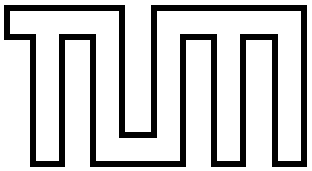


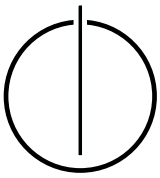
FAKULTÄT FÜR INFORMATIK

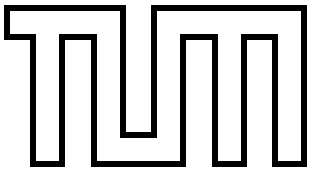
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Implementation and Evaluation of the NASA Breakup Model in modern C++

Jonas Schuhmacher





FAKULTÄT FÜR INFORMATIK

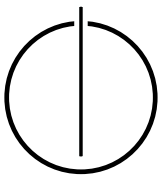
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Implementation and Evaluation of the NASA Breakup Model in modern C++

Effiziente Implementierung und Evaluation des NASA Breakup Model in modernem C++

Author:	Jonas Schuhmacher
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc. and Dr.-Ing. Pablo Gómez
Date:	15.09.2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2021 Jonas Schuhmacher

Acknowledgements

I want to thank all the people involved in this thesis. Foremost, I want to thank my two advisors Fabio Gratl and Pablo Gómez, for providing continuous support, answering all my questions, reviewing my code, and giving precious feedback on my thesis, including its "Denglish" all day and night. But most of all, I'm grateful for giving me the opportunity to write this thesis about this amazing topic. Who can say that they have never dreamed of clear space?

Furthermore, I want to thank my other helpers for proofreading and backing me. This covers Maxi, as well as my parents.

Abstract

As the population of satellites in a geocentric orbit is constantly increasing, it has reached a point where random collisions are no longer unlikely but more and more probable. Thus, the agencies of space-faring nations are interested in predicting the outcome of such fragmentation events in their long-term orbit models. In this context, one often recited model is the empirical NASA Breakup Model of EVOLVE 4.0, which can predict collision and explosion fragmentation events. It is crucial to model each fragment's features like characteristic Length, area-to-mass ratio, and ejection velocity from the place of occurrence to fully understand the breakup and its consequences.

Despite its significance, the model has not yet been implemented in an open-source, documented, and reliable project. Hence, this work aims to change this circumstance by providing an implementation[1] in C++ 17 that consolidates the characteristics mentioned above. Beyond that, the implementation stands out for some other vital attributes like efficiency, performance, and delivering an extensive toolbox whose components can stand alone. For instance, this toolbox contains the option of reading the TLE format, containing orbital elements, or in reverse producing orbital elements by conversion from cartesian state vectors.

The implementation has undergone multiple tests in order to be successfully verified. Its results were compared to the original NASA references and public available partial implementations of the NASA Breakup Model.

The resulting implementation is performant by extensively using parallelization and a cache-optimized data layout. For example, a testing system simulated breakups with millions of random fragments being generated in less than a second.

[1] `https://github.com/schuhmaj/nasa-breakup-model-cpp` last accessed: 01.09.2021

Zusammenfassung

Der Bestand an Satelliten in einem geozentrischen Orbit wächst beständig, so dass mittlerweile ein Punkt erreicht wurde, an welchem Kollisionen von Objekten nicht mehr unwahrscheinlich, sondern im Bereich des Möglichen liegen. Die Vorhersage der Fragmentierung spielt dabei eine entscheidende Rolle in Langzeitmodellen der Raumfahrtagenturen. Ein damit assoziiertes empirisches und häufig verwendetes Modell ist das NASA Breakup Model of EVOVLE 4.0, welches nicht nur Kollisionen, sondern auch Explosionen modelliert. Entscheidend für die Umsetzung ist dabei die Bestimmung der Fragment Eigenschaften, wie charakteristische Länge, Fläche zu Masse Verhältnis oder die Geschwindigkeit, um letztlich die Konsequenzen der Fragmentierung vollständig zu verstehen.

Trotz seiner Bedeutung ist das Modell bis heute von niemandem vollständig implementiert worden in einem zuverlässigen, dokumentierten, open-source Projekt. Diese Arbeit ändert diesen Umstand und präsentiert eine Implementierung[2] in C++ 17, welche die obigen Eigenschaften auf sich vereinigt und darüber hinaus auch noch eine Reihe weiterer Schlüsselmerkmale wie Effizienz, Performance und ein umfangreiches Werkzeugset mit sich bringt, das auch individuell in seinen Subkomponenten tragfähig ist. Dazu gehören beispielsweise TLE Format Input oder Kepler Element Output.

Zudem wurde die Implementierung umfangreichen Tests unterzogen und erfolgreich verifiziert auf ihre Gültigkeit mit den originalen NASA Referenzen und gegengeglichen mit anderen verfügbaren partiellen Implementierungen.

Im Großen und Ganzen konnten im finalen Resultat durch Einsatz von Parallelisierung und Cacheoptimierung auf einem Testsystem Laufzeiten von unter einer Sekunde, selbst für Millionen von zu erzeugenden Zufallsfragmenten, erreicht werden.

[2] `https://github.com/schuhmaj/nasa-breakup-model-cpp` last accessed: 01.09.2021

Contents

Part I.

Introduction and Background

1. Introduction

Some "BEEP, BEEP, ..." ushered in a new era in humankind's history in 1957 - the beginning of the space age with Sputnik I. Since then, significant achievements have been accomplished, deepening our knowledge about the stars and even the composition of our own planet. However, as it always has been in human history, people leave something behind wherever they go. This circumstance manifests in Earth's orbit in the form of space debris, including dysfunctional satellites or old burned-out rocket bodies. For instance, the oldest object, Vanguard-1, launched in 1958, is still in orbit today, with its reentry date estimated for the next century. Figure 1.1 illustrates the trend of the growing object population in Earth's orbit. Notably, there were two jumps in the diagram for payload debris from 2007-2008 and 2009-2010. These jumps are conditioned by a Chinese anti-satellite weapon test and the collision of Iridium-33 and Cosmos-2251. Both incidents serve as case studies and shall be examined later in this paper.

