ORBITAL DEBRIS CLOUD EVOLUTION: AN ANALYSIS OF FRAGMENTATION EVENTS IN LOW EARTH ORBIT

by

Reece Humphreys

A Thesis Submitted to the Faculty of

The Wilkes Honors College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Liberal Arts and Sciences

with a Concentration in Physics

Wilkes Honors College of

Florida Atlantic University

Jupiter, Florida

May 2021

ORBITAL DEBRIS CLOUD EVOLUTION: AN ANALYSIS OF FRAGMENTATION EVENTS IN LOW EARTH ORBIT

by

Reece Humphreys

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Yaouen Fily, and has been approved by the members of their supervisory committee. It was submitted to the faculty of the Harriet L. Wilkes Honors College and was accepted in partial fulfillment of the requirements for the degree of Bachelor of Science in Liberal Arts and Sciences.

SUPERVISORY COMMITTEE:

____

Dr. Yaouen Fily

____

[second reader]

____

Interim Dean Timothy Steigenga, Harriet L. Wilkes Honors College

____

Date

Abstract

Author:	Reece Humphreys
Title:	Orbital Debris Cloud Evolution: An analysis of fragmentation events in low Earth orbit
Institution:	Harriet L. Wilkes Honors College, Florida Atlantic University
Thesis Advisor:	Dr. Yaouen Fily
Degree:	Bachelor of Science in Liberal Arts and Sciences
Concentration:	Mathematics
Year:	2021

Orbital debris has quickly become one of the newest sources of pollution as a result of mankind's desire to work in, explore, and utilize space. However, unlike most types of pollution which people experience daily, this is pollution that is impossible for the average person to ever encounter, yet poses just as grand of a threat as the other types. Orbital debris are the remnants of orbital collisions, weapons tests, decommissioned satellites, and spent rocket stages that are passing over our heads faster than bullets and containing similar energy to hand grenades. This paper explores the existing models of orbital debris generation, how clouds of debris evolve with respect to time, and the ramifications that they pose.

Contents

List of Figures

List of Tables

List of Symbols

A Cross-sectional area [m2].

H Scale height for exponential atmospheric model [km].

I_k Modified Bessel function of the first kind and order k .

J_2 Second zonal harmonic coefficient for the Earth .

L_c Fragment characteristic length [m] .

M_e Reference mass for collisions[kg] .

M_p Projectile mass[kg] .

M_t Target mass[kg] .

M Mass [kg] .

Δv Relative velocity [km/s].

Ω Longitude of the ascending node [rad or deg].

Φ Flux of fragments.

$\mathcal{N}$ Normal distribution .

μ_E Earth's planetary gravitational constant [km3/ s2].

μ Mean value.

ν True anomaly [rad or deg].

ω Argument of the periapsis [rad or deg].

ρ Atmosphere density [kg / m3].

a Semi-major axis [km].

e Eccentricity.

h Altitude [km].

i Inclination [rad or deg].

v_c Relative impact velocity [km/s] .

1 Introduction

1.1 Motivation

Due to the accelerating launch cadence in the space sector, increased accessibility and resources for small teams to create cube satellites, and satellite mega constellation constructions underway, researchers have become increasingly concerned about the implications of potential orbital collisions. These worries have been compounded by the actions taken by foreign nations with regards to anti-satellite weapons which create substantial amounts of debris. In one such instance, a 2007 Chinese anti-satellite missile test was universally condemned and received statements from government officials such as U.K. prime minister who stated "We are concerned about the impact of debris in space and we expressed that concern". These fragmentation events can be difficult to track due to the small size of some of the debris fragments that are generated, yet they can pose a great hazard to other satellites and crewed operations being conducted in space. Tens of millions of pieces of orbital debris currently exist within Low Earth Orbit (LEO) with an average size smaller than 1cm. While minuscule in size, these pieces of debris have an average impact velocity of 10 km/s which generates similar energy to that of an exploding hand grenade. It is, therefore, paramount to study the phenomena that arise within orbital debris clouds to gather methods for mitigating debris cloud formations. Without such a study, the future commercialization of space, the potential for humanity to become a multi-planetary species, and the benefits that the advanced satellites provide researchers will remain in jeopardy.

1.2 Methodology

The first component in modeling orbital debris is to implement a breakup model which makes predictions about the outcome of an orbital collision or explosion. Breakup models use experimental data to create statistical models that predict the size, mass, speed, and number of debris pieces generated in a fragmentation event. Once these characteristics are obtained, equations of motion can be implemented that account for the significant forces acting on each debris fragment, such as drag and solar radiation, to model how the debris position and velocity will evolve over time.

This paper implements the NASA Standard Satellite Breakup Model to simulate the orbital collisions and gather information regarding the behavior of orbital debris [3]. This was accomplished by utilizing the model to create an implementation in python which has been made open source on GitHub for others wishing to build on the foundations of this research.

Following the implementation of the breakup model, the implementation of dominant orbital perturbations is given. These are forces that act on debris to change their orbits over time and include effects such as atmospheric drag and solar radiation. The optimal way to represent these effects is through changes in orbital parameters, which is an alternate way of expressing the current state of an object rather than using euclidean coordinates (x, y, z, v_x, v_y, v_z) . The benefits of using orbital parameters is explored more in-depth later in the paper.

Finally a case study showing the magnitude of orbital collisions is shown along with an analysis on the potential for the Kessler syndrome to occur given the increase in launch cadence and recent expansions into constructing satellite mega constellations.

1.3 Roadmap

Chapter 1: Introduction to Orbital debris and the process of modeling them

Chapter 2: Details the implementation of the NASA breakup model to simulation a fragmentation event

Chapter 3: Reparameterization of the orbital debris to orbital elements and the benefits this yields

Chapter 4: Explains the different phases of the orbital debris cloud and how the propagation of each phase is conducted

2 Modeling Satellite Breakups

2.1 The NASA Standard Satellite Breakup Model

A satellite is any artificial body placed in orbit around the earth or moon or another planet. The definition of the term is intentionally general and as such, can be used to reference spacecraft (SC), remnants of rockets (RB), and communication devices (SAT) in orbit. A satellite breakup model is a mathematical model used to describe the outcome of a satellite breakup due to an explosion or a collision [6]. The outcome of any satellite breakup model should describe the size, area-to-mass (AM) ratio, and the ejection velocity of each fragment produced in the satellite breakup [3]. The NASA Standard Satellite Breakup Model is an industry-standard breakup model developed by NASA and is used by most major space agencies such as the European Space Agency (ESA) and the Japanese Aerospace Exploration Agency (JAXA). The implementation of it was provided by [3] but was later clarified by Kristo in 2011 [4].

The way that the model works is by using experimental observations performed both on Earth and in orbit to characterize the breakup using statistical distributions. The choice to use statistical distributions is a result of the stochastic nature of the breakup event in the sense that it would be impossible to reproduce the same circumstances each time. For example, explosions are stochastic due to the complex chemical reactions that lead to the explosion. By using a statistical distribution and sampling from it we can more accurately represent the fragments that would be generated during a collision or explosion. For this paper, I will be focusing on how the collision fragments are generated, but it should be noted that the explosion case just involves slightly different parameters in many cases.

2.2 Implementing the NASA Breakup Model

2.2.1 Characteristic Length and Number of fragments

To account for the different characteristics of each fragment of debris, the statistical distributions must be expressed as a function of some independent variable [3]. In the latest version of the NASA breakup model this variable is called the characteristic length, L_c, which assumes that the debris particles are spherical and have the density of aluminum for objects smaller than 1 cm with a diminishing density for larger debris [3]. By defining the distributions using characteristic length we ensure that the mass, area, and velocity of each fragment are not constant for all debris with the same characteristic length. This in turn guarantees that the stochastic nature of the breakups is reflected in our models. It should be noted that in prior versions of the NASA breakup model, the mass of each piece of debris was used as the independent variable [4]. However, it was found to be more directly linked to both on-orbit and terrestrial breakup data [3].

The implementation of the characteristic length distribution can be cumbersome to follow. As such, a flow chart illustrating the steps of the algorithm is provided below. Each of the steps listed in the flow chart will be explored in further detail in the rest of this subsection.

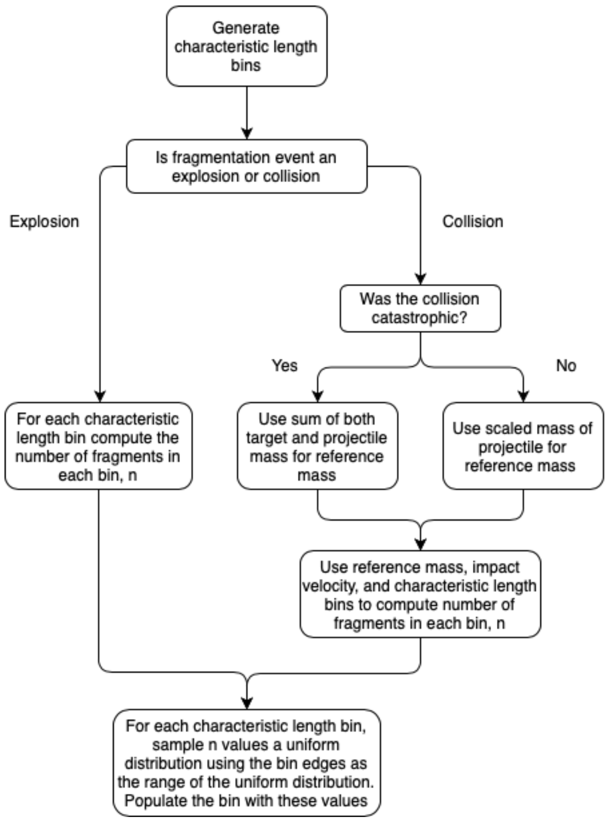

Generate characteristic length bins
Is fragmentation event an explosion or collision
Explosion
Collision
Was the collision catastrophic?
Yes
No
For each characteristic length bin compute the number of fragments in each bin, n
Use sum of both target and projectile mass for reference mass
Use scaled mass of projectile for reference mass
Use reference mass, impact velocity, and characteristic length bins to compute number of fragments in each bin, n
For each characteristic length bin, sample n values a uniform distribution using the bin edges as the range of the uniform distribution. Populate the bin with these values

Figure 2.1: The algorithm used for producing the characteristic length distribution used in the NASA breakup model

The creation of the characteristic lengths was not given in the original specification of the NASA breakup model by Johnson but was included in the corrections by Krisko in 2001. Krisko specifies that the NASA breakup model deposits fragments of L_c from 1mm to over 1m in bins and that the number of fragments in each bin is determined by a power law that will be discussed later. However, Latizia's implementation for CiELO modified this to first create 100 bins that are equally spaced on a logarithmic scale between 1mm and 10 cm. For this paper, we will be following Latizia's methodology as it has the most up-to-date information about the NASA breakup model. A common theme among the prior research on the NASA breakup model is that each implementation is slightly different due to the lack of information publicly available about the original implementation specified by Johnson et. al. The pieces of debris that are larger than 10cm will be handled separately to ensure that the breakup model conserves mass. Figure (X) below illustrates the process described above for how the characteristic length illustration works.

