Louis Grunfelder, Student at ISAE-SUPAERO

Helioswarm Configuration Plotting User Guide

Introduction

Greetings !

The central focus of this work is to develop basic visualization tools for the Helioswarm mission. The Helioswarm mission aims at understanding the plasmas and magnetic fields around the Earth and in the solar environments. It will help collecting data useful to better understand space weather around Earth; such a knowledge will in turn give information to help protect astronauts, and space technologies from extreme space weather events. Although these near-Earth space environments have already been studied , our current knowledge comes from the space missions where each of them comprises a single satellite is launched with several instruments. HelioSwarm, in contrast, will offer unique, new perspectives thanks to its nine spacecraft that will co-fly while performing simultaneous measurements in the Sun-Earh environments. The main objective of HelioSwarm is to better investigate plasma turbulence that s ubiquitous in the Universe and is key to understand particle acceleration and energy dissipation in astrophysics.

The multi-spacecraft missions Cluster and MMS (Magnetospheric MultiScale) advanced our understanding of the interactions between Earth's magnetic field and the solar wind thanks to their four spacecraft that co-fly in space. The methods developed for analyzing data from these two missions are a good starting point for a future analysis with nine spacecraft; we will focus on tools used for four-spacecraft constellations in this work. The NASA's HelioSwarm mission is planned to be launched in 2029 with trajectories in the vicinity of the Earth along the Lunar-resonance orbit.

Table of contents

Description and purpose of the tools

The 9 satellites of Helioswarm comprise one HUB (also called mothe spacecraft) and 8 satellites (also called daughter spacecraft). The 8 daughter spacecraft revolve around the HUB. . Among the overall nine spacecraft, we can select subsets of them by choosing certain spacecraft pairs (selecting 2 spacecraft) or spacecraft tetrahedra (selecting 4 spacecraft). To facilitate the four spacecraft forming tetrahedra later, we had the idea to develop a visualization tool in two referent frames:

- In the GSE (Geocentric Solar Ecliptic) frame, the tool would visualize the trajectory of Helioswarm around the Earth. This tool allow the user to locate the satellites in space to have a global context including the distance from the Earth and respective position regarding the Sun-Earth line.

- In the HUB frame, using the HUB as center, the tool will visualize and plot the tetrahedra that best meet certain criteria given the configuration of Helioswarm at a certain time. Two certain criteria are choosen here: (1) the most regular tetrahedra because analyses using regular tetrahedra are most reliable (i.e., they would provide minimal uncertainties) and (2) the largest scales that HelioSwarm satellites can be used for four-spacecraft analyses.

User Guide of the tool in the GSE frame

As described earlier, this frame the tool is to be used to describe the trajectory of Helioswarm in the GSE referent frame.

```
import plotly.graph_objects as go
import numpy as np
import re
```

All the following code was successfully run on Python 3.10

First, we import the Python modules necessary to run the main code. All the plots we made with the plotly modules because these are more interactive , compared to other libraries such as matplotlib, providing what we need

In order to plot the positions of Helioswarm the first step is to extract the trajectory data from the .txt file downloaded from Amda. Below is an example of a slice of the HUB position during the first 14 hours of the mission.

The following code extracts the positions of the HUB in the .txt file from Amda based on the pattern with which they are written in the file (that is why the “re” module was imported). The file used here is a slice of the global file “n0”. The slice “s0” was obtained, up to 3 weeks of the data, in order to have one complete revolution around the Earth. The global file “n0” contains one year of data.

```
def satellites_coordinates(file_name):
    """
    Parameters
    ----------
    file_name : string
        Name of the file containing the positions of the wanted satellite.
    Returns
    -------
    array
        The array containing the values of x, y and z.
    """
    with open(file_name, "r") as file:
        lines = file.readlines()

    extracted_lines = lines[6:]  # Ignore the lines that don't have values

    list_values = []

    for line in extracted_lines:
        line = line.strip()  # Remove blank spaces and other symbols of the new line

        values = re.findall(r"[-+]?\d+\.\d+", line)

        # Extraction of the x, y, z values
        x = float(values[1])
        y = float(values[2])
        z = float(values[3])

        list_values.append([x, y, z])  # Add the x, y, z coordinates at the list of values

    # Create a 3 rows table with the values of x , y and z
    final_extracted_lines = np.array(list_values)

    return final_extracted_lines
```

The function is used as such :

```
HUB = coordonnées_satellites("./Data/s0.txt")
```

Here the user only needs to call the function on one satellite because the aim is to plot the position of Helioswarm that will be represented by one point, the HUB. Indeed, one could think of plotting all the position of the 9 satellites in the GSE frame but it would not provide any new insights because the scales of distances between the satellites are much smaller than the Earth-Helioswarm distance.

