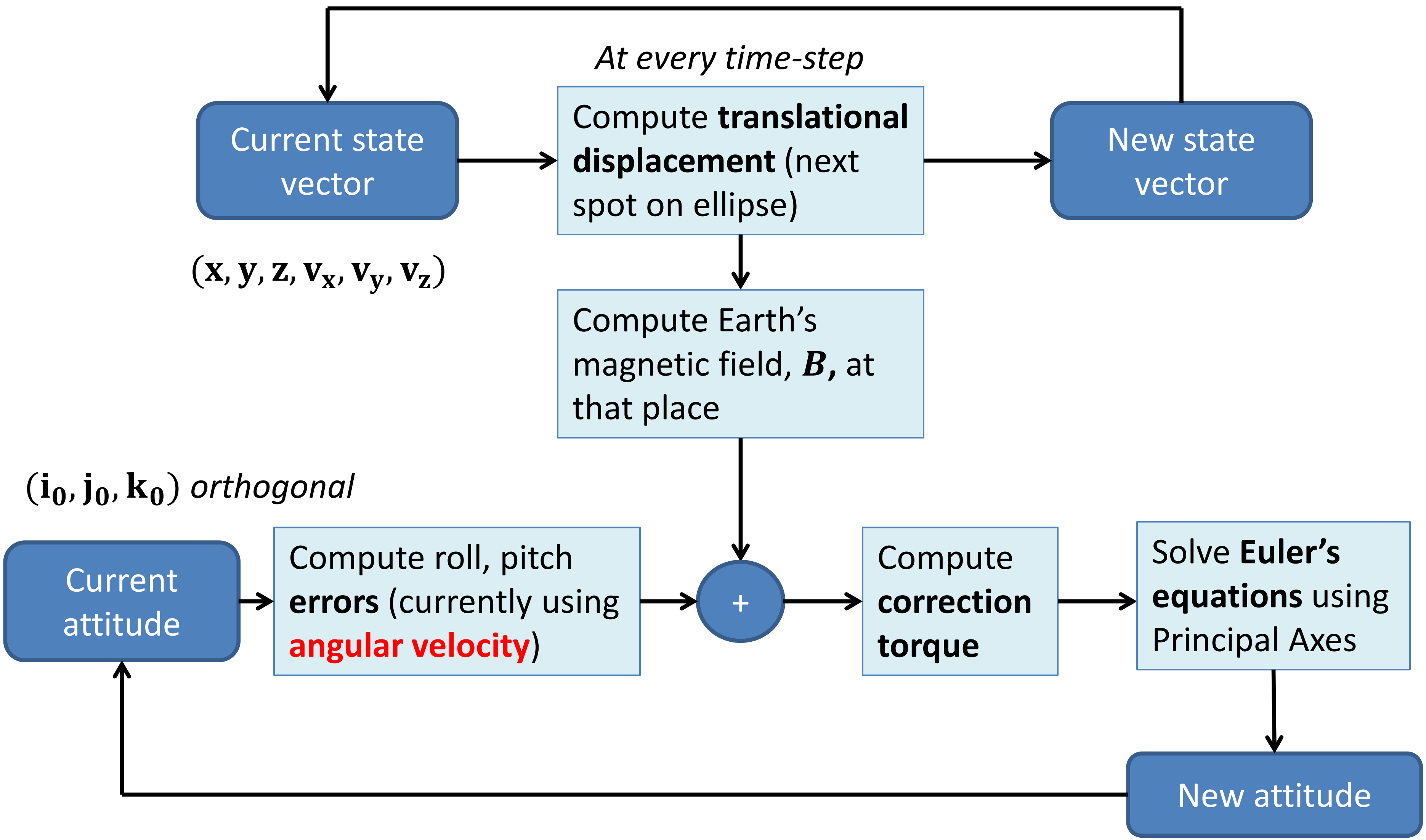
- Work out orbit position (propagation step) and velocity
- Find magnetic field at that point, inertial then transform into satellite frame *(a tricky matrix multiplication)*
- Determine what torque to apply (bang bang refers to On, Off only → no proportionality yet)
- Apply $\tau = I\alpha$ in **inertial frame** to find next step omegas *(don't want fictitious forces! More matrices)*
- Based on Sun's angle and satellite longitude angle, work out solar flux received for power.
- Back to propagation…

# Propagation

- Orbit propagation is the basic step because we need to know where the satellite is and, in this model, what magnetic field is available for control
- Great to compare against STK here
- **Six** degrees of freedom so at least six inputs



# Process Flowchart




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

# Key Equations

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

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

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


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

Attitude_Simulator_v3.py | Controller.py | MagneticField.py | Power.py | Propagator.py | Solver.py

***Main code***

```python
"""
CubeSat Attitude Determination and Control System Simulation
Version 3: Bang Bang Control with Sun added (24 Nov 2020)

Orbit, omega, power outputs look reasonable
@author: Yu Jun
"""
import math
import numpy as np
from numpy import linalg as LA
import csv
import Propagator
import MagneticField
import Controller
import Solver
import Power

'''=========Test conditions (change this part only=========='''

iniPos = np.array([-6719.400, 385.319, 2.669, -0.272368, -4.77507, 6.03443])
#ISS [x,y,z,vx,vy,vz] from STK, distances in km and velocity in km/s

dt = 1 # unified throughout
q = 10   # data record rate (every q frames)
Duration = 400*60 # seconds
i0 = np.array([1,0,0])    # initial attitude of spacecraft, in inertial coord
j0 = np.array([0,1,0])   # these '0' vectors must be orthogonal and unit mag.
k0 = np.cross(i0,j0)
omegaX = 0.2    # Starting test values, in spacecraft frame. rad/s
omegaY = 0.1    # so correspond to roll/pitch/yaw
omegaZ = 0.3
'''=========================================================='''
```

Import functions

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

Tailor to our CubeSat inertia, coil geometry. **Time-step 1 s**

x,y,z for roll, pitch, yaw

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

Attitude_Simulator_v3.py Controller.py MagneticField.py Power.py Propagator.py Solver.py

***Main code***

```
'''Satellite constants (input once the design is finalised)'''
Ix = 1   # moment of inertia along *principal* axes
Iy = 1
Iz = 1
turns = 10
area = 0.001
Kp = 0.02

'''======Computation constants (don't need to change)======'''
totalSteps = int(Duration/dt)
Pos = iniPos                 # initialize state vector
i = np.array([1,0,0])        # unit vectors in Earth non-rotating inertial frame
j = np.array([0,1,0])
k = np.array([0,0,1])
TestData = np.array([omegaX,omegaY,omegaZ,Kp])   # save initial test data
Jx = 0 # initialise current in x torque coil
Jy = 0
Jz = 0
M_old = [0.0, 0.0, 0.0] # initialize torque history
sunAngle = 0   # assume in ecliptic
History = []   # initialize records
i0data = []
Time = 0       # seconds

orbitDebug = []  # initialize empty list for testing
```

