

**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 they poses just as grand of a threat as the other types. Orbital debris is 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

<b>Abstract</b>	<b>iii</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Symbols</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Methodology . . . . .	2
1.3 Roadmap . . . . .	3
<b>2 Modeling Satellite Breakups</b>	<b>4</b>
2.1 The NASA Standard Satellite Breakup Model . . . . .	4
2.2 Implementing the NASA Breakup Model . . . . .	6
2.2.1 Characteristic Length and Number of fragments . . .	6
2.2.2 Area to Mass Distribution . . . . .	10
2.2.3 Change in Velocity Distribution . . . . .	13
2.3 Validating the Implementation . . . . .	15
<b>3 State Representation</b>	<b>17</b>
3.1 Orbital State vectors and their advantages . . . . .	18
3.2 Keplerian elements and their advantages . . . . .	20
<b>4 Debris Cloud Evolution</b>	<b>25</b>
4.1 Ellipsoid and Ring Phase . . . . .	25
4.2 Transition to Toroid and Band phase . . . . .	28
4.3 Toroid and Band phase . . . . .	30
4.3.1 Aerodynamic Drag . . . . .	31
4.3.2 Atmospheric Models . . . . .	31
4.3.3 Effects of Drag on Orbital Elements . . . . .	33
4.3.4 Nodal Precession . . . . .	37
<b>5 Analysis</b>	<b>39</b>
5.1 Data Source . . . . .	39
5.2 Decay Time . . . . .	40
5.3 Spread . . . . .	40
<b>Appendices</b>	<b>44</b>
<b>A Flux</b>	<b>45</b>

<b>B</b>	<b>Code</b>	<b>47</b>
B.1	Breakup Model . . . . .	47
B.2	Coordinate Transforms . . . . .	57
B.3	Orbit Propagation . . . . .	62

## List of Figures

2.1	The algorithm used for producing the characteristic length distribution used in the NASA breakup model . . . . .	7
2.2	A sketch illustrating the process for determining the number of fragments in each characteristic length bin . . . . .	8
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. . . . .	10
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. . . . .	12
3.5	A diagram illustrating the Keplerian elements related to the orbital plane intersecting a reference plane. Data Source: Orbital elements Wikipedia page . . . . .	21
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 . . . . .	21
3.7	The relationship between the eccentricity and the resulting conic section. Data Source: <a href="https://3.bp.blogspot.com">https://3.bp.blogspot.com</a> . . . . .	22
4.8	The three phases of debris cloud evolution. Data Source: Letizia . . . . .	25
4.9	To DO . . . . .	26
4.10	To DO . . . . .	28
4.11	Measuring the flux at the time of the ellipsoid phase <b>(a)</b> and measuring the flux near the completion of the ring phase <b>(b)</b> . . . . .	29
4.12	The flux of the fragments as a function of time <b>(a)</b> and the convergence ratio of the flux as a function of time <b>(b)</b> . . . . .	30
4.13	The atmospheric density of Earth as a function of altitude according to the Exponential Atmospheric Model . . . . .	32
4.14	From Wikipedia Nodal Precession . . . . .	37
5.15	From Wikipedia Nodal Precession . . . . .	40

## List of Tables

1	(Data Source: NASA Breakup Model Implementation Comparison of results, A. Rossi, 24th IADC Meeting April 2006)	
	.....	16

## List of Symbols

- $A$  Cross-sectional area [m<sup>2</sup>].
- $H$  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] .
- $\Delta v$  Relative velocity [km/s].
- $\Omega$  Longitude of the ascending node [rad or deg].
- $\Phi$  Flux of fragments.
- $\mathcal{N}$  Normal distribution .
- $\mu_E$  Earth's planetary gravitational constant [km<sup>3</sup>/ s<sup>2</sup>].
- $\mu$  Mean value.
- $\nu$  True anomaly [rad or deg].
- $\omega$  Argument of the periapsis [rad or deg].
- $\rho$  Atmosphere density [kg / m<sup>3</sup>].
- $a$  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 the U.K. prime minister whose spokesperson 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 in chapter 3.

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 it

**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 collision [6]. A 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 Johnson et al. in 2001 [3] but was later clarified by Kristo in 2011 [4].

The NASA standard breakup model uses 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. In this paper, I will be focusing on how the collision fragments are generated, but it should be noted that the explosion

case often involves slightly different parameters.

## 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, denoted  $L_c$ , 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 in-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.

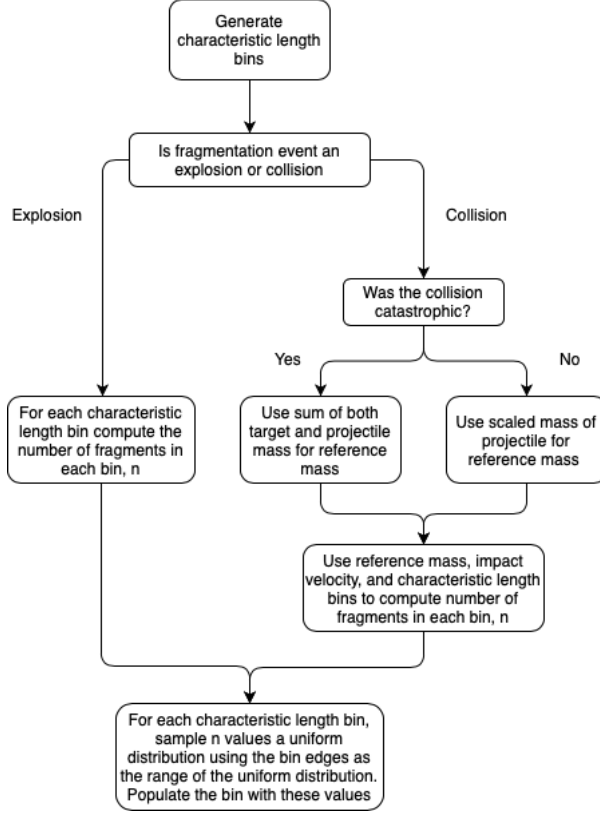


Figure 2.1: The algorithm used for producing the characteristic length distribution used in the NASA breakup model

The creation of the characteristic lengths was not given in the original specification of the NASA breakup model by Johnson [3] 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. In 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. [3]. The pieces of debris that are larger than 10cm will be handled separately to ensure that the breakup model

conserves mass. Figure 2.2 below illustrates the process of using bins to determine the number of fragments of each size.

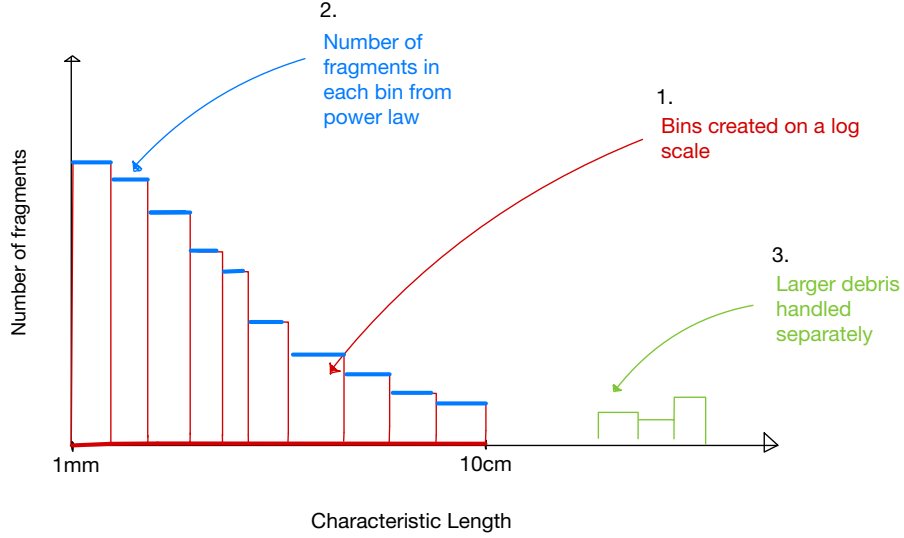


Figure 2.2: A sketch illustrating the process for determining the number of fragments in each characteristic length bin

To determine the number of fragments in each characteristic length bin, we acknowledge 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

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

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 and anti-satellite tests, a value of  $S$  between 0.1 and 1 was found to fit the experimental data better [4].

In the case of a collision, a distinction must be made between 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

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

where  $M_e$  is defined as follows:

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

In equation 3,  $M_t$  is the target mass,  $M_p$  is the projectile mass, and  $V_c$  is the relative impact velocity between the target and the projectile.

Figure 2.3 shows sample results from ODAP to illustrate the characteristic length distribution described above. Additionally, the relevant ODAP function can be found in the appendix in section (X).

