Working with the AWP_attitude_addition python code

introduction

This is my variation of the AWP repository form Alfonso Gonzales. The main goal of it was to add in attitude dynamics and a few more perturbations (mostly for LEO orbit). Besides that i also added comments in the code trying to explain what everything does, added a feature to animate the 3d orbit, added the option of tracking the sun direction relative to the spacecraft and fixed a few bugs and flaws that occurred on my machine. I did this during my internship at SAB Aerospace in Brno.

A big portion of my time here was spent researching the theory. As such I have linked at the end the various sources I have used. I've also tried to reference the sources of the formula's / software implementation used to code this program.

This document does not describe the full python code by far, but it does describe the elements that I believe are most important, and on which I made edits. I believe that the other functions are adequately commented on, self explanatory, or just not important to know about.

As mister Gonzales purposely did not add any documentation to this code (for learning purposes). A chunk of my time was also spent on deciphering the different functions and values. But worry not! I did add comments for you :)

I recommend that any python script using these functions be placed in the "example usage" folder. Although not strictly necessary, this will prevent any issues that might occur with directory paths.

As such the examples in that folder are not mandatory, one can write their own script to e.g simulate multiple orbits based on the results of the previous one, or ones that are slightly offset by a certain value. I tried to keep it as general as possible.

All of the commits and updates that I made can be tracked on the github page.

When spice complains that the correct kernels aren't added. Add the needed file to the data folder, update the spice_data.py file and then add that new element to the furnish_kernels function in spacecraft.py.

SC class

Initializing the spacecraft object:

This is the main file used to define and propagate an orbit.
You start by defining the initial parameters of the orbit and the spacecraft.

```
{
        'cb'            : pd.earth, #defines the central body
        'date0'         : '2020-04-01', #defines the start date
        'et0'           : None, #define the start ephemeris time (if no date)
        'frame'         : 'J2000', #defines the inertial reference frame of the simulation
        'dt'            : 200, #defines the dt at which points are LOGGED
        'tspan'         : '1',
        'orbit_state'   : [],  #orbit defined by position and velocity vector quaternion and angular velocity
        'actuators'     : [0., 0., 0., 0., 0., 0.], #implement a body axis centered linear and rotaional force
        'coes'          : [], #[semi-major axis(km) ,eccentricity ,inclination (deg) , true anomaly, aop(deg), raan(deg)]
        'orbit_perts'   : {}, #defines a list of what pertubations are to be included
        'propagator'    : 'DOP853', #defines which ODE solver is used
        'atol'          : 1e-6, #absolute max error
        'rtol'          : 1e-6, #relative max error
        'stop_conditions': {}, #list of condistions to stop propagations
        'print_stop'    : True,
        'dense_output'  : False,
        'mass0'         : 1, #initial mass
        'inertia0'      : np.array([[1., 0., 0.], #moment of inertia tensor
                                    [0., 1., 0.],
                                    [0., 0., 1.],]),
        'drag_Cp'       : np.zeros(3), #position of centre of pressure relative to body fixed axes
        'solarPress_Cp' : np.zeros(3), #position of centre of pressure relative to body fixed axes
        'output_dir'    : '.',
        'propagate'     : True #bool for propagation on initialisation
    }
```

The snippet above is the “base” spacecraft. When initializing the object you can input a dictionary with the values that apply to you. It will then override these base values with the new ones. An example of this is given below

```
sc   = SC(
           {
           'cb'         : earth,
           'date0'      : '2020-01-01',
           'coes'       : coes,
           #'orbit_state': state,
           'actuators'  : [0., 0., 0., 0., 0., 0.], #these are the forces and torques that act with respect to the body axis
           'mass0'      : 100.,
           'inertia0'   : np.array([[0.1, 0., 0.],
                                    [0., 1., 0.],
                                    [0., 0., 1.]]),
           'drag_Cp'        : np.array([-1., 0., 0.]), #position of the Cp's in the attitude body fixed frame
           'solarPress_Cp'  : np.array([-1., 0., 0.]),
           'tspan'      : '10', #Tspan is either the amount or seconds. If it is a string,it is the amount of orbits
           'dt' : 200, #this decides at which points the integrator STORES points to be plotted
           'orbit_perts': {'J2': True,
                           'n_bodies': [pd.moon, pd.sun, pd.jupiter, pd.saturn ],
                           'grav_grad': True,
                           'atmos_drag': {'CD': 2.2, 'A':10},
                           'solar_press': {'ref': 1, 'A': 10},
                           'mag_torque':{'di_moment': np.array([1., 0., 0.])}
                           }
           } )
```

The initial state of the spacecraft can be given in 2 ways:

1. You can give the orbital kepler elements (under `'coes'`)

```
[semi-major axis(km) ,eccentricity ,inclination (deg) , true anomaly, aop(deg), raan(deg)]
```

2. You can give the initial state of the spacecraft (under `'orbit_state'`)

```
[rx(km), ry, rz, vx(km/s), vy, vz, q0, q1, q2, q3, w1(rad/s), w2, w3]
```

When the `'tspan'` is given as a string it counts it as the amount of *ideal* orbits, otherwise it's in seconds.

One can also choose to add stop conditions such as a minimum/maximum height, enter/leave sphere of influence..etc.

One can also add the perturbations. Whichever is present in the `'orbit_perts'` dict, will be taken into account.

Propagation/integration of the orbit

Unless stated otherwise, the program integrates the orbit immediately after loading the necessary files and the class has been called. It then saves the resulting states under the object.

```
The EOP file 'finals2000A.all' in C:\Users\MichaelGerits/src/iers/ is already the latest.
The Leap Second file 'Leap_Second.dat' in C:\Users\MichaelGerits/src/iers/ is already the latest.

LOADING SPICE KERNELS...

 [Propagating Orbit: 0:00:10] |********************************************************************************************************************************************** |
(ETA:   0:00:00)
```

i've added a progress bar to give an idea of how long the propagation should take.

You can specify which ODE solver you use in the initialisation of the spacecraft under `'propagator'` there you can choose from the options that scipy.solve_ivp gives you.