Data structures

## *Main code*

```python
'''*****======Start loop======*****'''
print("Starting simulation... iniPos:", Pos)
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)
    x  = newPos[0]
    y  = newPos[1]
    z  = newPos[2]
    vx = newPos[3]
    vy = newPos[4]
    vz = newPos[5]
    Pos = newPos
    #print(newPos)   # debug

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

    """II. Magnetic Field Calculation"""
    BfieldGCI = MagneticField.TiltedDipoleXYZ(lat, long, r_vectorMag)
    # in Earth non-rotating frame, nanoTesla, r_vectorMag in km

    BfieldBFPA = MagneticField.GCItoBFPAtransform(i,j,k,i0,j0,k0,BfieldGCI[0],\
                                                  BfieldGCI[1],BfieldGCI[2])

    BfieldNED = MagneticField.TiltedDipoleNED(lat, long, r_vectorMag)
    # transforms magnetic field to satellite principal axes frame
    # sub 0's: actual spacecraft orientation, unit vectors

```

Compute translational displacement in Geocentric Inertial Frame

Convert x,y,z to latitude/ longitude

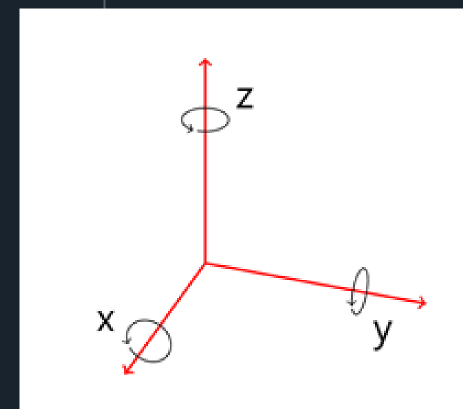
```python
    """III. Magnetorquer Output"""
    '''The B-field information is used to calculate torque by working out,
    under nominal Bang Bang control. Then convert to BFPA frame. '''

    NetTorque = Controller.nominalTorqueBFPA(omegaX,omegaY,omegaZ,BfieldBFPA)
    #print(NetTorque)

    """IV. Numerical Integration for omegas"""
    nextOmega = Solver.EulerEqnSolver(omegaX,omegaY,omegaZ, NetTorque[0],
                                      NetTorque[1], NetTorque[0],Ix,Iy,Iz,dt)

    omegaX = nextOmega[0]
    omegaY = nextOmega[1]
    omegaZ = nextOmega[2]
```

Solve Euler's equations for angular velocity

*Main code*

```python
    """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 valu

    # 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
```

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




Save data at preset rate

**Main code**

```python
    """ VI. Power calculation with dark side"""
    sunAngle = sunAngle + 2*np.pi/(24*60*60)*dt  # sun moves
    if sunAngle >= 2*np.pi:
        sunAngle = sunAngle - 2*np.pi             # keep to within 0, 2pi range
    # because at ISS inclinations the sun's 23 deg tilt won't affect coverage
    power = Power.flux(long, sunAngle, i0, j0, k0) # compute power

    if (n//q)*q == n:                             # recording the data
        Time = Time + q*dt
        Data = [Time,Pos[0]/1000,Pos[1]/1000,Pos[2]/1000,Pos[3]/1000,Pos[4]/1000,Pos[5]/1000,BfieldNE
            BfieldNED[1],BfieldNED[2],NetTorque[0],NetTorque[1],NetTorque[2],\
            omegaX, omegaY, omegaZ, power]
        History.append(Data)     # B vectors experienced

    orbitDebug.append([x,y,z,vx,vy,vz])

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

# save data
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_North (nT)","B_East (nT)",\
                     "B_Down (nT)","RollTorque (Nm)","PitchTorque (Nm)", \
                     "YawTorque (Nm)","OmegaX (rad/s)",\
                     "OmegaY (rad/s)", "OmegaZ (rad/s)","Power (Watt)"])
    for row in History:
        writer.writerow(row)
f.close()
```

# Earth magnetic field

- Analytical tilted dipole model
  - Refer to notes from MIT 16.684 Space Systems Product Development
  - Need to convert to inertial frame

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

Where: C=cos , S=sin, $\phi$=latitude, $\lambda$=longitude

Units: nTesla

# Function: B field

Implement matrix

```python
# -*- coding: utf-8 -*-
"""
Magnetic Field File (24 Nov 2020)
Contains two B field calculations: one in rotating Earth frame (North, East, Down)
and another in the x,y,z frame (with a vector transformation)

See MIT notes for tilted dipole mode (Slide 34):
https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-851-satellite-engineering-fall-2003/lectur
"""
import numpy as np

def TiltedDipoleXYZ(lat,long, r_vectorMag):
    """outputs magnetic field in ECI -> ""x,y,z" frame"""
    matrix = np.array([[-np.cos(lat), np.sin(lat)*np.cos(long), np.sin(lat)*np.sin(long)],\
                       [0,np.sin(long), -np.cos(long)],\
                       [-2*np.sin(lat), -2*np.cos(lat)*np.cos(long), -2*np.cos(lat)*np.sin(long)]])
    vector = np.array([-29900, -1900, 5530])

    Bfield = (6378/r_vectorMag)**3*matrix.dot(vector)
    #In north, east and down currently. Use 6378 which is Earth radius in km.

    Bfieldx = Bfield[0]*(-np.sin(lat)*np.cos(long)) + \
              Bfield[1]*np.sin(long) + Bfield[2]*(-np.cos(lat)*np.cos(long))
    Bfieldy = Bfield[0]*(-np.sin(lat)*np.sin(long)) + Bfield[1]*np.cos(long) +\
              Bfield[2]*(-np.cos(lat)*np.sin(long))
    Bfieldz = Bfield[0]*np.cos(lat) - Bfield[2]*np.sin(lat)
    # nanoTesla
    return np.array([Bfieldx*10**(-9) , Bfieldy*10**(-9) , Bfieldz*10**(-9) ])
```

# Function: B field

```python
def GCItoBFPAtransform(i,j,k,i0,j0,k0,x,y,z):
    """Transforms vector x,y,z in coordinate frame with unit vectors i,j,k
    into vector x0,y0,z0 in coordinate frame with unit vectors i0,j0,k0.
    Use: transform B field from Geocentric Inertial Frame to spacecraft
    Body-Fixed Principal axes frame. See MIT Dynamics lecture for math.
    Outputs: new vector x0,y0,z0."""

    x0 = np.dot(i0,i)*x + np.dot(i0,j)*y + np.dot(i0,k)*z
    y0 = np.dot(j0,i)*x + np.dot(j0,j)*y + np.dot(j0,k)*z
    z0 = np.dot(k0,i)*x + np.dot(k0,j)*y + np.dot(k0,k)*z

    return np.array([x0,y0,z0])

def TiltedDipoleNED(lat,long, r_vectorMag):
    """Debug function to compare against STK.
    Returns array of B field, in North, East, Down (NED) components
    in nanoTesla"""

    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 = np.array([B_north,B_east,B_down])*(6378/r_vectorMag)**3*10**(-9)    # in nanoTesla

    return BfieldRot
```

# About propagators…

- There are different levels of accuracy for this
  - 2 Body problem: textbook, six classical elements/ [x,y,z,vx,vy,vz] state vector fully sufficient to describe orbit
  - J2, J4… : considers Earth “fatness” at the equatorial mass bulge. I am using J4 at the moment, relatively simple to implement under Runge Kutta 4 integration (no atmospheric drag)
  - SGP4 and above: much more advanced, considers other Earth mass distribution and other celestial bodies

In general for short durations (~days of orbit) no big deviation is expected; but for **long term** mission planning higher fidelity models are necessary.