### Characteristic Length Distribution

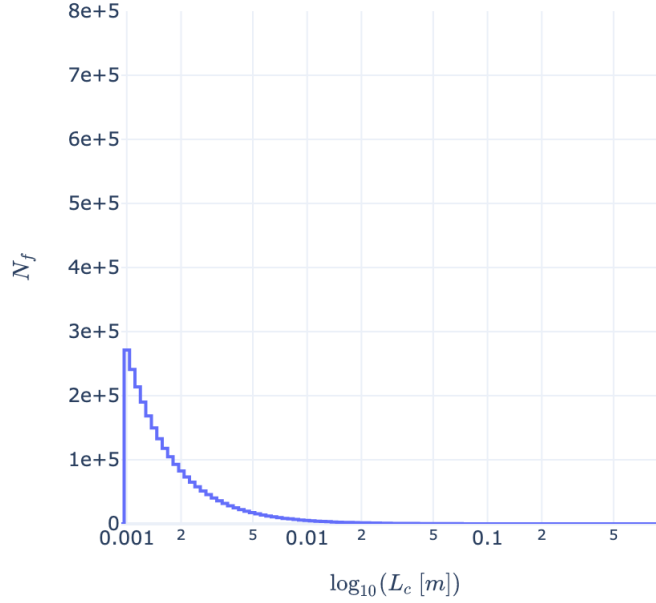


Figure 2.3: The characteristic length distribution produced by the python implementation of a catastrophic collision involving a rocket body with a mass of 1000kg, a projectile of mass 10kg, and an impact velocity of 10 km/s.

#### 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 determine the mass of each fragment of debris. The discrete distributions were found by using a  $\chi^2$  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 ODAP.

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

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

$D_{A/M}$  is the distribution function of  $\chi$  as a function of  $\lambda_c$ , where

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

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

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

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

$$\text{and} \quad (8)$$

$$\sigma_{A/M} = \begin{cases} 0.2, & \lambda_c \leq -3.5 \\ 0.2 + 0.1333(\lambda_c + 3.5) & \lambda_c > -3.5 \end{cases} \quad (9)$$

Every fragment of debris has a corresponding  $A/M$  distribution since both  $\mu_{A/M}$  and  $\sigma_{A/M}$  are functions of  $\lambda_c$ . 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.

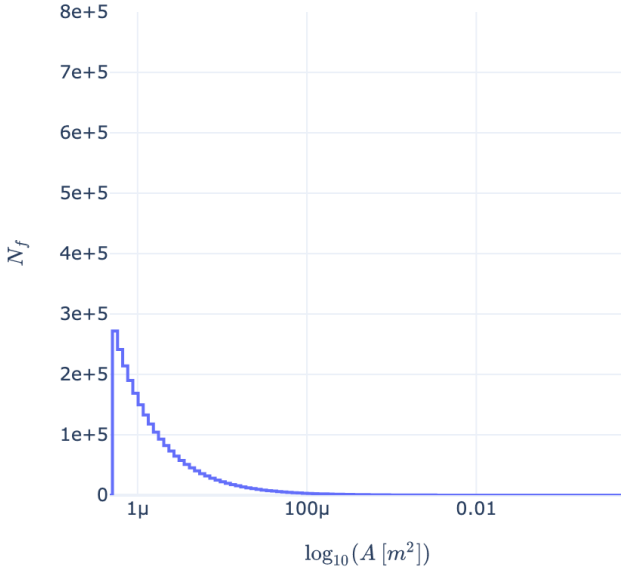
The  $A/M$  ratio alone does not provide enough information to determine both the area and mass of a fragment. As such, the average cross-sectional

area,  $A$ , can also be obtained through a one-to-one correspondence with  $L_c$  using the following expression [3]:

$$A_x = \begin{cases} 0.540424 * L_c^2 & \text{where } L_c < 0.00167 \text{ m} \\ 0.556945 * L_c^{2.0047077} & \text{where } L_c \geq 0.00167 \text{ m} \end{cases} \quad (10)$$

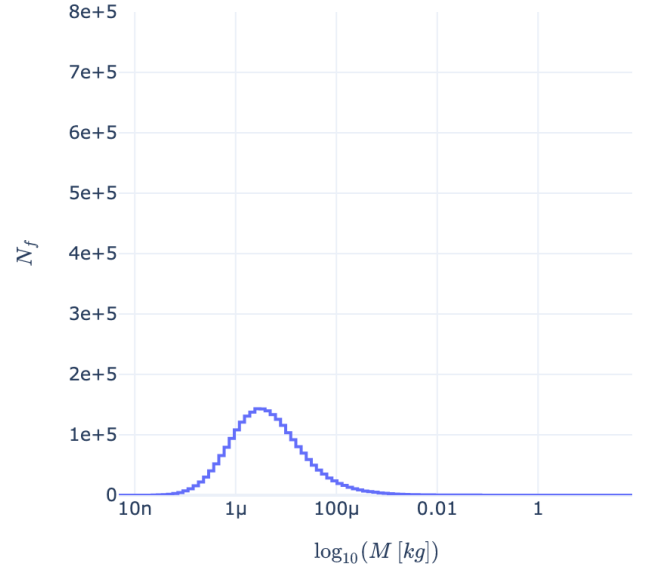
Utilizing both the  $A/M$  ratio and the cross sectional area  $A$ , we can now obtain the mass  $M$  easily using  $M = A_x/(A/M)$ . Figure (X) shows a simulated A/M distribution and average cross sectional area. The code to reproduce this is found in section (X).

Area distribution



(a)

Mass Distribution



(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,  $\Delta v$ , is modeled by log-normal distribution that is function of the  $A/M$  ratio by

$$D_{\Delta v} = \mathcal{N}(\mu_v(\chi), \sigma_v(\chi), \xi) \quad (11)$$

where

$$\xi = \log_{10}(\Delta v), \quad (12)$$

$$\chi = \log_{10}(A/M), \quad (13)$$

$$\mu_v(\chi) = 0.2\chi + 1.85, \quad (14)$$

$$\sigma_v(\chi) = 0.4, \quad (15)$$

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, ODAP's implementation of the velocity distribution often takes tens of thousands of iterations to complete. One or two fragments often 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 ODAP's implementation of the breakup model 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 with a projectile mass that weighs 10kg that has an impact velocity of 10km/s. The data was provided in a presentation given by Rossi et al. [7].

As shown in the table 1, the data from the various space agencies has some significant differences, especially for the characteristic length in the  $> 1\text{mm}$  range, despite 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 ODAP is to be within the original deviations of the data.

Model	Number of Fragments						
	Length				Mass	Area	Velocity
	>1mm	>1cm	>10cm	>1m	>1g	>1cm <sup>2</sup>	>100ms <sup>-1</sup>
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
<i>Mean</i>	478,663	8195	-	-	-	-	-
<i>Standard Deviation</i>	393,726	3813	-	-	-	-	-
<b><i>Python Implementation</i></b>	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,  $\Omega$ , the argument of periapsis,  $\omega$ , and the true anomaly,  $\nu$ .

Both of these methods have advantages and disadvantages for modeling orbital debris, and as such, it is often convenient to switch back and forth between them. 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

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

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

$$m_1 \vec{a}_1 = G \frac{m_1 m_2 \vec{r}}{|\vec{r}|^3} \quad (16)$$

$$\vec{a}_i = G \sum_{j \neq i} m_j \frac{\vec{r}_j - \vec{r}_i}{|\vec{r}_j - \vec{r}_i|^3} \quad (\text{More general}) \quad (17)$$

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 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 each 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 cap-

tured with this parameterization. More precisely, knowing 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 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). \*\*\*  
FIX

The reference body is called the primary body. In our case, this is the Earth, while the other body is called 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,  $\vec{v}$ , to define the **specific angular momentum** vector as

$$\vec{h} = \vec{r} \times \vec{v}. \quad (18)$$

Using the Earth's 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**—the plane created by tracing out the  $\vec{r}$  vector, which contains both the  $\vec{r}$  and  $\vec{v}$ . 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. The vector  $\vec{n}$  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

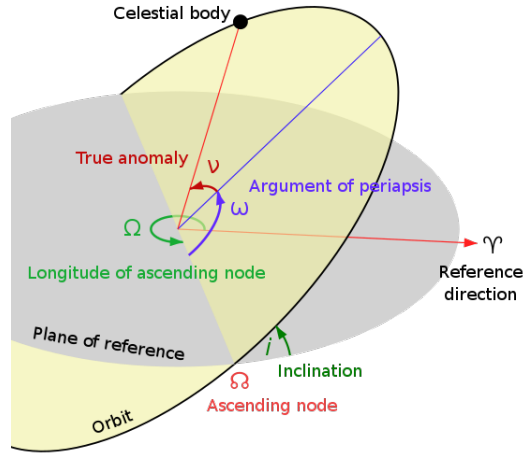


Figure 3.5: A diagram illustrating the Keplerian elements related to the orbital plane intersecting a reference plane. Data Source: Orbital elements Wikipedia page

momentum  $\vec{h}$ .

$$\vec{n} = \hat{k} \times \vec{h} \quad (19)$$

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.