This integrator is used in the `SC.propagate_orbit()` function, as seen below.

```python
def propagate_orbit( self ):
        #sets a format for the loading bar
        self.widgets = [' [',
            progressbar.Timer(format= 'Propagating Orbit: %(elapsed)s'),
            '] ',
            progressbar.Bar('*'),' (',
            progressbar.ETA(), ') ',
        ]
        self.bar = progressbar.ProgressBar(max_value=(self.config['tspan']+1)/(self.config['dt']), widgets=self.widgets).start()

        self.ode_sol = solve_ivp(
            t_eval       = np.arange(self.et0, self.et0 + self.config['tspan'], self.config[ 'dt']), #desides at which timesteps the values should be stored
            fun          = self.diffy_q, #array with vel and acc (time derivatives of state)
            t_span       = ( self.et0, self.et0 + self.config[ 'tspan' ] ),#time span (from et0 to et0+time span)
            y0           = self.state0, #initial state
            method       = self.config[ 'propagator' ], #what ODE solver is used
            events       = self.stop_condition_functions, #stopping conditions
            rtol         = self.config[ 'rtol' ], #relative accuracy lim
            atol         = self.config[ 'atol' ], #absolute accuracy lim
            #max_step    = 50,
            dense_output = self.config[ 'dense_output' ] )

        self.states  = self.ode_sol.y.T
        self.ets     = self.ode_sol.t
        self.n_steps = self.states.shape[ 0 ]
```

On its own, the integrator decides how big of a step to take based on the error limits it has been given. You can make use of the `max_step` argument to limit the step size it takes in seconds.

The differential equation

for this software it was chosen that it'd be the orbital state that would get integrated, instead of the coefficients. **Before I started working on this, the point mass 2-body problem and the J2 perturbation were already added. I expanded upon this**.

The "state" is a vector with the spacecraft's position, velocity, direction quaternion, angular velocity and the mass.

- Both the position and velocity are with respect to the Inertial frame
- Quaternion is used to rotate the body with respect to the inertial frame
- angular velocity is with respect to the body frame

Since the integrator is only able to solve first order differentials. An intermediary state was needed, which is just the time derivative.

Two body problem:

The base equation of the differential equation is Newton's law of gravity.

$$\ddot{\vec{r}} = -\frac{\mu}{r^3}\vec{r}$$

when you add this with the acceleration due to e.g. thruster forces, you get this implementation in python.

```
a = -r * self.cb[ 'mu' ] / nt.norm( r ) ** 3 + a_g #km/s^2
```

Rigid body kinematics & attitude.

For the implementation of rotation. Quaternions were used, due to the fact that once set up, they are easier to use and have no singularities. This was made even easier by the tinyQuaternions python package. (a good explanation of how they work can be seen here).

To get the time derivative of the quaternion these formulations were used:

$$\dot{q} = \frac{1}{2} q \otimes \vec{\omega}$$

This is equivalent to the following formula, which was used in the software.

$$\begin{bmatrix} \dot{w} \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \frac{1}{2} \cdot \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \cdot \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

more explanation about this (and quaternions in general) can be seen here.

Now if we ever want to transform from the **body frame to inertial frame** we use the **normal** quaternion, and if we want to go from **inertial- to body frame** we use the **conjugate.**

The time derivative of the angular velocity isnot just the angular acceleration.
Since our angular velocity is not around an inertial frame, and our body axis might be placed in multiple ways. The software makes use of an Inertia Tensor and Euler's equation of rigid body kinematics. This makes that the total time derivative of the angular velocity this:

$$\dot{\vec{\omega}}_{b_{icrf}} = \left[\overrightarrow{M_b} - \vec{\omega}_{b_{icrf}} \times \left(I_{mom}\vec{\omega}_{b_{icrf}}\right) - \dot{I}_{mom}\vec{\omega}_{b_{icrf}}\right]\text{inv}(I_{mom})$$

Orbital perturbations

The bulk of my work was spent on these, as only the "J2 effect" was already complete when I started. The original creator of the AWP software did discuss a lot of these topics already, so for most of these I followed that theory. Some of these only work for an earth-centered orbit, as certain models are used to calculate effects. However, when found, models for other planets can easily be added.

The effects of these are added one by one at every iteration of the integration.

each returns an `'effect'` tuple, this consists of the linear and angular acceleration.

J2 effect

This perturbation effect was already added. It takes into account the oblateness of the earth due to its spin. In software, it is implemented this way:

The source for these equations can be found here (the mistake in this video has already been corrected).

```
def calc_J2( self, et, state ):
        '''
        calc the J2 effect on acceleration in km/s^2
        '''
        z2     = state[ 2 ] ** 2
        norm_r = nt.norm( state[ :3 ] )
        r2     = norm_r ** 2
        tx     = state[ 0 ] / norm_r * ( 5 * z2 / r2 - 1 )
        ty     = state[ 1 ] / norm_r * ( 5 * z2 / r2 - 1 )
        tz     = state[ 2 ] / norm_r * ( 5 * z2 / r2 - 3 )
        return (1.5 * self.cb[ 'J2' ] * self.cb[ 'mu' ] * self.cb[ 'radius' ] ** 2 / r2 ** 2 * np.array( [ tx, ty, tz ] ),
                np.zeros(3))
```

n-body perturbations

This function requires a list of bodies to be taken into account. These are taken from the planetary data file. The position of these bodies is accurately calculated at the specific time using SPICE kernels. Using these it calculates the gravitational effect on the spacecraft's position and attitude. more info here.