|
Satellite-Chief

7 Feb 2022 13:58:22

Time (UTCG)	x (km)	y (km)	z (km)	Magnitude (km)	True_Anomaly (deg)	Total_Mass_Flow_Rate (g/hr)
7 Sep 2026 11:40:00.000	62212.586000	-47800.507483	15792.058423	80029.266407	0.447	0.000000000
7 Sep 2026 12:39:59.992	67849.933004	-39325.301279	17399.422275	80329.525237	7.856	0.000000000
7 Sep 2026 13:39:59.992	72825.399715	-30466.530750	18833.823920	81157.016105	15.161	0.000000000
7 Sep 2026 14:39:59.992	77111.826493	-21320.153344	20086.653216	82487.916455	22.275	0.000000000
7 Sep 2026 15:39:59.992	80703.406477	-11982.878883	21155.000429	84286.198434	29.124	0.000000000
7 Sep 2026 16:39:59.992	83613.478128	-2546.203770	22041.147545	86507.277517	35.655	0.000000000
7 Sep 2026 17:39:59.992	85870.767979	6907.972430	22751.624579	89101.881559	41.831	0.000000000
7 Sep 2026 18:39:59.992	87514.989706	16309.854832	23296.068658	92019.517510	47.637	0.000000000
7 Sep 2026 19:39:59.992	88592.579690	25602.560774	23686.096349	95211.172950	53.070	0.000000000
7 Sep 2026 20:39:59.992	89153.065636	34741.654926	23934.327500	98631.149908	58.138	0.000000000
7 Sep 2026 21:39:59.992	89246.277225	43693.896039	24053.622941	102238.110928	62.857	0.000000000
7 Sep 2026 22:39:59.992	88920.396922	52435.610602	24056.541196	105995.506613	67.248	0.000000000
7 Sep 2026 23:39:59.992	88220.731818	60950.990621	23954.986548	109871.571212	71.333	0.000000000
8 Sep 2026 00:39:59.992	87189.044852	69230.496826	23760.007811	113839.049558	75.137	0.000000000
8 Sep 2026 01:39:59.992	85863.287132	77269.458073	23481.706875	117874.780109	78.682	0.000000000

Once the user has the list of the needed positions, these are plotted using the following function.

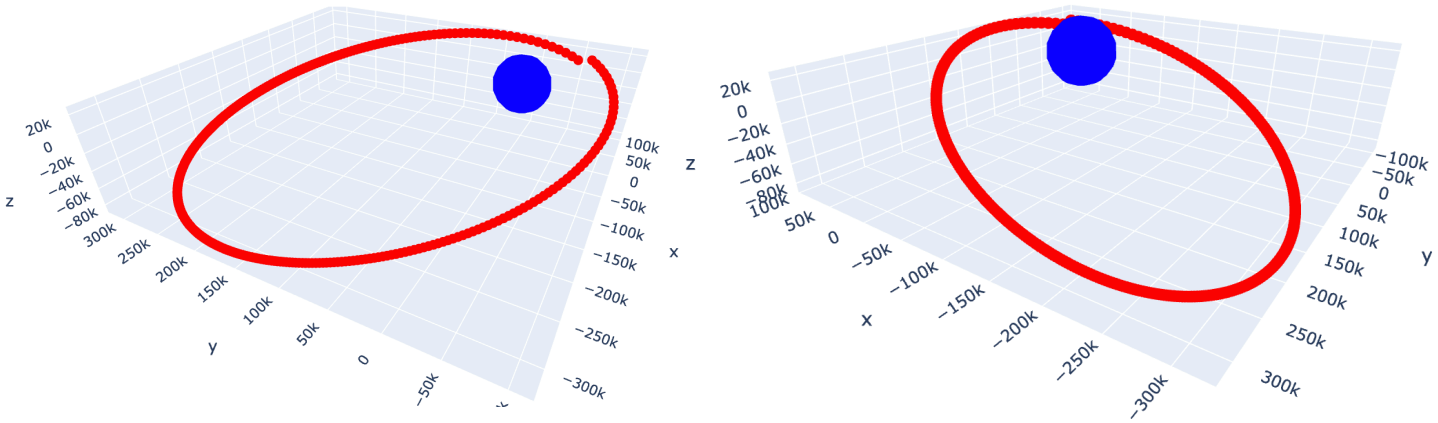
```python
def display_satellites_positions(positions):
    fig = go.Figure()

    x = np.append(positions[:, 0],0)
    y = np.append(positions[:, 1],0)
    z = np.append(positions[:, 2],0)

    labels = ['HEALIOSWARM'] * len(positions[:,0]) + ['EARTH']
    text = [f"{label} (x: {x[i]}, y: {y[i]}, z: {z[i]})" for i, label in enumerate(labels)]

    for i, label in enumerate(labels):
        if label == 'EARTH':
            marker_symbol = 'circle'
            marker_color = 'blue'
            marker_size = 30
        else:
            marker_symbol = 'circle'  # Use of a red circle for the satellite points
            marker_color = 'red'
            marker_size = 5

        fig.add_trace(go.Scatter3d(x=[x[i]],y=[y[i]],z=[z[i]],
            mode='markers',
            marker=dict(symbol=marker_symbol,color=marker_color,size=marker_size,),name=label,text=[text[i]],
            hoverinfo='text'))
    fig.show()
```

The given function is used as such :

```python
display_satellites_positions(HUB)
```

The result of this set of function in the GSE frame is the following :

Here are two of the many points of view the user can get from the interactive plot. One can notice the elliptic trajectory Helioswarm revolves on. If the user wants to see all the trajectories on which Helioswarm will revolve on he can either slice the data in sets of 3 weeks to see them one by one or make the plot from the global n0.txt file to see them all at once on one plot.