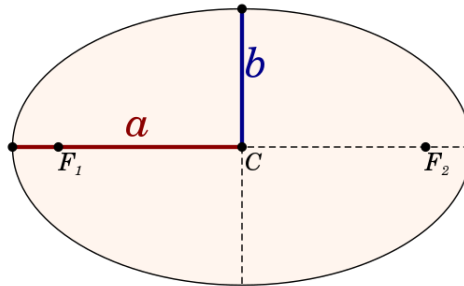


Figure 3.6: The semi-major axis,  $a$ , along with the semi-minor axis,  $b$ , and the foci of an ellipse,  $F_1$  and  $F_2$ . Data Source: Semi-major and semi-minor axis Wikipedia page

Using the vis-viva equation, the semi-major axis,  $a$ , can be defined as

$$a = \frac{1}{\frac{2}{|\vec{r}|} - \frac{|\vec{v}|^2}{\mu}}, \quad (20)$$

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

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:

$$\vec{e} = \left( \frac{|\vec{v}|^2}{\mu} - \frac{1}{|\vec{r}|} \right) \vec{r} - \frac{(\vec{r} \cdot \vec{v})}{\mu} \vec{v} \quad (21)$$

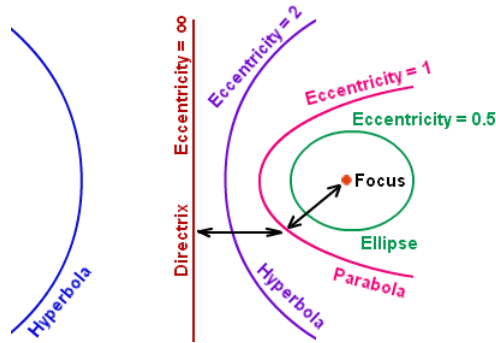


Figure 3.7: The relationship between the eccentricity and the resulting conic section. Data Source: <https://3.bp.blogspot.com>

The **inclination**,  $i$ , is the angle formed between the unit vector pointing

in the  $\hat{k}$  direction and can be calculated by

$$i = \arccos \frac{K_z}{|\vec{h}|}, \quad (22)$$

where  $\vec{h}$  is the angular momentum.

The **true anomaly**,  $\nu$ , 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.

$$\nu[rad] = \begin{cases} \arccos \frac{\vec{e} \cdot \vec{r}}{|\vec{e}| |\vec{r}|} & \text{for } \vec{r} \cdot \vec{v} \geq 0 \\ 2\pi - \arccos \frac{\vec{e} \cdot \vec{r}}{|\vec{e}| |\vec{r}|} & \text{otherwise} \end{cases} \quad (23)$$

The **eccentric anomaly**,  $E$ , is also used to measure an object's position in an orbit. The eccentric anomaly is defined using the magnitude of the eccentricity vector,  $e = |\vec{e}|$ , and the true anomaly as follows:

$$E = 2 \arctan \frac{\tan \frac{\nu}{2}}{\sqrt{\frac{1+e}{1-e}}} \quad (24)$$

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

$$\Omega[rad] = \begin{cases} \arccos \frac{n_x}{|\vec{n}|} & \text{for } n_y \geq 0 \\ 2\pi - \arccos \frac{n_x}{|\vec{n}|} & \text{for } n_y < 0. \end{cases} \quad (25)$$

The **argument of periapsis**,  $\omega$ , is the angle in the orbital plane that is

between the ascending node and the periapsis and is calculated using

$$\omega[rad] = \begin{cases} \arccos \frac{\vec{n} \cdot \vec{e}}{|\vec{n}| |\vec{e}|} & \text{for } e_z \geq 0 \\ 2\pi - \arccos \frac{\vec{n} \cdot \vec{e}}{|\vec{n}| |\vec{e}|} & \text{for } e_z < 0 \end{cases} \quad (26)$$

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

$$M = E - e \sin E \quad (27)$$

We can now parameterize each of the debris fragments using  $(e, a, i, \Omega, \omega, M)$ . 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

$$\frac{dM}{dt} = \sqrt{\frac{\mu}{a^3}}. \quad (28)$$

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, as illustrated in Figure 4.8. In addition to having different shapes, those phases differ in how long they last and the type of model used to describe them. For example, the ellipsoid and ring formations occur 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.

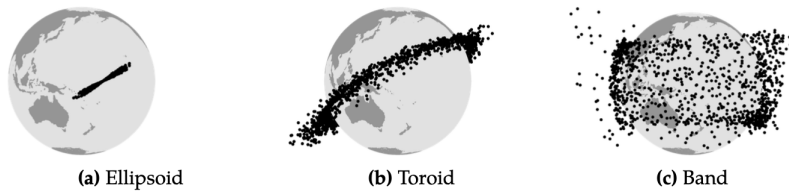


Figure 4.8: The three phases of debris cloud evolution. Data Source: Letizia

### 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 fragment has a different velocity in both direction and magnitude. This causes the debris to spread

out and form first an ellipsoid (Figure 4.9) then a complete ring around the Earth. As the debris cloud expands, the number of other satellites it can potentially impact grows. On the other hand, the cloud also gets sparser, and the probability of any specific satellite getting hit decreases. In other words, the odds of subsequent collision spread out as well.

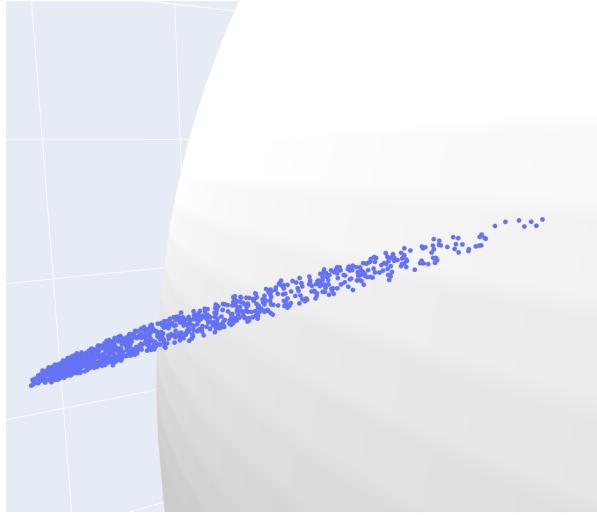


Figure 4.9: To DO

It takes two to three days for the debris to form a uniform ring around the Earth. On this time scale, weak effects such as air drag and gravitational forces between debris are negligible. As a result, each debris fragment orbits around the Earth as if the other fragments did not exist. From a practical point of view, we can use standard results for the two-body gravitational problem (CITE TEXTBOOK).

While the two-body problem is well characterized, it is not computationally efficient enough to perform long duration and accurate propagation. 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. Fram-

ing 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

$$M = E - e \sin(E) = \sqrt{\frac{\mu}{a^3}} (t - t_0), \quad (29)$$

where  $E$  is the eccentric anomaly,  $e$  is the eccentricity,  $\mu$  is the standard gravitational parameter,  $a$  is the semi-major axis,  $t$  is some time in the future, and  $t_0$  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. This 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 re-

sulting eccentric anomaly to the true anomaly. This form is given as

$$\nu = 2 \arctan \left( \sqrt{\frac{1+e}{1-e}} \tan \frac{E}{2} \right). \quad (30)$$

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.

## 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

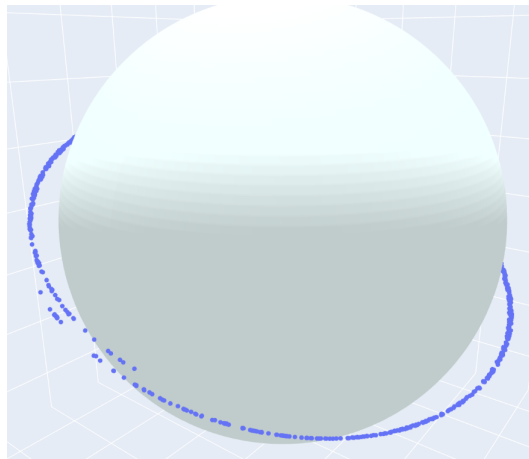


Figure 4.10: To DO

such, detecting when the ring has formed allows us to switch to this new propagation method.

To assess 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.

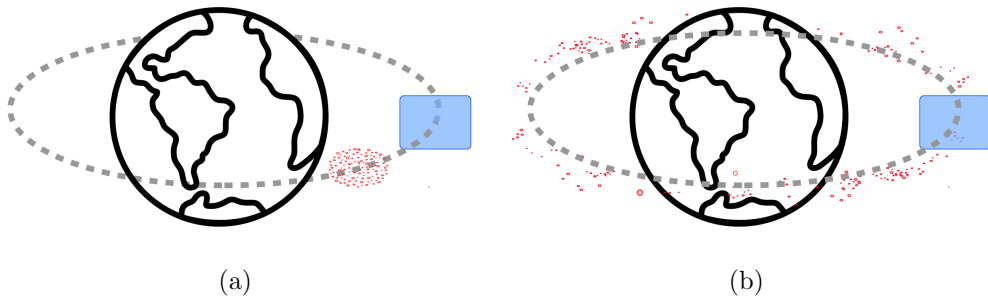


Figure 4.11: Measuring the flux at the time of the ellipsoid phase **(a)** and measuring the flux near the completion of the ring phase **(b)**

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.