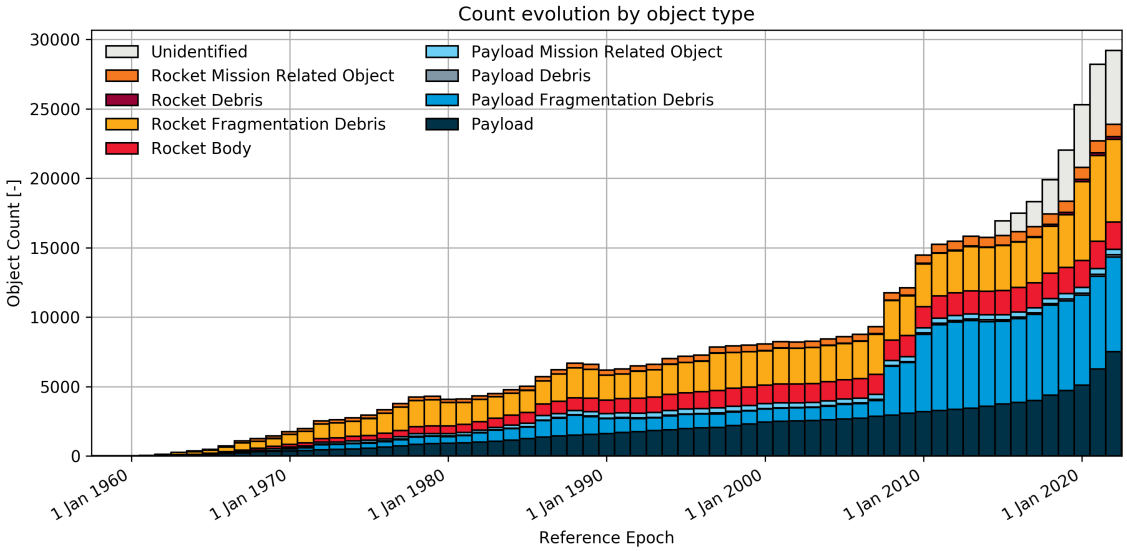


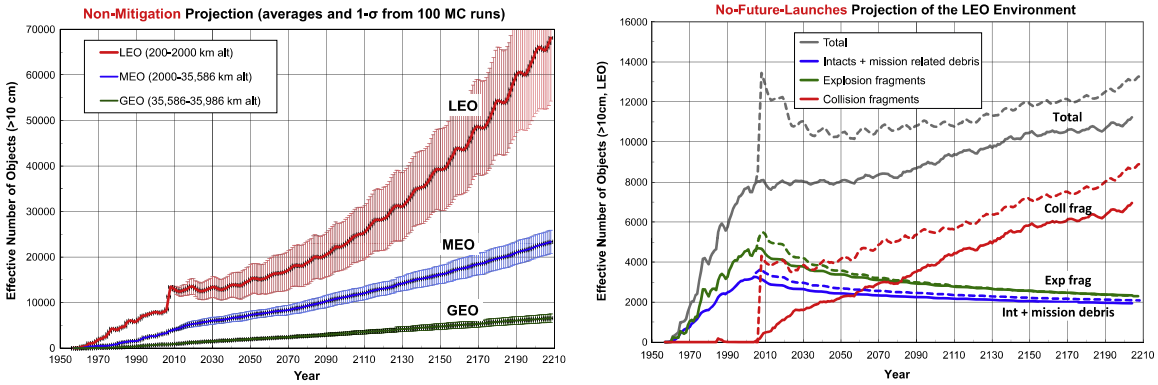
Figure 1.1.: Chart showing the number of objects in a geocentric orbit; from [aE]

Besides this dilemma of human-created objects in orbit, the problem has started to develop its own dynamics since the number of objects has reached a critical point [Lio11] where collisions will ongoingly occur, consequently increasing the fragment count as the "most dominating debris-generating mechanism" [LJ06]. Therefore, literature also calls the debris population to be above "the point of critical density" [KOK01], which marks the moment when random collisions balance the number of decaying objects.

Another more prominent term, which the reader might be familiar with because the press

often works with it, is the expression "Kessler Syndrome" [KJLM10], which originates from Donald J. Kessler. He already proposed the scenario of cascading collisions generating debris, making future space flight more complicated, in 1978.

A problem in this context is the missing regulation of space with different actors, coming from military, university, or commercial origin, working only in their self-interest, although space is a significant part of our globalized economy [Sch20].



(a) Development of debris population without mitigation as predicted by *LEGEND* for LEO (low-earth-orbit), MEO (medium-earth-orbit) and GEO (geosynchronous-equatorial-orbit); each curve is the average of 100 Monte-Carlo runs

(b) Projection of population growth in LEO based on a no-future-launch scenario; Solid curves are based on historical data until 2006; Dashed curves are based on historical data until 2009

Figure 1.2.: Different scenarios of object population growth; from [Lio11]

Figure 1.2a portrays this situation when no measures are undertaken to prevent random collisions like through evasive maneuvers or sustainably designed rockets [Joh10], and satellites are not disposed of by deorbiting or moving them to a grave-yard orbit. In contrast, Figure 1.2b depicts impressively the Kessler Syndrom of collisions being the dominant debris source, as this scenario implies no further launches. In conclusion, Figure 1.2 summarizes the two extreme scenarios of total apathy and total restriction.

It would be impossible to produce those scenarios and predictions without reliable breakup models describing the fragmentation process of, for instance, a collision. If one searches across literature, one often finds the NASA Breakup Model of EVOLVE 4.0 as a central reference source in this domain. Despite that, there are rarely publicly available implementations able to produce every scenario. Further, they are written by scientists without a computer science background and only rarely documented. This work closes this existing gap by creating an extensive open-source implementation of the NASA Breakup Model in modern C++.[1] Moreover, the quality of the implementation is the focus of this work and not only an accessory. This thesis will guide the reader from the theoretical background to the implementation, and it completes with an evaluation containing verification, results, and time measurements.

[1] `https://github.com/schuhmaj/nasa-breakup-model-cpp` last accessed: 01.09.2021

2. Theoretical Background

This chapter aims to give the reader a qualified understanding of the Breakup Model's characteristics, constraints, and history. In the beginning, we will thereby examine the basic concepts of the NASA Breakup Model of EVOLVE 4.0, going over its history, structure, and its features. Afterward, the sections will introduce the Keplerian Elements and their exemplary utilization in the TLE format before finally completing the topic with a quick summary.

2.1. General Idea of the NASA Breakup Model

In the first place, the NASA Breakup Model of EVOLVE 4.0 describes disintegrations of satellites in orbit, which could either be a collision of two spacecraft or an explosion of one rocket body. Thus the model gets one or two satellites as input and outputs the resulting debris. The model is empirical in its nature and based on the results of well-prepared experiments like on-orbit or terrestrial hypervelocity tests and observed breakup events in orbit. The Nimbus 6 R/B explosion or the Solwind/ P-78 "collision" with an anti-satellite-missile are two of multiple data sources for the NASA Breakup Model of Evolve 4.0 [LJKAM02]. One series of experiments to emphasize is the ground-based Satellite Orbital Debris Characterization Impact Test ($SOCIT$), which is still of interest these days as a reference [ACH$^+$17] and had a significant impact on the NASA Breakup Model of EVOLVE 4.0. As the name might suggest, this is not the first model describing how orbit breakups occur. The predecessor was the Breakup Model of Evolve 3.0, which was retired because new data led to the realization that the former model was imprecise in certain situations, like e. g. the estimation of debris smaller than 10 cm for explosions. The main difference between those two generations is the choice of the independent parameter. Whereas the predecessor chooses mass as the independent variable, the new model chooses the characteristic Length, the satellite's diameter, as the preferred variable for function input. The motive behind this decision is the circumstance that the characteristic Length "is more directly linked to both the on-orbit and terrestrial breakup data" [JKLAM01]. This phrase references the radar observations with which it is possible to derive the characteristic Length from the radar cross section (RCS). If we assume that the body is a sphere, the characteristic Length (diameter) L_c can then be calculated from the radar cross section, which is considered to be a two-dimensional circular area by the following Equation 2.1:

$$L_c = 2 \cdot \sqrt{\frac{RCS}{\pi}} \tag{2.1}$$

Another possible approach of calculating the characteristic Length L_c consists of forming the average over the three maximum orthogonal projected dimensions, which is presented

below in Equation 2.2 [MPFC+15]:

$$L_c = \frac{1}{3} \cdot (X + Y + Z) \tag{2.2}$$

Although L_c is the independent variable, sometimes it is desirable to use the mass as input. This fact can be especially helpful for breakup events where the mass of the bodies involved is well-known. Two equations are required, once more assuming spherical objects:

$$\rho(L_c) = \begin{cases} \rho_{\text{Al}} = 2698.9 & \text{if } L_c < 0.01\text{m} \\ 92.937 \cdot L_c^{-0.74} & \text{otherwise} \end{cases} \tag{2.3}$$

$$M(L_c) = \frac{4}{3}\pi \cdot \left(\frac{L_c}{2}\right)^3 \cdot \rho(L_c) \tag{2.4}$$

$$\text{where } \rho \text{ in } \left[\frac{kg}{m^3}\right], L_c \text{ in } [m], M \text{ in } [kg], \rho_{\text{Al}} = \text{density of aluminum}$$

Equation 2.3 [JKLAM01] calculates the density ρ based on an empirical approach, whereas Equation 2.4 combines the former strategy with the sphere volume to determine the mass M. In order to achieve the previously mentioned objective, Equation 2.4 has only to be resolved for L_c.