User Guide of the tool in the HUB frame

From now on the user will work in the HUB frame where the goal is to plot the configuration of Helioswarm at a certain time with the two most interesting tetrahedra will be drawn from all possible configurations (126 tetrahedra). The choice of interest for the criteria of the tetrahedra is given to the user from two possible choices, either the regularity of the tetrahedron or the size of it.

First, we import the necessary modules that will be used in the main code. All the plots we made with the plotly modules because they are more interactive while keeping important spatial and geometric information.

```
import plotly.graph_objects as go
import numpy as np
import math as m
import re
```

In order to plot the positions of Helioswarm the first step is to extract the given position from the .txt file containing it that was download from Amda. Below is an example of a slice of the HUB position during the first 14 hours of the mission.

```
|                                                                                                   7 Feb 2022 13:58:22
Satellite-Chief
```

Time (UTCG)	x (km)	y (km)	z (km)	Magnitude (km)	True_Anomaly (deg)	Total_Mass_Flow_Rate (g/hr)
7 Sep 2026 11:40:00.000	62212.586000	-47800.507483	15792.058423	80029.266407	0.447	0.000000000
7 Sep 2026 12:39:59.992	67849.933004	-39325.301279	17399.422275	80329.525237	7.856	0.000000000
7 Sep 2026 13:39:59.992	72825.399715	-30466.530750	18833.823920	81157.016105	15.161	0.000000000
7 Sep 2026 14:39:59.992	77111.826493	-21320.153344	20086.653216	82487.916455	22.275	0.000000000
7 Sep 2026 15:39:59.992	80703.406477	-11982.878883	21155.000429	84286.198434	29.124	0.000000000
7 Sep 2026 16:39:59.992	83613.478128	-2546.203770	22041.147545	86507.277517	35.655	0.000000000
7 Sep 2026 17:39:59.992	85870.767979	6907.972430	22751.624579	89101.881559	41.831	0.000000000
7 Sep 2026 18:39:59.992	87514.989706	16309.854832	23296.068658	92019.517510	47.637	0.000000000
7 Sep 2026 19:39:59.992	88592.579690	25602.560774	23686.096349	95211.172950	53.070	0.000000000
7 Sep 2026 20:39:59.992	89153.065636	34741.654926	23934.327500	98631.149908	58.138	0.000000000
7 Sep 2026 21:39:59.992	89246.277225	43693.896039	24053.622941	102238.110928	62.857	0.000000000
7 Sep 2026 22:39:59.992	88920.396922	52435.610602	24056.541196	105995.506613	67.248	0.000000000
7 Sep 2026 23:39:59.992	88220.731818	60950.990621	23954.986548	109871.571212	71.333	0.000000000
8 Sep 2026 00:39:59.992	87189.044852	69230.496826	23760.007811	113839.049558	75.137	0.000000000
8 Sep 2026 01:39:59.992	85863.287132	77269.458073	23481.706875	117874.780109	78.682	0.000000000

The following function extracts the desired values based on the pattern with which they are written in the file (that is why the "re" module was imported). The file that was used here is a slice of the global file. The slice "s0" was sliced (to 3 weeks of data) in order to have one revolution around the earth. The global file "n0" contains one year of data.

```
def satellites_coordinates(file_name):
    """
    Parameters
    ----------
    file_name : string
        Name of the file containing the positions of the wanted satellite.
    Returns
    -------
    array
        The array containing the values of x, y and z.
    """
    with open(file_name, "r") as file:
        lines = file.readlines()

    extracted_lines = lines[6:]  # Ignore the lines that don't have values

    list_values = []

    for line in extracted_lines:
        line = line.strip()  # Remove blank spaces and other symbols of the new line

        values = re.findall(r"[-+]?\d+\.\d+", line)

        # Extraction of the x, y, z values
        x = float(values[1])
        y = float(values[2])
        z = float(values[3])

        list_values.append([x, y, z])  # Add the x, y, z coordinates at the list of values

    # Create a 3 rows table with the values of x , y and z
    final_extracted_lines = np.array(list_values)

    return final_extracted_lines
```

Until now the data retrieval is the same as in the tool for the GSE frame. Now, in the HUB frame, another function will be added to extract the value of Time(UTGC) that will be used afterwards.

```
def retrieve_time(file_name):
    with open(file_name, "r") as file:
        lines = file.readlines()

    extracted_lines = lines[6:]  # Ignore the lines that don't have values

    list_values = []

    for line in extracted_lines:
        line = line.strip()  # Remove blank spaces and other symbols of the new line
        value = re.findall(r"\d{1,2} [A-Za-z]{3} \d{4} \d{2}:\d{2}:\d{2}.\d{3}", line)
        list_values.append(value)

    return list_values
```

Next, we implement a series of functions that will be used to calculate the properties of the 126 tetrahedra, which can be calculated from 9 satellites.