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

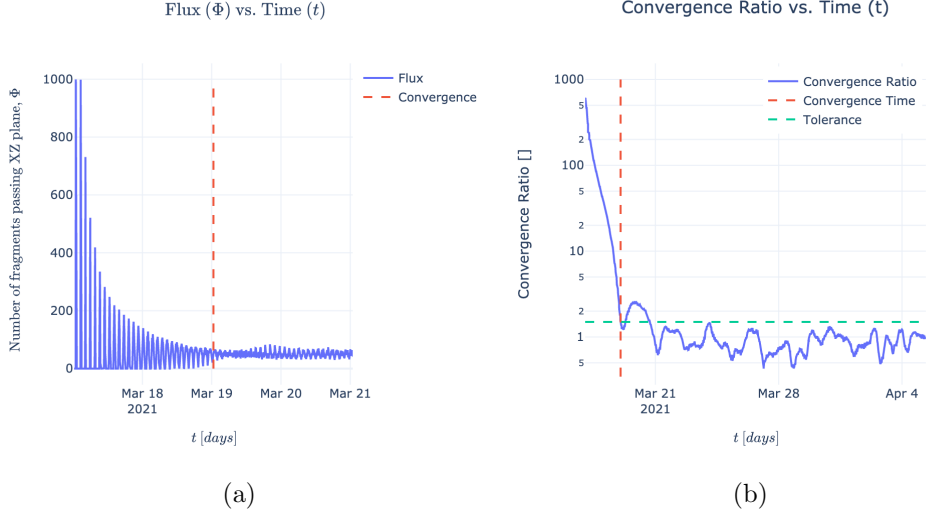


Figure 4.12: The flux of the fragments as a function of time **(a)** and the convergence ratio of the flux as a function of time **(b)**

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  $J_2$  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 its 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 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  $\rho$  varies according to

$$\rho = \rho_0 e^{-\frac{h_{ellp}-h_0}{H}} \quad (31)$$

where  $\rho_0$  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  $\rho$ . The combination of the U.S. Standard Model and CIRA will yield moderately accurate results for general purposes and as such will be utilized in for computing the force of drag [8].<sup>1</sup>

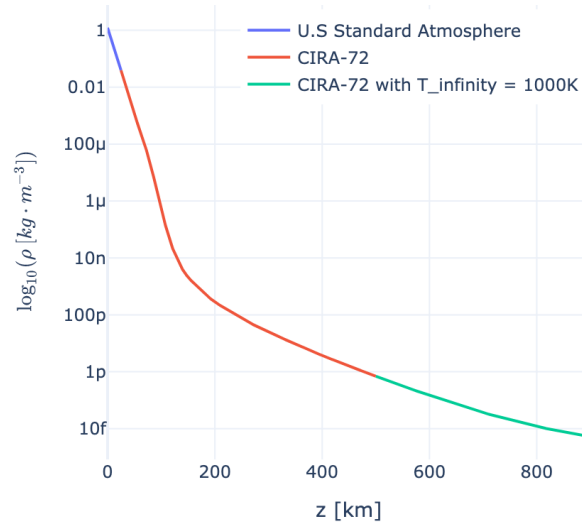


Figure 4.13: The atmospheric density of Earth as a function of altitude according to the Exponential Atmospheric Model

<sup>1</sup>The atmospheric model implementation, including the tabulated values that are used, are included in the appendix of this paper



The air density,  $\rho$ , 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

$$F_D = \frac{1}{2m} \rho v^2 C_D A \quad (32)$$

where  $\rho$  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).

$$\frac{da}{dt} = \begin{cases} -\frac{C_DA}{M} \sqrt{\mu_E a \rho} \exp\left(-\frac{a - R_h}{H}\right) [I_0 + 2eI_1 + \frac{3}{4}e^2(I_0 + I_2) + \frac{e^3}{4}(3I_1 + I_3)] & \text{for } 0.01 \leq e \leq 0.2 \\ -\frac{C_DA}{M} \sqrt{\mu_E a \rho} \exp\left(-\frac{a - R_h}{H}\right) [I_0 + 2eI_1] & \text{for } 0.001 \leq e < 0.01 \\ -\frac{C_DA}{M} \sqrt{\mu_E a \rho} \exp\left(-\frac{a - R_h}{H}\right) & \text{for } e < 0.001 \end{cases} \quad (33)$$

$$\frac{de}{dt} = \begin{cases} -\frac{C_DA}{M} \sqrt{\frac{\mu_E}{a}} \rho \exp\left(-\frac{a - R_h}{H}\right) & \text{for } 0.01 \leq e \leq 0.2 \\ -\frac{C_DA}{M} \sqrt{\frac{\mu_E}{a}} \rho \exp\left(-\frac{a - R_h}{H}\right) [I_1 + \frac{e}{2}(I_0 + I_2)] & \text{for } 0.001 \leq e < 0.01 \\ 0 & \text{for } e < 0.001 \end{cases} \quad (34)$$

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

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

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:

$$\frac{da}{dt} = -k_a * \exp(-z) * (\mathbf{e}^T * \mathbf{K}_a^l * \mathbf{I} + O(e^6)) \quad (35)$$

$$\frac{de}{dt} = -k_e * \exp(-z) * (\mathbf{e}^T * \mathbf{K}_e^l * \mathbf{I} + O(e^6)) \quad (36)$$

where:

$$\mathbf{e}^T = (1 \quad e \quad e^2 \quad e^3 \quad e^4 \quad e^5) \quad (37)$$

$$\mathbf{I}^T = (I_0 \quad I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \quad I_6) \quad (38)$$

$$\mathbf{K}_a^l = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{3}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 \\ \frac{21}{64} & 0 & \frac{28}{64} & 0 & \frac{7}{64} & 0 & 0 \\ 0 & \frac{30}{64} & 0 & \frac{15}{64} & 0 & \frac{3}{64} & 0 \end{bmatrix} \quad (39)$$

$$\mathbf{K}_e^l = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & -\frac{5}{8} & 0 & \frac{1}{8} & 0 & 0 & 0 \\ -\frac{5}{16} & 0 & -\frac{4}{16} & 0 & \frac{1}{16} & 0 & 0 \\ 0 & -\frac{18}{128} & 0 & -\frac{1}{128} & 0 & \frac{3}{128} & 0 \\ -\frac{18}{256} & 0 & -\frac{19}{256} & 0 & \frac{2}{256} & 0 & \frac{3}{256} \end{bmatrix} \quad (40)$$

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].<sup>2</sup>

#### 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.

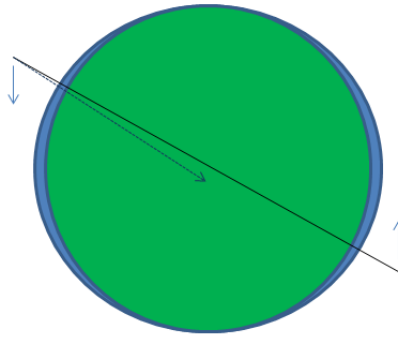


Figure 4.14: From Wikipedia Nodal Precession

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

---

<sup>2</sup>The implementation of drag, and other perturbations is contained in the appendix

is expressed by

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

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

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 Earth's second dynamic form factor.

The  $J_2$  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  $J_2$  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  $0.108 \times 10^{-2}$  whereas its  $J_3$  term has a value of  $0.253 \times 10^{-5}$ .

## 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 <sup>3</sup>.

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

---

<sup>3</sup>Additional information about TLE's and CelesTrak can be found on their <https://www.celstrak.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

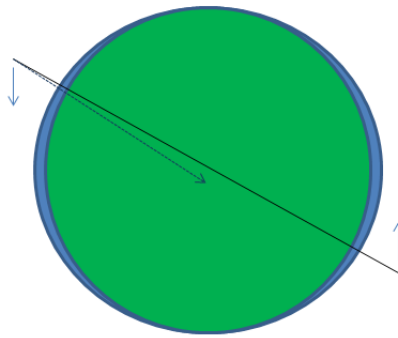


Figure 5.15: From Wikipedia Nodal Precession

## 5.3 Spread



# Index

- argument of periapsis, 23
- ascending node, 20
- atmospheric drag, 31
- descending node, 20
- Earth-Centered Inertial (ECI)
  - Coordinate system, 17
- eccentric anomaly, 23
- Exponential Atmospheric Model,
  - 31
- fundamental plane, 20
- inclination, 22
- Kepler orbits, 26
- Kepler's Equation, 27
- Keplerian elements, 17
- Keplers equation, 24
- line of nodes, 20
- longitude of the ascending node,
  - 23
- mean anomaly, 24
- Nodal Precession, 37
- Nodal precession, 37
- orbital eccentricity, 22
- orbital plane, 20
- orbital state vectors, 17
- semi-major axis, 21
- specific angular momentum, 20
- true anomaly, 23
- Two-lines elements, 39