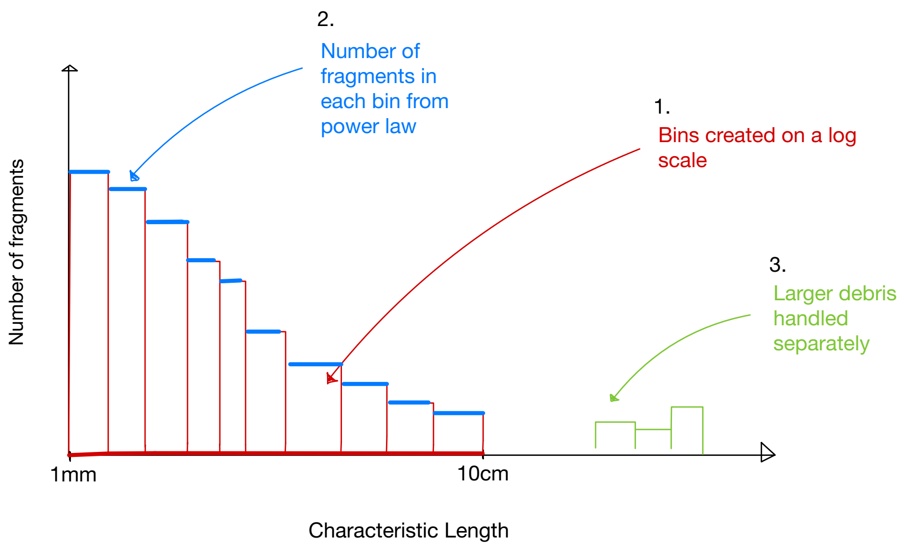

2.
Number of fragments in each bin from power law
1.
Bins created on a log scale
3.
Larger debris handled separately
Number of fragments
1mm
10cm
Characteristic Length

Figure 2.2: A sketch illustrating the process for determining the number of fragments in each characteristic length bin

To determine the number of fragments in each characteristic length bin it is crucial to first recognize that collisions and explosions will produce different types of fragments. Explosions will produce larger debris fragments with smaller velocities while collisions tend to generate a large number of small fragments with high velocities [1]. As such, the number of fragments in each bin will be determined by different power laws based on the type of breakup event.

The number of explosive fragments of size L_c is governed by the following equation:

$$N(L_c) = 6 * S * L_c^{-1.6} \tag{1}$$

With S = 1, the relationship has been observed to be valid for rocket upper stages with masses in the range of 600-1000 kg [3]. However, for explosions due to other malfunctions such as battery explosions, anti-satellite tests etc a value of S between 0.1 to 1 was found to be a good approximate solution [4] .

In the case of a collision, a distinction must be made, if the collision was catastrophic or non catastrophic. A collision is categorized as catastrophic if it causes the complete fragmentation of both the impactor and the target [5]. This occurs when the energy per target mass exceeds $40J * g^{-1}$ [4]. The number of produced fragments for a collision is governed by the following :

$$N(L_c[m]) = 0.1 * (M_e)^{0.75} * L_c^{-1.71} \tag{2}$$

Where M_e is defined as follows:

$$M_e[kg] = \begin{cases} M_t[kg] + M_p[kg] & \text{Catastrophic collision:} \\ M_p[kg] * (v_c[km/s]/1[km/s])^2 & \text{Non Catastrophic collision} \end{cases} \tag{3}$$

M_t is the target mass, M_p is the projectile mass,V_c is the relative impact velocity between the target and the projectile.

Figure (x) shows the algorithm for the creating the characteristic length distribution described above. The resulting distribution that is produced by the python implementation for a test scenario is given in figure (X). Additionally, the python implementation can be found in the appendix in section (X).

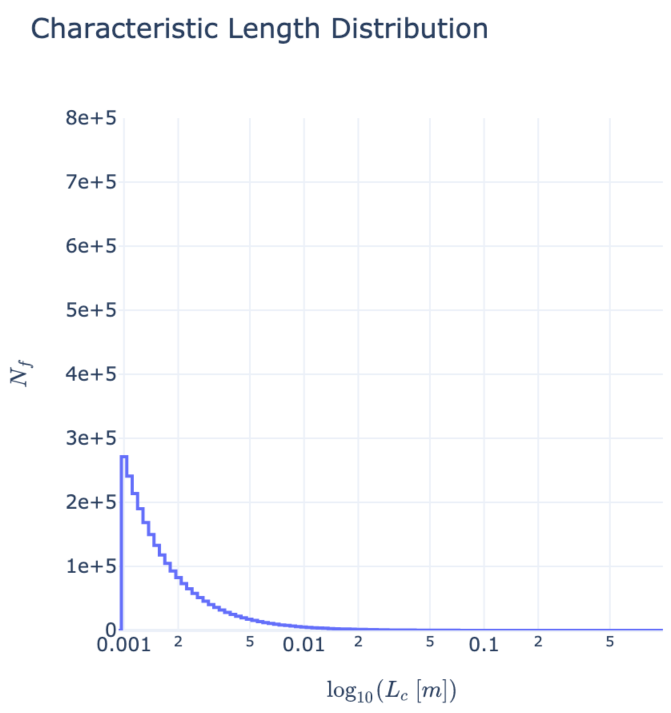


Figure 2.3: The characteristic length distribution produced by the python implementation of a catastrophic collision involving a rocket body with a mass of 1000kg, a projectile of mass 10kg, and an impact velocity of 10 km/s.

2.2.2 Area to Mass Distribution

The area-to-mass ratio, A/M, for fragments is a distribution that was based on analysis of thousands of fragmentation debris and provides us with a method to the mass of each fragment of debris. The discrete distributions were found by using a χ^2 fit to orbital decay characteristics for 1,780 upper stage explosion fragments, and similar data was developed for spacecraft fragments [3]. Each type of debris producer, RB, SC, and SAT, will produce different size debris. As such, the distribution that determines the area to mass ratio has three variants. All three are based on a normal distribution but use different expressions for determining the mean and standard deviation of the distribution. For simplicity, the rest of this section will only provide the details of the SC distribution. However, the implementation of the other two categories is included in the python implementation.

For small objects, with $L_c < 8\text{cm}$, SAT, the A/M distribution is expressed as

$$D_{A/M}(\lambda_c, \chi) = \mathcal{N}(\mu_{A/M}, \sigma_{A/M}(\lambda_c), \chi). \tag{4}$$

$D_{A/M}$ is the distribution function of χ as a function of λ_c, where

$$\lambda_c = \log_{10}(L_c), \tag{5}$$

$$\chi = \log_{10}(A/M) \tag{6}$$

$\mathcal{N}$ is the normal distribution function with mean $\mu_{A/M}$ and standard deviation $\sigma_{A/M}$, where

$$\mu_{A/M} = \begin{cases} -0.3, & \lambda_c \leq -1.75 \\ -0.3 - 1.4(\lambda_c + 1.75), & -1.75 < \lambda_c < -1.25 \\ -1.0, & \lambda_c \geq -1.25 \end{cases} \tag{7}$$

$$\sigma_{A/M} = \begin{cases} 0.2, & \lambda_c \leq -3.5 \\ 0.2 + 0.1333(\lambda_c + 3.5) & \lambda_c > -3.5 \end{cases} \tag{8}$$

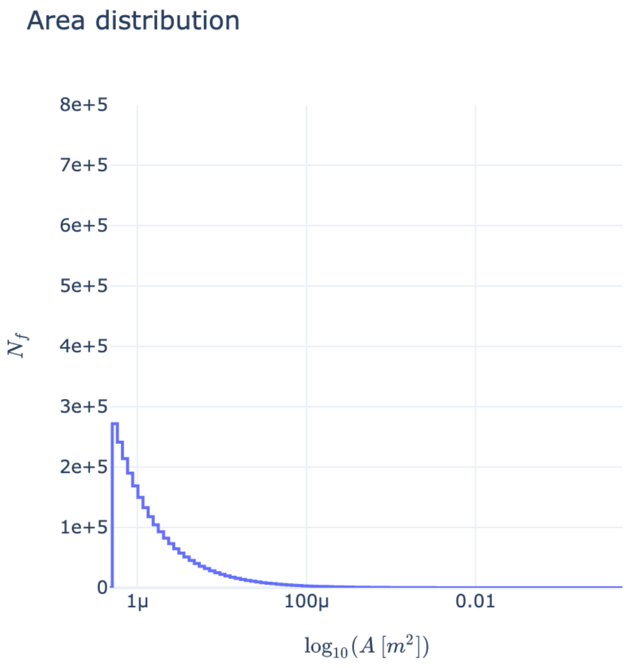
Every fragment of debris has a corresponding A/M distribution since both $\mu_{A/M}$ and $\sigma_{A/M}$ are functions of λ_c. To determine the corresponding A/M ratio for each debris, a random value is drawn from the distribution. This accounts for stochastic nature of breakup events mentioned previously.

The A/M ratio alone does not provide enough information to determine both the area and mass of a fragment. As such the average cross-sectional area, A, can also be obtained through a one-to-one correspondence with L_c

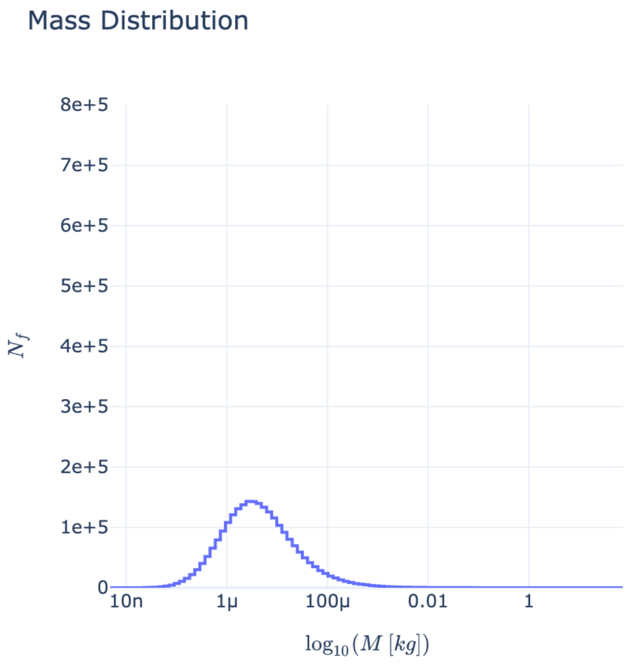
using the following expression [3]:

$$A_x = \begin{cases} 0.540424 * L_c^2 & \text{where } L_c < 0.00167m \\ 0.556945 * L_c^{2.0047077} & \text{where } L_c \geq 0.00167m \end{cases} \tag{9}$$

Utilizing both the A/M ratio and the cross sectional area A, we can now obtain the mass M simply using $M = A_x/(A/M)$. Figure (X) shows a simulated A/M distribution and average cross sectional area. The code to reproduce this is found in section (X).



(a)



(b)

Figure 2.4: The characteristic length distribution produced by the python implementation of a catastrophic collision involving a rocket body with a mass of 1000kg, a projectile of mass 10kg, and an impact velocity of 10 km/s.

2.2.3 Change in Velocity Distribution

The differential amount of velocity that each fragment will gain due to the breakup event is determined in a similar manner to the A/M ratio. The notable differences are that the distribution is now a log-normal distribution and that an additional check is implemented to ensure that extremely high ejection velocities are not included in the case of collisions [5].

More explicitly the velocity check is performed by sampling a value from the velocity distribution, and checking if it is lower than $1.3v_c$, where v_c is the relative collision velocity. If the value fails the check then new values are drawn until the check is passed [5].

The change in velocity, Δv, log-normal distribution a function of the A/M ratio as additionally it is a log-normal distribution modeled as

$$D_{\Delta v} = \mathcal{N}(\mu_v(\chi), \sigma_v(\chi), \xi) \tag{10}$$

where,

$$\xi = \log_{10}(\Delta v) \tag{11}$$

$$\chi = \log_{10}(A/M) \tag{12}$$

$$\mu_v(\chi) = 0.2\chi + 1.85 \tag{13}$$

$$\sigma_v(\chi) = 0.4 \tag{14}$$

The differential velocity vector is constructed by sampling from a uniform distribution for a velocity unit vector and then scaling it with the values from the log-normal distribution.

It should be noted that details for efficiently handling the resampling of velocities are sparse. As such, the python implementation of the velocity distribution often takes tens of thousands of iterations to complete. Often one or two fragments have an area to mass ratio that causes the log-normal distribution to be highly unlikely to return a value that is lower than $1.3V_c$. As such an iteration limit exists in the code that is set at 100,000 iterations to try and alleviate the computation time. If no suitable value is found before the maximum iteration limit is hit, a value of $1.3V_c$ is assigned to that piece of debris.