$$Gm_2\left(\frac{1}{d^3}\mathbf{d}-\frac{1}{r'^3}\mathbf{r}'\right) \quad , \quad \frac{3\mu}{\left|\mathbf{R}_{C/P_i}\right|^5}\mathbf{R}_{C/P_i}\times[I_c]\,\mathbf{R}_{C/P_i}$$

```
def calc_n_bodies( self, et, state ):
        a = np.zeros( 3 )
        alpha = np.zeros( 3 )
        q = Quaternion(q=state[6:10])
        _q = q.conjugate
        #get theposition vector of each body relative to the sc
        for body in self.config[ 'orbit_perts' ][ 'n_bodies' ]:
            r_cb2body  = spice.spkgps( body[ 'SPICE_ID' ], et,
                self.config[ 'frame' ], self.cb[ 'SPICE_ID' ] )[ 0 ]
            r_sc2body = r_cb2body - state[ :3 ]

            #calc acceleration
            a += body[ 'mu' ] * (\
                 r_sc2body / nt.norm( r_sc2body ) ** 3 -\
                 r_cb2body / nt.norm( r_cb2body ) ** 3 )

            #calc attitude effect
            _r = _q.rotatePoint(r_sc2body) #rotate position vector to the body axis frame
            norm_r = nt.norm( _r )

            mult = np.matmul(self.config[ 'inertia0' ], _r)
            cross = np.cross(_r, mult)
            #same as grav gradient torque
            T = 3*self.cb[ 'mu' ]/(norm_r**5) * cross

            alpha += np.matmul(np.linalg.inv(self.config[ 'inertia0' ]), T)
        return (a, alpha)
```

Aerodynamic drag

for the aerodynamic drag, a separate python library was used called PyAtmos. With this, I could find the density of the air based on the altitude. It also includes a more accurate model, so later iterations could use that instead of the simpler exponential. theory for the application here. This does also take into account the spin of the atmosphere, which is assumed to be a constant vector.

$$F_d = \frac{1}{2}\rho u^2 C_d A$$

```
def calc_atmos_drag(self, et, state):
        '''
        calculates the atmospheric drag by making use of an atmosphere model up to 1000km.
        Anything above is consodered no drag.
        includes the relative airspeed using the rotational velocity of the atmosphere
        '''

        r = state[:3]
        v = state[3:6]
        mass = state[13]
        q = Quaternion(q=state[6:10])
        _q = q.conjugate

        alt = nt.norm(r) - self.cb[ 'radius' ]
```

```
        #if the spacecraft is to high for the atmosphere calc
        if alt > 1000:
            return (np.zeros(3), np.zeros(3))

        #load the exponential model at alt
        expo_geom = expo(alt)

        #get the density
        rho = expo_geom.rho
        CD = self.config['orbit_perts']['atmos_drag']['CD']
        A = self.config['orbit_perts']['atmos_drag']['A']

        #get the relative velocity with the atmosphere (atmosphere rotates)
        v_rel = v*1000-np.cross(self.cb['atm_rotation_vec'],r*1000) #change to si units

        #calc drag force
        force = -v_rel * nt.norm(v_rel) * 0.5 * rho * CD * A
        _force = _q.rotatePoint(force) #convert to body fixed frame to calc torque
        torque = np.cross(self.config[ 'drag_Cp' ], _force) #calc torque

        alpha = np.matmul(np.linalg.inv(self.config[ 'inertia0' ]), torque)
        a = force/mass/1000 #convert to km/s^2

        return (a, alpha)
```

Gravity gradient

This perturbation takes into consideration that part of the spacecraft is closer to the central body than the others, thus it will rotate. The formula for this implementation was taken from here. This was also added in the n-bodies perturbation.

$$\frac{3\mu}{\left|\mathbf{R}_{C/P_i}\right|^5}\mathbf{R}_{C/P_i} \times [I_c]\,\mathbf{R}_{C/P_i}$$

```
def calc_grav_gradient( self, et, state):
        '''
        calc the gravity gradiens effect.
        '''
        q = Quaternion(q=state[6:10])
        _q = q.conjugate
        r =  state[ :3 ]
        _r = _q.rotatePoint(r) #rotate position vector to the body axis frame
        norm_r = nt.norm( _r )

        mult = np.matmul(self.config[ 'inertia0' ], _r)
        cross = np.cross(_r, mult)
        T = 3*self.cb[ 'mu' ]/(norm_r**5) * cross #the units work themselves out

        alpha = np.matmul(np.linalg.inv(self.config[ 'inertia0' ]), T)

        return(np.zeros(3), alpha)
```

Magnetic torque

For the magnetic Torque an external module was also used (ppigrf), this time to find the magnetic field. This returned a magnetic field in the ENU coordinate system. This was thus converted to the IAU_Earth frame, then to the initialized inertial reference frame and then to the body fixed frame. Luckily only the ENU to IAU wasn't implemented in the SPICE documentation, so a simple rotation matrix was used for that. Afterwards, using the Dipole moment, it is quite simple to get the torque.

$$\text{ENU to IAU\_EARTH (ECEF)} = \begin{bmatrix} -\sin(\lambda) & \cos(\lambda) & 0 \\ -\sin(\phi)\cos(\lambda) & -\sin(\phi)\sin(\lambda) & \cos(\phi) \\ \cos(\phi)\cos(\lambda) & \cos(\phi)\sin(\lambda) & \sin(\phi) \end{bmatrix}$$