The beforehand addressed aspects primarily focus on the recent additions in EVOLVE 4.0 and conversions between the mass and characteristic Length, but not on those output-related features. Hence, a practical model must produce a valid fragmentation for a presented breakup event with specific initial conditions and determine each fragment's size, area-to-mass ratio (which also implies mass and area), and ejection velocity. Besides, the model cannot be deterministic and always produces identical debris distributions as the same satellite breakup would lead to different results in reality. [JKLAM01]

Of course, the runtime is also an area of concern when creating a Breakup Model. Thus, the model cannot be infinitesimally complex and has to adapt to the computational capabilities of the time, for example, presented in [VO17] for the evolution of debris fragments or in terms of the sheer generated number of debris [PCDL18].

2.2. Explosion and Collision Archetype

As Section 2.1 already mentioned, the NASA Breakup Model consists of two categories: An explosion type with one participating object and a collision type with two partaking satellites where the smaller one, mass-wise, is the projectile, and the remaining one the target. On the one hand, tank explosions, residual propellant-related explosions, or "deliberately [with explosive charges] destroyed" [JKLAM01] spacecraft do use the first archetype for the description of their fragmentation. On the other hand, the collision variant is used when two spacecraft crash into each other. Here one must distinguish two types of collisions. A catastrophic collision happens if both target and projectile are entirely fragmented. The model assumes that this event occurs when the kinetic energy of the smaller object divided by the mass of the larger vessel is greater than 40 J/g (see Equation 2.5). In the opposite state, a non-catastrophic collision means that only the projectile is thoroughly fragmented, and the target is littered with impacts.

$$\text{Catastrophic if } 40 \text{ J/g} < \frac{m_{projectile} \cdot v_{impact}^2}{2 \cdot m_{target}} \tag{2.5}$$

2.3. Breakup Fragment Properties

2.3.1. Size Distribution

A breakup model has to provide the absolute quantity of fragments and values for multiple traits of the resulting pieces. So the NASA Breakup Model does both tasks by assigning each breakup-event type a power-law distribution, approximating the amount of debris and its presumable size (= characteristic Length) for each of the fragments. Before the size distributions are presented, it is necessary to know that the power-law "for NASA's breakup model was validated for a size range between 1 mm and 1 m via observational" [BLR+17] and experimental data.

Explosion Case

A simple power-law, shown in Equation 2.6, has been developed based on experiments and observations for the explosion case, "taking [...] natural variations due to structure and energy" [JKLAM01] into account. Usually, the scaling factor S of the power-law distribution equals one for a regular rocket body with 600 - 1000 kg masses. However, sometimes a scaling is required. The value S adopts thereby multiple different values, for example, "0.1 for a Soviet/Russian Proton ullage motor; 0.1 for a Molniya orbit Soviet Early Warning satellite; 0.25 for an Ukrainian Tsyklon third stage; 0.3 for a Soviet Anti-Satellite (ASAT); 0.5 for a Soviet battery-related explosion; 0.6 for a Soviet Electronic Ocean Reconnaissance Satellite (EORSAT); and 2 for a Titan Transtage." [LJKAM02] Accordingly to these results, the equation presented below describes the number of fragments for a given size L_c or larger in meters:

$$N(L_c) = S \cdot 6 \cdot L_c^{-1.6} \tag{2.6}$$
$$\text{with } S \text{ as type-dependent, unitless number (normally } S = 1)$$

Collision Case

Collisions are, as stated above, categorized into two classes: catastrophic and non-catastrophic. Nevertheless, the associated power-law equation is kept relatively simple, and the cases only differ in one characteristic: the calculation of M. Equation 2.7 shows this correlation, which depicts the number of fragments equal to or larger than a given size L_c in a similar fashion as in an explosion. Here, m_1 and m_2 are respectively the target's mass and the projectile's mass in [kg]. As previously mentioned, m_1 is the bigger of the two masses. [JKLAM01]

$$N(L_c) = 0.1 \cdot M^{0.75} \cdot L_c^{-1.71} \tag{2.7}$$
$$\text{with } M = \begin{cases} m_1 + m_2 & \text{if catastrophic} \\ m_2 \cdot \|v_{impact}\|_2 & \text{otherwise} \end{cases}$$
$$v_{impact} = v_1 - v_2 \text{ in [km/s]}$$

2.3.2. Area-to-Mass Distribution

Generation of Area-to-Mass values

Aside from the characteristic Length, the area-to-mass ratio is the second parameter generated by the NASA Breakup Model. Therefore, the simulation has to draw values from Equation 2.8 or Equation 2.11, which both rely on the normal distribution. Further, the A/M distribution does not distinguish between the two types, explosion and collision, but rather depends on the previously computed characteristic Lengths of the debris, which should already be defined for continuation. [JKLAM01]

$$D_{A/M}(\lambda_c, \chi) = \alpha(\lambda_c) \cdot N(\mu_1(\lambda_c), \sigma_1(\lambda_c), \chi) + (1 - \alpha(\lambda_c)) \cdot N(\mu_2(\lambda_c), \sigma_2(\lambda_c), \chi) \tag{2.8}$$

where $\lambda_c = \log_{10}(L_c)$

$\chi = \log_{10}(A/M)$ is the variable in the distribution

$N =$ the normal distribution function: $N(\mu, \sigma, \chi) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\Big(-\frac{(\chi-\mu)^2}{2\sigma^2}\Big)$

Equation 2.8 determines the area-to-mass ratio for particles larger than 11 cm. The corresponding parameters are then chosen depending on the given context and the characteristic Length. Debris originating from a spacecraft uses the parameters presented in 2.10; those of rocket bodies utilize the parameters of 2.9.