2.3 Validating the Implementation

Due to the niche nature of orbital debris analysis, it can generally be challenging to find any full implementation of the breakup model. Additionally, many of various space agencies around the world do not share the details of how they implemented the models. For this reason, the above implementations were largely modeled after the details given in the CiELO [5] which mimicks the information provided in NASA literature.

To validate that the python implementation was performed correctly, a comparison is made to existing data provided by various space agencies for a given scenario. The two scenarios are an explosion event with a rocket body that weighs 1000kg and a catastrophic collision event for a rocket body that weighs 1000kg and a projectile mass that weighs 10kg with an impact velocity of 10km/s. The data was provided by Rossi et. al. in "NASA Breakup Model Implementation Comparison of Results"[7].

As shown in the table below, the data from the various space agencies has some significant differences, especially for the characteristic length in the ¿ 1mm range despite all implementing the same NASA breakup model. This is most likely a result of ambiguities in the original NASA breakup model specification literature as well as differences in how various programming languages perform random sampling. The goal of the python implementation is to be within the original deviations of the data.

Model	Number of Fragments						
	Length				Mass	Area	Velocity
	>1mm	>1cm	>10cm	>1m	>1g	>1cmˆ2	>100msˆ-1
ASI	378,581	9,403	234	7	2,472	5,878	112,932
CNSA	37,865	960	32	9	254	-	11,380
DLR	1,217,054	11,724	230	0	25,844	31,124	31,124
ESA	324,886	8,159	206	6	2,093	5,024	98,717
NASA	434,928	10,731	248	8	2,525	6,416	132,032
Mean	478,663	8195	-	-	-	-	-
Standard Deviation	393,726	3813	-	-	-	-	-
Python Implementation	378,525	9,479	223	6	3613	5,850	120,481

Table 1: (Data Source: NASA Breakup Model Implementation Comparison of results, A. Rossi, 24th IADC Meeting April 2006)

... **will elaborate more in this section once the python implementation has been fixed and validated**

3 State Representation

Following the successful computation of the fragmentation event, we need to construct a state representation for each piece of orbital debris. There are two primary methods for representing the state of an orbital body.

The first is using **orbital state vectors** to represent the debris. Orbital state vectors consist of the Cartesian position, $\vec{r}$, and velocity $\vec{v}$ along with time, t. Additionally, the preferred coordinate system for this representation is the **Earth-Centered Inertial (ECI) Coordinate system**. This is a coordinate frame with the origin at the center of Earth's mass and has orbiting satellites moving relative to Earth. The orbital state vector representation is a natural result of Newton's laws of motion and the Universal Law of Gravitation.

The second representation is using **Keplerian elements**, which result from Kepler's laws of planetary motion. These consist of the eccentricity, e, Semimajor axis, a, inclination, i, longitude of the ascending node, Ω, the argument of periapsis, ω, and the true anomaly, ν.

Both of these methods have advantages and disadvantages for modeling orbital debris, and as such, it is often convenient to switch back and forth. Keplerian elements are a more abstract representation of an object's motion but provide significant benefits for performing propagations forward in time. Orbital state vectors are often more suited to performing visualizations and give a more intuitive sense of each piece of debris's current state. Both of these representations are explored in this section, as well as their benefits and construction.

3.1 Orbital State vectors and their advantages

The orbital state vectors utilize Newton's laws of motion and Newton's universal law of gravitation to describe the motion of an object in space. Newton's universal law of gravitation states that

$$\vec{F}_{grav} = G\frac{m_1 m_2 \vec{r}}{\mid \vec{r} \mid^3}$$

This is the dominant force that acts on orbital objects, neglecting drag. As such application of newtons second law yields

$$m_1 * \vec{a_1} = G\frac{m_1 m_2 \vec{r}}{\mid \vec{r} \mid^3} \tag{15}$$

$$\vec{a_i} = G\sum_{i \neq j} m_j \frac{\vec{r_j} - \vec{r_i}}{\mid r_j \vec{-} r_i \mid^3} \quad \text{(More general)} \tag{16}$$

Therefore, the change in an object's velocity can be expressed as the sum of the contributions of gravitational attraction from all other surrounding bodies. Gravitational force depends on mass, meaning the gravitational attraction for a piece of debris to nearby debris is negligible compared to the gravitational attraction from a nearby planetary body. Thus, for computational efficiency, inter-particle interactions will be neglected.

The benefits of using orbital state vectors are that they provide a natural intuition for how the motion of a piece of debris will change over time and they provide the necessary information for doing 3D visualization. However, a large drawback of using state vectors is that all six degrees of freedom, (x, y, z, v_x, v_y, v_z), will be changing during every timestep. As such when simulating a large number of fragments orbiting a body, the memory usage will grow quite large. Additionally, the periodic nature of orbits is not captured with this parameterization, meaning to know the state of a piece of debris in the far future will require propagating that debris from its initial conditions up to the desired time step. This again is an undesirable consequence and one of the primary motivations for preferring the Keplerian element parameterization.

3.2 Keplerian elements and their advantages

When viewed from an inertial plane, two orbiting bodies trace distinctive trajectories, where each has a focus at the common center of mass. When switching to a non-inertial frame centered on one of the bodies, only the opposite body's trajectory is viewable. Keplerian elements are a parameterization that describes these non-inertial trajectories (Wikibook).

The reference body is called the primary body, in our case this is the Earth, while the other body is the secondary body. It should be noted that there is no preferred primary or secondary body.

The first step to reparameterize the motion of the orbital debris is to use the position vector, $\vec{r}$, and velocity vector, $\vec{v}$, defined in the above subsection to define the **specific angular momentum** vector as follows

$$\vec{h} = \vec{r} \times \vec{v} \tag{17}$$

Using the Earths equator as the **fundamental plane**, which is the plane that is used as a reference, we can begin constructing the parameters of the **orbital plane**, which is the plane created by tracing out the $\vec{r}$ vector, which contains both the $\vec{(r)}$ vector and $\vec{v}$ vector. The intersection of the orbital plane with the fundamental plane is called the **line of nodes**

The **ascending node** is the spot where the orbiting body crosses the plane of reference / equatorial plane in a northerly direction. Similarly, the **descending node** is where it crosses the plane of reference in a southerly direction. $\vec{n}$ is a vector that points in the direction of the ascending node and is found by taking the cross product of the unit vector $\hat{k}$ with the angular momentum $\vec{h}$

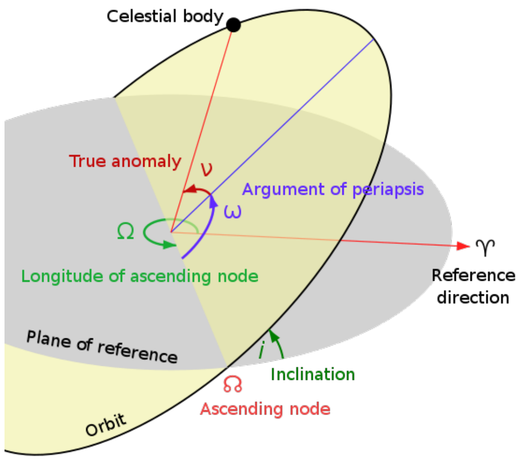


Figure 3.5: A diagram illustrating the Keplerian elements related to the orbital plane intersecting a reference plane. Data Source: Orbital elements Wikipedia page

$$\vec{n} = \hat{k} \times \vec{h} \tag{18}$$

Using the parameters typically used for specifying an ellipse, we can also describe an orbit

The **Semi-Major Axis**, a, is one half of the major axis. The major axis is a line that goes through both foci of the ellipse and the center, and ends at the widest point of the perimeter.

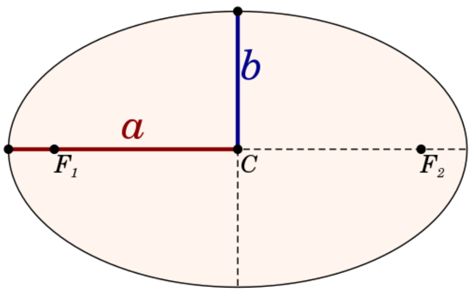


Figure 3.6: The semi-major axis, a, along with the semi-minor axis, b, and the foci of an ellipse, F_1 and F_2. Data Source: Semi-major and semi-minor axis Wikipedia page

Using the Vis-Viva equation, the semi-major axis, a, can be defined as:

$$a = \frac{1}{\frac{2}{|\vec{r}|} - \frac{|\vec{v}|^2}{\mu}} \tag{19}$$

where μ is the standard gravitational parameter of the primary body. For Earth the value of μ is $\mu = 3.986 * 10^{14} \frac{m^3}{s^2}$.

The **orbital eccentricity** is a dimensionless parameter that indicates to what degree an orbit around another body deviates from a perfect circle. It has the value of 0 for a circular orbit, a value between 0 and 1 for elliptic orbits, 1 for parabolic escape orbits, and greater than 1 for hyperbolic orbits. For the purpose of orbital debris in LEO, the eccentricities will be in the elliptic orbit range.

The equation for eccentricity can be expressed from the the Orbital State vectors as follows:

$$\vec{e} = (\frac{| \vec{v} |^2}{\mu} - \frac{1}{| \vec{r} |})\vec{r} - \frac{(\vec{r} \cdot \vec{v})}{\mu}\vec{v} \tag{20}$$

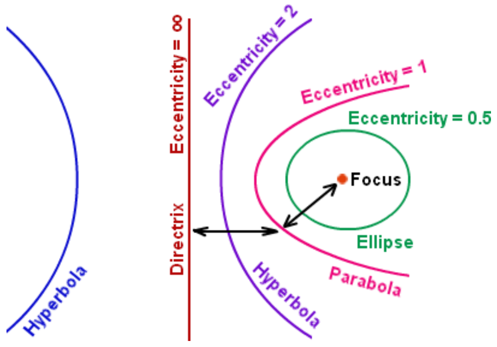


Figure 3.7: The relationship between the eccentricity and the resulting conic section. Data Source: https://3.bp.blogspot.com

The **inclination**, i, is the angle formed between the unit vector pointing in the $\hat{K}$ direction and the angular momentum vector $\vec{h}$ and can be calculated using

$$i = \arccos \frac{K_z}{\mid \vec{h} \mid} \tag{21}$$

The **true anomaly**, ν, is used to define the position of the body in its orbit, and is the angle between the direction of the periapsis and the current position of the body.

$$\nu[rad] = \begin{cases} \arccos \frac{\vec{e}\cdot\vec{r}}{|\vec{e}||\vec{r}|} & \text{for } \vec{r}\cdot\vec{v} \geq 0 \\ 2\pi - \arccos \frac{\vec{e}\cdot\vec{r}}{|\vec{e}||\vec{r}|} & \text{otherwise} \end{cases} \tag{22}$$

The **Eccentric anomaly**, E, is also used to measure an objects in an orbit. The Eccentric Anomaly is defined using the magnitude of the eccentricity vector, e, and the true anomaly as follows

$$E = 2\arctan \frac{\tan \frac{\nu}{2}}{\sqrt{\frac{1+e}{1-e}}} \tag{23}$$

The **longitude of the ascending node**, Ω, is the angle between the ascending node and the unit vector $\hat{i}$

$$\Omega[rad] = \begin{cases} \arccos \frac{n_x}{|\vec{n}|} & \text{for } n_y \geq 0 \\ 2\pi - \arccos \frac{n_x}{|\vec{n}|} & \text{for } n_y < 0 \end{cases} \tag{24}$$

The **argument of periapsis**, ω, is the angle in the orbital plane that is between the ascending node and the periapsis and is calculated using

$$\omega[rad] = \begin{cases} \arccos \frac{\vec{n}\cdot\vec{e}}{|\vec{n}||\vec{e}|} & \text{for } e_z \geq 0 \\ 2\pi - \arccos \frac{\vec{n}\cdot\vec{e}}{|\vec{n}||\vec{e}|} & \text{for } e_z < 0 \end{cases} \tag{25}$$

Finally, using **Keplers equation** we can define the **mean anomaly** as

$$M = E - e \sin E \tag{26}$$

We can now parameterize each of the debris fragments using (e, a, i, Ω, ω, M). This step is crucial to ensure that in the absence of orbital perturbations only one of these parameters, the mean anomaly, will change with respect to time, t This will allow for more efficient computations in the band formation phase of the simulation. Moreover, the change in mean anomaly with respect to time is analytic and expressed as

$$\frac{dM}{dt} = \sqrt{\frac{\mu}{a^3}} \tag{27}$$

As such, the position of a piece of debris in orbit can be found at any time in the future without having to numerically approximate its change in position over time. Due to the vast amount of debris generated, this will provide major benefits for the band formation phase of the orbital debris cloud.