# Function: Propagator

```python
def grad(p0,p1,p2,p3,p4,p5):     # RK4 gradient function
    r = sqrt(p0**2 + p1**2 + p2**2)   # Earth radius for J term calculations

    Jx = 1 - J2*(3./2.)*(rE/r)**2*(5*p2**2/r**2-1) + \
        J3*(5./2.)*(rE/r)**3*(3*p2/r-7*p2**3/r**3) - \
        J4*(5./8.)*(rE/r)**4*(3-42*p2**2/r**2+63*p2**4/r**4) - \
        J5*(3./8.)*(rE/r)**5*(35*p2/r-210*p2**3/r**3+231*p2**5/r**5) + \
        J6*(1./16.)*(rE/r)**6*(35-945*p2**2/r**2+3465*p2**4/r**4-3003*p2**6/r**6)
    Jz = 1 + J2*(3./2.)*(rE/r)**2*(3-5*p2**2/r**2) + \
        J3*(3./2.)*(rE/r)**3*(10*p2/r-(35./3.)*p2**3/r**3-r/p2) - \
        J4*(5./8.)*(rE/r)**4*(15-70*p2**2/r**2+63*p2**4/r**4) - \
        J5*(1./8.)*(rE/r)**5*(315*p2/r-945*p2**3/r**3+693*p2**5/r**5-15*p2/r) + \
        J6*(1./16.)*(rE/r)**6*(315-2205*p2**2/r**2+4851*p2**4/r**4-3003*p2**6/r**6)
    thetaP = 0.00007292115
    v = sqrt((p3+thetaP*p1)**2+(p4-thetaP*p0)**2+p5**2)
    return [p3,p4,p5,-GM*(p0)/r**3*Jx,-GM*(p1)/r**3*Jx,-GM*(p2)/r**3*Jz]

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

# Controller

- Linearized calculation used in python
- Literally $\omega_{n+1} = \omega_n + \alpha \cdot \Delta t$
- Where $\alpha$ is the acceleration from the magnetorquer (no other torques considered **yet***, and bang bang control for now)
- Gravity boom can also be modelled for $\alpha$

*eventually atmospheric drag, solar pressure will conspire to deviate the spacecraft :0

# Controller

```python
"""
Controller File (24 Nov 2020)
Calculates output torque based on input error
Proposed Two Axis (pitch and roll) stabilization code assuming circular orbit
"""
import numpy as np
import math

def nominalTorqueBFPA(OmegaX, OmegaY, OmegaZ, BfieldBFPA):
    """
    Input: current errors and Bfield to find corrective
    torques (all in BPFA). Omegas are floats, BfieldBFPA is array
    Output: Torque array in BFPA, to use directly in Solver module.

    0.25 Amp*m2 is the nominal working dipole strength. This method is
    BANG-BANG CONTROL : magnetorquer switches on or off only, at 0.25 Ampm^2
    dipole strength. This value from Nanoavionics spec sheet, projeted to
    consume 140 mW nominally.
    """

    TorqueX = np.array([0,0,0])   # initialize local variables
    TorqueY = np.array([0,0,0])   # default bang bang buffer zone
    TorqueZ = np.array([0,0,0])
    dipoleX = np.array([0,0,0])   # vectors
    dipoleY = np.array([0,0,0])
    dipoleZ = np.array([0,0,0])
    omega = np.array([OmegaX, OmegaY, OmegaZ])
```

# Controller

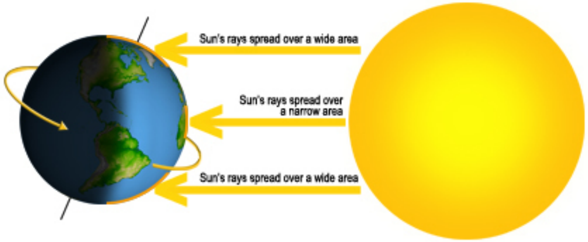
```python

    dipoleX = np.cross(np.array([1,0,0]),BfieldBFPA)
    if dipoleX.dot(omega) < -10E-7:
        TorqueX = 0.25*dipoleX        # run in +ive i0 direction
#        print("dipoleX dot:", dipoleX.dot(omega), "+i0")
    elif dipoleX.dot(omega) > 10E-7:
        TorqueX = -0.25*dipoleX       # run in -ve i0 direction
#        print("dipoleX dot:", dipoleX.dot(omega), "-i0")

    dipoleY = np.cross(np.array([0,1,0]),BfieldBFPA)
    if dipoleY.dot(omega) < -10E-7:
        TorqueY = 0.25*dipoleY        # run in +ive i0 direction
#        print("dipoleY dot:", dipoleY.dot(omega), "+j0")
    elif dipoleY.dot(omega) > 10E-7:
        TorqueY = -0.25*dipoleY       # run in -ve i0 direction
#        print("dipoleY dot:", dipoleY.dot(omega), "-j0")

    dipoleZ = np.cross(np.array([0,0,1]),BfieldBFPA)
    if dipoleZ.dot(omega) < -10E-7:
        TorqueZ = 0.25*dipoleZ        # run in +ive i0 direction
#        print("dipoleZ dot:", dipoleZ.dot(omega), "+k0")
    elif dipoleZ.dot(omega) > 10E-7:
        TorqueZ = -0.25*dipoleZ       # run in -ve i0 direction
#        print("dipoleZ dot:", dipoleZ.dot(omega), "-k0")

    return TorqueX + TorqueY + TorqueZ  # do all at once
```

# Solver

```python
# -*- coding: utf-8 -*-
"""
v3 24 Nov 2020
Solver module
Goal: output accurate, fast omegax/y/z values after one timestep
input: this step's omega, dt
Design notes: At first we used single timestep forward march, now upgraded to
Runge Kutta 4 -> much faster when timestep is 1 sec instead of 1 ms before
IN SATELLITE BFPA
@author: user
"""