Parameter for the A/M Distribution for rocket body fragments (R/B) and greater than 11 cm:

$$\begin{aligned}
\alpha^{R/B} &= \begin{cases} 1 & \lambda_c \leq -1.4 \\ 1 - 0.3571(\lambda_c + 1.4) & 1.4 < \lambda_c < 0 \\ 0.5 & \lambda_c \geq 0 \end{cases} \\
\mu_1^{R/B} &= \begin{cases} -0.45 & \lambda_c \leq -0.5 \\ -0.45 - 0.9(\lambda_c + 0.5) & -0.5 < \lambda_c < 0 \\ -0.9 & \lambda_c \geq 0 \end{cases} \\
\sigma_1^{R/B} &= 0.55 \\
\mu_2^{R/B} &= -0.9 \\
\sigma_2^{R/B} &= \begin{cases} 0.28 & \lambda_c \leq -1.0 \\ 0.28 - 0.1636(\lambda_c + 1) & -1.0 < \lambda_c < 0.1 \\ 0.1 & \lambda_c \geq 0.1 \end{cases}
\end{aligned} \tag{2.9}$$

Parameter for the A/M distribution for spacecraft fragments (S/C) greater than 11 cm:

$$\begin{aligned}
\alpha^{S/C} &= \begin{cases} 0 & \lambda_c \leq -1.95 \\ 0.3 + 0.4(\lambda_c + 1.2) & -1.95 < \lambda_c < 0.55 \\ 1 & \lambda_c \geq 0.55 \end{cases} \\
\mu_1^{S/C} &= \begin{cases} -0.6 & \lambda_c \leq -1.1 \\ -0.6 - 0.318(\lambda_c + 1.1) & -1.1 < \lambda_c < 0 \\ -0.95 & \lambda_c \geq 0 \end{cases} \\
\sigma_1^{S/C} &= \begin{cases} 0.1 & \lambda_c \leq -1.3 \\ 0.1 + 0.2(\lambda_c + 1.3) & -1.3 < \lambda_c < -0.3 \\ 0.3 & \lambda_c \geq -0.3 \end{cases} \\
\mu_2^{S/C} &= \begin{cases} -1.2 & \lambda_c \leq -0.7 \\ -1.2 - 1.333(\lambda_c + 0.7) & -0.7 < \lambda_c < -0.1 \\ -2.0 & \lambda_c \geq -0.1 \end{cases} \\
\sigma_2^{S/C} &= \begin{cases} 0.5 & \lambda_c \leq -0.5 \\ 0.5 - (\lambda_c + 0.5) & -0.5 < \lambda_c < -0.3 \\ 0.3 & \lambda_c \geq -0.3 \end{cases}
\end{aligned} \tag{2.10}$$

The second displayed Equation 2.11 calculates the A/M values for debris smaller than 8 cm. Unlike the former Equation 2.8, it does not distinguish between the particles' origin, and therefore, it has an unique parameter set (2.12). The here used abbreviation SOC is founded on the previously in Section 2.1 mentioned experimental series SOCIT.

$$D_{A/M}^{SOC}(\lambda_c, \chi) = N(\mu^{SOC}(\lambda_c), \sigma^{SOC}(\lambda_c), \chi) \tag{2.11}$$

Parameter for the A/M distribution for fragments smaller than 8 cm:

$$\begin{aligned}
\mu^{SOC} &= \begin{cases} -0.3 & \lambda_c \leq -1.75 \\ -0.3 - 1.4(\lambda_c + 1.75) & -1.75 < \lambda_c < -1.25 \\ -1.0 & \lambda_c \geq -1.25 \end{cases} \\
\sigma^{SOC} &= \begin{cases} 0.2 & \lambda_c \leq -3.5 \\ 0.2 + 0.1333(\lambda_c + 3.5) & \lambda_c > -3.5 \end{cases}
\end{aligned} \tag{2.12}$$

The reader might ask what happens to fragments between 8 cm and 11 cm in length because, until now, no equation was delivered. In order to solve the issue, a bridge function is used between those two lengths [JKLAM01]. As will be shown in Section 4.4, the implementation uses a linear approach to close the gap between these points sleekly. This aforementioned approach fits the strategy presented in [BJR+98], which combines both functions, too.

Application of A/M values

The A/M values give more insights about the debris than apparent in the first moment. To start with the fundamentals, it is now possible to calculate mass M and the cross-sectional

area A_x of every particle. Using Equation 2.13, one can determine the area with the help of the characteristic Length. Furthermore, by applying the relation presented in Equation 2.14, one can derive the mass.

$$A_x = \begin{cases} 0.540424 \cdot L_c^2 & \text{where } L_c < 0.00167 \text{ m} \\ 0.556945 \cdot L_c^{2.0047077} & \text{where } L_c \geq 0.00167 \text{ m} \end{cases} \tag{2.13}$$

$$M = \frac{A_x}{A/M} \tag{2.14}$$

Nevertheless, Equation 2.14 is not applicable in every situation. This fact holds for A/M values greater than $1\frac{m^2}{kg}$ because then "the effects of solar radiation pressure cannot be ignored" [JKLAM01]. In combination with the fact that "[o]bjects with an [A/M] close to or above $1\frac{m^2}{kg}$ are extremely lightweight" [LS09], the received force conducts to a movement change. Thus, we can use this fact for orbital lifetime calculation [JKLAM01]. Here lies already a second purpose of the A/M value: It allows estimating the material of a debris part. Large values indicate a lightweight material like multi-layer insulation, whereas smaller A/M values indicate composite values and even smaller values heavy-metal debris [LS09]. Moreover, it is possible to use the A/M values to calculate the ballistic coefficient, although the exact shape of the fragment may be unclear, which makes the determination of the coefficient of drag C_d difficult [BJR$^+$98].

Mass conservation

One thing still to consider as part of the A/M value study is the problem of mass conservation. In general, the NASA Breakup Model is not mass conserving if exactly as many particles are generated as calculated with the size distribution. Depending on the stochastic process of drawing values from Equation 2.8 or Equation 2.11, one gets either too much or not enough mass. Nevertheless, it is possible to neglect the issue by re-generating more fragments if one has too little mass or, on the opposite, throw some generated particles away by introducing a threshold [BJR$^+$98] [CIO$^+$21].

It always makes sense to apply the latter because it is not reasonable nor physically correct if one has mass produced out of none. In contrast to deleting a mass excess, the first ansatz of fulfilling the mass budget is more controversial, mainly because of three arguments. First of all, one can argue that L_c in the power-law distribution always is a lower bound, hence missing mass is concentrated in particles smaller than this bound. Secondly, the fulfillment of the mass budget is not always target-oriented. In cases of a non-catastrophic collision, only the projectile is entirely fragmented, whereas the target is only cratered. So the relation of input mass as sum of projectile and target equals output mass cannot be true since the target is not fully converted to debris. Moreover, as the third point, the paper [JKLAM01] states that the new model stands out for producing different results with altering fragment counts for equally sized satellites "unlike EVOLVE 3.0 [...] which [...] produced exactly the same number of debris for" [JKLAM01] satellites' explosions of the same size.

The here presented implementation solves this issue by making the mass budget as upper bound an obligation whereas the replenishment is optional. This will be presented in Section 4.4.

2.3.3. Ejection Velocity Distribution

Finally, the third feature to be determined by the NASA Breakup Model is the ejection velocity which allows for tracking and predicting whether the debris is going to burn up or remain in orbit - respectively determining the Kepler Elements. Consequently, it is modeled by the NASA Breakup Model with Equation 2.15. As presented below, the equation differentiates between collisions and explosions concerning the utilized parameters of 2.16. Furthermore, the normal distribution is once again the central part of the equation. [JKLAM01]

$$\begin{aligned} D_{\Delta V}(\chi, v) &= N(\mu(\chi), \sigma(\chi), v) \\ \text{where } \chi &= \log_{10}(A/M) \\ v &= \log_{10}(\Delta V) \\ \mu, \sigma &= \text{depending on case, see below (2.16)} \end{aligned} \tag{2.15}$$

$$\begin{array}{ll} \mu^{EXP} = \text{mean} = 0.2\chi + 1.85 & \mu^{COLL} = \text{mean} = 0.9\chi + 2.9 \\ \sigma^{EXP} = \text{standard deviation} = 0.4 & \sigma^{COLL} = \text{standard deviation} = 0.4 \end{array} \tag{2.16}$$