4 Debris cloud evolution

Debris cloud evolution is the collective change in the orbits and positions of fragments with respect to time. This evolution goes through a few distinctive phases, notably the ellipsoid, ring formation, toroid, and band formation, illustrated in figure 4.8. The primary differences in each phase are a result of how long each phase lasts. For example, the ellipsoid and ring formation occurs within a few days; as such only the force of gravity will have a significant effect during this period. However, the toroid and band formation phases take over a year. As a result, additional forces such as drag will have more prominent effects due to the increased duration. Different methodologies need to be applied for each phase to ensure fast and accurate computations. These different phases, the forces being considered, and the methodologies used will be analyzed in-depth in the following subsections.

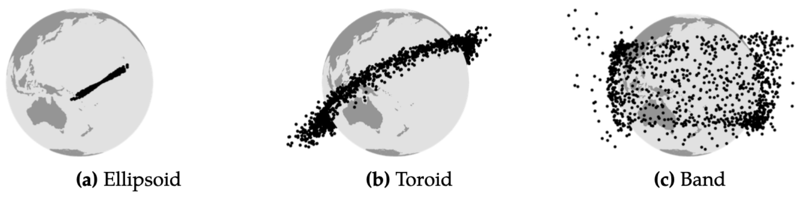


Figure 4.8: The three phases of debris cloud evolution. Data Source: Letizia

4.1 Ellipsoid and Ring Phase

At the time of the fragmentation event, all debris fragments have the same position. However, as seen in section 2, each of the fragments' velocities is different and point in random directions. These differences in velocities cause the debris to spread out as their orbits progress until uniformity is reached. The ellipsoid formation is a minor phase that occurs within a few hours, characterized by the debris starting to spread out. The ellipsoid phase's importance is that the fragments of debris have a relatively high density and, as such, can have a high probability of colliding with other satellites. Since this research is not concerned with collision probabilities, no analysis will be performed on the ellipsoid phase. A visualization that was created from the python implementation of this phase is provided in figure 4.9.

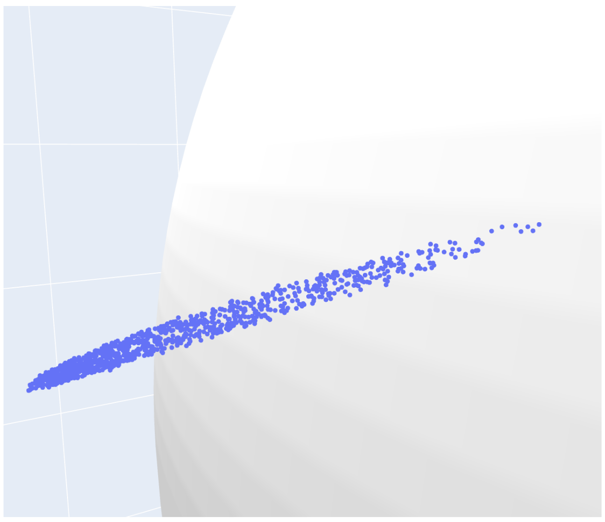


Figure 4.9: To DO

The ring phase occurs once the fragments have reached uniformity, which takes around two to three days depending on the fragmentation event, and is a continuation of the ellipsoid phase. As such, the methodology and assumptions for both of these phases are the same.

Since these phases occur on a short time span and inter-fragment gravitational interactions are negligible, two-body dynamics drive the equations of motion throughout this phase between an individual fragment and the

Earth. As such, propagating the fragments during these phases is a special case of the two-body problem. While the two-body problem is well characterized, it is not computationally efficient enough to perform long duration and accurate propagations. As such, we reframe the problem as propagating Kepler orbits.

Kepler orbits are orbits in which no perturbations of inter gravitational interactions are considered, a special case of the two-body problem. Framing the problem this way lends itself naturally to using Keplerian elements. For Kepler orbits, only the orbits' anomalies will be changing over time which reduces the number of computations that need to be performed and aids in maintaining accuracy.

This benefit is a result of **Kepler's Equation** which is expressed in terms of the mean anomaly as

$$M = E - e\sin(E) = \sqrt{\frac{\mu}{a^3}}\,(t - t_0) \tag{28}$$

where E is the eccentric anomaly, e is the eccentricity, μ is the standrad gravitational parameter, a is the semi-major axis, t is some time in the future, t_0 is the current time. Since we have an analytic expression for the mean anomaly, we do not need to use any integration methods for propagating Kepler orbits forward in time which results in the aforementioned computational efficiencies.

The next step is to find the corresponding eccentric and true anomalies from the propagated mean anomaly. The eccentric anomaly is found from the Kepler equation; however, there is no closed-form solution given the mean anomaly. As such, numerical integration must be used to find the eccentric anomaly. The processes for performing the integration of Kepler's equation are the subject of many other research papers due to the importance it plays in orbit propagation. It is not included in this paper for conciseness, but a python implementation of one of the methods can be found in the appendix in section (TO DO INSERT REFERENCE TO APPENDIX SECTION HERE)(Maybe talk about the new methods in this area that do not require using integration).

Kepler's equation has another form which enables us to convert the resulting eccentric anomaly to the true anomaly. This form is given as

$$\nu = 2\arctan\left(\sqrt{\frac{1+e}{1-e}}\tan\frac{E}{2}\right). \tag{29}$$

Since we now have expressions for how the three anomalies will change over time for Kepler orbits, we can accurately propagate the debris until the ring formation phase is completed. However, we now need to have a method to detect once the ring formation phase is completed to switch to performing the propagation for future phases. A visualization of end result of the ring formation phase is provided in figure 4.10.

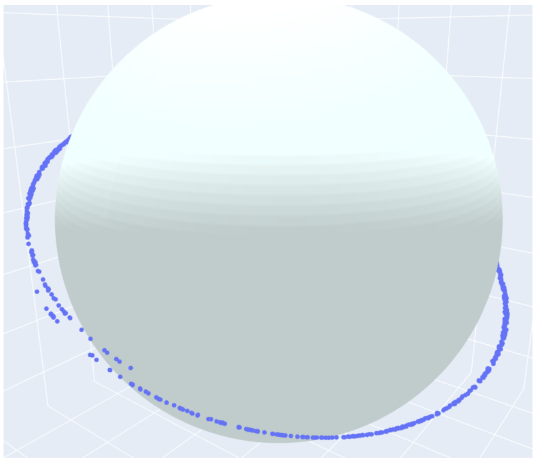


Figure 4.10: To DO

4.2 Transition to Toroid and Band phase

Detecting the end of the ring formation phase is crucial in propagating the debris cloud. When transitioning from the ring phase to the toroid and band phases, additional forces such as drag must be considered. Additionally, these new phases take a much longer amount of time to form. Both of these factors result in needing to switch to a new propagation method. As such, detecting when the ring has formed allows us to switch to this new propagation method.

To asses whether the system has formed a roughly uniform ring we monitor the number of debris passing through a certain region of space as a function of time. A visualization of how this process works is provided in figure 4.11.

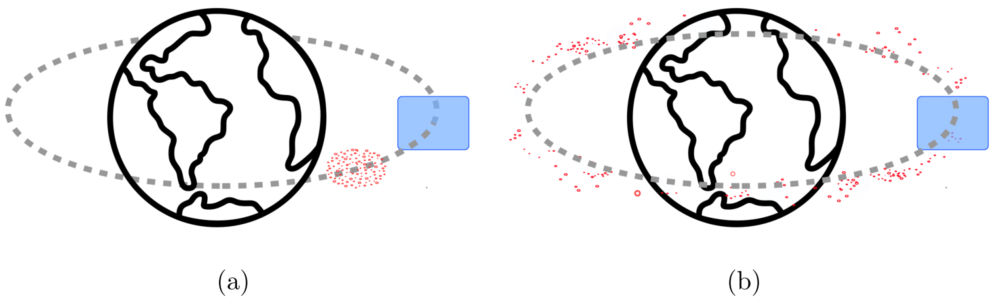
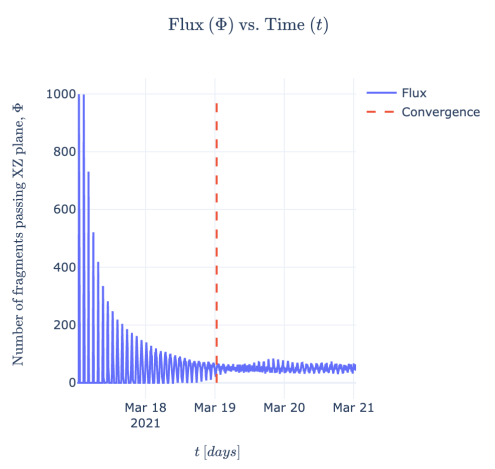
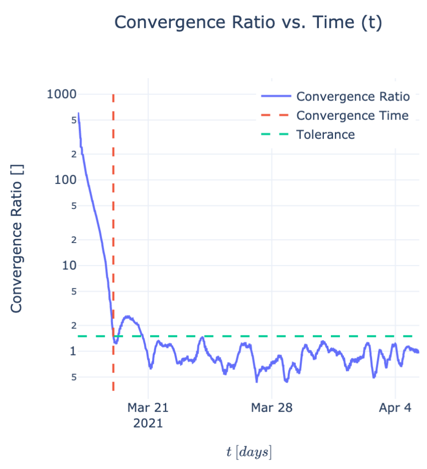


Figure 4.11: Measuring the flux at the time of the ellipsoid phase **(a)** and measuring the flux near the completion of the ring phase **(b)**

More specifically, to measure the fragments' spread we will define a particle-based flux. This is accomplished by creating an xz plane in the equatorial coordinate system and detecting when particles have switched from one side of the plane to the other. Plotting the results will show peaks when the fragments pass through, which will converge to some value as the fragments become uniformly spread out.



(a)



(b)

Figure 4.12: The flux of the fragments as a function of time **(a)** and the convergence ratio of the flux as a function of time **(b)**

Plotting the flux over time shows these expected peaks and what seems to be it converging to some value. However, the data is quite noisy and we need a concrete method to determine when the fragments will be distributed uniformly. As such, we need to develop a method to test for when the data has converged. A property that is true of all uniform distributions is that the variance approximately equals the mean. As such, we can define a converge ratio using the variance and mean of the flux. When the convergence ratio is within some defined tolerance we can and test for when the flux has become uniform and as such the conclusion of the ring formation phase. It should be noted that the ring never reaches true uniformity which results in osculating flux. An example of how the flux and convergence ratio evolves with time is shown in figure 4.12.

4.3 Toroid and Band phase

For the evolution of the cloud to continue, we now must consider orbital perturbations that will cause the orbits to change over time. Beginning to consider perturbations such as drag and J2 will enable the ring shape of the cloud to evolve towards the Toroid band.

4.3.1 Aerodynamic Drag

A significant perturbation that causes changes in the fragments' orbits is **atmospheric drag**. Atmospheric drag acts on a fragment as a result of molecules in the atmosphere colliding with it's surface and is a natural consequence of conservation of momentum.

