Orbit Determinator Documentation

Release 1.0.0

Nilesh Chaturvedi, Alexandros Kazantzidis

Aug 07, 2018

Contents:

CHAPTER 1

About Orbit Determinator

The orbitdeterminator package provides tools to compute the orbit of a satellite from positional measurements. It supports both cartesian and spherical coordinates for the initial positional data, two filters for smoothing and removing errors from the initial data set and finally two methods for preliminary orbit determination. The package is labeled as an open source scientific package and can be helpful for projects concerning space orbit tracking.

Lots of university students build their own cubesat's and set them into space orbit, lots of researchers start building their own ground station to track active satellite missions. For those particular space enthusiasts we suggest using and trying our package. Any feedback is more than welcome and we wish our work to inspire other's to join us and add more helpful features.

Our future goals for the package is to add a 3d visual graph of the final computed satellite orbit, add more filters, methods and with the help of a tracking ground station to build a server system that computes orbital elements for many active satellite missions.

CHAPTER 2

Copyright and License

The project's idea belongs to AerospaceResearch.net and Andreas Hornig and it has been developed under Google summer of code 2017 by Nilesh Chaturvedi and Alexandros Kazantzidis.

CHAPTER 3

Installation

Open up your control panel, pip install git if you do not already have it and then clone the github repository of the program https://github.com/aerospaceresearch/orbitdeterminator. Create a new virtual environment for python version 3.4. Then, all you need to do is go to the directory where the package has been cloned with cd orbitdeterminator and run **python setup.py install**. That should install the package into your Lib/site-packages and you will be able to import and use it. Other than import you can just use it immediately from the clone directory (preferred).

3.1 Modules documentation

3.1.1 Filters:

Triple Moving Average

Here we take the average of 3 terms x0, A, B where, x0 = The point to be estimated A = weighted average of n terms previous to x0 B = weighted avreage of n terms ahead of x0 n = window size

orbitdeterminator.filters.triple_moving_average.**generate_filtered_data**(*filename*, *window*)

Apply the filter and generate the filtered data

Parameters

- **filename** (*string*) – the name of the .csv file containing the positional data
- **window** (*int*) – window size applied into the filter

Returns the final filtered array

Return type numpy array

orbitdeterminator.filters.triple_moving_average.**triple_moving_average**(*signal_array*, *window_size*)

Apply triple moving average to a signal

Parameters

- **signal_array** (*numpy array*) – the array of values on which the filter is to be applied
- **window_size** (*int*) – the no. of points before and after x0 which should be considered for calculating A and B

Returns a filtered array of size same as that of signal_array

Return type numpy array

orbitdeterminator.filters.triple_moving_average.**weighted_average**(*params*)
Calculates the weighted average of terms in the input

Parameters **params** (*list*) – a list of numbers

Returns weighted average of the terms in the list

Return type list

Savintzky - Golay

Takes a positional data set (time, x, y, z) and applies the Savintzky Golay filter on it based on the polynomial and window parameters we input

orbitdeterminator.filters.sav_golay.**golay**(*data*, *window*, *degree*)
Apply the Savintzky-Golay filter to a positional data set.

Parameters

- **data** (*numpy array*) – containing all of the positional data in the format of (time, x, y, z)
- **window** (*int*) – window size of the Savintzky-Golay filter
- **degree** (*int*) – degree of the polynomial in Savintzky-Golay filter

Returns filtered data in the same format

Return type numpy array

3.1.2 Interpolation:

Lamberts-Kalman Method

Takes a positional data set and produces sets of six keplerian elements using Lambert's solution for preliminary orbit determination and Kalman filters

orbitdeterminator.kep_determination.lamberts_kalman.**check_keplerian**(*kep*)
Checks all the sets of keplerian elements to see if they have wrong values like eccentricity greater that 1 or a negative number for semi major axis

Parameters **kep** (*numpy array*) – all the sets of keplerian elements in [semi major axis (a), eccentricity (e), inclination (i), argument of perigee (ω), right ascension of the ascending node (Ω), true anomaly (v)] format

Returns the final corrected set of keplerian elements that will be inputed in the kalman filter

Return type numpy array

orbitdeterminator.kep_determination.lamberts_kalman.**create_kep**(*my_data*)
Computes all the keplerian elements for every point of the orbit you provide using Lambert's solution It implements a tool for deleting all the points that give extremely jittery state vectors

Parameters **data** (*numpy array*) – contains the positional data set in (Time, x, y, z) Format

Returns array containing all the keplerian elements computed for the orbit given in [semi major axis (a), eccentricity (e), inclination (i), argument of perigee (ω), right ascension of the ascending node (Ω), true anomaly (v)] format

Return type numpy array

orbitdeterminator.kep_determination.lamberts_kalman.**kalman**(*kep*, *R*)
Takes as an input lots of sets of keplerian elements and produces the fitted value of them by applying kalman filters

Parameters

- **kep** (*numpy array*) – containing keplerian elements in this format (a, e, i, ω, Ω, v)
- **R** – estimate of measurement variance

Returns final set of keplerian elements describing the orbit based on kalman filtering

Return type numpy array

orbitdeterminator.kep_determination.lamberts_kalman.**lamberts**(*x1*, *x2*, *traj*)
Takes two position points - numpy arrays with time,x,y,z as elements and produces two vectors with the state vector for both positions using Lamberts solution

Parameters

- **x1** (*numpy array*) – time and position for point 1 [time1,x1,y1,z1]
- **x2** (*numpy array*) – time and position for point 2 [time2,x2,y2,z2]

Returns velocity vector for point 1 (vx, vy, vz)

Return type numpy array

orbitdeterminator.kep_determination.lamberts_kalman.**orbit_trajectory**(*x1_new*, *x2_new*, *time*)
Tool for checking if the motion of the sallite is retrogade or counter - clock wise

Parameters

- **x1** (*numpy array*) – time and position for point 1 [time1,x1,y1,z1]
- **x2** (*numpy array*) – time and position for point 2 [time2,x2,y2,z2]
- **time** (*float*) – time difference between the 2 points

Returns true if we want to keep retrogade, False if we want counter-clock wise

Return type bool

Gibb's Method

class orbitdeterminator.kep_determination.gibbsMethod.**Gibbs**

classmethod convert_list(*vec*)
Type casts the input data for the ease of use.

Converts the values of the input list with string datatype into float for the ease of furthur computation.