Ixx = 1   # moment of inertia along *principal* axes
Iyy = 1
Izz = 1

def EulerEqnSolver(omegaX,omegaY,omegaZ, MomentX, MomentY, MomentZ, Ixx, Iyy, Izz, 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/Ixx + (Iyy-Izz)/Ixx*omegaY0*omegaZ0)*dt #(n+1)th step
    omegaY = omegaY0 + (MomentY/Iyy + (Izz-Ixx)/Iyy*omegaZ0*omegaX0)*dt
    omegaZ = omegaZ0 + (MomentZ/Izz + (Ixx-Iyy)/Izz*omegaX0*omegaY0)*dt
    return [omegaX,omegaY,omegaZ]
```

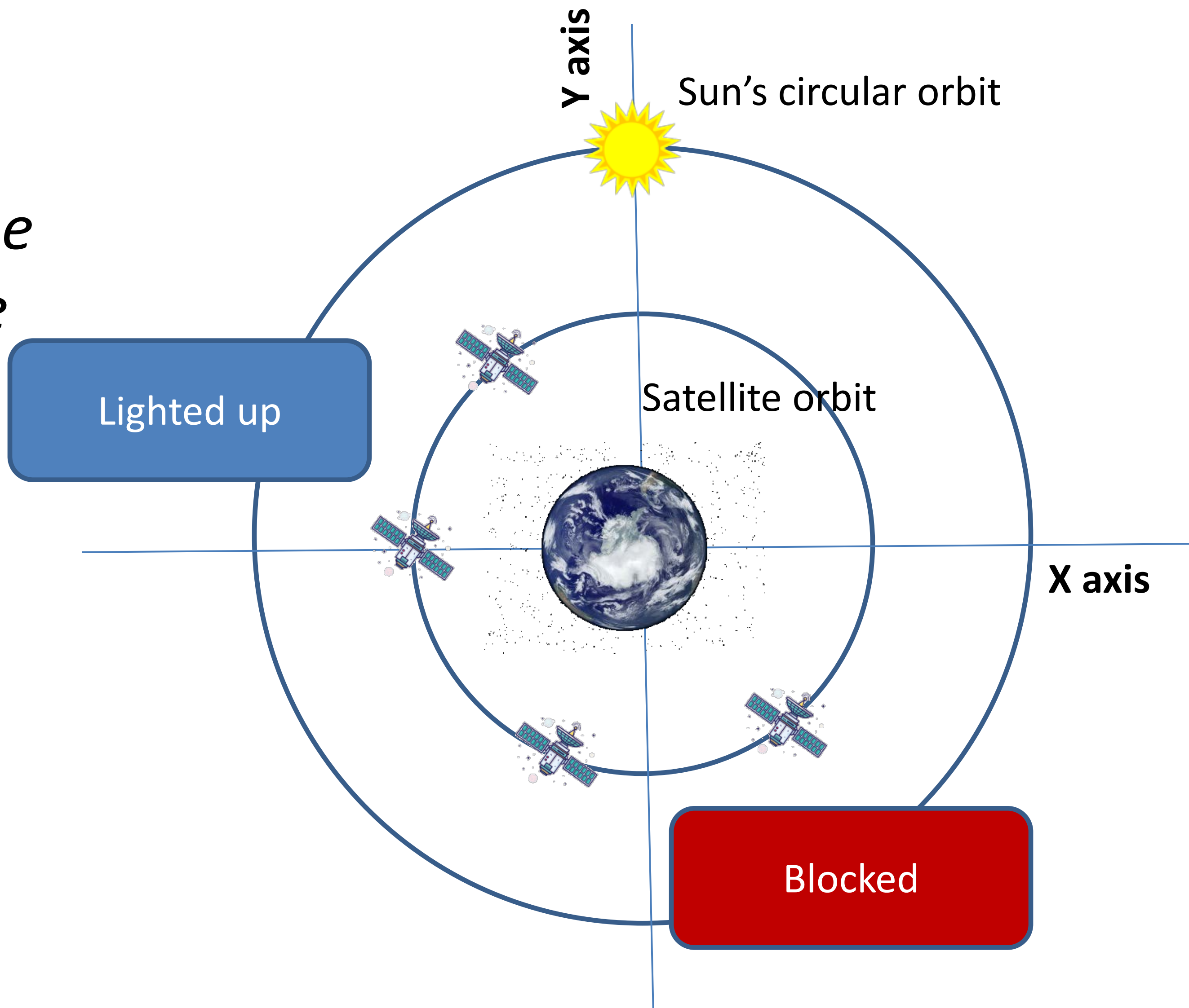
# Solar power methodology

- Added a rotating sun
- Sunlight incident if and only if $|\theta_{Sun} - \theta_{Sat}| \leq \frac{\pi}{2}$
- Furthermore, $\theta_{Sun}, \theta_{Sat} \in [0,2\pi]$ and considering the Equator/Ecliptic, $\theta_{Sat} = longitude = \tan^{-1}\left(\frac{y}{x}\right)$
- Can safely ignore 23 degrees tilt because our inclination (@ ~ISS) is less than 90 – 23 = 67 at which this would be significant



# Solar power illustration

*We are looking down from the "North Pole"; the circles are in the plane of the equator*



# Power

```python
def flux(long, sunAngle, i0, j0, k0):
    """Calculates solar power received by satellite
    Input: satellite longitude, sun angle and current satellite orientations in i0, j0, k0 vectors
    Output: power (Watt)
    k0 points upwards from top of satellite (solar panel exists there)"""

    efficiency = 0.307 * 0.88 * (1 - (75-28) * 0.0022) #efficiency of solar panel
    Area2U = 0.01076664    # Area of one 2U panel in m^2
    # using arrays for inertial i and k unit vectors

    phi = 1373*(math.cos(23)*np.array([1,0,0]) - math.sin(23)*np.array([0,0,1]))
    # solar flux vector

    powerTop = phi.dot(k0)*Area2U/2
    if powerTop < 0:   # top is sunlit
        powerTop = efficiency*abs(powerTop)
    else:
        powerTop = 0

    powerSidei0 = efficiency*abs(phi.dot(i0)*Area2U)  # don't double count
    powerSidej0 = efficiency*abs(phi.dot(j0)*Area2U)*0.75    #reduced to 1U on a side

    powerAll = powerTop + powerSidei0 + powerSidej0    # before considering dark side

    if abs(sunAngle - long) < np.pi/2:
        powerAll = powerAll              # lighted up
    else:
        powerAll = 0                     # in shadow of Earth