First, the "combinations" function calculates all the possible combinations of n elements among all the elements in the list L. In the end we will compute the combinations of 4 points in a set of 9 to have all the possible tetrahedra.

```
def combinations(L, n):
    """
    Parameters
    ----------
    L : list
        List of the elements we are working with, e.g., [0, 1, 2, 3, ..., 8]
    n : int
        Number of elements in a combination.

    Returns
    -------
    list
        The list of the combinations of n elements of L.
    """
    if n == 0:
        return [[]]
    else:
        comb = []
        for i in range(len(L)):
            element = L[i]
            rest = L[i+1:]
            for c in combinations(rest, n-1):
                comb.append([element]+c)
        return comb
```

Then we implement a function that can compute the barycenter of a tetrahedron defined by the indexes of the satellites (from 0 to 8) a time t.

```
def barycenter_t(list_sat,list_index,time_index):
    """
    Parameters
    ----------
    list_sat : list
        List of the position of each satellite, over time.
    list_index : list
        List of the indexes of the studied satellites, e.g., [0, 1, 2, 3, ..., 8]
    time_index : int
        Time.

    Returns
    -------
    bar_t : Array of size (3,)
        Barycenter of the satellites at time t.

    """
    bar_t = np.zeros(3, dtype=float)
    for i in list_index:
        bar_t += list_sat[i][time_index][...]
    bar_t /= len(list_index)
    return bar_t
```

Then, thanks to mathematical formulation, we can compute the volumetric tensor of a given tetrahedron defined by the indexes of the satellites (also called vertices). (cf the ISSI Scientific Report : Analysis Methods for Multi-Spacecraft Data by Götz Paschmann and Patrick W.Daly). Here is the mathematical formula to compute the volumetric tensor given the satellite positions 'r'.

$$R_{jk} = \frac{1}{N}\sum_{\alpha=1}^{N}\left(r_{\alpha j} - r_{bj}\right)\left(r_{\alpha k} - r_{bk}\right) = \frac{1}{N}\sum_{\alpha=1}^{N} r_{\alpha j} r_{\alpha k} - r_{bj} r_{bk} \qquad (13.5)$$

Where N is the number of satellites (4), \alpha labels the satellite (vertex) number, j is the cartesian component (x,y,z) of vertex α, and b is the barycenter.

Then the formula is transcribed into programming code below :

```
def volumetricTensor_t(list_sat,list_index,time_index):
    """
    Computes the coefficients of the volumetric tensor, at time t.

    Parameters
    ----------
    list_sat : list
        List of the positions of each satellite, over time.
    list_index : list
        List of the indexes of the studied satellites.
    time_index : int
        Time.

    Returns
    -------
    vt_t : Array of size (3,3)
        Volumetric tensor, at time time_index.

    """
    vt_t = np.zeros((3,3))
    bar_t = barycenter_t(list_sat,list_index,time_index)
    for j in range(3):
        for k in range(j,3):
            r_jk = 0
            for i in list_index:
                r_jk += (list_sat[i][time_index][j] - bar_t[j])*(list_sat[i][time_index][k] - bar_t[k])
            r_jk /= len(list_index)
            vt_t[j,k] = r_jk
            if (j != k):
                vt_t[k,j] = r_jk
    return vt_t
```

In the final part of these calculation on tetrahedra, the user will compute the 3 parameters of a tetrahedron : elongation, planarity and characteristic length. These parameters will be used to characterize the best tetrahedra following our criteria.

```
def polyhedronProperties_t(list_sat,list_index,time_index):
    """
    Computes L,E and P, to caracterize the tetrahedron whose vertices are the satellites.

    Parameters
    ----------
    list_sat : list
        List of the positions of each satellite, over time.
    list_index : list
        List of the indexes of the studied satellites.
    time_index : int
        Time.

    Returns
    -------
    L : float
        Caracteristic size of the tetrahedron.
    E : float
        Elongation of the tetrahedron.
    P : float
        Planarity of the tetrahedron.

    """
    if len(list_index) ==4:
        vt_t = volumetricTensor_t(list_sat,list_index,time_index)
        w, v = np.linalg.eig(vt_t)
        w.sort()
        a = np.sqrt(w[2])
        b = np.sqrt(w[1])
        c = np.sqrt(w[0])
        L = 2*a
        E = 1 - b/a
        P = 1 - c/b
        return L,E,P
```

In this code, the values a, b and c are the eigenvalues of the volumetric tensor.

Note : In this page the three functions were already presented, but instead of making the calculations at a time t , we will calculate the parameters given the list of time stamps. Here, the time parameter is no longer an input (i.e., the function is time independent). For the moment, these 3 functions will not be used to make the plots