Parameters **vec** (*list*) – input vector

Returns vector converted to float values

Return type list

classmethod cross_product(*a*, *b*)

Computes cross product of the given vectors. Returns a vector.

Parameters

- **a** (*list/array*) – first vector
- **b** (*list/array*) – second vector

Returns cross product of given vectors

Return type list/array

classmethod dot_product(*a*, *b*)

Computes dot product of two vectors. Multiplies corresponding axis with each other and then adds them. Returns a single value.

Parameters

- **a** (*list/array*) – first vector
- **b** (*list/array*) – second vector

Returns dot product of given vectors

Return type float

classmethod find_length(*path*)

Finds the length of the input file.

Calculates the length of the file with the given path containing all the position vectors. File should contain a header line describing all of its attributes in a single line only. This function removes the header line before it calculates file length. If the file does not contains the header line then the first line of data will get removed and you have to add 1 to the output after it returns the result.

Parameters **path** (*str*) – file path

Returns length of file

Return type int

classmethod gibbs(*r1*, *r2*, *r3*)

Computes state vector from the given set of three position vectors using Gibb's implementation.

Computes velocity vector (part of state vector) using Gibb's Method and takes r2 (input argument) as its position vector (part of state vector). Both combined forms state vector.

Parameters

- **r1** (*list*) – first position vector
- **r2** (*list*) – second position vector
- **r3** (*list*) – third position vector

Returns velocity vector

Return type list

classmethod magnitude(*vec*)
Computes magnitude of the input vector.

Parameters **vec** (*list*) – vector

Returns magnitude of vector

Return type float

classmethod operate_vector(*a*, *b*, *flag*)
Adds or subtracts input vectors based on the flag value.

If flag is 1 then add both vectors with corresponding values else if flag is 0 (zero) then subtract two vectors with corresponding values. Returns a vector.

Parameters

- **a** (*list/array*) – first vector
- **b** (*list/array*) – second vector
- **flag** (*int*) – checks for operation (addition/subtraction)

Returns sum/difference of vector based on flag value

Return type list/array

classmethod orbital_elements(*r*, *v*)
Computes orbital elements from state vector.

Orbital elements is a set of six parameters which are semi-major axis, inclination, right ascension of the ascending node, eccentricity, argument of perigee and mean anomaly.

Parameters

- **r** (*list*) – position vector
- **v** (*list*) – velocity vector

Returns set of six orbital elements

Return type list

read_file(*path*)
Invokes the Gibb's implementation and stores the result in a list.

Read the file with the given path and forms a set of three position vectors then applies Gibb's Method on that set and computes state vector for every set. After computing state vectors it stores the result into a list. Now, these state vectors can be used to find orbital elements.

Parameters **path** (*str*) – path to input file

Returns list of all pair of position and velocity vector

Return type numpy.ndarray

classmethod unit(*vec*)
Finds unit vector of the given vector. Divides each value of vector by its magnitude. Returns a vector.

Parameters **vec** (*list*) – input vector

Returns unit vector

Return type list

Spline Interpolation

Interpolation using splines for calculating velocity at a point and hence the orbital elements

orbitdeterminator.kep_determination.interpolation.**compute_velocity**(*spline*, *point*)

Calculate the derivative of spline at the point(on the points the given spline corresponds to). This gives the velocity at that point.

Parameters

- **spline** (*list*) – component wise cubic splines of orbit data points of the format [spline_x, spline_y, spline_z].
- **point** (*numpy array*) – point at which velocity is to be calculated.

Returns velocity vector at the given point

Return type numpy array

orbitdeterminator.kep_determination.interpolation.**cubic_spline**(*orbit_data*)

Compute component wise cubic spline of points of input data

Parameters

- **orbit_data** (*numpy array*) – array of orbit data points of the
- **[time, x, y, z]** (*format*) –

Returns component wise cubic splines of orbit data points of the format [spline_x, spline_y, spline_z]

Return type list

orbitdeterminator.kep_determination.interpolation.**main**(*data_points*)

Apply the whole process of interpolation for keplerian element computation

Parameters **data_points** (*numpy array*) – positional data set in format of (x, y, z, time)

Returns computed keplerian elements for every point of the orbit

Return type numpy array

Ellipse Fit

Finds out the ellipse that best fits to a set of data points and calculates its keplerian elements.

orbitdeterminator.kep_determination.ellipse_fit.**determine_kep**(*data*)

Determines keplerian elements that fit a set of points.

Parameters **data** (*nx3 numpy array*) – A numpy array of points in the format [x y z].

Returns

(kep,res) - The keplerian elements and the residuals as a tuple. kep: 1x6 numpy array res: nx3 numpy array

For the keplerian elements: kep[0] - semi-major axis (in whatever units the data was provided in) kep[1] - eccentricity kep[2] - inclination (in degrees) kep[3] - argument of periapsis (in degrees) kep[4] - right ascension of ascending node (in degrees) kep[5] - true anomaly of the first row in the data (in degrees)

For the residuals: (in whatever units the data was provided in) res[0] - residuals in x axis res[1] - residuals in y axis res[2] - residuals in z axis

orbitdeterminator.kep_determination.ellipse_fit.**plot_kep**(*kep*, *data*)
Plots the original data and the orbit defined by the keplerian elements.

Parameters

- **kep** (*1x6 numpy array*) – keplerian elements
- **data** (*nx3 numpy array*) – original data

Returns nothing

3.1.3 Propagation:

Propagation Model

class orbitdeterminator.propagation.sgp4.**SGP4**

__init__()
Initializes flag variable to check for FlagCheckError (custom exception).

compute_necessary_kep(*kep*, *b_star=2.1109e-05*)
Initializes the necessary class variables using keplerian elements which are needed in the computation of the propagation model.

Parameters

- **kep** (*list*) – kep elements in order [axis, inclination, ascension, eccentricity, perigee, anomaly]
- **b_star** (*float*) – bstar drag term

Returns NIL

compute_necessary_tle(*line1*, *line2*)
Initializes the necessary class variables using TLE which are needed in the computation of the propagation model.

Parameters

- **line1** (*str*) – line 1 of the TLE
- **line2** (*str*) – line 2 of the TLE

Returns NIL

propagate(*t1*, *t2*)
Invokes the function to compute state vectors and organises the final result.

The function first checks if compute_necessary_xxx() is called or not if not then a custom exception is raised stating that call this function first. Then it computes the state vector for the next 8 hours (28800 seconds in 8 hours) at every time epoch (28800 time epcohs) using the sgp4 propagation model. The values of state vector is formatted upto five decimal points and then all the state vectors got appended in a list which stores the final output.