One step is still missing before getting the Orbital Elements which are further illustrated in Subsection 2.4.1. After drawing values from Equation 2.15, the ejection velocity $v_{ejection}$ is only a scalar which does not help in determining the Orbital Elements. To achieve this goal, we have to transfer the scalar $v_{ejection}$ into a cartesian vector $\vec{v}_{ejection}$. This transformation can be performed by using the following Equation 2.17, which uses the uniform distribution [KBS+16]:

$$\begin{aligned} \vec{v}_{ejection} &= \begin{pmatrix} v_{ejection} \cdot \cos(\theta) \cdot \sqrt{1-u^2} \\ v_{ejection} \cdot \sin(\theta) \cdot \sqrt{1-u^2} \\ v_{ejection} \cdot u \end{pmatrix} \\ &\text{where } u \in \mathcal{U}_{[-1;1]} \text{ and } \theta \in \mathcal{U}_{[0;2\pi]} \end{aligned} \tag{2.17}$$

Henceforth, it is now feasible to determine the Orbital Elements of the debris and start to propagate their orbits, predicting and avoiding possible collisions with those fragments in the future by summing up the parent's velocity $\vec{v}_{parent}$ with the presently calculated ejection velocity vector $\vec{v}_{ejection}$ to the total velocity $\vec{v}_{total}$.

2.3.4. Assigning parent bodies

The attentive reader may have noticed that no way of identifying the parent satellite has been given for the collision case until now, despite this being necessary to sum up ejection velocity and base velocity. In fact, the original paper [JKLAM01] does not specify an inheritance hierarchy in that case. This paper's solution consists of two steps. First, fragments greater than the small parent satellite cannot originate from it and thus must origin from the bigger satellite. Secondly, the remaining assignment is executed randomly but with respect to the parent satellites' masses which means we norm the parent satellites' masses to the actually generated total output mass. More details on this ansatz will follow in Section 4.4. The

subsequently in Subsection 3.4.3 presented implementation in Python follows here the same approach [KBS+16].

2.4. Description of an Orbit

2.4.1. Kepler Elements

In order to describe the position of a satellite around a celestial body like the Earth unambiguously, one requires six parameters plus the moment in time, also called epoch, when these position parameters are valid. One way of propagating an orbit with a given input is thereby utilizing the velocity vector $\vec{v}$ and the position vector $\vec{r}$. The previous section shows that this is relatively easy to achieve because the Breakup Model already uses three-dimensional cartesian velocity vectors for the calculation. If additionally the position vector is given, one has every required parameter for an orbit. Nonetheless, without some kind of reference frame, those two cartesian vectors are useless. For this purpose, we can use one of two feasible Earth-centered reference frames. The first possibility is to utilize the Earth-centered inertial reference frame where X (I) and Y (J) axis span the equatorial plane and the Z (K) axis points upwards to the north pole. Figure 2.1 shows the aforementioned equatorial coordinate system (ECI). One has to add that long-term observations typically require a fixed reference frame. Since the ECI is not a fixed frame due to "gravitational perturbations of neighboring planets" [Wal18], one has to additionally define a reference time, which is usually the standard Epoch (January 1, 2000, 12:00 h). In combination with the epoch, coordinates are clear and precise, independent of the Earth's characteristics.

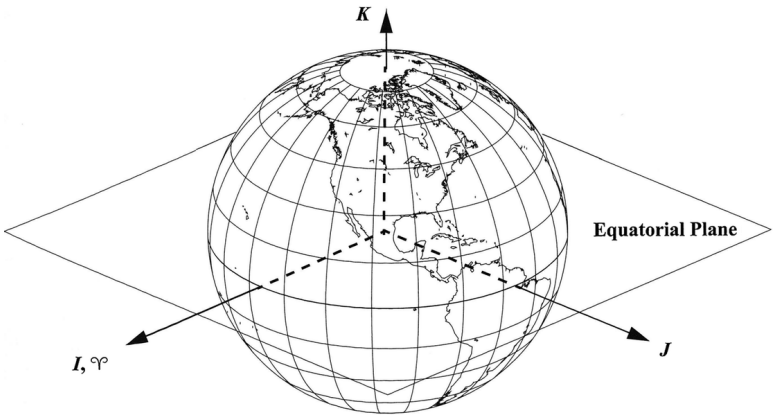


Figure 2.1.: The IJK equatorial coordinate system (ECI); from [Wal18]

Another option is the so-called "Earth-centered Earth-fixed (ECEF) reference system where the coordinate system co-rotates with Earth and hence is fixed to the Earth's surface" [Wal18]. It is standardized in the International Terrestrial Reference Frame (ITRF). Both systems have their advantages and disadvantages, but to summarize it quickly: The ECI is more valuable to represent the state of celestial bodies and spacecraft around the Earth, whereas the ECEF is beneficial in describing objects' positions on the Earth's ground.

Although this would be sufficient, it is desirable to have the Orbital Elements in their more expressive Keplerian Form. This section does not explain the conversion, but instead,

it focuses more on introducing the Keplerian elements and their meaning to the reader.

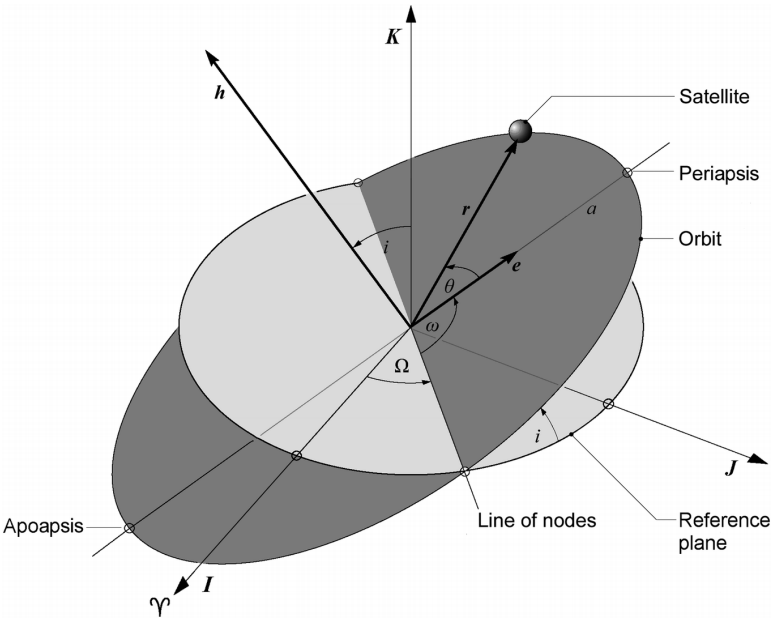


Figure 2.2.: The chart shows the semi-major axis a, the inclination i, the RAAN Ω, the argument of periapsis ω and the true anomaly θ; from [Wal18]

Figure 2.2 shows the different orbital elements and how they affect the orbit of a satellite around the Earth. Semi-major-axis a and eccentricity e are the two elements describing the shape of an orbit and are therefore called metric elements. a is, as the name suggests, half of the major-axis of an ellipsoid, whereas the eccentricity reflects how circular the orbit is (2.18).

$$shape = \begin{cases} \text{circular} & \text{if } e = 0 \\ \text{elliptic} & \text{if } 0 < e < 1 \\ \text{parabolic} & \text{if } e = 1 \\ \text{hyperbolic} & \text{if } e > 1 \end{cases} \tag{2.18}$$

The following three elements characterize the orbit's orientation around a celestial body; hence, they are called angular elements. The inclination i is the angle "between the angular momentum vector h and the z-direction of the reference frame" [Wal18] or the angle between the reference plane, commonly the equatorial plane, and the orbital trajectory plane. In contrast to the inclination, the right ascension of ascending node (RAAN) Ω describes the "angle between the line from the origin O of the reference frame to the vernal point and from O to the ascending node" [Wal18]. The vernal point is located where the sun would be seen from the Earth at the beginning of spring. For further reading, it is sufficient to know that this point is clearly defined. The last of the three elements is the argument of periapsis ω, which is the angle "in the orbital plane between the line of nodes and the periapsis measured in the direction of the motion" [Wal18].