(they are just bonuses for the user if one want to pursue the calculations for all values of time).

```
def barycenter(list_sat,list_index):
    """
    Computes the coordinates of the barycenter, for all time.

    Parameters
    ----------
    list_sat : list
        List of the position of each satellite, over time.
    list_index : list
        List of the indexes of the studied satellites.

    Returns
    -------
    bar : list
        Barycenter of the satellites for all t.

    """
    bar = []
    max_time_index = len(satellites[0][...])
    for i in range(max_time_index):
        bar.append(barycenter_t(list_sat,list_index,i))
    return bar
def volumetricTensor(list_sat,list_index):
    """
    Computes the coefficients of the volumetric tensor, for all time.

    Parameters
    ----------
    list_sat : list
        List of the positions of each satellite, over time.
    list_index : list
        List of the indexes of the studied satellites.

    Returns
    -------
    vt : Array of size (max_time_index,3,3)
        Volumetric tensor of the satellites for all time.

    """
    max_time_index = len(satellites[0][...])
    vt = np.zeros((max_time_index,3,3))
    for t in range(max_time_index):
        vt[t] = volumetricTensor_t(list_sat,list_index,time_index)
    return vt
def polyhedronProperties(listSat,list_index):
    """
    Computes L,E and P for all time

    Parameters
    ----------
    list_sat : list
        List of the positions of each satellite, over time.
    listIndex : list
        List of the indexes of the studied satellites.

    Returns
    -------
    L : Array of size (tMax,)
        Characteristic size of the tetrahedron, over time.
    E : Array of size (tMax,)
        Elongation of the tetrahedron, over time.
    P : Array of size (tMax,)
        Planarity of the tetrahedron, over time.

    """
    if len(list_index) ==4:
        max_time_index = len(satellites[0][...])
        L = np.zeros(max_time_index)
        E = np.zeros(max_time_index)
        P = np.zeros(max_time_index)
        for t in range(max_time_index):
            L[t],E[t],P[t] = polyhedronProperties_t(list_sat,list_index,time_index)
        return L,E,P
```

To sum up, for now the user is able to compute all the possible tetrahedra given a Helioswarm configuration at a time t. In addition, the user can characterize them by calculating their three characteristic parameters.

At this points, the user is not able to retrieve the best tetrahedra that are the most interesting for furthur scientific analyses.

The first case that was described earlier is to extract the biggest tetrahedra at a certain time t because these ones are the ones that cover the biggest volume where data can be analyzed (for example magnetic field data). This is the aim of the following code.

```
def tetra_duo_big_volume_t(poly_list,combination):
    vol = []
    for x in poly_list:
        a = x[0]/2
        b = a*(1-x[1])
        c = b*(1-x[2])
        vol.append((8/3)*a*b*c)

    vol_sort = sorted(vol, reverse=True)

    # Extraction of the two biggest tetrahedra volumes and the indexes of their summits

    max_tetra = vol_sort[:2]
    max_indexes = [vol.index(tetra) for tetra in max_tetra]

    return combination[max_indexes[0]],combination[max_indexes[1]]
```

In this code the user computes the volume of a tetrahedron from the three eigenvalues of the volumetric tensor. All the volumes of the 126 tetrahedra are calculated then sorted to extract the two biggest ones.

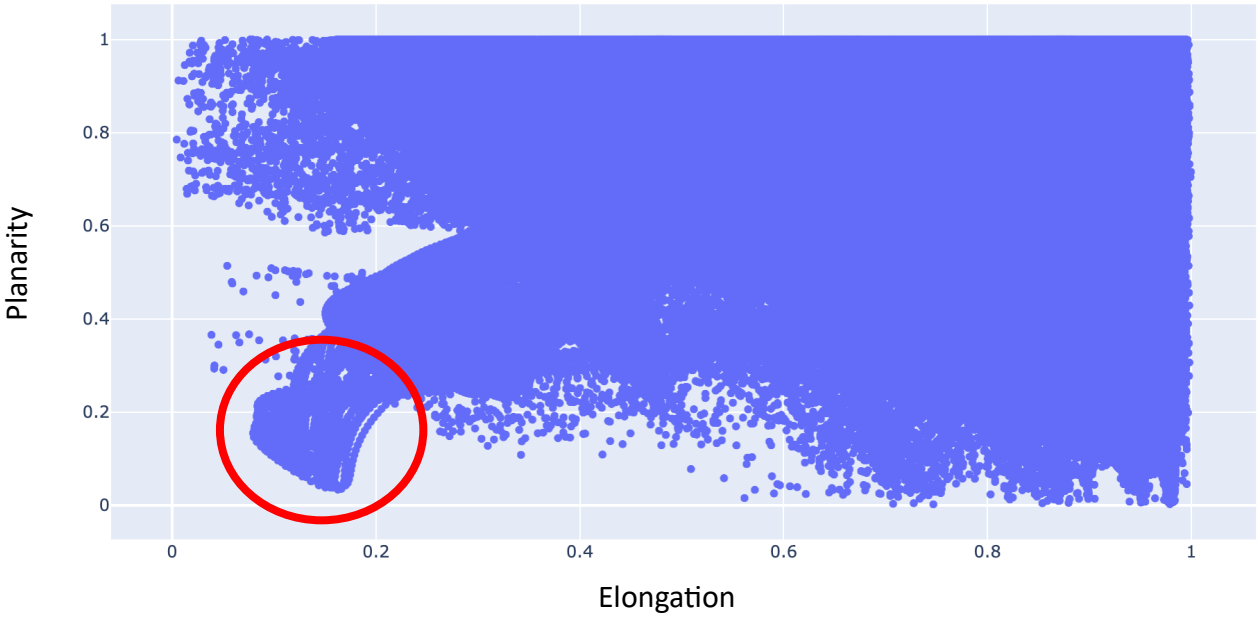
<table>
<tr><th>P \ E</th><th>0</th><th>low</th><th>interme-diate</th><th>large</th><th>1</th></tr>
<tr><td>1</td><td>Circle</td><td colspan="3">Ellipse of increasing eccentricity</td><td rowspan="5">Straight Line</td></tr>
<tr><td>large</td><td rowspan="3">Ellipsoid of increasing oblateness</td><td>Pancake</td><td>Elliptical Pancake</td><td>Knife Blade</td></tr>
<tr><td>interme-diate</td><td>Thick Pancake</td><td>Potatoes</td><td>Flattened Cigar</td></tr>
<tr><td>low</td><td>Pseudo-Sphere</td><td>Short Cigar</td><td>Cigar</td></tr>
<tr><td>0</td><td>Sphere</td><td colspan="3">Ellipsoid of increasing prolateness</td></tr>
</table>