Parameters

- **t1** (*int*) – start time epoch
- **t2** (*int*) – end time epoch

Returns vector containing all state vectors

Return type numpy.ndarray

propagation_model(*tsince*)
From the time epoch and information from TLE, applies SGP4 on it.

The function applies the Simplified General Perturbations algorithm SGP4 on the information extracted from the TLE at the given time epoch 'tsince' and computes the state vector from it.

Parameters **tsince** (*int*) – time epoch

Returns position and velocity vector

Return type tuple

classmethod recover_tle(*pos*, *vel*)
Recovers TLE back from state vector.

First of all, only necessary information (which are inclination, right ascension of the ascending node, eccentricity, argument of perigee, mean anomaly, mean motion and bstar) that are needed in the computation of SGP4 propagation model are recovered. It is using a general format of TLE. State vectors are used to find orbital elements which are then inserted into the TLE format at their respective positions. Mean motion and bstar is calculated separately as it is not a part of orbital elements. Format of TLE: x denotes that there is a digit, c denotes a character value, underscore(_) denotes a plus/minus(+/-) sign value and period(.) denotes a decimal point.

Parameters

- **pos** (*list*) – position vector
- **vel** (*list*) – velocity vector

Returns line1 and line2 of TLE

Return type list

class orbitdeterminator.propagation.sgp4.**FlagCheckError**
Raised when compute_necessary_xxx() function is not called.

Cowell Method

Numerical orbit propagator based on RK4. Takes into account J2 and drag perturbations.

orbitdeterminator.propagation.cowell.**drag**(*s*)
Returns the drag acceleration for a given state.

Parameters **s** (*1x6 numpy array*) – the state vector [rx,ry,rz,vx,vy,vz]

Returns the drag acceleration [ax,ay,az]

Return type 1x3 numpy array

orbitdeterminator.propagation.cowell.**j2_pert**(*s*)
Returns the J2 acceleration for a given state.

Parameters **s** (*1x6 numpy array*) – the state vector [rx,ry,rz,vx,vy,vz]

Returns the J2 acceleration [ax,ay,az]

Return type 1x3 numpy array

orbitdeterminator.propagation.cowell.**propagate_state**(*s*, *t0*, *tf*)
Equivalent to the rk4 function.

orbitdeterminator.propagation.cowell.**rk4**(*s*, *t0*, *tf*, *h=30*)
Runge-Kutta 4th Order Numerical Integrator

Args: s(1x6 numpy array): the state vector [rx,ry,rz,vx,vy,vz] t0(float) : initial time tf(float) : final time h(float) : step-size

Returns the state at time tf

Return type 1x6 numpy array

orbitdeterminator.propagation.cowell.**rkf45**(*s*, *t0*, *tf*, *h=10*, *tol=1e-06*)
Runge-Kutta Fehlberg 4(5) Numerical Integrator

Args: s(1x6 numpy array): the state vector [rx,ry,rz,vx,vy,vz] t0(float) : initial time tf(float) : final time h(float) : step-size tol(float) : tolerance of error

Returns the state at time tf

Return type 1x6 numpy array

orbitdeterminator.propagation.cowell.**sdot**(*s*)
Returns the time derivative of a given state.

Parameters **s** (*1x6 numpy array*) – the state vector [rx,ry,rz,vx,vy,vz]

Returns the time derivative of s [vx,vy,vz,ax,ay,az]

Return type 1x6 numpy array

orbitdeterminator.propagation.cowell.**time_period**(*s*, *h=30*)
Returns the nodal time period of an orbit.

Parameters

- **s** (*1x6 numpy array*) – the state vector [rx,ry,rz,vx,vy,vz]
- **h** (*float*) – step-size

Returns the nodal time period of the orbit

Return type float

Simulator

class orbitdeterminator.propagation.simulator.**Simulator**(*params*)
A class for the simulator.

__init__(*params*)
Initializes the simulator.

Parameters **params** – A SimParams object containing kep,t0,t,period,speed, and op_writer

Returns nothing

calc()
Calculates the satellite state at current time and calls itself after a certain amount of time.

simulate()
Starts the calculation thread and waits for keyboard input. Press q or Ctrl-C to quit the simulator cleanly.

stop()
Stops the simulator cleanly.

class orbitdeterminator.propagation.simulator.**SimParams**
SimParams class. This is just a container for all the parameters required to start the simulation.

kep(1x6 numpy array): the intial osculating keplerian elements epoch(float): the epoch of the above kep period(float): maximum time period between observations t0(float): starting time of the simulation speed(float): speed of the simulation op_writer(OpWriter): output handling object

class orbitdeterminator.propagation.simulator.**OpWriter**
Base output writer class. Inherit this class and override the methods.

close()
Anything that has to be executed after finishing writing the output. Runs once.

Example: Closing connection to a database

open()
Anything that has to be executed before starting to write output. Runs once.

Example: Establishing connection to database

static write(*t*, *s*)
This method is called everytime the calc thread finishes a computation.

Parameters

- **t** – the current time of simulation
- **s** – the state vector at t [rx,ry,rz,vx,vy,vz]

class orbitdeterminator.propagation.simulator.**print_r**
Bases: *orbitdeterminator.propagation.simulator.OpWriter*

Prints the position vector

class orbitdeterminator.propagation.simulator.**save_r**(*name*)
Bases: *orbitdeterminator.propagation.simulator.OpWriter*

Saves the position vector to a file

__init__(*name*)
Initialize the class.

Parameters **name** (*string*) – file name

DGSN Simulator

class orbitdeterminator.propagation.dgsn_simulator.**DGSNSimulator**(*params*)
A class for the simulator.

__init__(*params*)
Initializes the simulator.

Parameters **params** – A SimParams object containing kep,t0,t,period,speed, op_writer, dgsn_period, and dgsn_thresh. For a description of the parameters, look at the documentation of the SimParams class.

Returns nothing

calc()
Calculates the satellite state at current time and calls itself after a certain amount of time.

simulate()
Starts the calculation thread and waits for keyboard input. Press q or Ctrl-C to quit the simulator cleanly.

stop()
Stops the simulator cleanly.

class orbitdeterminator.propagation.dgsn_simulator.**SimParams**
SimParams class. This is just a container for all the parameters required to start the simulation.

kep(1x6 numpy array): the intial osculating keplerian elements epoch(float): the epoch of the above kep period(float): maximum time period between observations t0(float): starting time of the simulation speed(float): speed of the simulation op_writer(OpWriter): output handling object

r_jit(float): std of Gaussian noise applied to observations dgsn_period(float): average time period between gaps dgsn_thresh(float): used to control the duration of the gap.

it is a number between 0 and 1. a higher number means a bigger gap.

class orbitdeterminator.propagation.dgsn_simulator.**OpWriter**
Base output writer class. Inherit this class and override the methods.

close()
Anything that has to be executed after finishing writing the output. Runs once.