    return powerAll
```

# 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

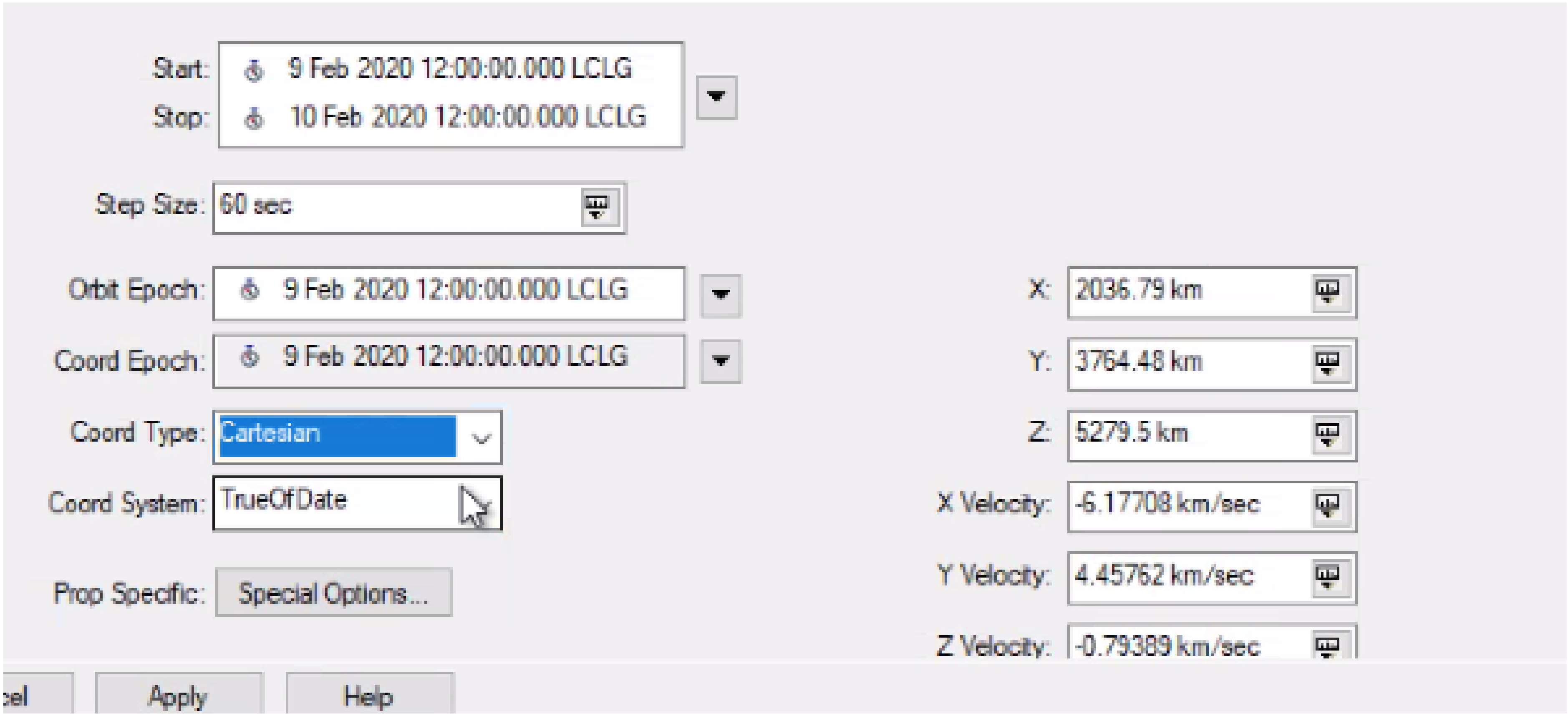
# Notes and some issues

1) ~~Orbit duration is a little shorter than STK~~
2) The other set of values Jess and Michael plotted in STK on 10/11/20 surprisingly broke the same propagator. I have no idea why ☹ so it's back to the ISS orbit here
3) Solar power considers Earth blockage (yay!)
4) Magnetic field looks sensible?
5) Bang Bang control is very crude and might give instabilities, though it works as a first approximation (omegas decrease as desired)