Those five elements clearly define an orbit. Nevertheless, one argument is still missing to

tell where the satellite is located at a specific epoch. For this last element, there are multiple equivalent possibilities. The simplest one is the true anomaly θ (shown in Figure 2.2) which describes the angle between the satellite's position and its periapsis in the direction of the movement. Alternatively, one can use the mean anomaly M to describe the position (see Equation 2.19, based on the mean motion in Equation 2.20):

$$M = n \cdot (t - t_p) \tag{2.19}$$

where t_p = point in time at periapsis

$$n = \frac{2\pi}{T} \overset{elliptic}{=} \sqrt{\frac{\mu}{a^3}} \text{ T = Orbital Period}, \mu = \text{gravitational parameter} \tag{2.20}$$

Lastly, the eccentric anomaly E fulfills the requirement of describing the position of a satellite around a celestial body, too (see Equation 2.21):

$$M = E - e \cdot \sin E \tag{2.21}$$

2.4.2. The Two Line Element Set

The utility of the Orbital Elements opens up in two applications. On the one hand, they are far more expressive than simple coordinates, and a human can imagine the orbit without further calculations. On the other hand, the Orbital Elements are widely used in space flight. One example is the TLE format, short for Two Line Element Set, which is the standardized format for orbit descriptions. It consists of two lines of 69 characters each, containing the orbital state and some additional data like the International Designator or the satellite's classification. Thereby, it is realized as possible input in the implementation, as shown in Section 4.3. For further details about the TLE format, it is advised to visit the following site [st].

2.5. Summary of the Calculation Steps

In the prior section, The NASA Breakup Model of EVOLVE 4.0 has been thoroughly examined, and some first decisions regarding the implementation have been addressed. The following Figure 2.3 has the intention to summarize those lessons about the Breakup Model and guide the reader to the next chapter. The simulation needs the minimal characteristic Length L_c, the masses and velocities of our satellites required as input, whereby the last one is optional in the explosion case. The resulting debris can then be associated with the event location. Following that, one transforms the orbital states vectors to Keplerian Elements and has the possibility to process the fragments further in e. g., an orbit propagator.

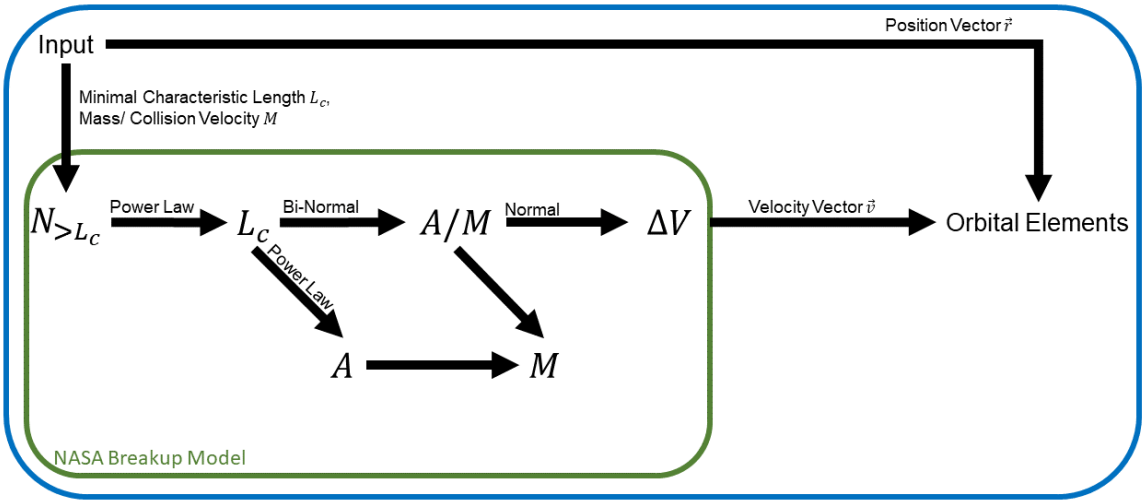


Figure 2.3.: The whole chapter visualized as a process chart; adapted and expanded from [Kli06]

3. Related Work

This chapter aims to give an extensive view of the different breakup models, their corresponding purposes, and applications before finally presenting the implementations of the NASA Breakup Model, which were used as a reference for this work.

3.1. Introduction to Breakup Model Classification

In general, this section differentiates between short-term and long-term models. The short-term analysis focuses directly on the breakup event itself and its direct consequences "in the order of days" [LSWG01]. Typically, they consist of high precision "models of the fragmentation process, orbit propagators containing short period perturbative expressions, and collision risk assessment tools that provide detailed information about the hazards to on-orbit systems" [LSWG01]. On the other hand, long-term debris models analyze the fragmentation event and its longer-term effects on the whole satellite population, including future collisions and explosion risks. Therefore, these models also include "orbit propagators that incorporate long-term and secular perturbations" [LSWG01].

Another classification organizes breakup models into three categories: "complex, semi-analytic and empirical" [Bar96], with the first ones "based on fundamental physical principles" [Bar96]. In contrast, empirical models are derived from experiments and observed data and partly follow the physical principles - like the NASA Breakup Model. Lastly, the category in-between combines both approaches and portrays models "developed from theoretical expressions [combined with a calibration] through [...] experimental data" [Bar96].

3.2. Short-term Modeling

This section will first present an overview of the different existing breakup models and their corresponding use cases before presenting recent improvements to the NASA Standard Breakup Model.

3.2.1. Overview of Breakup Models in other areas of application

Chapter 2 introduced the NASA Standard Breakup Model as one of many different available models that differ in their possible application scenarios. For example, the *GRIM*, Grazing Impact Scattering Model, is a "statistical model [...] to predict the resulting fragment properties based on the projectile properties for very shallow impact angles on mirror surfaces" [FBT+21]. The model is based on a test series of hypervelocity tests, comparable to the *SOCIT* series of the NASA Breakup Model, with "spherical particles rang[ing] between 4-50 km/s [and] masses between 10^{-19} kg to 10^{-14} kg"[FBT+21]. Thus, the model enables future missions to evaluate the probability of instrument damage against optical sensors like X-ray detectors that typically focus those micro-particles.

Another paper, published in 2021, deals with the breakup of objects during reentry. They therefore use, as a new approach, "deep-learning to develop next-generation models for [...] aerothermodynamic modeling" [SM21]. Modeling reentry is essential to ensure that neither the original spacecraft nor its debris becomes a threat "to civilians, buildings, or populated areas on the ground" [SM21]. Furthermore, the topic has become more significant in the last years as the growing satellite population in low-earth-orbit demands precisely planned mission ends to mitigate its further pollution with debris.

Of course, the two models differ in their application scenarios and their calculation expenditure and required input parameters. The *GRIM*, for example, requires the projectile elevation angle for its workflow, whereas the here presented reentry model additionally needs aerodynamic information. This section shall conclude that every model has its place and use. Likewise, the NASA Breakup Model is only one of many available models and obviously not always the best choice since the decision depends on the concrete context and the prioritized parameters. Here, the NASA Breakup Model use case lies in the simulation of on-orbit collision and explosion fragmentation events in the size span of 1 mm to 1 m, as mentioned in Chapter 2.