## 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 uniformly 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

$$P = C_N^m * \left(\frac{1}{N}\right)^m * \left(\frac{N-1}{N}\right)^{N-m} \quad (41)$$

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

$$P_n = C_N^n * P^n * (1 - P)^{N-n} \quad (42)$$

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

$$\langle n \rangle = \sum_{n=0}^N n P_n = P * N = \frac{N}{M} \quad (43)$$

Similarly, we can compute the variance as

$$\langle (n - \langle n \rangle)^2 \rangle = \sum_{n=0}^N \left(n - \frac{N}{M}\right)^2 * P_n = P * N(1 - P) \quad (44)$$

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

```
1 import numpy as np
2 import scipy
3 from enum import IntEnum
4
5 debris_category = IntEnum('Category', 'rb sc soc')
6 from numba import njit, prange
7
8 """ ----- Mean ----- """
9 def make_mean_AM(debris_type):
10
11     def RB_mean_AM(lambda_c):
12
13         mean_am_1 = np.empty_like(lambda_c)
14         mean_am_2 = np.empty_like(lambda_c)
15
16         mean_am_1[lambda_c<=-0.5] = -0.45
17         I = (lambda_c>-0.5) & (lambda_c<0)
18         mean_am_1[I] = -0.45 - (0.9*(lambda_c[I] +0.5))
19         mean_am_1[lambda_c>=0] = -0.9
20
21         mean_am_2.fill(-0.9)
22
23         return np.array([mean_am_1,mean_am_2])
24
25     def SC_mean_AM(lambda_c):
26         mean_am_1 = np.empty_like(lambda_c)
27         mean_am_2 = np.empty_like(lambda_c)
28
29         mean_am_1[lambda_c<=-1.1] = -0.6
30         I = (lambda_c>-1.1) & (lambda_c<0)
31         mean_am_1[I] = -0.6 - (0.318*(lambda_c[I] +1.1))
32         mean_am_1[lambda_c>=0] = -0.95
33
```

```

34     mean_am_2[lambda_c<=-0.7] = -1.2
35     I = (lambda_c>-0.7) & (lambda_c<-0.1)
36     mean_am_2[I] = -1.2 - (1.333*(lambda_c[I] + 0.7))
37     mean_am_2[lambda_c>=-0.1] = -2.0
38
39     return np.array([mean_am_1,mean_am_2])
40
41 def SOC_mean_AM(lambda_c):
42
43     mean_am_1 = np.empty_like(lambda_c)
44     mean_am_2 = np.empty_like(lambda_c)
45
46     mean_am_1[lambda_c<=-1.75] = -0.3
47     I = (lambda_c>-1.75) & (lambda_c<-1.25)
48     mean_am_1[I] = -0.3 - (1.4*(lambda_c[I] +1.75))
49     mean_am_1[lambda_c>=-1.25] = -1.0
50
51     mean_am_2.fill(0)
52     return np.array([mean_am_1,mean_am_2])
53
54 if debris_type == debris_category.rb:
55     return RB_mean_AM
56 elif debris_type == debris_category.sc:
57     return SC_mean_AM
58 else:
59     return SOC_mean_AM
60
61 """ ----- Standard Deviation ----- """
62
63 def make_standard_dev_AM(debris_type):
64
65     def RB_std_dev_AM(lambda_c):
66
67         std_dev_1 = np.empty_like(lambda_c)
68         std_dev_2 = np.empty_like(lambda_c)
69
70         std_dev_1.fill(0.55)
71
72         std_dev_2[lambda_c<=-1.0] = 0.28
73         I = (lambda_c>-1.0) & (lambda_c<0.1)

```



```

74         std_dev_2[I] = 0.29 - (0.1636*(lambda_c[I] +1))
75         std_dev_2[lambda_c>=0.1] = 0.1
76
77         return np.array([std_dev_1, std_dev_1])
78
79
80     def SC_std_dev_AM(lambda_c):
81
82         std_dev_1 = np.empty_like(lambda_c)
83         std_dev_2 = np.empty_like(lambda_c)
84
85         std_dev_1[lambda_c<=-1.3] = 0.1
86         I = (lambda_c>-1.3) & (lambda_c<-0.3)
87         std_dev_1[I] = 0.1 + (0.2*(lambda_c[I] +1.3))
88         std_dev_1[lambda_c>=-0.3] = 0.3
89
90         std_dev_2[lambda_c<=-0.5] = 0.5
91         I = (lambda_c>-0.5) & (lambda_c<-0.3)
92         std_dev_2[I] = 0.5 - ((lambda_c[I] + 0.5))
93         std_dev_2[lambda_c>=-0.3] = 0.3
94
95         return np.array([std_dev_1, std_dev_1])
96
97
98     def SOC_std_dev_AM(lambda_c):
99         std_dev_1 = np.empty_like(lambda_c)
100         std_dev_2 = np.empty_like(lambda_c)
101
102         std_dev_1[lambda_c<=-3.5] = 0.2
103         I = (lambda_c>-3.5)
104         std_dev_1[I] = 0.2 + (0.1333*(lambda_c[I] +3.5))
105
106         std_dev_2.fill(0)
107
108         return np.array([std_dev_1, std_dev_1])
109
110     if debris_type == debris_category.rb:
111         return RB_std_dev_AM
112     elif debris_type == debris_category.sc:
113         return SC_std_dev_AM

```

```

114     else:
115         return SOC_std_dev_AM
116
117 """ ----- Alpha ----- """
118 def alpha_AM(lambda_c, debris_type):
119     def RB_alpha_AM(lambda_c):
120         alpha = 1
121         # dev1 rule
122         if lambda_c <= -1.4:
123             alpha = 1
124         elif (lambda_c > -1.4 and lambda_c < 0):
125             alpha = 1 - (0.3571*(lambda_c + 1.4))
126         else:
127             alpha = 0.5
128         return alpha
129
130     def SC_alpha_AM(lambda_c):
131         alpha = 1
132         # dev1 rule
133         if lambda_c <= -1.95:
134             alpha = 0
135         elif (lambda_c > -1.95 and lambda_c < 0.55):
136             alpha = 0.3 + (0.4*(lambda_c + 1.2))
137         else:
138             alpha = 1
139         return alpha
140
141     def SOC_alpha_AM(lambda_c):
142         # Is not used by SOC, for saftey returning 1
143         alpha = 1
144         return alpha
145
146     if debris_type == debris_category.rb:
147         return RB_alpha_AM(lambda_c)
148     elif debris_type == debris_category.sc:
149         return SC_alpha_AM(lambda_c)
150     else:
151         return SOC_alpha_AM(lambda_c)
152
153 alpha_AM = np.vectorize(alpha_AM)

```

```

154
155 """ ----- Distribution A/M ----- """
156 def distribution_AM(lambda_c, debris_type):
157
158     N = len(lambda_c)
159     lambda_c = np.array(lambda_c)
160
161     mean_factory = make_mean_AM(debris_type)
162     std_dev_factor = make_standard_dev_AM(debris_type)
163
164     mean_preSwitch = np.array(mean_factory(lambda_c))
165     std_dev_preSwitch = np.array(std_dev_factor(lambda_c))
166
167     alpha = np.array(alpha_AM(lambda_c, debris_category.rb)) # This
    takes a long time
168     switch = np.random.uniform(0,1, N)
169
170     if debris_type == debris_category.rb or debris_type ==
    debris_category.sc:
171
172         means = np.empty(N)
173         I,J = switch<alpha, switch>=alpha
174         means[I] = mean_preSwitch[0, I]
175         means[J] = mean_preSwitch[1, J]
176
177         devs = np.empty(N)
178         devs[I] = std_dev_preSwitch[0, I]
179         devs[J] = std_dev_preSwitch[1, J]
180
181         return np.random.normal(means, devs, N)
182
183     else:
184         means = mean_preSwitch[0]
185         devs = std_dev_preSwitch[0]
186
187         return np.random.normal(means, devs, N)
188
189 """ ----- Area ----- """
190 def avg_area(L_c):
191

```

```

192     A = np.copy(L_c)
193     I = A < 0.00167 # (m)
194     A[I] = 0.540424 * A[I]**2
195     I = A >= 0.00167 # (m)
196     A[I] = 0.556945 * A[I]**2.0047077
197
198     return A
199
200
201
202 """ ----- Mean ----- """
203 @jit()
204 def mean_deltaV(kai, explosion):
205     if explosion == True:
206         return (0.2 * kai) + 1.85
207     else:
208         # Is a collision
209         return (0.9 * kai) + 2.9
210
211 """ ----- Standard Deviation ----- """
212 def std_dev_deltaV():
213     return 0.4
214
215 """ ----- Distribution delta V ----- """
216 @jit()
217 def distriNormale(mu, sigma, x):
218     p = (1/(sigma*np.sqrt(2*np.pi))*np.exp(-1/2.*((x-mu)/sigma)**2))
219     return p
220
221 @jit()
222 def distriDeltaVExpl(nu, chi):
223     mu = 0.2*chi + 1.85
224     return distriNormale(mu, 0.4, nu)
225
226 @jit( parallel=True )
227 def distribution_deltaV(chi, v_c, explosion=False):
228     N = len(chi)
229     result = np.empty_like(chi)
230     progress = 0
231