# Simulation parameters

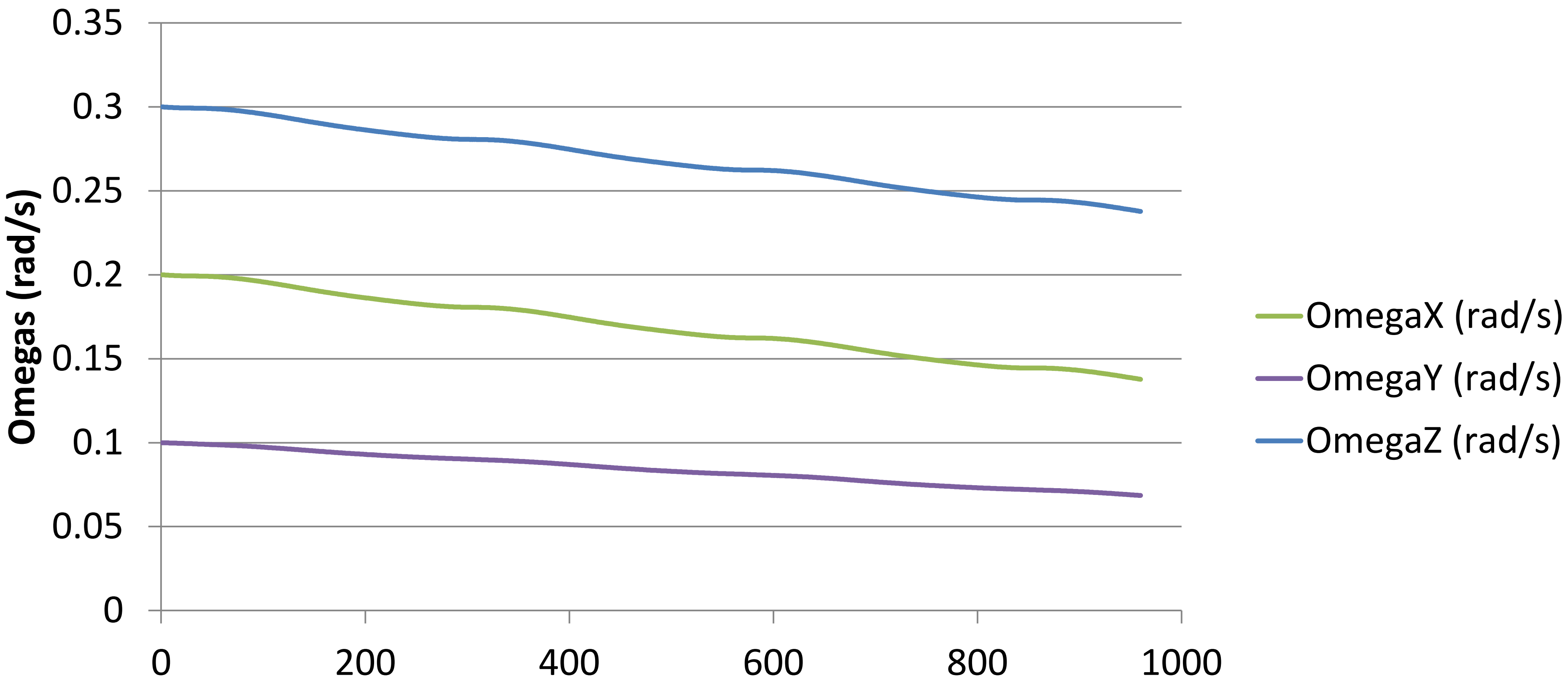
- Started off with (ISS?) orbit: (-5801232.89, 3520987.41, 6.751 [km], -2136.353, -3519.9, 6120.767 [km/s]) state vector (x,y,z,vx,vy,vz)
- Roll, Pitch, Yaw at 0.2, 0.1, 0.3 rad/s at first (omegaX, omegaY, omegaZ)
- 1 second time step, data saved every 10 second (so as to see power fluctuation in rotating satellite)
- Bang Bang maximum-effort control

# From STK



Start:
9 Feb 2020 12:00:00.000 LCLG
Stop:
10 Feb 2020 12:00:00.000 LCLG