Example: Closing connection to a database

open()
Anything that has to be executed before starting to write output. Runs once.

Example: Establishing connection to database

static write(*t*, *r*)
This method is called everytime the calc thread finishes a computation.

Parameters

- **t** – the current time of simulation
- **s** – the state vector at t [rx,ry,rz,vx,vy,vz]

class orbitdeterminator.propagation.dgsn_simulator.**print_r**
Bases: *orbitdeterminator.propagation.dgsn_simulator.OpWriter*

Prints the position vector

class orbitdeterminator.propagation.dgsn_simulator.**save_r**(*name*)
Bases: *orbitdeterminator.propagation.dgsn_simulator.OpWriter*

Saves the position vector to a file

__init__(*name*)
Initialize the class.

Parameters **name** (*string*) – file name

Kalman Filter

Kalman Filter to smoothen observations. It continuously reads a file where observations are being written and updates its estimate based on the observations and the cowell model.

class orbitdeterminator.propagation.kalman_filter.**KalmanFilter**
Kalman Filter class wrapper.

process(*s*, *t0*, *dgsn_file*)
The main Kalman Filter. Continuously reads an obervations file and updates the state estimate.

Parameters

- **s** (*1x6 numpy array*) – the state vector [rx,ry,rz,vx,vy,vz]
- **t0** (*float*) – epoch of s
- **dgsn_file** (*string*) – path to the observations file

Returns nothing

sgp4_prop

SGP4 propagator. This is a wrapper around the PyPI SGP4 propagator. However, this does not generate an artificial TLE. So there is no string manipulation involved. Hence this is faster than sgp4_prop_string.

orbitdeterminator.propagation.sgp4_prop.**kep_to_sat**(*kep*, *epoch*, *bstar=0.21109E-4*, *whichconst=wgs72*, *afspc_mode=False*)

Converts a set of keplerian elements into a Satellite object.

Args: kep(1x6 numpy array): the osculating keplerian elements at epoch epoch(float): the epoch bstar(float): bstar drag coefficient whichconst(float): gravity model. refer pypi sgp4 documentation afspc_mode(boolean): refer pypi sgp4 documentation

Returns an sgp4 satellite object encapsulating the arguments

Return type Satellite object

orbitdeterminator.propagation.sgp4_prop.**propagate_kep**(*kep*, *t0*, *tf*, *bstar=2.1109e-05*)

Propagates a set of keplerian elements.

Parameters

- **kep** (*1x6 numpy array*) – osculating keplerian elements at epoch
- **t0** (*float*) – initial time (epoch)
- **tf** (*float*) – final time

Returns the position at tf vel(1x3 numpy array): the velocity at tf

Return type pos(1x3 numpy array)

orbitdeterminator.propagation.sgp4_prop.**propagate_state**(*r*, *v*, *t0*, *tf*, *bstar=2.1109e-05*)

Propagates a state vector

Parameters

- **r** (*1x3 numpy array*) – the position vector at epoch
- **v** (*1x3 numpy array*) – the velocity vector at epoch
- **t0** (*float*) – initial time (epoch)
- **tf** (*float*) – final time

Returns the position at tf vel(1x3 numpy array): the velocity at tf

Return type pos(1x3 numpy array)

sgp4_prop_string

SGP4 propagator. This is a wrapper around PyPI SGP4 propagator. It constructs an artificial TLE and passes it to the PyPI module.

orbitdeterminator.propagation.sgp4_prop_string.**propagate**(*kep*, *init_time*, *final_time*, *bstar=2.1109e-05*)

Propagates a set of keplerian elements.

Parameters

- **kep** (*1x6 numpy array*) – osculating keplerian elements at epoch
- **init_time** (*float*) – initial time (epoch)
- **final_time** (*float*) – final time
- **bstar** (*float*) – bstar drag coefficient

Returns the position at tf vel(1x3 numpy array): the velocity at tf

Return type pos(1x3 numpy array)

3.1.4 Utils:

kep_state

Takes a set of keplerian elements (a, e, i, ω, Ω, v) and transforms it into a state vector (x, y, z, vx, vy, vz) where v is the velocity of the satellite

orbitdeterminator.util.kep_state.**kep_state**(*kep*)

Converts the keplerian elements to position and velocity vector

Parameters **kep** (*numpy array*) – a 1x6 matrix which contains the following variables kep(0): semi major axis (km) kep(1): eccentricity (number) kep(2): inclination (degrees) kep(3): argument of perigee (degrees) kep(4): right ascension of the ascending node (degrees) kep(5): true anomaly (degrees)

Returns 1x6 matrix which contains the position and velocity vector r(0),r(1),r(2): position vector (x,y,z) km r(3),r(4),r(5): velocity vector (vx,vy,vz) km/s

Return type numpy array

read_data

Reads the positional data set from a .csv file

orbitdeterminator.util.read_data.**load_data**(*filename*)

Loads the data in numpy array for further processing in tab delimiter format

Parameters **filename** (*string*) – name of the csv file to be parsed

Returns array of the orbit positions, each point of the orbit is of the format (time, x, y, z)

Return type numpy array

orbitdeterminator.util.read_data.**save_orbits**(*source*, *destination*)

Saves objects returned from load_data

Parameters

- **source** – path to raw csv files.

- **destination** – path where objects need to be saved.

state_kep

Takes a state vector (x, y, z, vx, vy, vz) where v is the velocity of the satellite and transforms it into a set of keplerian elements (a, e, i, ω, Ω, v)

orbitdeterminator.util.state_kep.**state_kep**(*r*, *v*)

Converts state vector to orbital elements.