λ = longitude, ϕ = latitude (source)

```python
def calc_mag_torque(self, et, state): #TODO: speed up
        r = state[:3]
        q = Quaternion(q=state[6:10])
        D = self.config['orbit_perts']['mag_torque']['di_moment']


        #gets the longitude, lattitude and the radius
        rad, lon, lat = nt.cart2lat( np.array([r]),self.config[ 'frame' ], self.cb[ 'body_fixed_frame' ], np.array([et]))[0]
        alt = rad-self.cb['radius']

        date = spice.et2utc(et, format_str='ISOC', prec=0).replace('T', ' ')[:10].split('-')
        date = datetime(int(date[0]), int(date[1]), int(date[2]))

        with warnings.catch_warnings(action="ignore"): #gets the magnetic field
            #TODO:Newer magnetic field model will be needed due to deprecated function
            Be, Bn, Bu = ppigrf.igrf(lon, lat, alt, date)
        ENU_vec = np.array([Be[0,0], Bn[0,0], Bu[0,0]])*1e-9

        lon_rad = lon*nt.r2d
        lat_rad = lat*nt.r2d

        R = np.array([
                    [-np.sin(lon_rad),                   np.cos(lon_rad),                   0             ],
                    [-np.sin(lat_rad) * np.cos(lon_rad), -np.sin(lat_rad) * np.sin(lon_rad), np.cos(lat_rad)],
                    [np.cos(lat_rad) * np.cos(lon_rad),  np.cos(lat_rad) * np.sin(lon_rad),  np.sin(lat_rad)]
                    ])

        IAU_vec = np.matmul(R, ENU_vec) #transforms the ENU vec to IAU vector

        J200_vec = nt.frame_transform(np.array([IAU_vec]), self.cb[ 'body_fixed_frame' ], self.config[ 'frame' ], np.array([et]))[0] #tranforms to the J200
inertial frame

        body_vec = q.rotatePoint(J200_vec) #transforms to the body frame

        Torque = np.cross(D, body_vec)

        alpha = np.matmul(np.linalg.inv(self.config[ 'inertia0' ]), Torque)

        return (np.zeros(3), alpha)
```

Solar radiation pressure

Just like with the n-body function. This uses a SPICE function to get the current position of the sun. Then the program is also able to calculate if it is in an eclipse or not. The theory about this perturbation was discussed here

$$g = \frac{\beta}{d^2}\ , \beta = (1+\rho)G_1/B, G_1 \sim 1 \times 10^8$$

```python
def calc_solar_press(self, et,  state):
        r = state[:3]
        mass = state[13]

        if oc.check_eclipse(et, r, self.config['cb'],self.config['frame']) != -1: #checks if it goes trough an eclipse
            return(np.zeros(3), np.zeros(3))

        q = Quaternion(q=state[6:10])
        _q = q.conjugate #rotation vector to vonvert to body axis

        r_cb2body  = spice.spkgps( pd.sun[ 'SPICE_ID' ], et, self.config[ 'frame' ], self.cb[ 'SPICE_ID' ] )[ 0 ] #get the vector form central
body to sun
        r_body2sc = r - r_cb2body
        d_sun = nt.norm(r_body2sc)


        a =
(1+self.config['orbit_perts']['solar_press']['ref'])*pd.sun['G1']*self.config['orbit_perts']['solar_press']['A']*1e-6/(d_sun**3)/mass*r_body2sc
        torque = np.cross(self.config['solarPress_Cp'], _q.rotatePoint(1000*a*mass)) #rotate to body frame and multiply by 1000 to go to SI units
(Nm)
```

```
        alpha = np.matmul(np.linalg.inv(self.config['inertia0']), torque) #rad/s^2
        return (a, alpha)
```

Other applied forces/torques

As mentioned before, I also added the option under `'actuators'` to add a constant force and torque vector with respect to the body fixed axis. This could be used to inspect the effect of parasitic torques or a small continuous thruster bur, or leak.

plotting tools

These are the functions present in the `'plotting_tools.py'` file (usually shortened to `pt`). Here there are a lot of functions to plot all kinds of results.

Most of these were already added, only `'sun_dirs_plot()'` and `'animate_orbits()'` are completely new functions. Most of these however, either needed a lot of fixing, or needed a lot of editing due to the expanded state, now that attitude was added.

For most of these the spacecraft class also has a wrapper function to more easily call these if only one spacecraft is being evaluated. These functions also automatically check if the needed values are calculated, and if not, request the class to do so. This way the calculation does not have to be called separately.

Important to know is that some of these, especially `'plot_groundtracks()'` and `'plot_orbits()'` are built to plot multiple orbits. As such they take a list of arrays. This is more evident in the `"many_orbits.py"` example file. If only one spacecraft is being simulated I recommend using the wrapped functions or making a list with only one element.

Just as with the spacecraft init. Most of these plotting functions also have some parameters that can be tweaked by inputting some arguments. such as the time unit, figure size..etc

3d plot

this plot shows a 3d plot of the orbits, which can be moved around and zoomed in. The dot in each orbit marks the beginning point of each spacecraft.

This plot can be called in 2 ways.

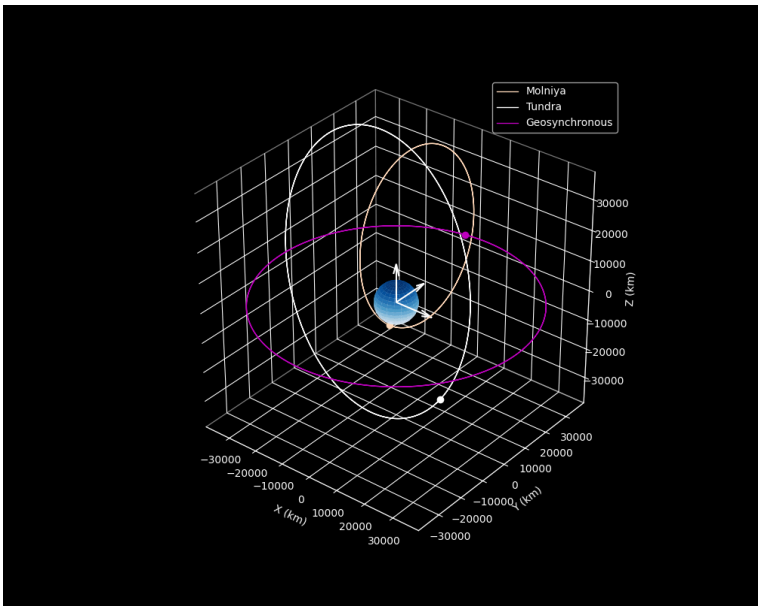
- spacecraft wrapper function

  `sc.plot_3d(ani = True,  args = { 'show': True, 'ani_name': 'orbit.gif', 'frames': None, 'showTime': True, 'fps': 5})`

  This includes both the plot and and the animation. To show the plot separately, set `'show'` to true

- calling the plotting function directly

```
rs = [ sc.states[ :, :3 ] for sc in scs ]
pt.plot_orbits(rs, {'show': True, 'labels': [ 'Molniya', 'Tundra', 'Geosynchronous' ]})
```