Step Size:
60 sec
Orbit Epoch:
9 Feb 2020 12:00:00.000 LCLG
Coord Epoch:
9 Feb 2020 12:00:00.000 LCLG
Coord Type:
Cartesian
Coord System:
TrueOfDate
Prop Specific:
Special Options...
X:
2036.79 km
Y:
3764.48 km
Z:
5279.5 km
X Velocity:
-6.17708 km/sec
Y Velocity:
4.45762 km/sec
Z Velocity:
-0.79389 km/sec
Apply
Help

# Omega under control

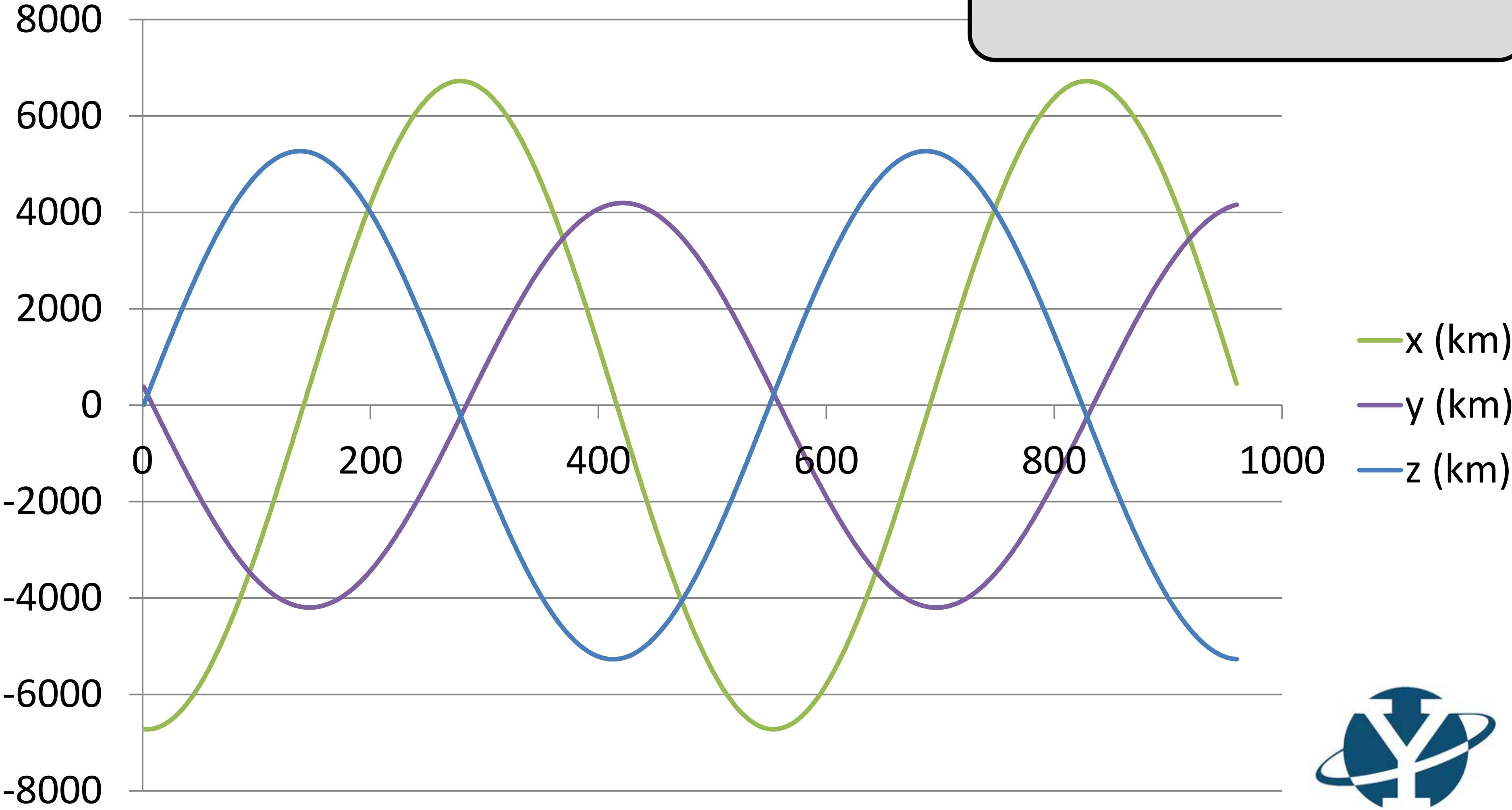



Exponential decay from **proportional** control $\frac{d\omega}{dt} = k\omega$

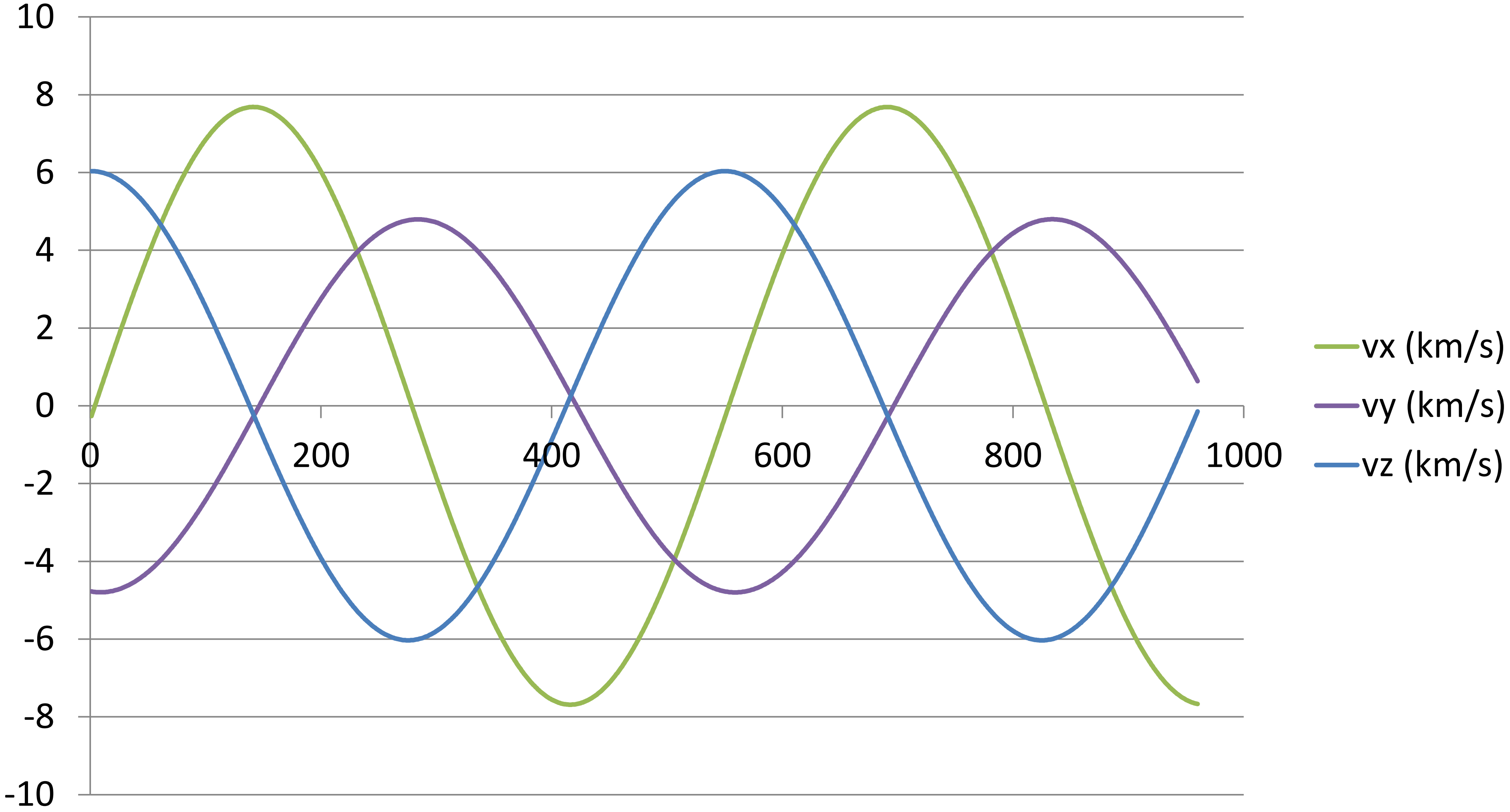
Greater Kp -> faster decay -> but power okay?