4.3.2 Atmospheric Models

To model the effects of drag, a suitable atmospheric model must be selected. An atmospheric model tells us information about the air pressure, air density, wind speeds, e.t.c. at varying locations in the atmosphere. However, it would be cumbersome and computationally inefficient to utilize a model that contains all of this information. To enable efficiency varying models of drag tend to focus on a select few variables rather than creating a universal "best" model that considers all relevant factors [8].

Choosing a model that best fits a given use case comes down to the desired criterion for speed, accuracy, and applicability. For example the DAMAGE orbital debris model assumes a rotating, oblate atmosphere with density and density scale height values taken from the 1972 COSPAR International Reference Atmosphere (CIRA) (Hugh G. Lewis, 2012). A detailed exploration of various atmospheric models is given by (Gaposchkin and Coster (1988), but for the purposes of this paper, we will be focusing on a model called the **Exponential Atmospheric Model**. This is a a static model that assumes a spherically symmetric distribution of particles, where the density decays exponentially with increasing altitude [8].

In the Exponential Atmospheric Model the density ρ varies according to

$$\rho = \rho_0 \, e^{-\frac{h_{ellp} - h_0}{H}} \tag{30}$$

where ρ_0 is a reference density that is used with h_0, a reference altitude, h_{ellp} the actual altitude above the ellipsoid, and H which is a scale height.

The reference density and reference altitude are tabulated values that come from sources such as the U.S. Standard Atmosphere and CIRA. The scale height is a value used to ensure continuity throughout ρ. The combination of the U.S. Standard Model and CIRA will yield moderately accurate results for general purposes and as such will utilized in for computing the force of drag [8]. [1]

The air density, ρ, as a function of altitude is shown above. Note that the y axis is given as a log scale due to the rapidly increasing density of air at high altitudes as the distance to Earth shrinks.

4.3.3 Effects of Drag on Orbital Elements

The aerodynamic drag on an object is typically expressed in the following form

$$F_D = \frac{1}{2m} \rho v^2 \, C_D A \tag{31}$$

[1]The atmospheric model implementation, including the tabulated values that are used, are included in the appendix of this paper

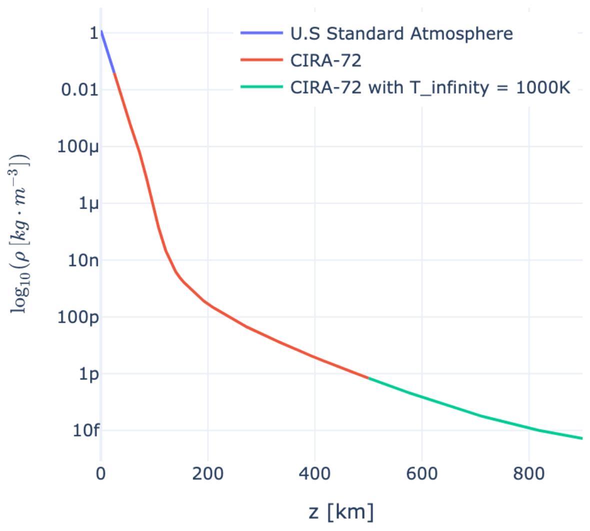


Figure 4.13: The atmospheric density of Earth as a function of altitude according to the Exponential Atmospheric Model

where ρ is the density of the fluid, v is the speed of the object relative to the fluid, C_D is an experimentally determined dimensionless number, A is the cross sectional area, and m is the mass of the satellite. A is the relevant cross section, determined by the method used to measure C_D.

Drag is a force that will cause an acceleration that opposes the direction of motion of an object. As such, drag produces a similar effect to a retrograde thrust which enables aerobraking, a useful orbital maneuver that can be performed around planetary bodies with an atmosphere. Similarly, for orbital debris drag is the predominant force behind changing the semi-major axis and eccentricity, gradually causing the debris to have a lower perigee.

As a result of the density increasing exponentially as altitude decreases, the effects of drag create a form of a feedback loop. A satellite experiences drag, which lowers its orbit, which in turn causes it to experience more drag. This will continue until a satellite is eventually deorbited.

The expressions for the effects of drag on orbital elements are derived by King-Hele and cover three different ranges of eccentricities(King-Hele 1987).

$$\frac{da}{dt} = \begin{cases} -\frac{C_D A}{M}\sqrt{\mu_E a\rho}\exp(-\frac{a-R_h}{H})[I_0 + 2eI_1 + \\ \frac{3}{4}e^2(I_0 + I_2) + \frac{e^3}{4}(3I_1 + I_3)] & \text{for } 0.01 \leq e \leq 0.2 \\ -\frac{C_D A}{M}\sqrt{\mu_E a\rho}\exp(-\frac{a-R_h}{H})[I_0 + 2eI_1] & \text{for for } 0.001 \leq e < 0.01 \\ -\frac{C_D A}{M}\sqrt{\mu_E a\rho}\exp(-\frac{a-R_h}{H}) & \text{for } e < 0.001 \end{cases} \tag{32}$$

$$\frac{de}{dt} = \begin{cases} -\frac{C_D A}{M}\sqrt{\frac{\mu_E}{a}}\rho\exp(-\frac{a-R_h}{H}) & \text{for } 0.01 \leq e \leq 0.2 \\ -\frac{C_D A}{M}\sqrt{\frac{\mu_E}{a}}\rho\exp(-\frac{a-R_h}{H})[I_1 + \frac{e}{2}(I_0 + I_2)] & \text{for } 0.001 \leq e < 0.01 \\ 0 & \text{for } e < 0.001 \end{cases} \tag{33}$$

$I_n(z)$ represents the modified Bessel function with order n, where $z = \frac{ae}{H}$ and is given by

$$I_k(z) = \frac{1}{\pi}\int_0^{\pi} e^{z\cos(\theta)}\cos(k\theta)d\theta \qquad k \in Z.$$

Following the precedents of many other texts, we will be assuming that the fragments have a drag coefficient of 0.7.

Modifications were made by Frey et. al to find more appropriate boundary conditions and to increase the accuracy of each phase by including more terms of the series expansion.

The first modification to the King-Hele implementation is to introduce two functions, k_a and k_e, that are used for describing the rate of change of a and e in all eccentricity regimes. [2]

$$k_a = \delta\sqrt{\mu a}\rho(h_p)$$

$$k_e = k_a/a$$

For circular orbits, e = 0, the change in a and e can be solved using the following expression:

$$\frac{da}{dt} = -k_a$$

$$\frac{de}{dt} = 0$$

For low eccentric orbits, $e < e_b(a, H)$, a series expansion in e is performed and integrated using the first kind modified Bessel function as:

$$\frac{da}{dt} = -k_a * \exp(-z) * (\boldsymbol{e}^T * \boldsymbol{K_a^l} * \boldsymbol{I} + O(e^6)) \tag{34}$$

$$\frac{de}{dt} = -k_e * \exp(-z) * (\boldsymbol{e}^T * \boldsymbol{K_e^l} * \boldsymbol{I} + O(e^6)) \tag{35}$$

where:

$$\boldsymbol{e}^T = (1 \quad e \quad e^2 \quad e^3 \quad e^4 \quad e^5) \tag{36}$$

$$\boldsymbol{I}^T = (I_0 \quad I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \quad I_6) \tag{37}$$

$$\boldsymbol{K_a^l} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ \frac{21}{64} & 0 & \frac{28}{64} & 0 & \frac{7}{64} & 0 & 0 \\ 0 & \frac{30}{64} & 0 & \frac{15}{64} & 0 & \frac{3}{64} & 0 \end{bmatrix} \tag{38}$$

$$\boldsymbol{K_e^l} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & -\frac{5}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 \\ -\frac{5}{16} & 0 & -\frac{4}{16} & 0 & \frac{1}{16} & 0 & 0 \\ 0 & -\frac{18}{128} & 0 & -\frac{1}{128} & 0 & \frac{3}{128} & 0 \\ -\frac{18}{256} & 0 & -\frac{19}{256} & 0 & \frac{2}{256} & 0 & \frac{3}{256} \end{bmatrix} \tag{39}$$

The vast majority of the debris fall in the low eccentricity range, as such only the circular and low eccentricity expressions are implemented. The high eccentricity range has a similar formulation provided by [2]. [2]

4.3.4 Nodal Precession

Nodal precession is the precession of the orbital plane of a satellite around the rotational axis of the central body. This is caused due to the non-spherical nature of the rotating central body. The non-spherical nature is a result of the centrifugal force produced by the rotation which deforms the body, causing an equatorial bulge

As a result, the planetary body creates a non-uniform gravitational field that induces a torque on satellites.

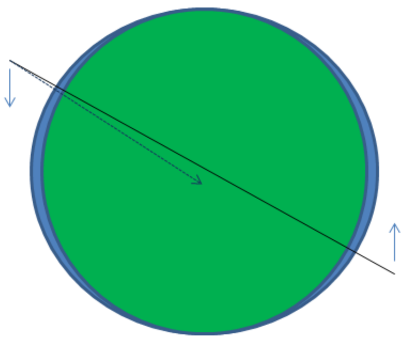


Figure 4.14: From Wikipedia Nodal Precession

Intuitively, it would appear that the torque would reduce the inclination of the orbit. However, due to the bulge, the gravitational force is not directed towards the center of the body but rather is offset toward the equatorial plane. As such, it causes torque-induced gyroscopic precession which causes the ascending and descending nodes to drift with time. This phenomenon is called **Nodal Precession**.

The effects of nodal precession on the ascending and descending nodes

[2]The implementation of drag, and other perturbations is contained in the appendix

is expressed by

$$\frac{d\omega}{dt} = \frac{3}{2} J_2 \frac{R_E^2}{p^2} \bar{n} (2 - \frac{5}{2} \sin^2(i))$$

$$\frac{d\Omega}{dt} = -\frac{3}{2} J_2 \frac{R_E^2}{p^2} \bar{n} \cos(i)$$

where $p = a(1 - e^2)$ is the semi-latus rectum of the orbit, $\bar{n} = \sqrt{\frac{\mu_E}{a^3}}$ is the mean motion, R_E is the radius of the Earth, and J_2 is the Earths second dynamic form factor.

The J2 term is a result of an infinite series equation that describes the perturbation effects of a rotating planetary body on the gravity of a planet. Each term of the series is denoted as J_n, however the J_2 term is more than 1000 times larger than the other terms (Ai soln. j2 perturbation). This is why the J2 effect is considered a relevant orbital perturbation for the evolution of an orbital debris cloud. For context, Earth's J_2 term has a value of 0.108×10^{-2} whereas its J_3 term has a value of 0.253×10^{-5}.

5 Analysis

Now that the critical components for debris cloud evolution are established, we can use the developed python implementation to gain insights into orbital debris's evolution. The first step for this is choosing a suitable scenario for the fragmentation event. Once the scenario is selected, the propagation methods defined in section 4 can be applied to study the debris throughout the various phases. This includes analyzing the estimated time for the fragments to deorbit and the fragments' spread over time.

5.1 Data Source

Selecting a realistic scenario to analyze can be difficult without prior knowledge of typical satellite orbits. As such, it is beneficial to perform analysis on existing satellites in low Earth orbit. This can be accomplished by utilizing a database of **Two-lines elements** (TLE's), which are a standardized way of describing information about a satellite's orbit.

The python implementation created for this researches utilizes a TLE database maintained by the website 'CelesTrak' to make all cataloged objects in orbit available for fragmentation events [3].

The analysis conducted in the following subsections is performed on two different satellites. The first is a satellite called OXP 1, which has an altitude of approximately 750 km and a 25-degree inclination. This is in a higher region of low Earth orbit and will allow us to analyze long-term changes on the debris cloud.