3d animate

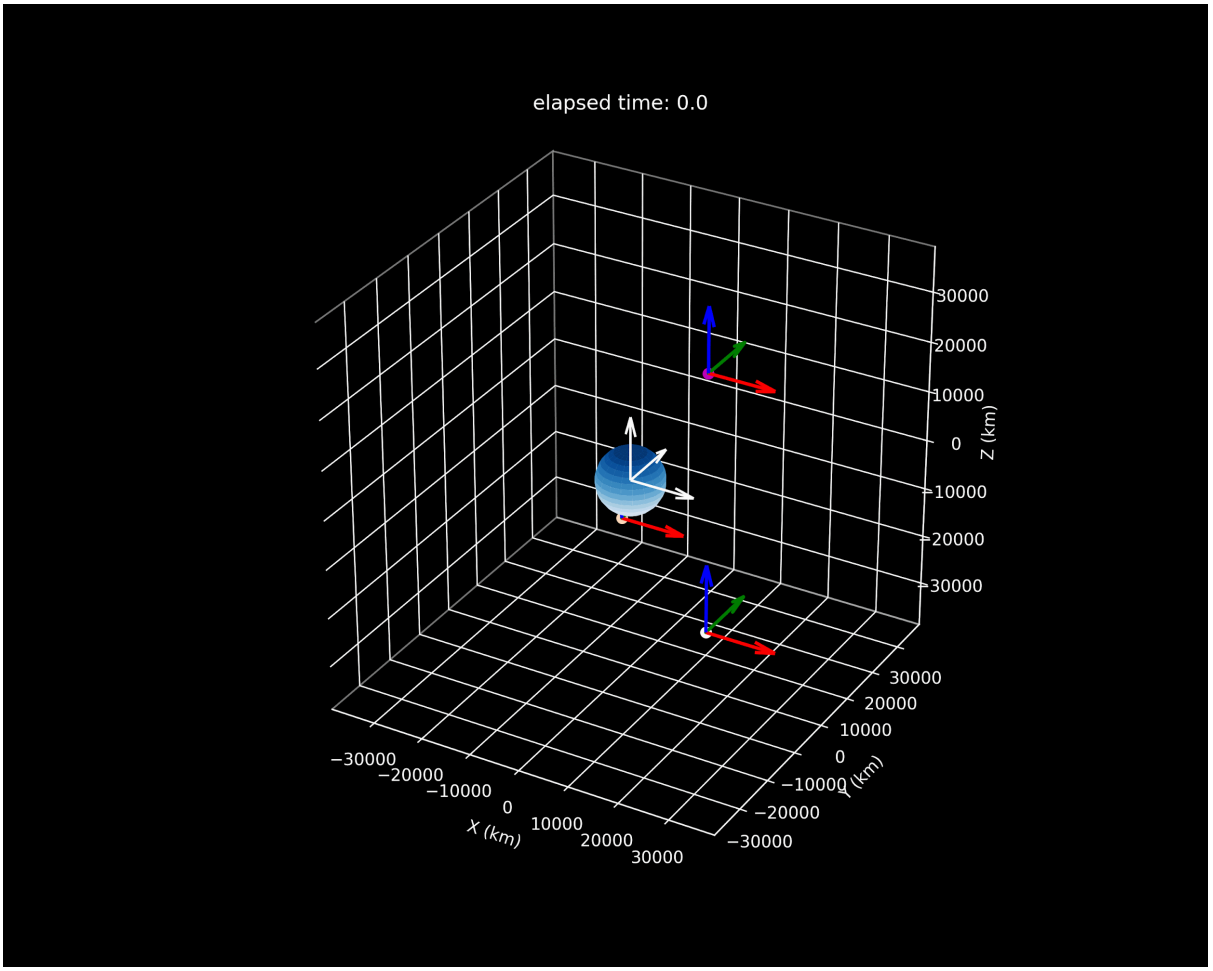
This function works by making multiple plots and stitching them together. When running it, it will start rendering the frames and saving them in the ‘Frames’ folder. When done, or when interrupted it will **automatically** delete the frame files and create the .gif file.

For the wrapper function `'ani'` has to be set to True. The `'frames'` key decides how many frames will be made. If this is below the amount of logged points it will stop the animation prematurely. If set to None, it will automatically set it so that all orbits are animated smoothly.

Animation also offers the ability to plot the inertial, orbital, and body fixed axes. To define which is shown, put True or false for each in the arguments.

When plotting multiple orbits, caution should be taken to set the `'steps'` to the smallest amount such that no errors are given. An example of this can be seen in the `'many_orbits.py'` example file.

```
pt.animate_orbits( max_steps, rs, vs, quats, times, args = { 'ani_name': 'mult_orbit.gif', 'lb_axes': True, 'or_axes': False,
'labels': [ 'Molniya', 'Tundra', 'Geosynchronous' ], 'frames': None })
```