Figure 13.2: The shape of the polyhedron as a function of E and P.

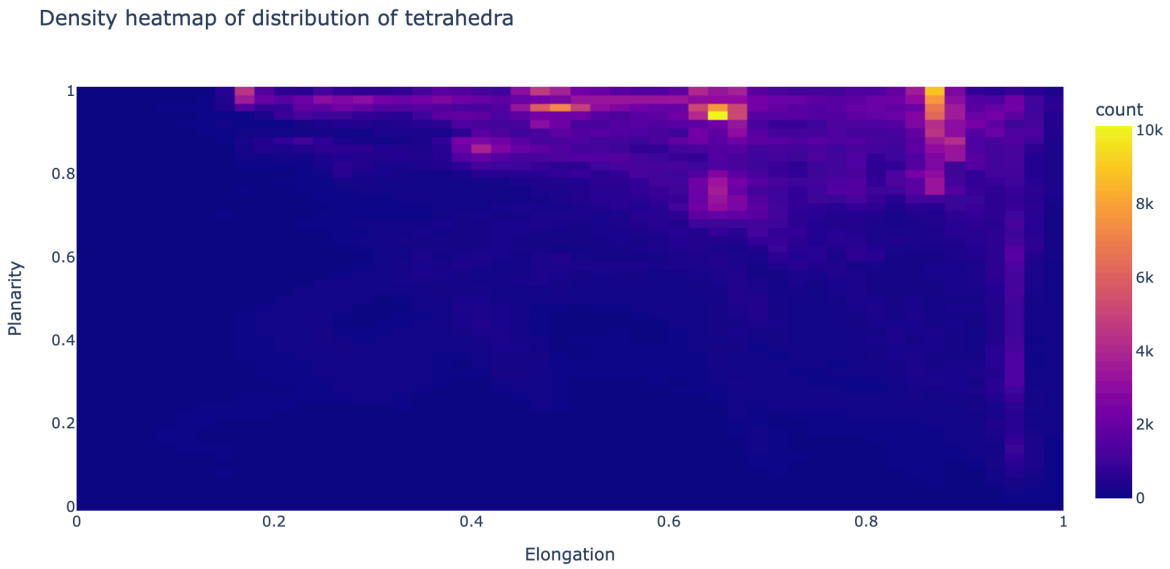
The second case that was described earlier aimed at extracting the tetrahedra that are the most regular. Indeed, the more the tetrahedron is regular the more reliable the results from four-spacecraft analysis applications. Here, we determine the regularity of all possible tetrahedra. Using the mathematical formulation from the ISSI Scientific Report : Analysis Methods for Multi-Spacecraft Data by Götz Paschmann and Patrick W.Daly the regularity of a tetrahedron can be described thanks to the elongation, planarity, size parameters.

Indeed, the lower the elongation and the planarity values are (both close to 0) the more regular the tetrahedron will be. Therefore, in order to have the two most regular tetrahedra, the user has to extract the two ones with the lowest set of elongation and planarity. In order to achieve that one has to minimize $\sqrt{E^2 + P^2}$.

Before going to the code, here is the distribution of the values of elongation and planarity using our values over the first year of data. One can see that a small portion of the tetrahedra have both planarity and elongation values under 0.2. In order to have the most regular tetrahedra, the tool will help to choose the ones that are in the red area.



Since the plot above does not reflect the density of the data points (they are overplotted), we additionally show a heat map to demonstrate the density of the data points below.



```
def tetra_most_regular(poly_list,combination):

    min_D = float('inf')
    tetra_regular = None
    index_tetra_regular = None

    for index, tuple in enumerate(poly_list):
        L,E,P = tuple
        D = m.sqrt(E**2 + P**2)
        if  D < min_D :
            min_D = D
            tetra_regular = tuple
            index_tetra_regular = index

    if tetra_regular is not None and index_tetra_regular is not None:
        del poly_list[index_tetra_regular]

    #print(tetra_regular)
    return combination[index_tetra_regular]
```

To extract the most and second-most regular ones, this function will have to be called twice.

The last step is to make the visualization tool in order to plot the given configuration of Helioswarm at a time t and to add the two tetrahedra that will either be the biggest ones or the most regular ones.