3.2.2. Enhancements to the NASA Breakup Model in recent years

Tuning input parameters

The improvement of the NASA Breakup Model is an ongoing process. New breakups happening are leading to new enhancement proposals for the model. One of them proposes "an iterative approach to estimate the tuning parameters for the Standard Breakup Model, i. e., the fragmented mass of the parent objects" [CIO$^+$21] for collision. This method improves the number of estimated fragments and therefore enhances the Breakup Model without affecting "the velocity distributions of the fragments" [CIO$^+$21].

Improving physical correctness

Finally, *FREMAT*, the Fragmentation Event Model and Assessment Tool, is a 2017 developed ESA project partly based on the ESA's *MASTER* (see Section 3.3), which in turn is based on the NASA Breakup Model. The tool consists of three components, with the first one being the Fragmentation Event Generator (*FREG*) able of "simulat[ing] fragmentation events (explosions and collisions)" [AIG$^+$17]. For completeness, the second one, Fragmentation Events on Spatial density Tool (*IFEST*), is a tool for long-term debris evolution, and the third one, Simulation of On-Orbit Fragmentation Tool (*SOFT*), fulfills its purpose in extracting debris information like parent object identification from actual observed fragmentation events.

FREG improves the NASA Breakup Model, especially in terms of its physical properties. Mass conservation was already introduced in subsubsection 2.3.2 as a flaw of the NASA Breakup Model. Furthermore, "linear momentum and kinetic energy" [VO17] are not conserved by the NASA Standard Breakup Model. *FREG* solves the mass conservation principle by rejecting "the most massive fragments until [it] conserve[d] or accumulate[d] less than the initial total mass" [AIG$^+$17]. If not enough mass was produced after this step, *FREG* produces one massive additional fragment "whose mass matches exactly the difference between the total mass and the initial mass" [AIG$^+$17]. In contrast, our presented

C++ implementation either discards fragments randomly exceeding the budget or fills the budget optionally by producing more fragments according to the original model's power-law distribution of L_c values.

FREG resolves the conservation of momentum by "distributing the delta-velocities vectors uniformly around the cent[er] of mass of each parent object" [AIG+17]. This approach can be compared to our approach in Subsection 2.3.3, which also uses a uniform distribution method. The C++ implementation does not sustain the last principle, conservation of kinetic energy, whereas *FREG* satisfies this physical principle with an additional scaling step of the previously calculated velocity vectors [AIG+17].

The above approaches show that the NASA Breakup Model is constantly improved in some aspects. Even the later presented C++ implementation is not the "pure" NASA Breakup Model as released in the original paper 2001 [JKLAM01].

Major upcoming revisions

A direct experimental approach to totally revise the Breakup Model is the *DebriSat* experiment, executed in 2014 [Kra02]. The necessity of an improvement results from the fact that the current Breakup Model is based on the *SOCIT* series, which took place between 1991 and 1992 [ACH+17]. However, the satellite used for the test series "characterized the breakup of a 1960's US Navy Transit satellite" [Kra02]. Thus, the new experiment involves today's satellites' characteristics and collects the highest possible precision for "fragments equal or larger than 2mm" [ACH+17], ultimately creating the foundation for new, improved models.

3.3. Long-term Debris Evolution

This section briefly introduces the embedding of fragmentation models into long-term evolutionary models that simulate the orbital population over centuries and their results.

In general, the long-term models are mainly used to forecast the future debris distributions in areas of interest like the LEO (low-earth-orbit) or the GEO (geostationary-earth-orbit) region. A noteworthy work in this area is NASA's *LEGEND*, the LEO-to-GEO Environmental Debris model, capable of giving extensive predictions of Earth's surrounding with debris. Here, the NASA Breakup Model of EVOLVE 4.0 fulfills the role of generating fragments when a collision occurs in the long-term context of *LEGEND* [LHKO04].

A long-term European model is ESA's *MASTER*, which stands for Meteoroid And Space Debris Terrestrial Environment Reference. Its latest revision is *MASTER-8* from March 2019 following *MASTER-2009* and "re-calibrate[s] the Fengyun-1C and Cosmos-Iridium events from the past [after their fragment amount has] converged to a stable value" [Bra19]. Similar to *LEGEND*, the *MASTER* model is based upon a modified NASA Breakup Model, which details can be found in the given paper: [BLR+17]. Later, Section 9.2 sketches how the C++ implementation could be further integrated into such a long-term model.

3.4. Other Implementations of the Breakup Model

This section deals with available implementations of the NASA Breakup Model and their corresponding purposes. Here, the word "available" already marks one motivation behind this work. Breakup models, as shown above, have various areas of application, and many papers use them, yet there are only three open-source implementations available. In addition, those implementations are not designed by software engineers nor substantially documented coupled with incompleteness in terms of the NASA Breakup Model's features. In the following subsection, we will therefore analyze those three reference implementations in detail and how they influenced the presented implementation in C++.

3.4.1. Implementation in C

A C implementation, which can be found on GitHub,[1] and its corresponding paper is about a proof of concept over the value of the Murchison Widefield Array (MWA) in terms of debris clouds detection and observation. Hence the implementation creates fragments of different sizes in order to then check their detectability against the MWA radio telescope.

With this in mind, their code can produce only the catastrophic collision case since they lack an implementation for other scenarios. However, they claim that one can "utili[z]e [their] simulations to characteri[z]e explosions" [JT21]. In order to run the collision, their implementation asks for two parameters: mass and minimal characteristic Length, which is the minimum of parameters required for a catastrophic collision scenario. Although an implementation for the breakup of rocket bodies exists, it is hardcoded to always use the parameters for the spacecraft case.

Unfortunately, the project does not feature any documentation neither externally nor in the source code, but apparently, they were "not seeking to develop an accurate or comprehensive simulation of any particular collision event" [JT21]; instead, they just aimed at generating physically appropriate debris clouds.

Hence, our presented C++ implementation makes no use of this C implementation at all.

3.4.2. Implementation in FORTRAN

In contrast, the FORTRAN implementation, available on GitLab,[2] only handles the explosion case to study debris in the GEO (Geostationary Earth Orbit) region. The paper proposes a "way to create a synthetic population of space debris from simulated data [coming from the implemented NASA Breakup Model], which are constrained by observational data" [PCDL18] from the GEO region.

Their simulation gets the Keplerian Elements and the mass for an existing satellite as input in order to calculate the explosion fragments. In addition, a second input file contains the parameters like α, σ, μ used by the different distribution functions. The values of those parameters can follow the in Subsection 2.3.2 and Subsection 2.3.3 presented parameters. But it is also possible to use custom parameters linked to a "historical fragmentation" [PCDL18] event. As already mentioned above, the FORTRAN implementation only is able to simulate explosions for both rocket bodies and spacecraft.

[1] `https://github.com/steven-tingay/Orbital-Collisions` last accessed: 03.09.2021
[2] `https://gitlab.obspm.fr/apetit/nasa-breakup-model` accessed: 03.09.2021

The FORTRAN model contains documentation, and the code is organized well in terms of tidiness and structure.