# Orbital positions

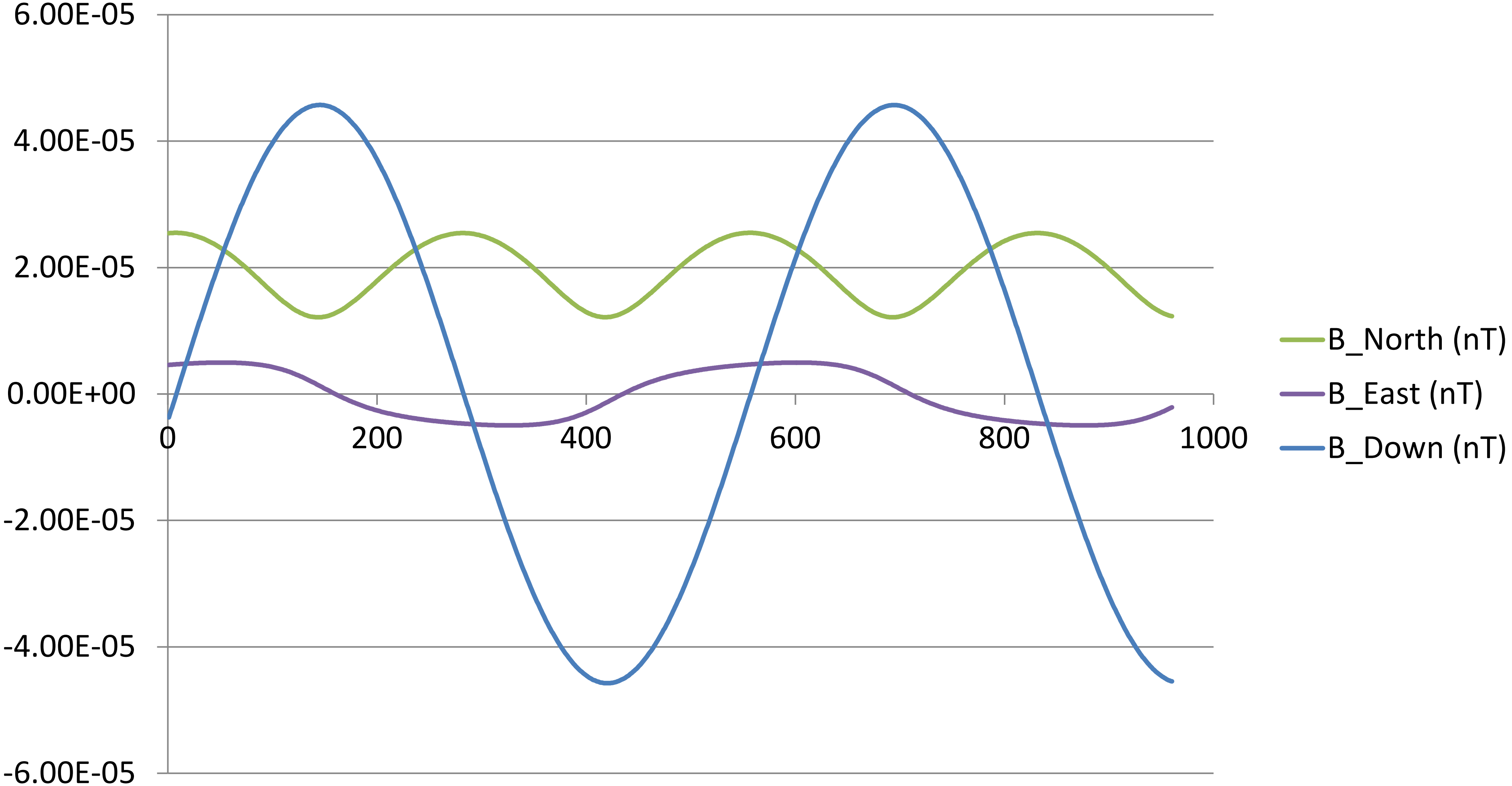
Period correct!



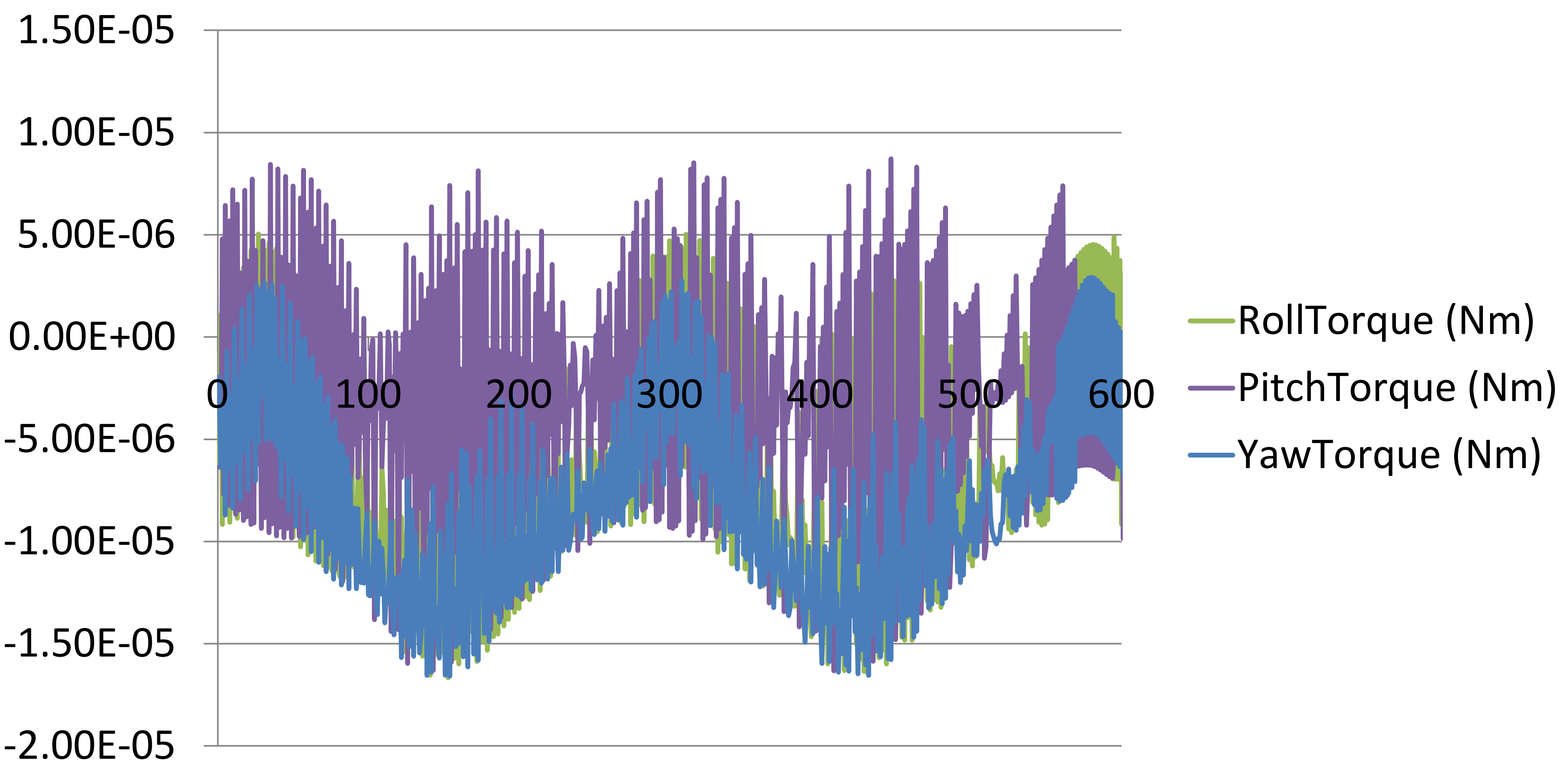
# Orbital Velocities



# Earth magnetic field

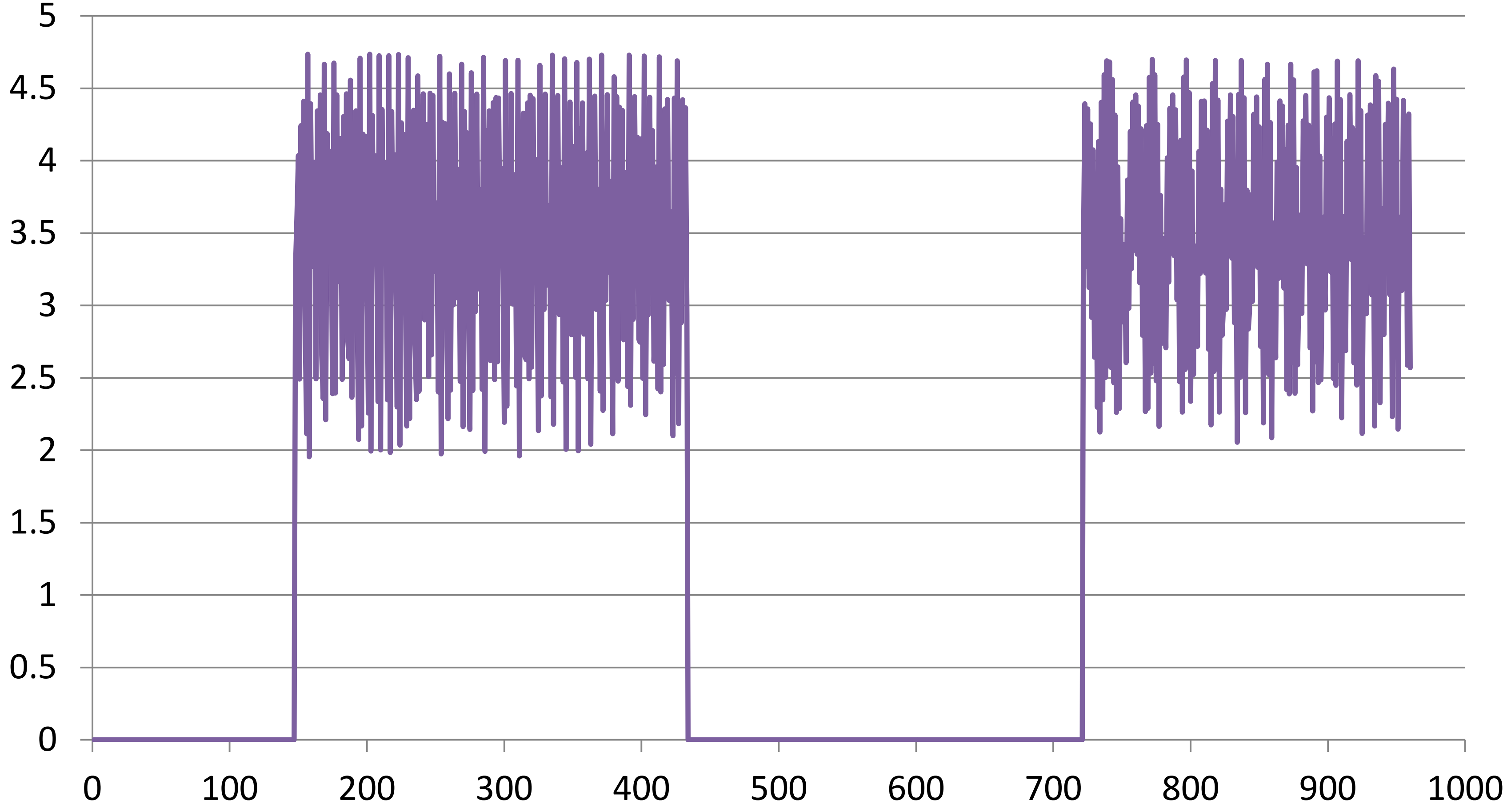


# Torques (Nm)



1.50E-05
1.00E-05
5.00E-06
0.00E+00
-5.00E-06
-1.00E-05
-1.50E-05
-2.00E-05
0
100
200
300
400
500
600
RollTorque (Nm)
PitchTorque (Nm)
YawTorque (Nm)

# Power (Watt)

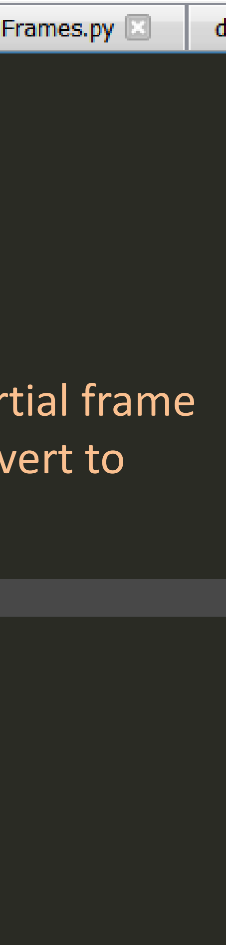


# Suggested next steps

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.