The second satellite is a Starlink satellite produced by SpaceX. This

[3]Additional information about TLE's and CelesTrak can be found on their https://www.celestrak.com/NORAD/elements/

satellite is at a much lower altitude, around (INSERT) km, and is a member of a satellite mega constellation currently being constructed with the goal of providing global internet coverage. It was selected as part of this analysis due to the increasing number of companies expressing interest in low Earth orbit mega-constellations. Additionally, the Starlink mega-constellation is quickly grown to having over one thousand satellites in orbit, with the end goal being 30,000 (ADD SOURCE). As such, it is relevant to study the debris cloud that would be produced in the case of a fragmentation event.

5.2 Decay Time

5.3 Spread

Index

References

[1] S. Barrows. EVOLUTION OF ARTIFICIAL SPACE DEBRIS CLOUDS.

[2] S. Frey, C. Colombo, and S. Lemmens. Extension of the King-Hele orbit contraction method for accurate, semi-analytical propagation of non-circular orbits. *Advances in Space Research*, 64(1): 1–17, July 2019. ISSN 0273-1177. doi: 10.1016/j.asr.2019.03.016. URL `https://www.sciencedirect.com/science/article/pii/S0273117719301978`.

[3] N. L. Johnson, P. H. Krisko, J. C. Liou, and P. D. Anz-Meador. NASA's new breakup model of evolve 4.0. *Advances in Space Research*, 28(9):1377–1384, Jan. 2001. ISSN 0273-1177. doi: 10.1016/S0273-1177(01)00423-9. URL `http://www.sciencedirect.com/science/article/pii/S0273117701004239`.

[4] P. H. Krisko. Proper Implementation of the 1998 NASA Breakup Model. *Orbital Debris Quarterly News*, 15(4):4,5, Oct. 2011. URL `http://orbitaldebris.jsc.nasa.gov/`.

[5] F. Letizia. *Space debris cloud evolution in Low Earth Orbit.* PhD thesis, temp, Feb. 2016.

[6] J. C. Liou. Orbital Debris Modeling. URL `https://ntrs.nasa.gov/citations/20120003286`.

[7] A. Rossi. NASA Breakup Model Implementation Comparison of results. *temp*, page 29, 2021.

[8] D. A. Vallado and M. C. W. D. *Fundamentals of astrodynamics and applications.* Microcosm Press., 2013.

Appendices

Write a description of what each section of the appendix covers

A Flux

The first assumption is that by neglecting inter gravitational forces, the motion of an individual fragment is independent of the motion of all other fragments. (ANOTHER ASSUMPTION HERE)

If we assume that each fragment crossing the plane is uniformally distributed in time, picking the last N points that will pass in some amount of time greater than 1 period. Call the total fragments crossing during that time M.

The probability a fragment crossing at time t_i is $\frac{1}{N}$. Thus the probability of m fragments crossing between time t_i and t_{i+1} is defined as follows

$$P = C_N^m * (\frac{1}{N})^m * (\frac{N-1}{N})^{N-m} \tag{40}$$

Using this expression, we can define the probability of n fragments crossing as:

$$P_n = C_N^n * P^n * (1-P)^N \tag{41}$$

Notice that this is a **Binomial distribution** which makes sense given the assumptions that were made about the distribution of the fragments.

From this, we can compute the mean number of fragments crossing, n, as

$$< n >= \sum_{n=0}^{N} nP_n = P * N = \frac{N}{M} \tag{42}$$

Similarly, we can compute the variance as

$$< (n- < n >)^2 >= \sum_{n=0}^{N} (n - \frac{N}{m})^2 * P_n = P * N(1-P) \tag{43}$$

With N, M being much larger than 1 we get a variation approximately equal to the mean.

We now have enough information to define a convergence ratio. If we divide the variation by the mean over a given interval, it will approximately equal 1 if the number of fragments crossing the plane during that interval is uniform.

B Code

B.1 Breakup Model

Listing 1: Perturbation Differential Equations Implementation

```
import numpy as np
import scipy
from enum import IntEnum

debris_category = IntEnum('Category', 'rb sc soc')
from numba import njit, prange

""" ------------------ Mean ------------------ """
def make_mean_AM(debris_type):

    def RB_mean_AM(lambda_c):

        mean_am_1 = np.empty_like(lambda_c)
        mean_am_2 = np.empty_like(lambda_c)

        mean_am_1[lambda_c<=-0.5] = -0.45
        I = (lambda_c>-0.5) & (lambda_c<0)
        mean_am_1[I] = -0.45 - (0.9*(lambda_c[I] +0.5))
        mean_am_1[lambda_c>=0] = -0.9

        mean_am_2.fill(-0.9)

        return np.array([mean_am_1,mean_am_2])

    def SC_mean_AM(lambda_c):
        mean_am_1 = np.empty_like(lambda_c)
        mean_am_2 = np.empty_like(lambda_c)

        mean_am_1[lambda_c<=-1.1] = -0.6
        I = (lambda_c>-1.1) & (lambda_c<0)
        mean_am_1[I] = -0.6 - (0.318*(lambda_c[I] +1.1))
        mean_am_1[lambda_c>=0] = -0.95

```

```python
        mean_am_2[lambda_c<=-0.7] = -1.2
        I = (lambda_c>-0.7) & (lambda_c<-0.1)
        mean_am_2[I] = -1.2 - (1.333*(lambda_c[I] + 0.7))
        mean_am_2[lambda_c>=-0.1] = -2.0

        return np.array([mean_am_1,mean_am_2])

    def SOC_mean_AM(lambda_c):

        mean_am_1 = np.empty_like(lambda_c)
        mean_am_2 = np.empty_like(lambda_c)

        mean_am_1[lambda_c<=-1.75] = -0.3
        I = (lambda_c>-1.75) & (lambda_c<-1.25)
        mean_am_1[I] = -0.3 - (1.4*(lambda_c[I] +1.75))
        mean_am_1[lambda_c>=-1.25] = -1.0

        mean_am_2.fill(0)
        return np.array([mean_am_1,mean_am_2])

    if debris_type == debris_category.rb:
        return RB_mean_AM
    elif debris_type == debris_category.sc:
        return SC_mean_AM
    else:
        return SOC_mean_AM

""" ----------------- Standard Deviation ----------------- """

def make_standard_dev_AM(debris_type):

    def RB_std_dev_AM(lambda_c):

        std_dev_1 = np.empty_like(lambda_c)
        std_dev_2 = np.empty_like(lambda_c)

        std_dev_1.fill(0.55)

        std_dev_2[lambda_c<=-1.0] = 0.28
        I = (lambda_c>-1.0) & (lambda_c<0.1)
        std_dev_2[I] = 0.29 - (0.1636*(lambda_c[I] +1))
        std_dev_2[lambda_c>=0.1] = 0.1

        return np.array([std_dev_1,std_dev_1])


    def SC_std_dev_AM(lambda_c):

        std_dev_1 = np.empty_like(lambda_c)
        std_dev_2 = np.empty_like(lambda_c)

        std_dev_1[lambda_c<=-1.3] = 0.1
        I = (lambda_c>-1.3) & (lambda_c<-0.3)
        std_dev_1[I] = 0.1 + (0.2*(lambda_c[I] +1.3))
        std_dev_1[lambda_c>=-0.3] = 0.3

        std_dev_2[lambda_c<=-0.5] = 0.5
        I = (lambda_c>-0.5) & (lambda_c<-0.3)
        std_dev_2[I] = 0.5 - ((lambda_c[I] + 0.5))
        std_dev_2[lambda_c>=-0.3] = 0.3

        return np.array([std_dev_1,std_dev_1])


    def SOC_std_dev_AM(lambda_c):
        std_dev_1 = np.empty_like(lambda_c)
        std_dev_2 = np.empty_like(lambda_c)

        std_dev_1[lambda_c<=-3.5] = 0.2
        I = (lambda_c>-3.5)
        std_dev_1[I] = 0.2 + (0.1333*(lambda_c[I] +3.5))

        std_dev_2.fill(0)

        return np.array([std_dev_1,std_dev_1])

    if debris_type == debris_category.rb:
        return RB_std_dev_AM
    elif debris_type == debris_category.sc:
        return SC_std_dev_AM
    else:
        return SOC_std_dev_AM

""" ----------------- Alpha ----------------- """
def alpha_AM(lambda_c, debris_type):
    def RB_alpha_AM(lambda_c):
        alpha = 1
        # dev1 rule
        if lambda_c <= -1.4:
            alpha = 1
        elif (lambda_c > -1.4 and lambda_c < 0):
            alpha = 1 - (0.3571*(lambda_c + 1.4))
        else:
            alpha = 0.5
        return alpha

    def SC_alpha_AM(lambda_c):
        alpha = 1
        # dev1 rule
        if lambda_c <= -1.95:
            alpha = 0
        elif (lambda_c > -1.95 and lambda_c < 0.55):
            alpha = 0.3 + (0.4*(lambda_c + 1.2))
        else:
            alpha = 1
        return alpha

    def SOC_alpha_AM(lambda_c):
        # Is not used by SOC, for saftey returning 1
        alpha = 1
        return alpha

    if debris_type == debris_category.rb:
        return RB_alpha_AM(lambda_c)
    elif debris_type == debris_category.sc:
        return SC_alpha_AM(lambda_c)
    else:
        return SOC_alpha_AM(lambda_c)

alpha_AM = np.vectorize(alpha_AM)
```

```python

""" ----------------- Distribution A/M ----------------- """
def distribution_AM(lambda_c, debris_type):

    N = len(lambda_c)
    lambda_c = np.array(lambda_c)

    mean_factory = make_mean_AM(debris_type)
    std_dev_factor = make_standard_dev_AM(debris_type)

    mean_preSwitch = np.array(mean_factory(lambda_c))
    std_dev_preSwitch = np.array(std_dev_factor(lambda_c))

    alpha = np.array(alpha_AM(lambda_c, debris_category.rb)) # This takes a long time
    switch = np.random.uniform(0,1, N)

    if debris_type == debris_category.rb or debris_type == debris_category.sc:

        means = np.empty(N)
        I,J = switch<alpha, switch>=alpha
        means[I] = mean_preSwitch[0, I]
        means[J] = mean_preSwitch[1, J]

        devs = np.empty(N)
        devs[I] = std_dev_preSwitch[0, I]
        devs[J] = std_dev_preSwitch[1, J]

        return np.random.normal(means, devs, N)

    else:
        means = mean_preSwitch[0]
        devs = std_dev_preSwitch[0]

        return np.random.normal(means, devs, N)

""" ----------------- Area ----------------- """
def avg_area(L_c):

    A = np.copy(L_c)
    I = A < 0.00167 #(m)
    A[I] = 0.540424 * A[I]**2
    I = A >= 0.00167 #(m)
    A[I] = 0.556945 * A[I]**2.0047077

    return A



""" ----------------- Mean ----------------- """
@njit()
def mean_deltaV(kai, explosion):
    if explosion == True:
        return (0.2 * kai) + 1.85
    else:
        # Is a collision
        return (0.9 * kai) + 2.9

""" ----------------- Standard Deviation ----------------- """
def std_dev_deltaV():
    return 0.4

""" ----------------- Distribution delta V ----------------- """
@njit()
def distriNormale(mu,sigma,x):
    p = (1/(sigma*np.sqrt(2*np.pi))*np.exp(-1/2.*((x-mu)/sigma)**2))
    return p

@njit()
def distriDeltaVExpl(nu,chi):
    mu = 0.2*chi + 1.85
    return distriNormale(mu,0.4,nu)

@njit( parallel=True )
def distribution_deltaV(chi, v_c, explosion=False):
        N = len(chi)
        result = np.empty_like(chi)
        progress = 0

        for i in prange(N):
            mean = mean_deltaV(chi[i], explosion)
            dev  = 0.4
            x = np.random.rand()
            dv =x*1.3*v_c
            dist = distriDeltaVExpl(np.log10(dv),chi[i])
            y = np.random.rand()
            while y > dist:
                x = np.random.rand()
                dv = x*1.3*v_c
                dist =  distriDeltaVExpl(np.log10(dv),chi[i])
                y = np.random.rand()
            result[i] = dist
        return result

#     N = len(chi)
#     mean = mean_deltaV(chi, explosion)
#     dev  = std_dev_deltaV()
# # print(np.mean(mean),dev)
#     max_itr = 15000
#     i = 0
#     base = 10
#     centered = np.random.normal(0, dev, N)
#     I = np.nonzero(base**(mean+centered)>1.3*v_c)[0]
#     n = len(I)
#     while n != 0 and i<max_itr:
#         centered[I] = np.random.normal(0, dev, n)
#         #I = np.nonzero(base**(mean+centered)>1.3*v_c)[0]
#         J = np.nonzero(base**(mean[I] + centered[I])>1.3*v_c)[0]
#         I = I[J]
#         n = len(I)
#         i+=1
#     centered[I] = np.log10(1.3*v_c)
#     result = base**centered
#     return result

""" ----------------- Unit vector delta V ----------------- """
def unit_vector(N):
    vectors = np.random.normal(0, 1, np.array([N, 3]))
    vectors /= np.sqrt((vectors**2).sum(axis=1))[:, None]
```

```python
    return vectors

def velocity_vectors(N, target_velocity, velocities):
    unit_vectors = unit_vector(N)
    velocity_vectors = velocities[:, None] * unit_vectors
    return target_velocity + velocity_vectors

from numpy.random import uniform

""" ----------------- Num. Fragments & Char. Length-----------------
    """
def number_fragments(l_characteristic, m_target, m_projectile, v_impact
    , is_catastrophic, debris_type, explosion):
    # Defining reference Mass
    if explosion == True: # Can be multiplied by scaling factor S
        print("explosion")
        return 6*(l_characteristic)**(-1.6)
    else:
        m_ref = 0
        if is_catastrophic:
            m_ref = m_target + m_projectile
        else:
            m_ref = m_projectile * (v_impact)**2
        return 0.1 * ((m_ref)**0.75) * l_characteristic**(-1.71)


def characteristic_lengths(m_target, m_projectile, v_impact,
    is_catastrophic, debris_type, explosion):
    bins = np.geomspace(0.001, 1, 100)
    N_fragments = number_fragments(bins, m_target, m_projectile,
    v_impact, is_catastrophic, debris_type, explosion)
    N_per_bin = np.array(N_fragments[:-1] - N_fragments[1:]).astype(int
    )
    L_c = np.concatenate([uniform(bins[i], bins[i+1], size=N_per_bin[i
    ]) for i in range(len(bins) - 1)])

    return L_c

def fragmentation(m_target, m_projectile, v_impact, is_catastrophic,
    debris_type, explosion):
```

```python

    prelim_L_c = characteristic_lengths(m_target, m_projectile, v_impact, is_catastrophic, debris_type, explosion)
    prelim_lambda_c = np.log10(prelim_L_c)
    prelim_areas = avg_area(prelim_L_c)
    prelim_AM = np.array(distribution_AM(prelim_lambda_c, debris_type))
    prelim_masses = prelim_areas / 10**prelim_AM

    if explosion == True:

        unaccounted_mass = m_target - np.sum(prelim_masses)

        n_large_deb = np.random.randint(2, 8) # Pick 2-8 pieces of deb > 1m to spread out the rest of the mass

       # Using 10**-4 to enure endpoints are not included
        mass_range = np.linspace(10**-4, (unaccounted_mass - 10**-4), 10**4) # Create mass range, will use `n_large_deb` to split into sections
        ranges = np.sort(np.random.choice(mass_range, n_large_deb - 1, replace=False))
        ranges = np.concatenate([[0],ranges,[unaccounted_mass]])

        # Note adding zero for subtraction to work (correct dims) then dropping it afterward
        mass_per_deb = np.concatenate((ranges[1:],np.zeros(1))) - ranges
        mass_per_deb = np.resize(mass_per_deb, mass_per_deb.size - 1)


        # For L_c > 1, A/M Distribution is basically deterministic, therefore will just use avg value, can get using np.inf
        assumed_AM_factory = make_mean_AM(debris_type)
        assumed_len = np.ones(mass_per_deb.shape)
        assumed_AM = assumed_AM_factory(assumed_len)

        # Each mean has two possible values, randomly pick one of them for each piece of deb
        AM_choices = np.random.choice([0,1], len(mass_per_deb), replace=True)
        assumed_AM = 10**np.array([assumed_AM[AM_choices[i], i] for i in range(assumed_AM.shape[1])])

        # mass * AM = A(L_c), therefore can reverse Area function for L_c
        area = mass_per_deb * assumed_AM
        found_L_c = np.sort((area / 0.556945)**(1/2.0047077)) # Inversing the Area function defined above
        found_lambda_c = np.log10(found_L_c)
        found_areas = avg_area(found_L_c)

        found_AM = np.array(distribution_AM(found_lambda_c, debris_type))
        found_masses = found_areas / assumed_AM # Using assumed A/M since A/M is a distribution and could get diff values.

        L_c = np.concatenate([prelim_L_c, found_L_c])
        areas = np.concatenate([prelim_areas, found_areas])
        masses = np.concatenate([prelim_masses, found_masses])
        AM = np.concatenate([prelim_AM, assumed_AM])

        return L_c, areas, masses, AM
    else:
        # Is a collision
        if is_catastrophic == True:

            unaccounted_mass = (m_target + m_projectile) - np.sum(prelim_masses)
            # Put the rest of the mass in many fragments in last bin
            deposit_bin = (np.geomspace(0.001, 1, 100)[-1] + np.geomspace(0.001, 1, 100)[-2])/2



            n_large_deb = np.random.randint(15, 50) # Pick 2-8 pieces of deb > 1m to spread out the rest of the mass

           # Using 10**-4 to enure endpoints are not included
            mass_range = np.linspace(10**-4, (unaccounted_mass - 10**-4), 10**4) # Create mass range, will use `n_large_deb` to split into sections
            ranges = np.sort(np.random.choice(mass_range, n_large_deb - 1, replace=False))
            ranges = np.concatenate([[0],ranges,[unaccounted_mass]])

            # Note adding zero for subtraction to work (correct dims) then dropping it afterward
            found_masses = np.concatenate((ranges[1:],np.zeros(1))) - ranges
            found_masses = np.resize(found_masses, found_masses.size - 1)

            found_L_c = np.ones_like(found_masses) * deposit_bin
            found_areas = avg_area(found_L_c)
            found_AM = found_areas / found_masses

            L_c = np.concatenate([prelim_L_c, found_L_c])
            areas = np.concatenate([prelim_areas, found_areas])
            masses = np.concatenate([prelim_masses, found_masses])
```