The visual tool is made with plotly which allows to make the plot more interactive while retaining the tetrahedral properties and geometric information on it.

```
def display_helioswarm_satellites_tetrahedra(index, satellites,tetra):
    fig = go.Figure()

    x = [satellites[0][index][0], satellites[1][index][0], satellites[2][index][0], satellites[3][index][0],
         satellites[4][index][0], satellites[5][index][0], satellites[6][index][0], satellites[7][index][0],
         satellites[8][index][0]]
    y = [satellites[0][index][1], satellites[1][index][1], satellites[2][index][1], satellites[3][index][1],
         satellites[4][index][1], satellites[5][index][1], satellites[6][index][1], satellites[7][index][1],
         satellites[8][index][1]]
    z = [satellites[0][index][2], satellites[1][index][2], satellites[2][index][2], satellites[3][index][2],
         satellites[4][index][2], satellites[5][index][2], satellites[6][index][2], satellites[7][index][2],
         satellites[8][index][2]]

    labels = ['HUB', 'SAT_1', 'SAT_2', 'SAT_3', 'SAT_4', 'SAT_5', 'SAT_6', 'SAT_7', 'SAT_8']
    text = [f"{label} (x: {x[i]}, y: {y[i]}, z: {z[i]})" for i, label in enumerate(labels)]

    for i, label in enumerate(labels):
        if label == 'HUB':
            marker_symbol = 'diamond'
            marker_color = 'red'
        else:
            marker_symbol = 'diamond'
            marker_color = 'black'

        fig.add_trace(go.Scatter3d(
            x=[x[i]],y=[y[i]],z=[z[i]],
            mode='markers',marker=dict(symbol=marker_symbol,color=marker_color,  size=5,),
            name=label,
            text=[text[i]],
            hoverinfo='text'
        ))

    hub_index = labels.index('HUB')

    # Add the first tetrahedron
    fig.add_trace(go.Mesh3d( # X, Y, Z coordinates of the 4 summits of the first tetrahedron
        x=[x[tetra[0][0]], x[tetra[0][1]], x[tetra[0][2]], x[tetra[0][3]]],
        y=[y[tetra[0][0]], y[tetra[0][1]], y[tetra[0][2]], y[tetra[0][3]]],
        z=[z[tetra[0][0]], z[tetra[0][1]], z[tetra[0][2]], z[tetra[0][3]]],
       i=[0, 0, 0],  # Indexes of the summits composing the faces of the first tetrahedron
       j=[1, 2, 3],
       k=[2, 3, 1],
       opacity=0.4,
       name = 'first_tetrahedron',
       color='orange'
   ))
```

```
    # Add the second tetrahedron
    fig.add_trace(go.Mesh3d(
        x=[x[tetra[1][0]], x[tetra[1][1]], x[tetra[1][2]], x[tetra[1][3]]],
        y=[y[tetra[1][0]], y[tetra[1][1]], y[tetra[1][2]], y[tetra[1][3]]],
        z=[z[tetra[1][0]], z[tetra[1][1]], z[tetra[1][2]], z[tetra[1][3]]],
        i=[0, 0, 0],
        j=[1, 2, 3],
        k=[2, 3, 1],
        opacity=0.4,
        name = 'second_tetrahedron',
        color='green'
    ))

    position_offset = 1500

    fig.update_layout(
        scene=dict(
            aspectmode='cube',
            xaxis=dict(range=[x[hub_index] - position_offset, x[hub_index] + position_offset]),
            yaxis=dict(range=[y[hub_index] - position_offset, y[hub_index] + position_offset]),
            zaxis=dict(range=[z[hub_index] - position_offset, z[hub_index] + position_offset]),
            xaxis_title='X',
            yaxis_title='Y',
            zaxis_title='Z'
        )
    )

    fig.show()
```

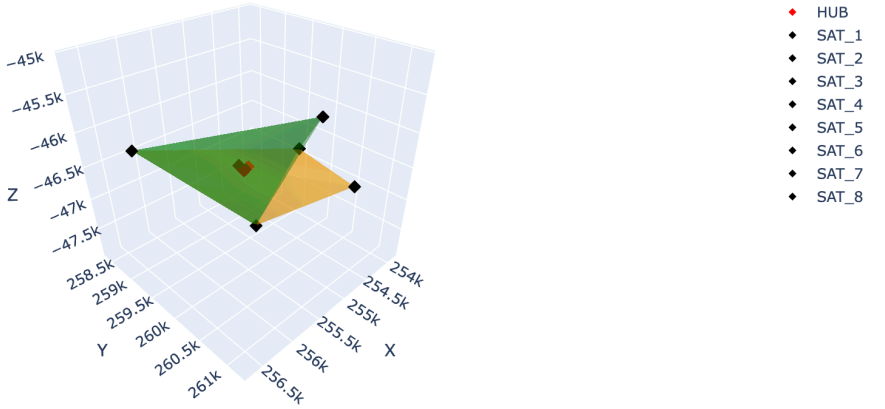
Now, we have covered everything and we are ready to make the visualization. To facilitate the use of this tool all the steps and the function calls are contained in the following main function.