state plot

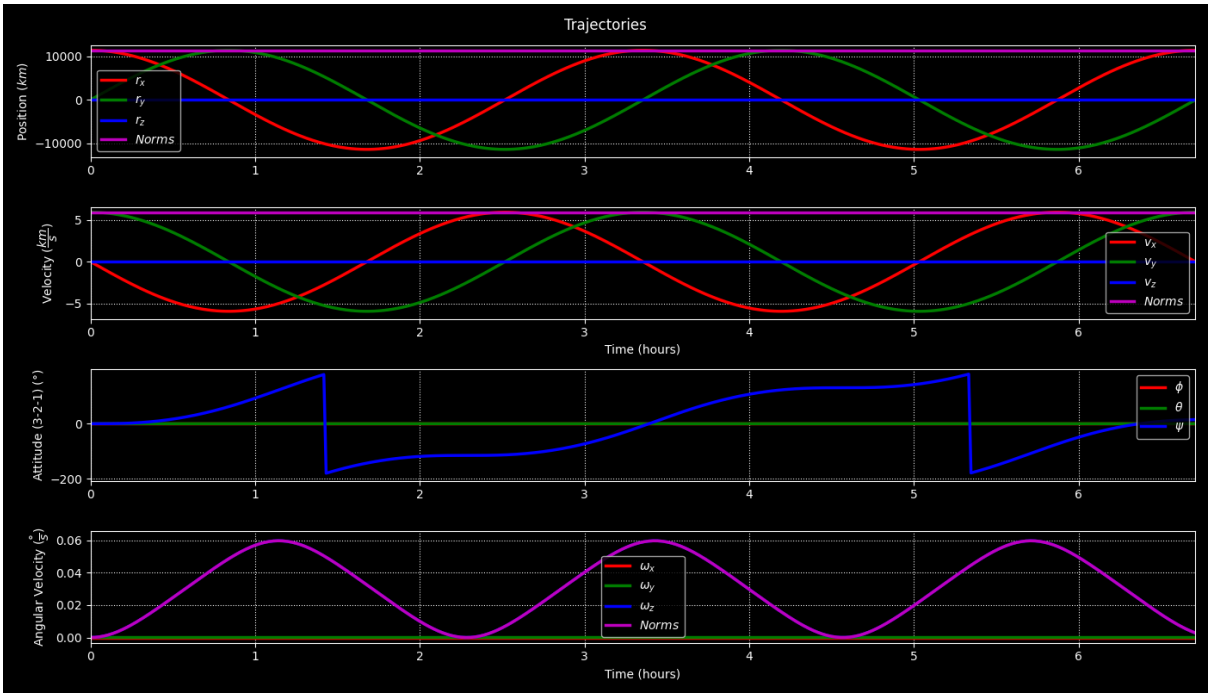
This plotting function only works for one spacecraft at a time. As such I would highly recommend just using the wrapper function.

This plots the states of the spacecraft over time (pretty self explanatory)

```
sc.plot_states(args = {'show': True, 'time_unit': 'days'})
```

For most of these functions the `'args'` parameter is optional. If the standard arguments (in `'plotting_tools.py'` are sufficient, you can leave this argument out of the function call the attitude now also is displayed in euler angles from the inertial frame. To convert them, this formula was used.

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \operatorname{atan2}\left(2(q_w q_x + q_y q_z), 1 - 2(q_x^2 + q_y^2)\right) \\ -\pi/2 + 2\operatorname{atan2}\left(\sqrt{1 + 2(q_w q_y - q_x q_z)}, \sqrt{1 - 2(q_w q_y - q_x q_z)}\right) \\ \operatorname{atan2}\left(2(q_w q_z + q_x q_y), 1 - 2(q_y^2 + q_z^2)\right) \end{bmatrix}$$

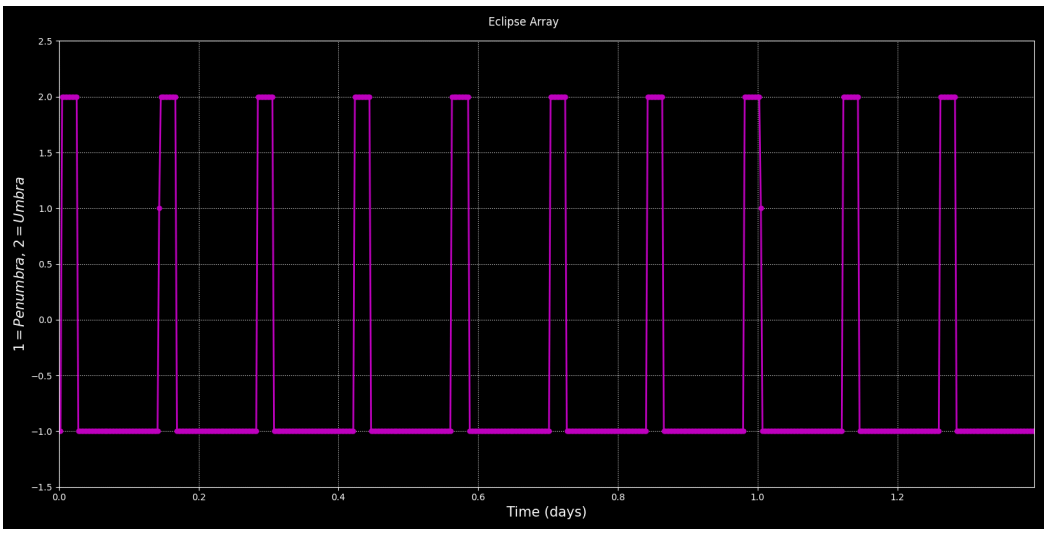


Eclipse plot

This function plots the eclipse state of the orbit over time. It can distinguish between an Umbra eclipse and a ênumbra eclipse. This is also a plot that can only be shown for one spacecraft at a time. It is called in a very similar way to the states plot.

In the most recent version i've also added the option of calculating eclipses by neighboring bodies. This can be done by adding a list of other bodies you want to consider before `'args'`

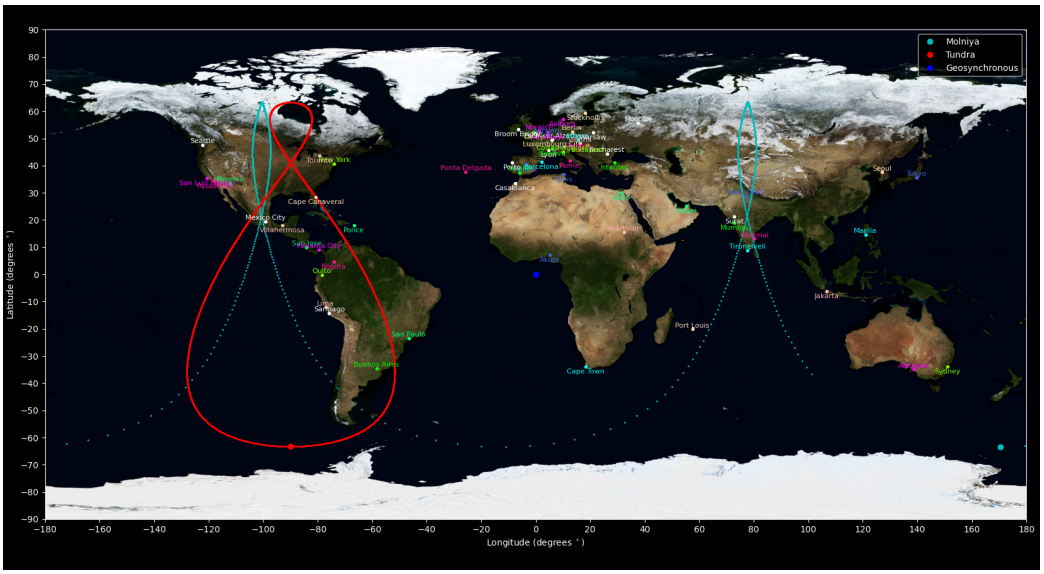
```
sc.plot_eclipse_array(args = {'show': True, 'time_unit': 'days'})
```