B.2 Coordinate Transforms

Listing 2: Perturbation Differential Equations Implementation

```
import numpy as np
from numpy import cross
from numba import njit as jit, prange
from numpy.core.umath import cos, sin, sqrt
from numpy.linalg import norm

"""
Converting from Keplerian to Cartesian
--------------------------------------
Helpful links:
    https://downloads.rene-schwarz.com/download/M001-Keplerian_Orbit_Elements_to_Cartesian_State_Vectors.pdf
    https://gitlab.eng.auburn.edu/evanhorn/orbital-mechanics/blob/a850737fcf4c43e295e79decf2a3a88acbbba451/Homework1/kepler.py

```

```python
Notes: Code was modified from Poliastro source, elements.py
"""

mu = 398600.4418 #km^3s^-2

@jit
def rotation_matrix(angle, axis):

    c = cos(angle)
    s = sin(angle)

    if axis == 0:
        return np.array([[1.0, 0.0, 0.0], [0.0, c, -s], [0.0, s, c]])
    elif axis == 1:
        return np.array([[c, 0.0, s], [0.0, 1.0, 0.0], [s, 0.0, c]])
    elif axis == 2:
        return np.array([[c, -s, 0.0], [s, c, 0.0], [0.0, 0.0, 1.0]])
    else:
        raise ValueError("Invalid axis: must be one of 'x', 'y' or 'z'")

@jit
def rv_pqw(k, p, ecc, nu):
    pqw = np.array([[cos(nu), sin(nu), 0], [-sin(nu), ecc + cos(nu), 0]]) * np.array(
        [[p / (1 + ecc * cos(nu))], [sqrt(k / p)]]
    )
    return pqw

@jit
def coe_rotation_matrix(inc, raan, argp):
    """Create a rotation matrix for coe transformation"""
    r = rotation_matrix(raan, 2)
    r = r @ rotation_matrix(inc, 0)
    r = r @ rotation_matrix(argp, 2)
    return r

@jit
def coe2rv(k, p, ecc, inc, raan, argp, nu):

    pqw = rv_pqw(k, p, ecc, nu)
    r, v = rv_pqw(k, p, ecc, nu)
    rm = coe_rotation_matrix(inc, raan, argp)
    ijk = pqw @ rm.T

    return ijk

# ks = np.array([a, e_mag, i, Omega, omega, M, nu, p_semi, T, E])
@jit(parallel=True)
def coe2rv_many_new(state, mu=mu):
    inc = np.deg2rad(state[2, :])
    raan = np.deg2rad(state[3, :])
    argp = np.deg2rad(state[4, :])
    nu = np.deg2rad(state[6, :])
    p = state[7, :]
    ecc = state[1, :]

    n = nu.shape[0]
    rr = np.zeros((n, 3), dtype=np.float64)
    vv = np.zeros((n, 3), dtype=np.float64)

    for i in prange(n):
        rr[i, :], vv[i, :] = (coe2rv(mu, p[i], ecc[i], inc[i], raan[i],
     argp[i], nu[i]))

    return rr, vv

@jit(parallel=True)
def coe2rv_many(k, p, ecc, inc, raan, argp, nu):
    inc = np.deg2rad(inc)
    raan = np.deg2rad(raan)
    argp = np.deg2rad(argp)
    nu = np.deg2rad(nu)

    n = nu.shape[0]
    rr = np.zeros((n, 3), dtype=np.float64)
    vv = np.zeros((n, 3), dtype=np.float64)

    for i in prange(n):
```

```python
        rr[i, :], vv[i, :] = (coe2rv(k, p[i], ecc[i], inc[i], raan[i], argp[i], nu[i]))

    return rr, vv

"""
Converting from Cartesian to Keplerian
--------------------------------------
"""

def rv2coe(r, v, mu):
    ''' Converts a position, `r`, and a velocity, `v` to the set of keplerian elements.'''
    """
    Parameters
    ----------
    r: array (3, n)
       Position of the body in 3 dim. Measured using center of Earth as origin. (m)
    v: array (3, n)
       Velocity of the body in 3 dim relative to Earth. (m / s)

    Returns
    -------
    ks: array (9, n)
        An array containing all of the keplerian elements + extra useful info.
        a: Float
        e: Float
        i: Float
        Omega: Float
        omega: Float
        nu: Float
        p_semi: Float
        T: Float
    """

    def testAngle(test, angle):
        """Checks test for sign and returns corrected angle"""
        angle *= 180./np.pi
        I = test < 0
        angle[I] = 360. - angle[I]
        return angle

    r_hat = np.divide(r, norm(r, axis=1)[:, None])

    # Orbital momentum vector, p
    p = np.cross(r, v)

    # Eccentricty vector, e, and magnitude, e_mag (used freq)
    e = (np.cross(v, p) / mu) - r_hat
    e_mag = norm(e, axis = 1)

    # Longitude of the ascending node, Omega
    Omega_hat = np.cross(np.array([0, 0, 1])[None, :], p)
    Omega = np.arccos(Omega_hat[:,0]/norm(Omega_hat, axis=1))
    Omega = testAngle(Omega_hat[:, 1], Omega)

    # Argument of periapsis, omega
    omega = np.arccos(np.sum(Omega_hat*e, axis=1) / (norm(Omega_hat, axis=1)*norm(e, axis=1)))
    B = e[:,2] < 0
    omega[B] = 2*np.pi - omega[B]
    omega *= 180. / np.pi

    # True Anomaly, nu
    nu = np.arccos( np.sum(e*r, axis=1) / (norm(e, axis=1) * norm(r, axis=1)))
    B = np.sum(r*v, axis=1)<0
    nu[B] = 2*np.pi - nu[B]
    nu *= 180. / np.pi

    # Inclination, i
    i = np.arccos(p[:, 2] / norm(p, axis=1))*180./np.pi

    # Eccentric anomaly, E
    E = 2*np.arctan(np.tan(np.deg2rad(nu)/2) / np.sqrt((1 + e_mag)/(1 - e_mag)))

    # Mean anomaly, M
    M = np.mod(E - e_mag * np.sin(E), 2*np.pi)
    M *= 180./np.pi

    # Semi-Major axis, a
    R = norm(r, axis =1)
    V = norm(v, axis =1)
    a = 1/((2 / R) - (V*V / mu))

    # Semi-parmeter, p_semi
    p_semi = norm(p, axis=1)**2 / mu

    # Orbital period
    T = 2*np.pi * np.sqrt(a**3 / mu)

    # Keplerian State + Extra info
    ks = np.array([a, e_mag, i, Omega, omega, M, nu, p_semi, T, E])

    return ks
```