Parameters

- **r** (*numpy array*) – position vector
- **v** (*numpy array*) – velocity vector

Returns

array of the computed keplerian elements kep(0): semimajor axis (kilometers) kep(1): orbital eccentricity (non-dimensional)

(0 <= eccentricity < 1)

kep(2): orbital inclination (degrees) kep(3): argument of perigee (degress) kep(4): right ascension of ascending node (degrees) kep(5): true anomaly (degrees)

Return type numpy array

input_transf

Converts cartesian co-ordinates to spherical co-ordinates and vice versa

orbitdeterminator.util.input_transf.**cart_to_spher**(*data*)

Takes as an input a data set containing points in cartesian format (time, x, y, z) and returns the computed spherical coordinates (time, azimuth, elevation, r)

Parameters **data** (*numpy array*) – containing the cartesian coordinates in format of (time, x, y, z)

Returns array of spherical coordinates in format of (time, azimuth, elevation, r)

Return type numpy array

orbitdeterminator.util.input_transf.**spher_to_cart**(*data*)

Takes as an input a data set containing points in spherical format (time, azimuth, elevation, r) and returns the computed cartesian coordinates (time, x, y, z).

Parameters **data** (*numpy array*) – containing the spherical coordinates in format of (time, azimuth, elevation, r)

Returns array of cartesian coordinates in format of (time, x, y, z)

Return type numpy array

rkf78

Uses Runge Kutta Fehlberg 7(8) numerical integration method to compute the state vector in a time interval tf

orbitdeterminator.util.rkf78.**rkf78**(*neq*, *ti*, *tf*, *h*, *tetol*, *x*)

Runge-Kutta-Fehlberg 7[8] method, solve first order system of differential equations

Parameters

- **neq** (*int*) – number of differential equations
- **ti** (*float*) – initial simulation time
- **tf** (*float*) – final simulation time
- **h** (*float*) – initial guess for integration step size
- **tetol** (*float*) – truncation error tolerance [non-dimensional]
- **x** (*numpy array*) – integration vector at time = ti

Returns array of state vector at time tf

Return type numpy array

orbitdeterminator.util.rkf78.**ypol_a**(*y*)

Computes velocity and acceleration values by using the state vector y and keplerian motion

Parameters **y** (*numpy array*) – state vector (position + velocity)

Returns derivative of the state vector (velocity + acceleration)

Return type numpy array

golay_window

orbitdeterminator.util.golay_window.**window**(*error*, *data*)

Calculates the constant c which is needed to determine the savintzky - golay filter window window = len(data) / c ,where c is a constant strongly related to the error contained in the data set

Parameters

- **error** (*float*) – the a-priori error estimation for each measurment
- **data** (*numpy array*) – the positional data set

Returns constant which describes the window that needs to be inputed to the savintzky - golay filter

Return type float

anom_conv

Vectorized anomaly conversion scripts

orbitdeterminator.util.anom_conv.**ecc_to_mean**(*E*, *e*)

Converts eccentric anomaly to mean anomaly.

Parameters

- **E** (*numpy array*) – array of eccentric anomalies (in radians)
- **e** (*float*) – eccentricity

Returns array of mean anomalies (in radians)

Return type numpy array

orbitdeterminator.util.anom_conv.**mean_to_t**(*M*, *a*)

Converts mean anomaly to time elapsed.

Parameters

- **M** (*numpy array*) – array of mean anomalies (in radians)

- **a** (*float*) – semi-major axis

Returns numpy array of time elapsed

Return type numpy array

orbitdeterminator.util.anom_conv.**true_to_ecc**(*theta*, *e*)

Converts true anomaly to eccentric anomaly.

Parameters

- **theta** (*numpy array*) – array of true anomalies (in radians)
- **e** (*float*) – eccentricity

Returns array of eccentric anomalies (in radians)

Return type numpy array

new_tle_kep_state

This module computes the state vector from keplerian elements.

orbitdeterminator.util.new_tle_kep_state.**kep_to_state**(*kep*)

This function converts from keplerian elements to the position and velocity vector

Parameters **kep** (*1x6 numpy array*) – kep contains the following variables kep[0] = semi-major axis (kms) kep[1] = eccentricity (number) kep[2] = inclination (degrees) kep[3] = argument of perigee (degrees) kep[4] = right ascension of ascending node (degrees) kep[5] = true anomaly (degrees)

Returns: r: 1x6 numpy array which contains the position and velocity vector

r[0],r[1],r[2] = position vector [rx,ry,rz] km r[3],r[4],r[5] = velocity vector [vx,vy,vz] km/s

orbitdeterminator.util.new_tle_kep_state.**tle_to_state**(*tle*)

This function converts from TLE elements to position and velocity vector

Parameters **tle** (*1x6 numpy array*) – tle contains the following variables tle[0] = inclination (degrees) tle[1] = right ascension of the ascending node (degrees) tle[2] = eccentricity (number) tle[3] = argument of perigee (degrees) tle[4] = mean anomaly (degrees) tle[5] = mean motion (revs per day)

Returns: r: 1x6 numpy array which contains the position and velocity vector

r[0],r[1],r[2] = position vector [rx,ry,rz] km r[3],r[4],r[5] = velocity vector [vx,vy,vz] km/s

teme_to_ecef

Converts coordinates in TEME frame to ECEF frame.

orbitdeterminator.util.teme_to_ecef.**conv_to_ecef**(*coords*)

Converts coordinates in TEME frame to ECEF frame.

Parameters **coords** (*nx4 numpy array*) – list of coordinates in the format [t,x,y,z]

Returns

list of coordinates in the format [t, latitude, longitude, altitude]

Note that these coordinates are with respect to the surface of the Earth. Latitude, longitude are in degrees.

Return type nx4 numpy array

3.2 Tutorials

3.2.1 * Run the program with main.py

For the first example we will showcase how you can use the full features of the package with main.py. Simply executing the main.py by giving the name of .csv file that contains the positional data of the satellite, as an argument in the function process(data_file):

```
def process(data_file, error_apriori):
    '''
    Given a .csv data file in the format of (time, x, y, z) applies both filters,␣
↪generates a filtered.csv data
    file, prints out the final keplerian elements computed from both Lamberts and␣
↪Interpolation and finally plots
    the initial, filtered data set and the final orbit.

    Args:
        data_file (string): The name of the .csv file containing the positional data
        error_apriori (float): apriori estimation of the measurements error in km

    Returns:
        Runs the whole process of the program
    '''
```

Simply input the name of the .csv file in the format of (time, x, y, z) and **tab delimiter** like the orbit.csv that is located in the src folder and the process will run. You also need to input a apriori estimation of the measurements errors, which in the example case is 20km per point (points every 1 second). In the case you are using your own positional data set you need to estimate this value and input it because it is critical for the filtering process:

```
run = process("orbit.csv")
```

Warning: If the format of you data is (time, azimuth, elevation, distance) you can use the input_transf function first and be sure that the delimiter for the data file is tab delimiter since this is the one read_data supports.