```

```

232         for i in prange(N):
233             mean = mean_deltaV(chi[i], explosion)
234             dev = 0.4
235             x = np.random.rand()
236             dv = x*1.3*v_c
237             dist = distriDeltaVExpl(np.log10(dv), chi[i])
238             y = np.random.rand()
239             while y > dist:
240                 x = np.random.rand()
241                 dv = x*1.3*v_c
242                 dist = distriDeltaVExpl(np.log10(dv), chi[i])
243                 y = np.random.rand()
244             result[i] = dist
245         return result
246
247 #     N = len(chi)
248 #     mean = mean_deltaV(chi, explosion)
249 #     dev = std_dev_deltaV()
250 #     print(np.mean(mean), dev)
251 #     max_itr = 15000
252 #     i = 0
253 #     base = 10
254 #     centered = np.random.normal(0, dev, N)
255 #     I = np.nonzero(base**(mean+centered)>1.3*v_c)[0]
256 #     n = len(I)
257 #     while n != 0 and i<max_itr:
258 #         centered[I] = np.random.normal(0, dev, n)
259 #         #I = np.nonzero(base**(mean+centered)>1.3*v_c)[0]
260 #         J = np.nonzero(base**(mean[I] + centered[I])>1.3*v_c)[0]
261 #         I = I[J]
262 #         n = len(I)
263 #         i+=1
264 #         centered[I] = np.log10(1.3*v_c)
265 #         result = base**centered
266 #         return result
267
268 """ ----- Unit vector delta V ----- """
269 def unit_vector(N):
270     vectors = np.random.normal(0, 1, np.array([N, 3]))
271     vectors /= np.sqrt((vectors**2).sum(axis=1))[:, None]

```

```

272     return vectors
273
274 def velocity_vectors(N, target_velocity, velocities):
275     unit_vectors = unit_vector(N)
276     velocity_vectors = velocities[:, None] * unit_vectors
277     return target_velocity + velocity_vectors
278
279 from numpy.random import uniform
280
281 """ ----- Num. Fragments & Char. Length-----
282 """
283 def number_fragments(l_characteristic, m_target, m_projectile, v_impact
284 , is_catastrophic, debris_type, explosion):
285     # Defining reference Mass
286     if explosion == True: # Can be multiplied by scaling factor S
287         print("explosion")
288         return 6*(l_characteristic)**(-1.6)
289     else:
290         m_ref = 0
291         if is_catastrophic:
292             m_ref = m_target + m_projectile
293         else:
294             m_ref = m_projectile * (v_impact)**2
295         return 0.1 * ((m_ref)**0.75) * l_characteristic**(-1.71)
296
297 def characteristic_lengths(m_target, m_projectile, v_impact,
298 is_catastrophic, debris_type, explosion):
299     bins = np.geomspace(0.001, 1, 100)
300     N_fragments = number_fragments(bins, m_target, m_projectile,
301 v_impact, is_catastrophic, debris_type, explosion)
302     N_per_bin = np.array(N_fragments[:-1] - N_fragments[1:]).astype(int
303 )
304     L_c = np.concatenate([uniform(bins[i], bins[i+1], size=N_per_bin[i
305 ]) for i in range(len(bins) - 1)])
306
307     return L_c
308
309 def fragmentation(m_target, m_projectile, v_impact, is_catastrophic,
310 debris_type, explosion):

```

```

305
306     prelim_L_c = characteristic_lengths(m_target, m_projectile,
307                                         v_impact, is_catastrophic, debris_type, explosion)
307     prelim_lambda_c = np.log10(prelim_L_c)
308     prelim_areas = avg_area(prelim_L_c)
309     prelim_AM = np.array(distribution_AM(prelim_lambda_c, debris_type))
310     prelim_masses = prelim_areas / 10**prelim_AM
311
312     if explosion == True:
313
314         unaccounted_mass = m_target - np.sum(prelim_masses)
315
316         n_large_deb = np.random.randint(2, 8) # Pick 2-8 pieces of deb
317         > 1m to spread out the rest of the mass
318
319         # Using 10**-4 to ensure endpoints are not included
320         mass_range = np.linspace(10**-4, (unaccounted_mass - 10**-4),
321                                  10**4) # Create mass range, will use `n_large_deb` to split into
322         sections
323         ranges = np.sort(np.random.choice(mass_range, n_large_deb - 1,
324                                             replace=False))
325         ranges = np.concatenate([[0], ranges, [unaccounted_mass]])
326
327         # Note adding zero for subtraction to work (correct dims) then
328         dropping it afterward
329         mass_per_deb = np.concatenate((ranges[1:], np.zeros(1))) -
330         ranges
331         mass_per_deb = np.resize(mass_per_deb, mass_per_deb.size - 1)
332
333         # For L_c > 1, A/M Distribution is basically deterministic,
334         therefore will just use avg value, can get using np.inf
335         assumed_AM_factory = make_mean_AM(debris_type)
336         assumed_len = np.ones(mass_per_deb.shape)
337         assumed_AM = assumed_AM_factory(assumed_len)
338
339         # Each mean has two possible values, randomly pick one of them
340         for each piece of deb
341         AM_choices = np.random.choice([0,1], len(mass_per_deb), replace
342                                         =True)

```

```

335         assumed_AM = 10*np.array([assumed_AM[AM_choices[i], i] for i
in range(assumed_AM.shape[1])])
336
337         # mass * AM = A(L_c), therefore can reverse Area function for
L_c
338         area = mass_per_deb * assumed_AM
339         found_L_c = np.sort((area / 0.556945)**(1/2.0047077)) #
Inversing the Area function defined above
340         found_lambda_c = np.log10(found_L_c)
341         found_areas = avg_area(found_L_c)
342
343         found_AM = np.array(distribution_AM(found_lambda_c, debris_type
))
344         found_masses = found_areas / assumed_AM # Using assumed A/M
since A/M is a distribution and could get diff values.
345
346         L_c = np.concatenate([prelim_L_c, found_L_c])
347         areas = np.concatenate([prelim_areas, found_areas])
348         masses = np.concatenate([prelim_masses, found_masses])
349         AM = np.concatenate([prelim_AM, assumed_AM])
350
351         return L_c, areas, masses, AM
352     else:
353         # Is a collision
354         if is_catastrophic == True:
355
356             unaccounted_mass = (m_target + m_projectile) - np.sum(
prelim_masses)
357             # Put the rest of the mass in many fragments in last bin
358             deposit_bin = (np.geomspace(0.001, 1, 100)[-1] + np.
geomspace(0.001, 1, 100)[-2])/2
359
360
361             n_large_deb = np.random.randint(15, 50) # Pick 2-8 pieces
of deb > 1m to spread out the rest of the mass
362
363             # Using 10**-4 to enure endpoints are not included
364             mass_range = np.linspace(10**-4, (unaccounted_mass -
10**-4), 10**4) # Create mass range, will use `n_large_deb` to
split into sections

```



```

365         ranges = np.sort(np.random.choice(mass_range, n_large_deb -
1, replace=False))
366         ranges = np.concatenate([[0], ranges, [unaccounted_mass]])
367
368         # Note adding zero for subtraction to work (correct dims)
then dropping it afterward
369         found_masses = np.concatenate((ranges[1:], np.zeros(1))) -
ranges
370         found_masses = np.resize(found_masses, found_masses.size -
1)
371
372         found_L_c = np.ones_like(found_masses) * deposit_bin
373         found_areas = avg_area(found_L_c)
374         found_AM = found_areas / found_masses
375
376         L_c = np.concatenate([prelim_L_c, found_L_c])
377         areas = np.concatenate([prelim_areas, found_areas])
378         masses = np.concatenate([prelim_masses, found_masses])

```

## B.2 Coordinate Transforms

### Listing 2: Perturbation Differential Equations Implementation

```

1 import numpy as np
2 from numpy import cross
3 from numba import njit as jit, prange
4 from numpy.core.umath import cos, sin, sqrt
5 from numpy.linalg import norm
6
7 """
8 Converting from Keplerian to Cartesian
9 -----
10 Helpful links:
11     https://downloads.rene-schwarz.com/download/M001-
Keplerian\_Orbit\_Elements\_to\_Cartesian\_State\_Vectors.pdf
12     https://gitlab.eng.auburn.edu/evanhorn/orbital-mechanics/blob/
a850737fcf4c43e295e79decf2a3a88acbbba451/Homework1/kepler.py
13