groundtrack plot

The groundtrack plot can be called for multiple spacecraft at a time, as such the syntax is similar to the 3d plot function. When calling the plot function on its own do not forget to run the `'calc_latlons()'` function first. When using a different central body, make sure to change the surface map to the correct central body with `'surface_body'` in arguments (also don't plot the cities). When other locations need to be added, one can follow a similar step as done with the `'cities_lat_long.py'` file and then specify the correct list in the arguments.

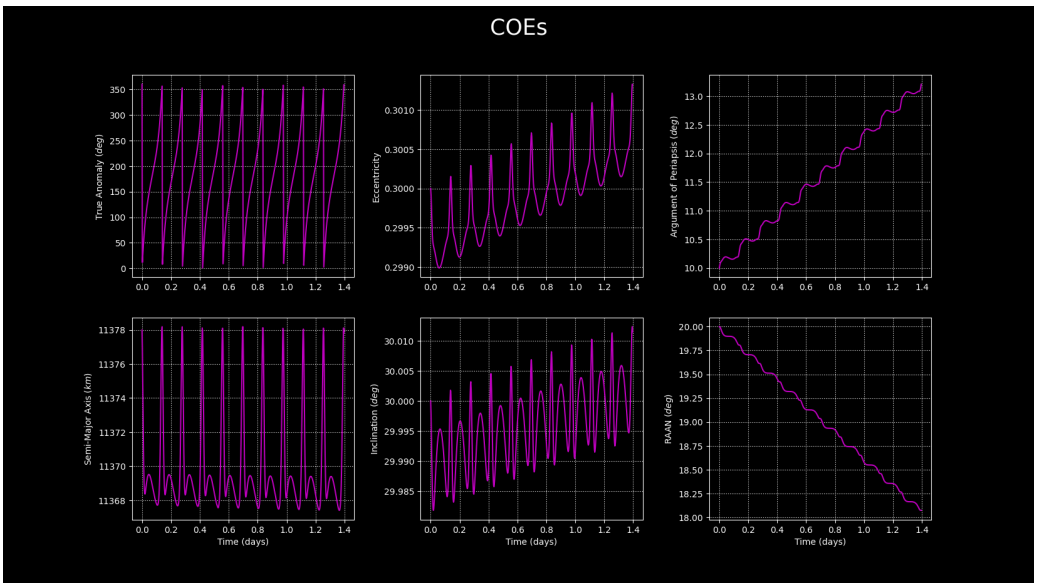
```
pt.plot_groundtracks (latlons, args = {'show': True, 'grid': False,
'labels': [ 'Molniya', 'Tundra', 'Geosynchronous' ],})
```



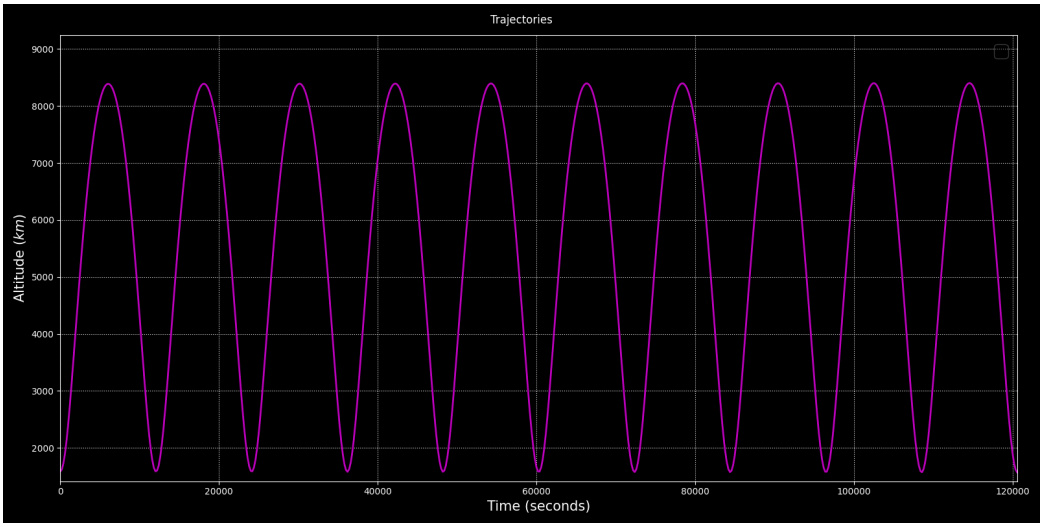
Orbit coefficients plot

When using this function it first converts the states into coefficients for each logged point. One may see that after a while numerical error builds up. This can be remedied by adding a max time step and a higher order integration. The original maker of AWP had a similar issue. While for most cases this error should not be too large, it should still be looked at critically. from this timestamp this error is talked about more. Currently it uses a SPICE function to convert the state to the coefficients, but i did try to implement the calculations myself, to the same result.

```
sc.plot_coes(args = {'show': True, 'time_unit': 'days'})
```