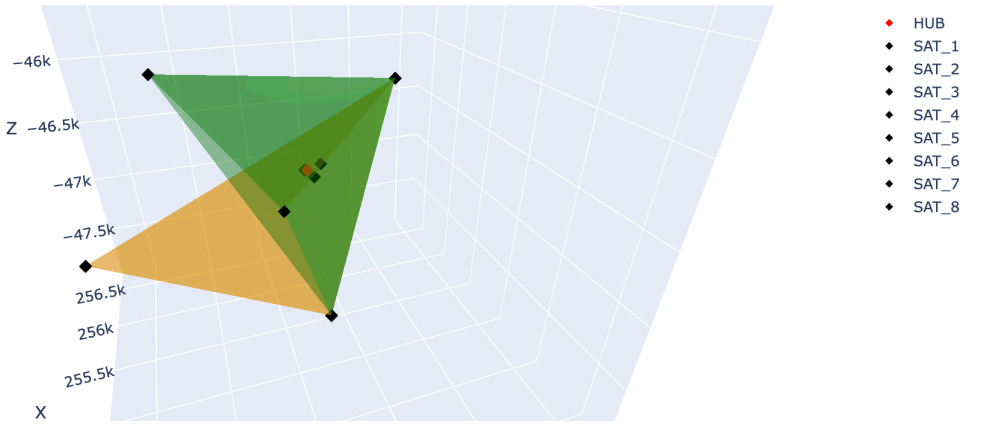
```
def main():

    satellites = np.empty((9, 8521, 3)) # Creation of the empty 3D table that will contain all the x,y,z
                                        # coordinates of the 9 satellites over the 8521 time indexes

    for i in range(9):                              # Extraction of the coordinates from the 9 .txt files
        file = "./Data/n" + str(i) + ".txt"
        coordinates = satellites_coordinates(file)
        satellites[i] = coordinates

    L = list(range(9))      # Creation of the list of indexes of the 9 satellites
    C = combinations(L,4)   # List of all the combinations of 4 elements that can be made in the list L
    pol_t = []              # Empty list that will contain (L,E,P) tuples of the 126 tetrahedra

    time_index = int(input("Please choose a value of time_index between 0 and 8521 : "))
    print('time_index corresponds to time : ' + str(retrieve_time("./Data/n0.txt")[time_index]))

    question = input("Want another date ? : Yes or No ?")
    while question == "Yes":
        time_index = int(input("Please choose a value of time_index between 0 and 8521 : "))
        print('time_index corresponds to time : ' + str(retrieve_time("./Data/n0.txt")[time_index]))
        question = input("Want another date ? : Yes or No ?")

    for comb in C :
        pol_t.append(polyhedronProperties_t(satellites,comb,time_index))

    function_choice = input("Please choose a function that rather : Computes the two tetrahedra that cover the bigg

    # Executes the given function depending on the choice of the user
    if function_choice == '1':
        display_helioswarm_satellites_tetrahedra(time_index, satellites,tetra_duo_big_volume_t(pol_t,C))
    elif function_choice == '2':
        tetra_duo_most_regular = [tetra_most_regular(pol_t,C),tetra_most_regular(pol_t,C)]
        display_helioswarm_satellites_tetrahedra(time_index, satellites,tetra_duo_most_regular)
    else:
        print("Invalid choice. Please choose 1 or 2.")
```

In this main function after creating the empty lists and all the combinations, the user will be asked to choose a time index between 0 and 8521 (index of the time stamps within one year of the data, at 1 hour time cadence). After making a choice the user will be asked to confirm the corresponding time value. The user will then be asked to keep the first value or to choose another one until satisfied. After that, the calculation of the list of tuples containing the characteristic values (L,E,P) of the 126 tetrahedra

the user will be asked to choose between (1) the two biggest tetrahedra, and (2) the two most regular ones. Finally, as an example, the tool will display the result as follows : (Two points of view of the same calculation given) (the units of x,y and z are in km)

```
Please choose a value of time_index between 0 and 8521 : 2536
time_index corresponds to time : ['22 Dec 2026 03:39:59.992']
Want another date ? : Yes or No ?Yes
Please choose a value of time_index between 0 and 8521 : 1765
time_index corresponds to time : ['20 Nov 2026 00:39:59.992']
Want another date ? : Yes or No ?No
Please choose a function that rather : Computes the two tetrahedra that cover the biggest volume (1) or the two tetra
hedra that are the most regular (2)1
```





Conclusion

To conclude, the tool presented is useful to provide an overview of the configuration of Helioswarm to understand the global context and to analyze subsets of the nine spacecraft forming tetrahedral configurations.

In the GSE frame, the tool is to display a large spatial scale to see where Helioswarm is with respect to the Sun-Earth line, and thus the solar wind main direction. In the HUB frame, the idea is to display the possible, best, or largest tetrahedral configuration.Here the user can choose tetrahedra that suit with their scientific purposes. A next, possible step is to add the magnetic field data to the latest plot to see how subsets of Helioswarm spacecraft might be chosen. Finally, analysis techniques with four spacecraft or more (e.g., gradient methods) may be implemented.