The process that will run with the use of the process function is, first the program reads your data from the .csv file then, applies both filters (Triple moving average and Savintzky - Golay), generates a .csv file called filtered, that included the filtered data set, computes the keplerian elements of the orbit with both methods (Lamberts - Kalman and Spline Interpolation) and finally prints and plots some results. More specifically, the results printed by this process will be first the sum and mean value of the residuals (difference between filtered and initial data), the computed keplerian elements in format of (a - semi major axis, e - eccentricity, i - inclination, ω - argument of perigee, Ω - right ascension of the ascending node, v - true anomaly) and a 3d matplotlib graph that plots the initial, filtered data set and the final computed orbit described by the keplerian elements (via the interpolation method).

Process

- Reads the data
- Uses both filters on them (Triple moving average and Savintzky - Golay)
- Generates a .csv file called filtered that includes the filtered data set
- Computes keplerian elements with both methods (Lamberts - Kalman and Spline Interpolation)
- Prints results and plot a 3d matplotlib graph

Results

- Sum and mean of the residuals (differences between filtered and initial data set)
- Final keplerian elements from both methods (first column : Lamberts - Kalman, second column : Spline Interpolation)
- 3d matplotlib graph with the initial, filtered data set and the final orbit described by the keplerian elements from Spline Interpolation

Warning: Measurement unit for distance is kilometer and for angle degrees

The output should look like the following image.

3.2.2 * Run the program with automated.py

automated.py is another flavour of main.py that is supposed to run on a server. It keeps listening for new files in a particular directory and processes them when they arrive.

Note: All the processing invloved in this module is identical to that of main.py.

For testing purpose some files have already put in a folder named src. These are raw unprocessed files. There is another folder named dst which contains processed files along with a graph saved in the form of svg.

To execute this script, change the directory to the script's directory:

```
cd orbitdeterminator/
```

and run the code using python3:

```
python3 automated.py
```

and thats it. This will keep listening for new files and process them as they arrive.

Process

- Initialize an empty git repository in src folder
- Read the untracked files of that folder and put them in a list
- Process the files in this list and save the results(processed data and graph) to dst folder
- Stage the processed file in the src folder in order to avoid processing the same files multiple times.
- Check for any untracked files in src and apply steps 2-4 again.