Altitude plot



Planetary data

This file mainly contains specific constants of celestial bodies, but also the reference frames they use and, most importantly, their identification to get accurate up-to-date info using SPICE. Here is an example from that file of earth. These can all be found in the `'planetary_data.py'` file in python tools.

```
earth = {
        'name'            : 'Earth',
        'spice_name'      : 'EARTH',
        'SPICE_ID'        : 399,
        'mass'            : 5.972e24,
        'mu'              : 5.972e24 * G, #km^3/s^2
        'radius'          : 6378.0, #km
        'J2'              : 1.081874e-3,
        'atm_rotation_vec': [0., 0., 72.9211e-6], #rad/s
        'sma'             : 149.596e6, # km
```

```
        'SOI'             : 926006.6608, # km
        'deorbit_altitude': 100.0, # km
        'cmap'            : 'Blues',
        'body_fixed_frame': 'IAU_EARTH',
        'traj_color'      : 'b'
        }
```

Extra

As a bonus (by request of one of my colleagues) I've also added in the option of Calculating and plotting the direction of the sun with respect to the body fixed frame. It does so by calculating the direction angles from the axes.

This is implemented such that it works in a similar way to all other plotting functions, specifically it follows the syntax of a single spacecraft plot. Thus using the wrapper function is recommended..

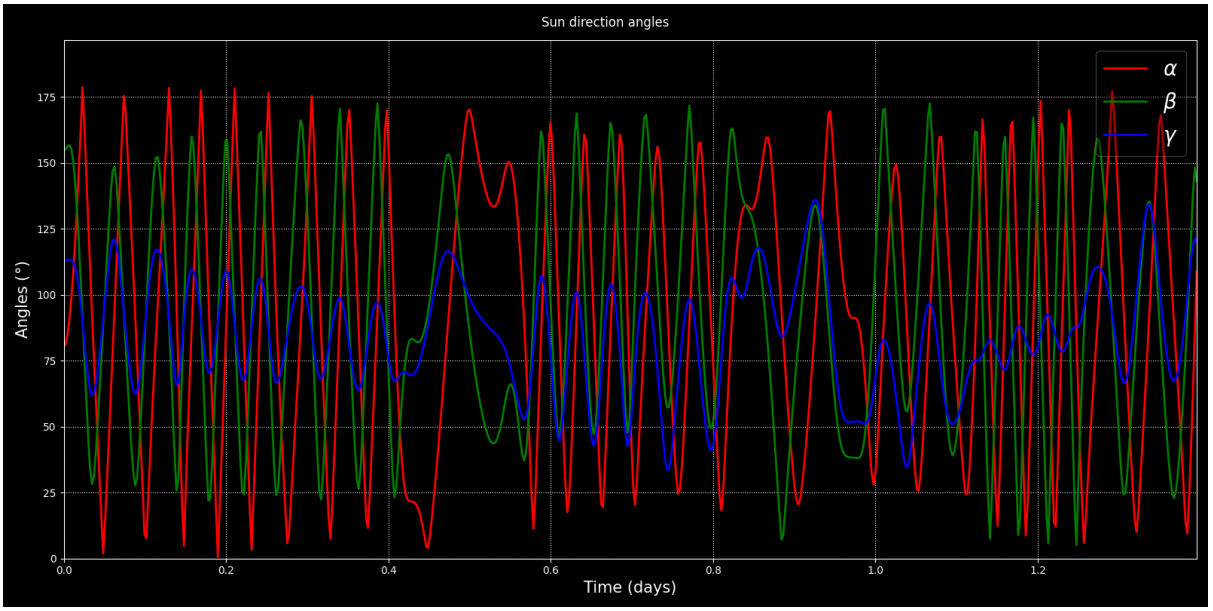
```
def calc_sun_dirs( self ):
        '''
        calculates the directional angles between the body fixed frame and the direction of the sun in degrees
        '''
        print( '\nCalculating sun directions..' )
        self.sun_dirs = np.zeros( ( self.n_steps, 3 ) ) #allocates the memory
        axes = np.array([[1,0,0],
                         [0,1,0],  #defines the body axes
                         [0,0,1]])

        for n in range( self.n_steps ):
            q = Quaternion(q=self.states[n, 6:10])
            _q = q.conjugate #rotation vector to convert to body axis

            r_cb2sun  = spice.spkgps( pd.sun[ 'SPICE_ID' ], self.ets[n], self.config[ 'frame' ], self.cb[ 'SPICE_ID' ] )[ 0 ] #get the vector
form central body to sun
            r_sc2sun = (r_cb2sun - self.states[n, :3])
            sun_dir = _q.rotatePoint(nt.normed(r_sc2sun))

            alpha = nt.vecs2angle(axes[0], sun_dir)
            beta = nt.vecs2angle(axes[1], sun_dir)
            gamma = nt.vecs2angle(axes[2], sun_dir)

            self.sun_dirs[n] = [alpha, beta, gamma]
        self.sun_dirs_calculated = True
```



Theory sources

- Fundamentals of Astrodynamics, Roger. R. Bate, Donald. D. Meuller, Jerry. E. White
- Ranjan Vepa - Dynamics and Control of Autonomous Space Vehicles and Robotics (2.2, 2.3, 2.4, 3.3, 3.4)
- Howard D. Curtis - Orbital Mechanics for Engineering Students (Chapters 6, 9)
- MARCEL J. SIDI - Spacecraft Dynamics and Control (4.5, 4.7, 4.8)
- dynamics and control of autonomous space vehicles and robotics, Ranjan Vepa
- Space mission engineering: the new SMAD, James. R. Wertz, David. F. Everett, Jeffrey. J. Puschell
- https://en.wikipedia.org/wiki/Euler%27s_equations_(rigid_body_dynamics)
- https://www.youtube.com/playlist?list=PLUeHTafWecAXDFDYEwunLL2V2kwqKzkvJ
- https://www.youtube.com/watch?v=HOWJp8NV5xU
- https://www.youtube.com/watch?v=AReKBoiph6g
- https://www.youtube.com/watch?v=LeJpWfyXJPw
-