```

```

14 Notes: Code was modified from Poliastro source, elements.py
15 """
16
17 mu = 398600.4418 #km^3s^-2
18
19 @jit
20 def rotation_matrix(angle, axis):
21
22     c = cos(angle)
23     s = sin(angle)
24
25     if axis == 0:
26         return np.array([[1.0, 0.0, 0.0], [0.0, c, -s], [0.0, s, c]])
27     elif axis == 1:
28         return np.array([[c, 0.0, s], [0.0, 1.0, 0.0], [s, 0.0, c]])
29     elif axis == 2:
30         return np.array([[c, -s, 0.0], [s, c, 0.0], [0.0, 0.0, 1.0]])
31     else:
32         raise ValueError("Invalid axis: must be one of 'x', 'y' or 'z'"
33             )
34
35 @jit
36 def rv_pqw(k, p, ecc, nu):
37     pqw = np.array([[cos(nu), sin(nu), 0], [-sin(nu), ecc + cos(nu),
38         0]]) * np.array(
39         [[p / (1 + ecc * cos(nu))], [sqrt(k / p)]]
40     )
41     return pqw
42
43 @jit
44 def coe_rotation_matrix(inc, raan, argp):
45     """Create a rotation matrix for coe transformation"""
46     r = rotation_matrix(raan, 2)
47     r = r @ rotation_matrix(inc, 0)
48     r = r @ rotation_matrix(argp, 2)
49     return r
50
51 @jit
52 def coe2rv(k, p, ecc, inc, raan, argp, nu):
53

```

```

52     pqw = rv_pqw(k, p, ecc, nu)
53     r, v = rv_pqw(k, p, ecc, nu)
54     rm = coe_rotation_matrix(inc, raan, argp)
55     ijk = pqw @ rm.T
56
57     return ijk
58
59 # ks = np.array([a, e_mag, i, Omega, omega, M, nu, p_semi, T, E])
60 @jit(parallel=True)
61 def coe2rv_many_new(state, mu=mu):
62     inc = np.deg2rad(state[2, :])
63     raan = np.deg2rad(state[3, :])
64     argp = np.deg2rad(state[4, :])
65     nu = np.deg2rad(state[6, :])
66     p = state[7, :]
67     ecc = state[1, :]
68
69     n = nu.shape[0]
70     rr = np.zeros((n, 3), dtype=np.float64)
71     vv = np.zeros((n, 3), dtype=np.float64)
72
73     for i in prange(n):
74         rr[i, :], vv[i, :] = (coe2rv(mu, p[i], ecc[i], inc[i], raan[i],
75             argp[i], nu[i]))
76
77     return rr, vv
78
79 @jit(parallel=True)
80 def coe2rv_many(k, p, ecc, inc, raan, argp, nu):
81     inc = np.deg2rad(inc)
82     raan = np.deg2rad(raan)
83     argp = np.deg2rad(argp)
84     nu = np.deg2rad(nu)
85
86     n = nu.shape[0]
87     rr = np.zeros((n, 3), dtype=np.float64)
88     vv = np.zeros((n, 3), dtype=np.float64)
89
90     for i in prange(n):

```

```

90         rr[i, :], vv[i, :] = (coe2rv(k, p[i], ecc[i], inc[i], raan[i],
    argp[i], nu[i]))
91
92     return rr, vv
93
94 """
95 Converting from Cartesian to Keplerian
96 -----
97 """
98
99 def rv2coe(r, v, mu):
100     ''' Converts a position, `r`, and a velocity, `v` to the set of
    keplerian elements.'''
101     """
102     Parameters
103     -----
104     r: array (3, n)
105         Position of the body in 3 dim. Measured using center of Earth as
    origin. (m)
106     v: array (3, n)
107         Velocity of the body in 3 dim relative to Earth. (m / s)
108
109     Returns
110     -----
111     ks: array (9, n)
112         An array containing all of the keplerian elements + extra
    useful info.
113         a: Float
114         e: Float
115         i: Float
116         Omega: Float
117         omega: Float
118         nu: Float
119         p_semi: Float
120         T: Float
121     """
122
123     def testAngle(test, angle):
124         """Checks test for sign and returns corrected angle"""
125         angle *= 180./np.pi

```

```

126         I = test < 0
127         angle[I] = 360. - angle[I]
128         return angle
129
130     r_hat = np.divide(r, norm(r, axis=1)[: , None])
131
132     # Orbital momentum vector, p
133     p = np.cross(r, v)
134
135     # Eccentricity vector, e, and magnitude, e_mag (used freq)
136     e = (np.cross(v, p) / mu) - r_hat
137     e_mag = norm(e, axis = 1)
138
139     # Longitude of the ascending node, Omega
140     Omega_hat = np.cross(np.array([0, 0, 1])[None, :], p)
141     Omega = np.arccos(Omega_hat[:,0]/norm(Omega_hat, axis=1))
142     Omega = testAngle(Omega_hat[:, 1], Omega)
143
144     # Argument of periapsis, omega
145     omega = np.arccos(np.sum(Omega_hat*e, axis=1) / (norm(Omega_hat,
146     axis=1)*norm(e, axis=1)))
147     B = e[:,2] < 0
148     omega[B] = 2*np.pi - omega[B]
149     omega *= 180. / np.pi
150
151     # True Anomaly, nu
152     nu = np.arccos( np.sum(e*r, axis=1) / (norm(e, axis=1) * norm(r,
153     axis=1)))
154     B = np.sum(r*v, axis=1)<0
155     nu[B] = 2*np.pi - nu[B]
156     nu *= 180. / np.pi
157
158     # Inclination, i
159     i = np.arccos(p[:, 2] / norm(p, axis=1))*180./np.pi
160
161     # Eccentric anomaly, E
162     E = 2*np.arctan(np.tan(np.deg2rad(nu)/2) / np.sqrt((1 + e_mag)/(1 -
    e_mag)))
163
164     # Mean anomaly, M

```

```

163     M = np.mod(E - e_mag * np.sin(E), 2*np.pi)
164     M *= 180./np.pi
165
166     # Semi-Major axis, a
167     R = norm(r, axis =1)
168     V = norm(v, axis =1)
169     a = 1/((2 / R) - (V*V / mu))
170
171     # Semi-parameter, p_semi
172     p_semi = norm(p, axis=1)**2 / mu
173
174     # Orbital period
175     T = 2*np.pi * np.sqrt(a**3 / mu)
176
177     # Keplerian State + Extra info
178     ks = np.array([a, e_mag, i, Omega, omega, M, nu, p_semi, T, E])
179
180     return ks

```

## B.3 Orbit Propagation

Listing 3: Perturbation Differential Equations Implementation

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numba import njit as jit, prange
4 from numpy import pi, sin, cos, sqrt
5 from scipy import integrate
6 from scipy.special import iv
7
8 # User defined libearayr
9 import planetary_data as pd
10 import CoordTransforms as ct
11 import Aerodynamics as aero
12
13 def null_perts():
14     return {
15         'J2': False,

```

```

16         'aero': False,
17         'moon_grav': False,
18         'solar_grav': False
19     }
20
21     class OrbitPropagator:
22
23     def __init__(self, states0, A, M, tspan, dt, rv=False, cb=pd.earth,
24                  perts=null_perts()):
25
26         # Need to add support for initializing with radius and velocity
27         if rv:
28             self.states = 0
29
30         else:
31             self.states = states0
32
33         # Setting the areas and masses
34         self.A = A
35         self.M = M
36
37         # Integration information
38         self.tspan = tspan
39         self.dt = dt
40
41         # Central body properties
42         self.cb = cb
43
44         # Defining perturbations being considered
45         self.perts = perts
46
47         # Defining constants for aerodynamic drag
48         if self.perts['aero']:
49             self.K_a = np.matrix([[1, 0, 0, 0, 0, 0, 0],
50                                   [0, 2, 0, 0, 0, 0, 0],
51                                   [3/4, 0, 3/4, 0, 0, 0, 0],
52                                   [0, 3/4, 0, 1/4, 0, 0, 0],
53                                   [21/64, 0, 28/64, 0, 7/64, 0, 0],
54                                   [0, 30/64, 0, 15/64, 0, 3/64, 0]])

```

```

55         self.K_e = np.matrix([[0, 1, 0, 0, 0, 0, 0],
56                                [1/2, 0, 1/2, 0, 0, 0, 0],
57                                [0, -5/8, 0, 1/8, 0, 0, 0],
58                                [-5/16, 0, -4/16, 0, 1/16, 0, 0],
59                                [0, -18/128, 0, -1/128, 0, 3/128, 0],
60                                [-18/256, 0, -19/256, 0, 2/256, 0, 3/256]])
61
62     def cartesian_representation(self):
63         # Returns the cartesian state representation of states for vis.
64         # purposes
65         N_t = self.states.shape[0]
66         N_frag = self.states.shape[2]
67         cartesian_states = np.empty(shape=(N_t, 2, N_frag, 3))
68
69         for i in prange(self.states.shape[0]):
70             cartesian_states[i, :, :] = ct.coe2rv_many_new(self.states[
71 i, :, :])
72
73         return cartesian_states
74
75     def diffy_q(self, t, state):
76         e, a, i, Omega, omega = state.reshape(5, len(self.A))
77         N_f = len(self.A)
78
79         # Central body information
80         mu = self.cb['mu']
81         radius = self.cb['radius'] # [m]
82         J2 = self.cb['J2']
83
84         # Local variables
85         delta_e = np.zeros_like(e)
86         delta_a = np.zeros_like(a)
87         delta_i = np.zeros_like(i)
88         delta_Omega = np.zeros_like(Omega)
89         delta_omega = np.zeros_like(omega)
90
91         # Current orbital information
92         peri = a * (1 - e) # [m]

```