B.3 Orbit Propagation

Listing 3: Perturbation Differential Equations Implementation

```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit as jit, prange
from numpy import pi, sin, cos, sqrt
from scipy import integrate
from scipy.special import iv

# User defined libearayr
import planetary_data as pd
import CoordTransforms as ct
import Aerodynamics as aero

def null_perts():
    return {
        'J2': False,
```

```python
        'aero': False,
        'moon_grav': False,
        'solar_grav': False
    }

class OrbitPropagator:

    def __init__(self, states0, A, M, tspan, dt, rv=False, cb=pd.earth,
     perts=null_perts()):

        # Need to add support for initializing with radius and velocity
        if rv:
            self.states = 0

        else:
            self.states = states0

        # Setting the areas and masses
        self.A = A
        self.M = M

        # Integration information
        self.tspan = tspan
        self.dt = dt

        # Central body properties
        self.cb = cb

        # Defining perturbations being considered
        self.perts = perts

        # Defining constants for aerodynamic drag
        if self.perts['aero']:
            self.K_a = np.matrix([[1 , 0, 0, 0 ,0 ,0, 0],
                       [0 , 2, 0, 0, 0, 0, 0],
                       [3/4, 0, 3/4, 0, 0, 0, 0],
                       [0 , 3/4, 0, 1/4, 0, 0, 0],
                       [21/64, 0, 28/64, 0, 7/64, 0, 0],
                       [0 , 30/64, 0, 15/64, 0, 3/64, 0]])

            self.K_e = np.matrix([[0 , 1, 0, 0, 0, 0, 0],
                   [1/2, 0 , 1/2, 0, 0, 0, 0],
                   [0 , -5/8, 0, 1/8, 0, 0, 0],
                   [-5/16, 0, -4/16, 0, 1/16, 0, 0],
                   [0 , -18/128, 0, -1/128, 0, 3/128, 0],
                   [-18/256, 0, -19/256, 0, 2/256, 0, 3/256]])

    def cartesian_representation(self):
    # Returns the cartesian state representation of states for vis. purposes
        N_t = self.states.shape[0]
        N_frag = self.states.shape[2]
        cartesian_states = np.empty(shape=(N_t, 2, N_frag, 3))

        for i in prange(self.states.shape[0]):
            cartesian_states[i, :, :] = ct.coe2rv_many_new(self.states[i, :, :])

        return cartesian_states


    def diffy_q(self, t, state):
        e, a, i, Omega, omega = state.reshape(5, len(self.A))
        N_f = len(self.A)

        # Central body information
        mu     = self.cb['mu']
        radius = self.cb['radius'] #[m]
        J2 = self.cb['J2']

        # Local variables
        delta_e = np.zeros_like(e)
        delta_a = np.zeros_like(a)
        delta_i = np.zeros_like(i)
        delta_Omega = np.zeros_like(Omega)
        delta_omega = np.zeros_like(omega)


        # Current orbital information
        peri = a * (1 - e) #[m]
        p    = a * (1 - e**2) #[m] (Semi parameter)
        n    = np.sqrt(mu / a**3) # (Mea motion)

        ############### Drag effects ###############
        if self.perts['aero']:
            h_p = (peri - radius) #[m]
            rho = aero.atmosphere_density(h_p/1e3) #[kg * m^-3]
            H = aero.scale_height(h_p/1e3) * 1e3 #[m]

            z = a*e / H
            Cd = 0.7
            tilt_factor =1
            delta = Cd * (self.A[0] * tilt_factor) / self.M[0]

            e_T = np.array([np.ones_like(e), e, e**2, e**3, e**4, e**5])
            I_T = np.array([ iv(i, z) for i in range(7)])
            k_a = delta * np.sqrt(mu * a) * rho
            k_e = k_a / a

            delta_e = np.zeros_like(e)
            delta_a = np.zeros_like(a)

            # CASE e < 0.001
            delta_e = np.zeros_like(e)
            delta_a = -k_a

            # CASE e>= 0.001
            I = e>= 0.001
            trunc_err_a = a[I]**2 * rho[I] * np.exp(-z[I]) * iv(0, z[I]) * e[I]**6
            trunc_err_e = a[I] * rho[I] * np.exp(-z[I]) * iv(1, z[I]) * e[I]**6

            transform_e = e_T.T.dot(self.K_e) * I_T
            coef_e = np.array([ transform_e[i,i] for i in range(N_f)])[I]

            transform_a = e_T.T.dot(self.K_a) * I_T
            coef_a = np.array([ transform_a[i,i] for i in range(N_f)])[I]

            delta_e[I] = -k_e[I] * np.exp(-z[I]) * (coef_e + trunc_err_e)
            delta_a[I] = -k_a[I] * np.exp(-z[I]) * (coef_a + trunc_err_a)

            delta_e[np.isnan(delta_e)] = 0
            delta_a[np.isnan(delta_a)] = 0

            # Deorbit check
            J = h_p < 100*1e3
            delta_a[J] = 0
            delta_e[J] = 0

        ###############  J2 effects  ###############
        if self.perts['J2']:
            base = (3/2) * self.cb['J2'] * (radius**2/p**2) * n
            i = np.deg2rad(i)
            delta_omega = base * (2 - (5/2)*np.sin(i)**2)
            delta_Omega = -base * np.cos(i)
            delta_omega = np.rad2deg(delta_omega) % 360
            delta_Omega = np.rad2deg(delta_Omega) % 360

        return np.concatenate((delta_e, delta_a, delta_i, delta_Omega, delta_omega))

    # Performing a regular propagation, i.e. w/ perturbations
    def propagate_perturbations(self):

        # Initial states
        a0, e0, i0, Omega0, omega0 = self.states[-1, :5, :]
        y0  = np.concatenate((e0, a0, i0, Omega0, omega0))

        # Propagation time
        T_avg = np.mean(self.states[-1, 8, :])
        times = np.arange(self.tspan[0], self.tspan[-1], self.dt)
        output = integrate.solve_ivp(self.diffy_q, self.tspan, y0, t_eval = times)
```

```python

        # Unpacking output (Need to drop first timestep as sudden introduction of drag causes discontinuities)
        N_f = len(self.A)
        de = output.y[0:N_f, 1:]
        da = output.y[N_f:2*N_f, 1:]
        di = output.y[2*N_f:3*N_f, 1:]
        dOmega = output.y[3*N_f:4*N_f, 1:]
        domega = output.y[4*N_f:, 1:]
        dnu = np.random.uniform(low=0., high=360., size=domega.shape)
        dp = da * (1 - de**2)

        # Results
        return de, da, di, dOmega, domega, dnu, dp

    # Performing a Keplerian propagation, i.e. w/o perturbations
    def propagate_orbit(self):

        times      = np.arange(self.tspan[0], self.tspan[-1], self.dt)

        # Mean anomaly rate of change
        M_dt       = sqrt(self.cb['mu']/self.states[0, :]**3)

        Nd         = len(M_dt)
        Nt         = len(times)

        # Mean anomaly over time
        M_t        = np.deg2rad(self.states[5, :, None]) + M_dt[:, None] * times[None, :]
        M_t        = np.rad2deg(np.mod(M_t, 2*pi))

        # Eccentric anomaly over time. Note need to use E_t in rad, thus convert to deg after using it in
        # x1 and x2
        E_t        = np.empty(shape=(Nd, Nt), dtype=np.float32)
        E_t        = M2E(self.states[1], np.deg2rad(M_t))

        x1         = sqrt(1 + self.states[1, :])[:, None] * sin(E_t / 2)
        x2          = sqrt(1 − self.states[1, :])[:, None] * cos(E_t / 2)
        E_t         = np.rad2deg(E_t)

        # True anomaly over time
        nu_t        = (2*np.arctan2(x1, x2) % (2*pi))
        nu_t        = np.rad2deg(nu_t).T

        n_times     = nu_t.shape[0]
        states      = np.empty(shape = (n_times, self.states.shape[0], self.states.shape[1]))

        for i in prange(n_times):
            state = self.states.copy()
            state[6, :] = nu_t[i, :]
            states[i] = state

        # Update internal states
        self.states = states



# Modified from OrbitalPy.utilities
@jit(parallel=True, fastmath=True)
def M2E(e_deb, M_t, tolerance=1e−14):
#Convert mean anomaly to eccentric anomaly.
#Implemented from [A Practical Method for Solving the Kepler Equation ][1]
#by Marc A. Murison from the U.S. Naval Observatory
#[1]: http://murison.alpheratz.net/dynamics/twobody/KeplerIterations_summary.pdf
    n_deb = M_t.shape[0]
    n_times = M_t.shape[1]

    E_t = np.empty_like(M_t)

    for i in prange(n_deb):
        e = e_deb[i]
        for j in prange(n_times):
            M = M_t[i, j]

            MAX_ITERATIONS = 100
            Mnorm = np.mod(M, 2 * pi)
            E0 = M + (-1 / 2 * e ** 3 + e + (e ** 2 + 3 / 2 * cos(M) * e ** 3) * cos(M)) * sin(M)
            dE = tolerance + 1
            count = 0
            while dE > tolerance:
                t1 = cos(E0)
                t2 = -1 + e * t1
                t3 = sin(E0)
                t4 = e * t3
                t5 = -E0 + t4 + Mnorm
                t6 = t5 / (1 / 2 * t5 * t4 / t2 + t2)
                E = E0 - t5 / ((1 / 2 * t3 - 1 / 6 * t1 * t6) * e * t6 + t2)
                dE = np.abs(E - E0)
                E0 = E
                count += 1
                if count == MAX_ITERATIONS:
                    print('Did not converge, increase number of iterations')
            E_t[i, j] = E
    return E_t
```

Listing 4: Perturbation Propagator

```

            delta_e[I] = -k_e[I] * np.exp(-z[I]) * (coef_e + trunc_err_e)
            delta_a[I] = -k_a[I] * np.exp(-z[I]) * (coef_a + trunc_err_a)

            delta_e[np.isnan(delta_e)] = 0
            delta_a[np.isnan(delta_a)] = 0

            # Deorbit check
            J = h_p < 100*1e3
            delta_a[J] = 0
            delta_e[J] = 0
```

```

        ###############  J2 effects  ###############
        if self.perts['J2']:
            base = (3/2) * self.cb['J2'] * (radius**2/p**2) * n
            i = np.deg2rad(i)
            delta_omega = base * (2 - (5/2)*np.sin(i)**2)
            delta_Omega = -base * np.cos(i)
            delta_omega = np.rad2deg(delta_omega) % 360
            delta_Omega = np.rad2deg(delta_Omega) % 360

        return np.concatenate((delta_e, delta_a, delta_i, delta_Omega,
    delta_omega))

    # Performing a regular propagation, i.e. w/ perturbations
    def propagate_perturbations(self):

        # Initial states
        a0, e0, i0, Omega0, omega0 = self.states[-1, :5, :]
```