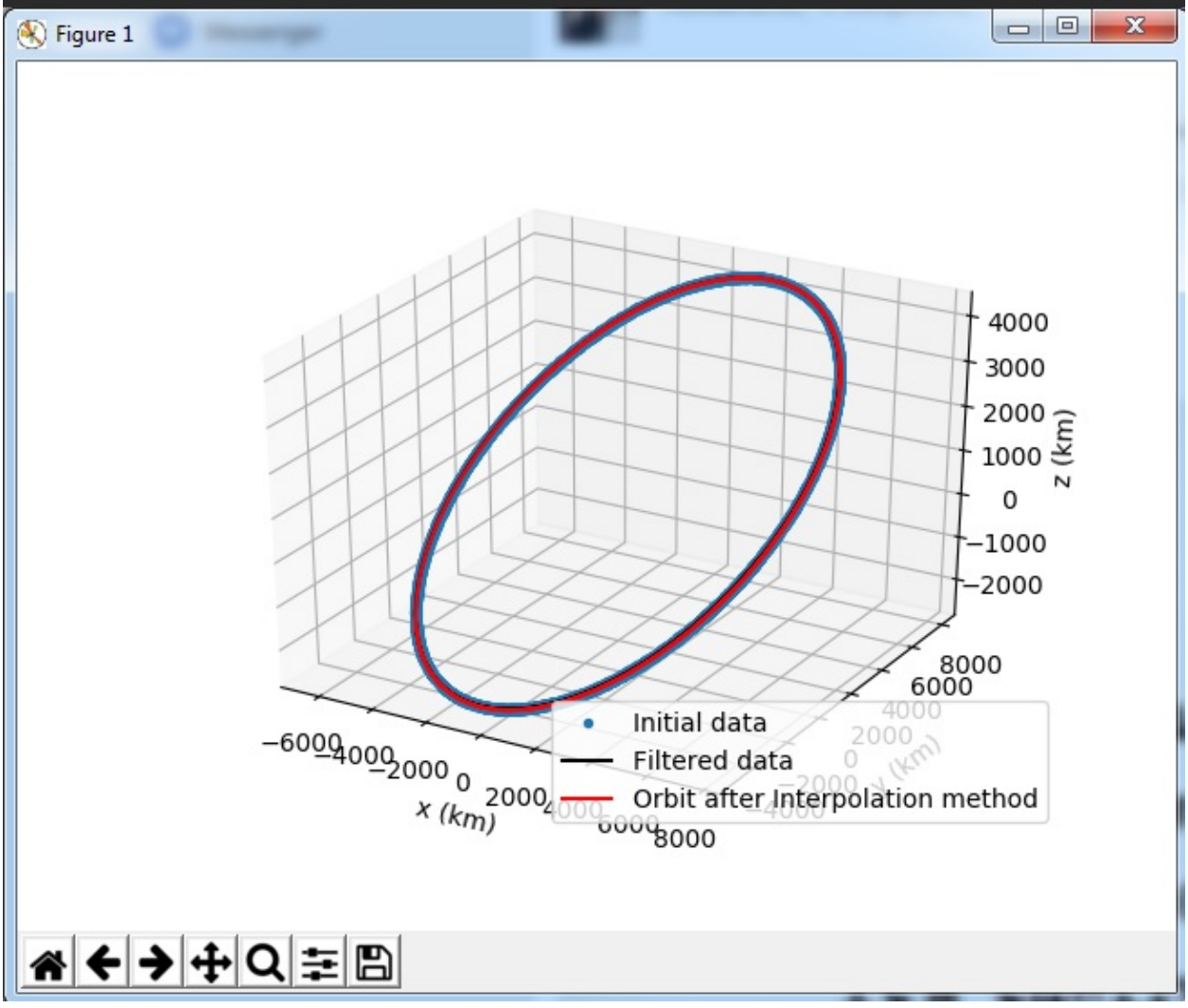
3.2.3 * Using certain modules

In this example we are not going to use the main.py, but some of the main modules provided. First of all lets clear the path we are going to follow which is fairly straightforward. Note that we are going to use the same orbit.csv that is located inside the src folder and has **tab delimeter** (read_data.py reads with this delimiter).

```
Displaying the sum of the residuals for each axis
[-411.9025779943013958  806.6275082264301091  312.802706832659112 ]

Displaying the mean of the residuals for each axis
[-0.0514942590316666  0.1008410436587611  0.0391052265073958]

Displaying the final keplerian elements, first row : Lamberts, second row : Interpolation
[[  7.2742037178552282e+03   7.2792955045547997e+03]
 [  2.3911078878650033e-01   2.3584322875812513e-01]
 [  2.9039053923422454e+01   2.9006753606193687e+01]
 [  2.7304613252432324e+02   2.7139493062060802e+02]
 [  3.3615785054162984e+02   3.3622985871131232e+02]
 [  3.1464426978147503e+02   3.1464426978147503e+02]]
```



```
Reinitialized existing Git repository in /home/nilesh/Documents/DGSN/orbitdeterm
inator/orbitdeterminator/src/.git/
processing
Displaying the sum of the residuals for each axis
[-7.3071114803598363  6.0195769922622517 -1.0865077005567656]

Displaying the mean of the residuals for each axis
[-0.0009135031229353  0.0007525411916817 -0.0001358304413748]

Displaying the final keplerian elements, first row : Lamberts, second row : Inte
rpolation
[[  7.4118904264173998e+03   7.4747408615121849e+03]
 [  2.4579100618594910e-01   2.5941412686356824e-01]
 [  2.8979148921840331e+01   2.9053134052803713e+01]
 [  2.7567928783143986e+02   2.7454481878050581e+02]
 [  3.3607209318385082e+02   3.3603901617390579e+02]
 [  3.1168910300232398e+02   3.1168910300232398e+02]]
File : orbit.csv has been processed


processing
Displaying the sum of the residuals for each axis
[ 4.1578835739922795 -2.0596593198753368  1.7371328427677071]

Displaying the mean of the residuals for each axis
[ 0.0005198004218018 -0.0002574896011846  0.0002171687514399]

Displaying the final keplerian elements, first row : Lamberts, second row : Inte
rpolation
[[  6.8981810916685954e+03   6.7844780735262648e+03]
 [  1.2036829917218617e-01   1.3620260613248805e-01]
 [  1.1200403901735038e+02   1.1194942033539589e+02]
 [  2.5583136978548816e+02   2.6708814175066198e+02]
 [  1.7727806864823512e+02   1.7696090462366834e+02]
 [  6.3323142006727799e+01   6.3323142006727799e+01]]
File : orbit1.csv has been processed


processing
```

Process

- Read the data
- Filter the data
- Compute keplerian elements for the final orbit

So first we read the data using the util/read_data.load_data function. Just input the .csv file name into the function and it will create a numpy array with the positional data ready to be processed:

```
data = read_data.load_data("orbit.csv")
```

Warning: If the format of you data is (time, azimuth, elevation, distance) you can use the util/input_transf.spher_to_cart function first. And it is critical for the x, y, z to be in kilometers.

We continue by applying the Triple moving average filter:

```
data_after_filter = triple_moving_average.generate_filtered_data(data, 3)
```

We suggest using 3 as the window size of the filter. Came to this conclusion after a lot of testing. Next we apply the second filter to the data set which will be of a larger window size so that we can smooth the data set in a larger scale. The optimal window size for the Savintzky - Golay filter is being computed by the function golay_window.c(error_apriori) in which we only have to input the apriori error estimation for the initial data set (or the measurements error):

```
error_apriori = 20.0
c = golay_window.c(error_apriori)

window = len(data) / c
window = int(window)
```

The other 2 lines after the use of the golay_window.c(error_apriori) are needed to compute the window size for the Savintzky - Golay filter and again for the polynomial parameter of the filter we suggest using 3:

```
data_after_filter = sav_golay.golay(data_after_filter, window, 3)
```

At this point we have the filtered positional data set ready to be inputed into the Lamberts - Kalman and Spline interpolation algorithms so that the final keplerian elements can be computed:

```
kep_lamb = lamberts_kalman.create_kep(data_after_filter)
kep_final_lamb = lamberts_kalman.kalman(kep_lamb, 0.01 ** 2)
kep_inter = interpolation.main(data_after_filter)
kep_final_inter = lamberts_kalman.kalman(kep_inter, 0.01 ** 2)
```

With the above 4 lines of code the final set of 6 keplerian elements is computed by the two methods. The output format is (semi major axis (a), eccentricity (e), inclination (i), argument of perigee (ω), right ascension of the ascending node (Ω), true anomaly (v)). So finally, in the variables kep_final_lamb and kep_final_inter a numpy array 1x6 has the final computed keplerian elements.

Warning: If the orbit you want to compute is polar (i = 90) then we suggest you to use only the interpolation method.

3.2.4 Using ellipse_fit method

If a lot of points are available spread over the entire orbit, then the ellipse fit method can be used for orbit determination. The module `kep_determination.ellipse_fit` has two methods - `determine_kep` and `plot_kep`. As the name suggests, `determine_kep` is used to determine the orbit and `plot_kep` is used to plot it. Call `determine_kep` with:

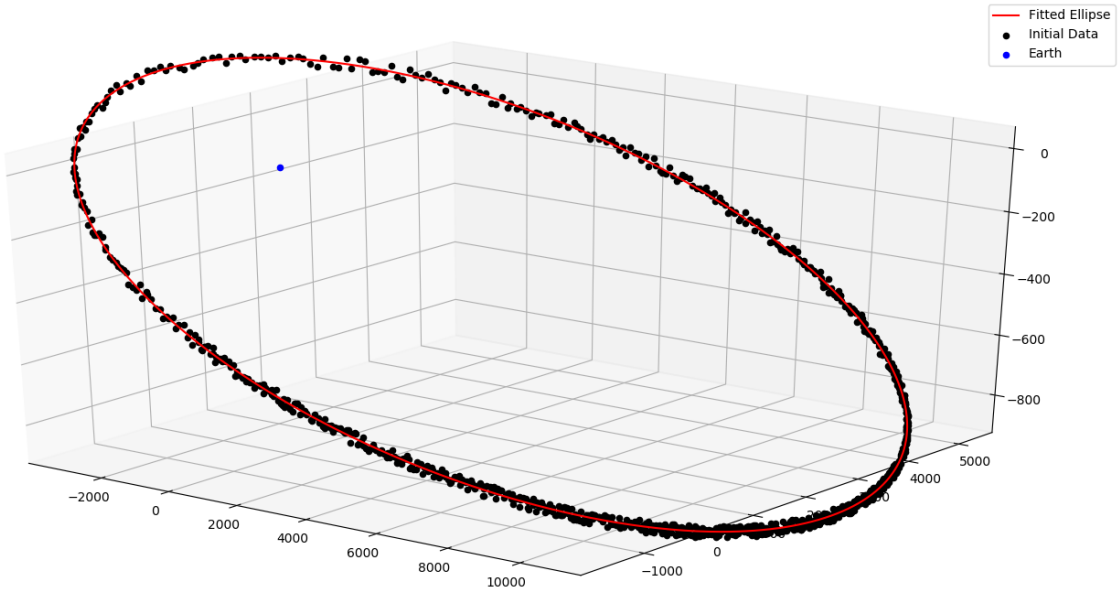
```
kep,res = determine_kep(data)
```

where *data* is a nx3 numpy array. The ellipse_fit method does not use time information at all. Hence, the input format is *[(x,y,z),...]*. The method results two arguments - the first output is the Keplerian elements while the second output is the list of residuals.

Plot the results using the `plot_kep` method. Call it with:

```
plot_kep(kep,data)
```

where *kep* is the Keplerian elements we got in the last step and data is the original data. The result should look like this.



3.2.5 Using propagation modules

Cowell Method

The module `propagation.cowell` propagates a satellite along its orbit using numerical integration. It takes into account the oblateness of the Earth and atmospheric drag. The module has many methods for calculating drag and J2 acceleration, and integrating them. However, here we will discuss only the important ones. One is `propagate_state` and the other is `time_period`. `propagate_state` propagates a state vector from t1 to t2. `time_period` finds out the nodal time period of an orbit, given a state vector. Call `propagate_state` like this.:

```
sf = propagate_state(si,t0,tf)
```

where si is the state at t0 and sf is the state at tf.

Note: In all propagation related discussions a state vector is the numpy array *[rx,ry,rz,vx,vy,vz]*.

Similarly to find out time period call `time_period` like this.:

```
t = time_period(s)
```

DGSN Simulator

The module `propagation.dgsn_simulator` can be used for simulating the DGSN. Given a satellite, it propagates the satellite along its orbit and periodically outputs its location. The location will have some associated with it. Observations will also not be exactly periodic. There will be slight variations. And sometimes observations might not be available (for example, the satellite is out of range of the DGSN).

To use this simulator, 3 classes are used.

- The SimParams class - This is a collection of all the simulation parameters.
- The OpWriter class - This class tells the simulator what to do with the output.
- The DGSNSimulator class - This is the actual simulator class.

To start, we must choose an OpWriter class. This will tell the simulator what to do with the output. To use it, extend the class and override its `write` method. Several sample classes have been provided. For this example we will use the default `print_r` class. This just prints the output.

Now create a SimParams object. For now, only set the kep, epoch and t0.:

```
epoch = 1531152114
t0 = epoch
iss_kep = np.array([6785.6420,0.0003456,51.6418,290.0933,266.6543,212.4306])

params = SimParams()
params.kep = iss_kep
params.epoch = epoch
params.t0 = t0
```

Now initialize the simulator with these parameters and start it.:

```
s = DGSNSimulator(params)
s.simulate()
```

The program should start printing the time and the corresponding satellite coordinates on the terminal.

Note: The module `propagation.simulator` is similar to this module. The only difference is that it doesn't add any noise. So it can be used for comparison purposes.

Kalman Filter

The module `propagation.kalman_filter` can be used to combine observation data and simulation data with a Kalman Filter. This module keeps on checking a file for new observation data and applies the filter accordingly. We can use the DGSN Simulator module to create observation data in real time. First, we must setup the simulator. We must configure it to save the output to a file instead of printing it. For this, we will use the in-built `save_r` class.

Run the simulator with the following commands.:

```
epoch = 1531152114
t0 = epoch
iss_kep = np.array([6785.6420,0.0003456,51.6418,290.0933,266.6543,212.4306])

params = SimParams()
params.kep = iss_kep
params.epoch = epoch
params.t0 = t0
params.r_jit = 15
params.op_writer = save_r('ISS_DGSN.csv')

s = DGSNSimulator(params)
s.simulate()
```

Now the program will start writing observations into the file `ISS_DGSN.csv`. Now we need to setup the Kalman Filter with the same parameters. Use `util.new_tle_kep_state` to convert Keplerian elements into a state vector. In this tutorial, it is already done. Run the filter by passing the state and the name of the file to read.:

```
s = np.array([2.87327861e+03,5.22872234e+03,3.23884457e+03,-3.49536799e+00,4.
↪87267295e+00,-4.76846910e+00])
t0 = 1531152114
KalmanFilter().process(s,t0,'ISS_DGSN.csv')
```

The program should start printing filtered values on the terminal.

3.2.6 Using utility modules

new_tle_kep_state

`new_tle_kep_state` is used to convert a TLE or a set of Keplerian elements into a state vector. To convert a TLE make an array out of the 2nd line of the TLE. The array should be of the form:

- tle[0] = inclination (in degrees)
- tle[1] = right ascension of ascending node (in degrees)
- tle[2] = eccentricity
- tle[3] = argument of perigee (in degrees)
- tle[4] = mean anomaly (in degrees)
- tle[5] = mean motion (in revs per day)

Now call `tle_to_state`. For example:

```
tle = np.array([51.6418, 266.6543, 0.0003456, 290.0933, 212.4518, 15.54021918])
r = tle_to_state(tle)
print(r)
```

Similarly a Keplerian set can also be converted into a state vector.

teme_to_ecef

`teme_to_ecef` is used to convert coordinates from TEME frame (inertial frame) to ECEF frame (rotating Earth fixed frame). The module accepts a list of coordinates of the form *[t1,x,y,z]* and outputs a list of latitudes, longitudes

and altitudes in Earth fixed frame. These coordinates can be directly plotted on a map.

For example:

```
ecef_coords = conv_to_ecef(np.array([[1521562500,768.281,5835.68,2438.076],
                                     [1521562500,768.281,5835.68,2438.076],
                                     [1521562500,768.281,5835.68,2438.076]]))
```

The resulting latitudes and longitudes can be directly plotted on an Earth map to visualize the satellite location with respect to the Earth.

3.3 Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

Symbols

C

D

E

F

G

R

S

T

U

W

Y