```

93     p    = a * (1 - e**2) #[m] (Semi parameter)
94     n    = np.sqrt(mu / a**3) # (Mea motion)
95
96     ##### Drag effects #####
97     if self.perts['aero']:
98         h_p = (peri - radius) #[m]
99         rho = aero.atmosphere_density(h_p/1e3) #[kg * m^-3]
100        H = aero.scale_height(h_p/1e3) * 1e3 #[m]
101
102        z = a*e / H
103        Cd = 0.7
104        tilt_factor = 1
105        delta = Cd * (self.A[0] * tilt_factor) / self.M[0]
106
107        e_T = np.array([np.ones_like(e), e, e**2, e**3, e**4, e
**5])
108        I_T = np.array([ iv(i, z) for i in range(7)])
109        k_a = delta * np.sqrt(mu * a) * rho
110        k_e = k_a / a
111
112        delta_e = np.zeros_like(e)
113        delta_a = np.zeros_like(a)
114
115        # CASE e < 0.001
116        delta_e = np.zeros_like(e)
117        delta_a = -k_a
118
119        # CASE e >= 0.001
120        I = e >= 0.001
121        trunc_err_a = a[I]**2 * rho[I] * np.exp(-z[I]) * iv(0, z[I
]) * e[I]**6
122        trunc_err_e = a[I] * rho[I] * np.exp(-z[I]) * iv(1, z[I]) *
e[I]**6
123
124        transform_e = e_T.T.dot(self.K_e) * I_T
125        coef_e = np.array([ transform_e[i,i] for i in range(N_f)])[
I]
126
127        transform_a = e_T.T.dot(self.K_a) * I_T

```

```

128         coef_a = np.array([ transform_a[i,i] for i in range(N_f)]) [
    I]
129
130         delta_e[I] = -k_e[I] * np.exp(-z[I]) * (coef_e +
trunc_err_e)
131         delta_a[I] = -k_a[I] * np.exp(-z[I]) * (coef_a +
trunc_err_a)
132
133         delta_e[np.isnan(delta_e)] = 0
134         delta_a[np.isnan(delta_a)] = 0
135
136         # Deorbit check
137         J = h_p < 100*1e3
138         delta_a[J] = 0
139         delta_e[J] = 0
140
141         ##### J2 effects #####
142         if self.perts['J2']:
143             base = (3/2) * self.cb['J2'] * (radius**2/p**2) * n
144             i = np.deg2rad(i)
145             delta_omega = base * (2 - (5/2)*np.sin(i)**2)
146             delta_Omega = -base * np.cos(i)
147             delta_omega = np.rad2deg(delta_omega) % 360
148             delta_Omega = np.rad2deg(delta_Omega) % 360
149
150             return np.concatenate((delta_e, delta_a, delta_i, delta_Omega,
delta_omega))
151
152         # Performing a regular propagation, i.e. w/ perturbations
153         def propagate_perturbations(self):
154
155             # Initial states
156             a0, e0, i0, Omega0, omega0 = self.states[-1, :5, :]
157             y0 = np.concatenate((e0, a0, i0, Omega0, omega0))
158
159             # Propagation time
160             T_avg = np.mean(self.states[-1, 8, :])
161             times = np.arange(self.tspan[0], self.tspan[-1], self.dt)
162             output = integrate.solve_ivp(self.diffy_q, self.tspan, y0,
t_eval = times)

```

```

163
164     # Unpacking output (Need to drop first timestep as sudden
introduction of drag causes discontinuities)
165     N_f = len(self.A)
166     de = output.y[0:N_f, 1:]
167     da = output.y[N_f:2*N_f, 1:]
168     di = output.y[2*N_f:3*N_f, 1:]
169     dOmega = output.y[3*N_f:4*N_f, 1:]
170     domega = output.y[4*N_f:, 1:]
171     dnu = np.random.uniform(low=0., high=360., size=domega.shape)
172     dp = da * (1 - de**2)
173
174     # Results
175     return de, da, di, dOmega, domega, dnu, dp
176
177 # Performing a Keplerian propagation, i.e. w/o perturbations
178 def propagate_orbit(self):
179
180     times      = np.arange(self.tspan[0], self.tspan[-1], self.dt)
181
182     # Mean anomaly rate of change
183     M_dt       = sqrt(self.cb['mu']/self.states[0, :]**3)
184
185     Nd         = len(M_dt)
186     Nt         = len(times)
187
188     # Mean anomaly over time
189     M_t        = np.deg2rad(self.states[5, :, None]) + M_dt[:, None]
] * times[None, :]
190     M_t        = np.rad2deg(np.mod(M_t, 2*pi))
191
192     # Eccentric anomaly over time. Note need to use E_t in rad,
thus convert to deg after using it in
193     # x1 and x2
194     E_t        = np.empty(shape=(Nd, Nt), dtype=np.float32)
195     E_t        = M2E(self.states[1], np.deg2rad(M_t))
196
197     x1         = sqrt(1 + self.states[1, :])[:, None] * sin(E_t /
2)

```

```

198         x2          = sqrt(1 - self.states[1, :])[:, None] * cos(E_t /
2)
199         E_t          = np.rad2deg(E_t)
200
201         # True anomaly over time
202         nu_t          = (2*np.arctan2(x1, x2) % (2*pi))
203         nu_t          = np.rad2deg(nu_t).T
204
205         n_times       = nu_t.shape[0]
206         states        = np.empty(shape = (n_times, self.states.shape[0],
self.states.shape[1]))
207
208         for i in prange(n_times):
209             state = self.states.copy()
210             state[6, :] = nu_t[i, :]
211             states[i] = state
212
213         # Update internal states
214         self.states = states
215
216
217
218 # Modified from OrbitalPy.utilities
219 @jit(parallel=True, fastmath=True)
220 def M2E(e_deb, M_t, tolerance=1e-14):
221     #Convert mean anomaly to eccentric anomaly.
222     #Implemented from [A Practical Method for Solving the Kepler Equation
    ][1]
223     #by Marc A. Murison from the U.S. Naval Observatory
224     #[1]: http://murison.alphératz.net/dynamics/twobody/
    KeplerIterations\_summary.pdf
225     n_deb = M_t.shape[0]
226     n_times = M_t.shape[1]
227
228     E_t = np.empty_like(M_t)
229
230     for i in prange(n_deb):
231         e = e_deb[i]
232         for j in prange(n_times):
233             M = M_t[i, j]

```

```

234
235         MAX_ITERATIONS = 100
236         Mnorm = np.mod(M, 2 * pi)
237         E0 = M + (-1 / 2 * e ** 3 + e + (e ** 2 + 3 / 2 * cos(M) *
e ** 3) * cos(M)) * sin(M)
238         dE = tolerance + 1
239         count = 0
240         while dE > tolerance:
241             t1 = cos(E0)
242             t2 = -1 + e * t1
243             t3 = sin(E0)
244             t4 = e * t3
245             t5 = -E0 + t4 + Mnorm
246             t6 = t5 / (1 / 2 * t5 * t4 / t2 + t2)
247             E = E0 - t5 / ((1 / 2 * t3 - 1 / 6 * t1 * t6) * e * t6
+ t2)
248             dE = np.abs(E - E0)
249             E0 = E
250             count += 1
251             if count == MAX_ITERATIONS:
252                 print('Did not converge, increase number of
iterations')
253             E_t[i, j] = E
254         return E_t

```

#### Listing 4: Perturbation Propagator

```

1
2         delta_e[I] = -k_e[I] * np.exp(-z[I]) * (coef_e +
trunc_err_e)
3         delta_a[I] = -k_a[I] * np.exp(-z[I]) * (coef_a +
trunc_err_a)
4
5         delta_e[np.isnan(delta_e)] = 0
6         delta_a[np.isnan(delta_a)] = 0
7
8         # Deorbit check
9         J = h_p < 100*1e3
10        delta_a[J] = 0
11        delta_e[J] = 0

```

```

12
13     ##### J2 effects #####
14     if self.perts['J2']:
15         base = (3/2) * self.cb['J2'] * (radius**2/p**2) * n
16         i = np.deg2rad(i)
17         delta_omega = base * (2 - (5/2)*np.sin(i)**2)
18         delta_Omega = -base * np.cos(i)
19         delta_omega = np.rad2deg(delta_omega) % 360
20         delta_Omega = np.rad2deg(delta_Omega) % 360
21
22         return np.concatenate((delta_e, delta_a, delta_i, delta_Omega,
23                                delta_omega))
24
25     # Performing a regular propagation, i.e. w/ perturbations
26     def propagate_perturbations(self):
27
28         # Initial states
29         a0, e0, i0, Omega0, omega0 = self.states[-1, :5, :]

```