Orbit Determinator Documentation

Release 1.0.0

Nilesh Chaturvedi, Alexandros Kazantzidis

Aug 18, 2017

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).

Modules documentation

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

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

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

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): right ascension of ascending node (degrees) kep(4): argument of perigee (degress) 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

Tutorials

* 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.

* 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.

* 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).

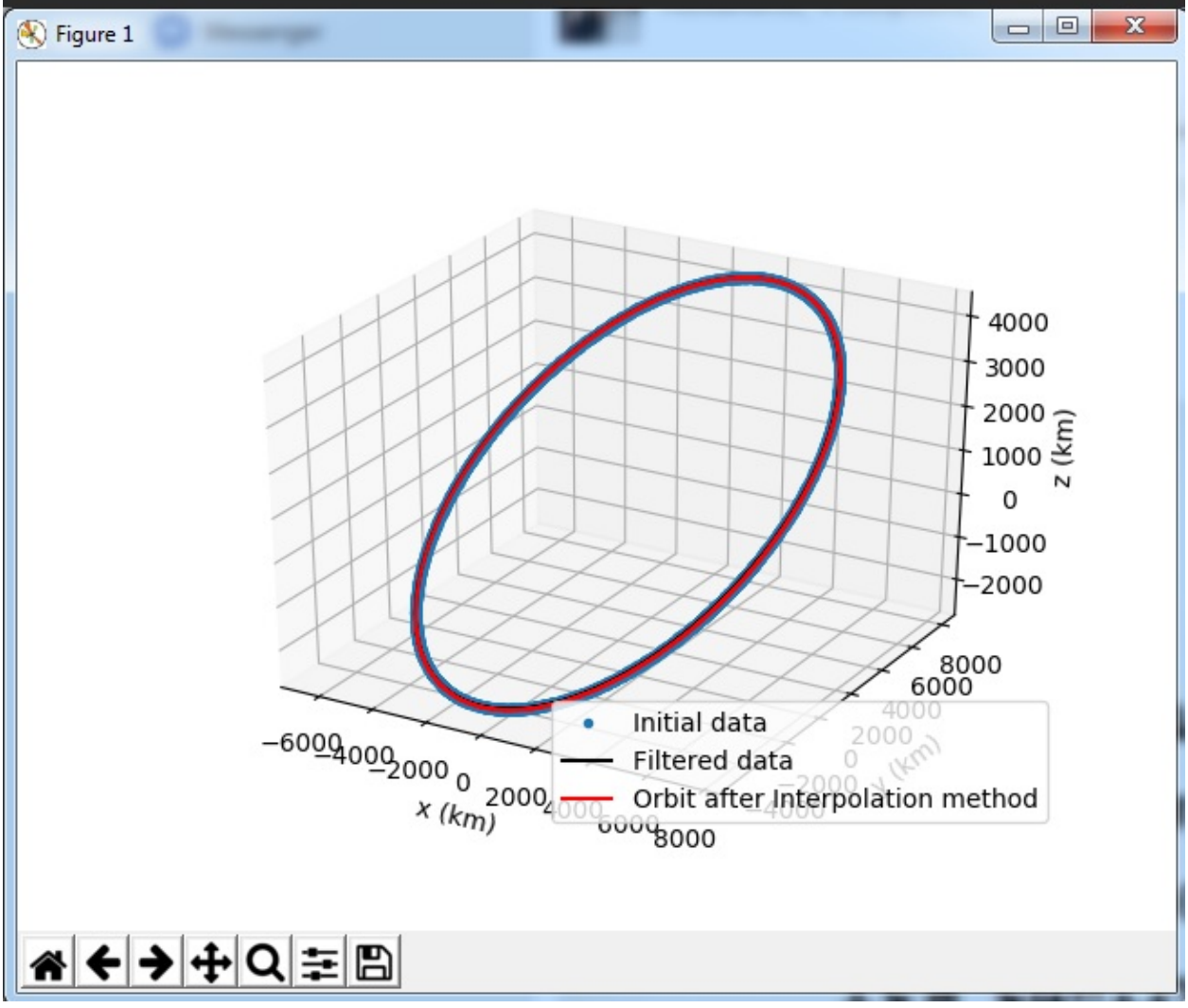
Process

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

```
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
```

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.

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index