Overall, the FORTRAN code proved helpful in choosing the correct area-to-mass ratio distribution function for a given fragment because it implements all three cases: smaller than 0.08 m, bigger than 0.11 m, and in-between. The two first-mentioned cases are well described in the original paper, so the bridge function was the focus of interest. Here the FORTRAN implementation chooses either the small or the big function depending on a comparison between a randomly drawn number r and a transformed fragment's characteristic Length L_b using Equation 3.1:

$$L_b = 10 \cdot (\log_{10}(L_c) + 1.05) \tag{3.1}$$

If $r > L_b$ applies, their implementation uses the big A/M distribution for fragments greater than 11 cm; otherwise, the smaller one is used. The exact reason for this decision remains unclear since no documentation is given, but we found Equation 3.1 in a second independent paper describing a collision analysis tool called *LUCA2* [RMSS17]. There, Equation 3.1 is applied for payloads, but no random number is drawn. Instead, $L_c > L_b$ is used for determining the correct equation.

Our implementation will use neither of those two approaches but instead linear interpolation as presented in Chapter 4.

3.4.3. Implementation in Python

The last one of the three presented implementations is in Python and is available on GitHub.[3] It has been developed within an ESA Ariadna study project and is used to study active debris removal with a game-theoretic approach. Hence, the implementation consists of two modules: the NASA Breakup Model and the game-theoretic approach. The latter handles active debris removal as "a non-cooperative game between self-interested agents" [KBS+16], representing the space-fairing nations. Those actors can either take costly actions to remove debris threatening orbital assets or wait until another player makes an action.

Hence, the implementation has been split into two parts: The Breakup Model consisting only of the collision component as the basis and the pykep library[4] as orbit propagator. Since those two parts need to be combined, they use the *Cube* approach to calculate collision probabilities [LKMS03]. The game theoretical analysis takes place on top of the earlier described model with different player nations executing debris removal operations alike, removing the satellite with the highest threat potential.

As mentioned, the Python implementation only models both collision types and only uses the A/M parameter set of 2.10, which refers to spacecraft fragments with L_c greater than 11 cm. Hence, smaller debris uses this function, too. This procedure is acceptable for the range of 8 cm - 11 cm, as one could define a bridge function, which always uses Equation 2.8 for this range. However, it definitely does not follow the empirical observations, which led to Equation 2.11, which portrays fragments smaller than 8 cm based on the *SOCIT* series. As we will see in Section 6.2, the small size distribution of Equation 2.11 leads to mainly bigger A/M values indicating, as described in Subsection 2.3.2, more lightweight material.

[3] `https://github.com/richardklima/Space_debris_removal_model` last accessed: 04.09.2021
[4] `https://esa.github.io/pykep/` last accessed: 04.09.2021

As a consequence of the neglected function, these particles are missing when comparing the results of the Python implementation to this work's results. One thing to add, it always uses as minimal characteristic Length $L_c = 5$ cm.

The Python implementation is only sparsely documented. However, the functional division of components is well-formed.

As input, the Python simulation needs the current TLE data[5] plus the NORAD Satellite catalog[6] [KBS+16] in order to be executed. Here, it has greatly influenced our C++ implementation as it accepts the TLE format and the NORAD satellite catalog, too. Furthermore, the overall concepts of execution order, how mass conservation is applied, and the transformation of a scalar ejection velocity to a vector come from the Python implementation and are reused in this C++ work. Notably, the Python implementation had the most significant impacts on our product.

3.4.4. Conclusion

The motivation of this work results from the incompleteness and lack of clarity of the above presented open-source implementations. Another point to consider is that none of these implementations come with proper tests that verify their functionality and correctness.

For example, the Python implementation has been intensively investigated since it has been the primary reference. While implementing our C++ version, we found two functional bugs inside the Python code. In the first case, the implemented area-to-mass ratio functionality used σ_1 for the second normal distribution instead of σ_2.[7] Secondly, the non-catastrophic collision case lacked a division by 1000. As Equation 2.7 shows, the factor M is in the non-catastrophic case the product of the projectile's mass $m_{projectile}$ in [kg] and the impact velocity v_{impact} in [km/s]. Nonetheless, the developers used v_{impact} in [m/s].[8] Hence, the results, in this case, had an error of order $\mathcal{O}(10^{3 \cdot 0.75}) = \mathcal{O}(10^{\frac{9}{4}})$.

Furthermore, none of the above implementations can handle every use case of the NASA Breakup Model.

So the NASA Breakup Model often cited and used, with an increasing importance because of the presented problems in Chapter 1, lacks any freely available helpful and exemplary documented implementation. Thus, the following implementation in C++ handles those problems with the aim of delivering a comprehensible, documented, extensible, rapid, and well-tested implementation covering all scenarios of the NASA Breakup Model of EVOLVE 4.0.

[5] `https://www.space-track.org/` last accessed: 01.09.2021
[6] `https://www.celestrak.com/pub/satcat.csv` last accessed: 01.09.2021
[7] `https://github.com/richardklima/Space_debris_removal_model/pull/1` last accessed: 11.09.2021
[8] `https://github.com/richardklima/Space_debris_removal_model/pull/2` last accessed: 11.09.2021

Part II.

Implementation

4. Architecture

After introducing the fundamentals, this chapter will introduce the architecture of the NASA Breakup Model's implementation in C++. Before going into the details, the chapter will first focus on giving a general sketch of the implementation.[1]

4.1. Overview of Components

Figure 4.1 shows the component buildup of the C++ implementation and gives a general idea about the design concepts and how the components interact.

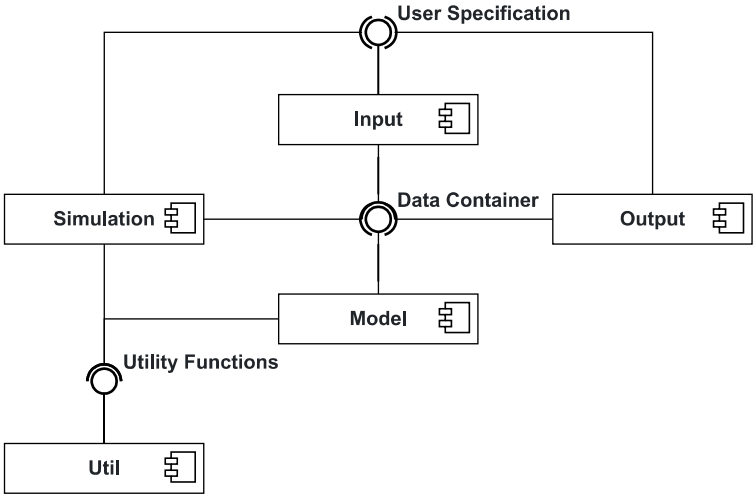


Figure 4.1.: Architecture visualized as UML Component Diagram

In the following, we concentrate on a brief summary of every element while discussing the design decisions and concepts behind the given architectural structure. Note that each component will be presented in detail in the upcoming sections. We start at the top with the `Input`, which is in charge of collecting the settings like input satellites or minimal characteristic Length given by the user. It then provides an interface to the `Simulation` and `Output` components which both can access and handle the collected information. It is important to remark that the `Input` utilizes only the `Model`, whereas it is used by `Simulation` and `Output`. So if they change, the `Input` is not directly affected. The next component in the diagram to describe is the `Output` which uses the `Model` in order to produce results in the form of CSV or VTK files and is configured by the `Input`. Again we have the advantage that `Output` only has to change if the `Model` changes. In charge of what the name suggests,

[1] `https://github.com/schuhmaj/nasa-breakup-model-cpp` last accessed: 01.09.2021

the `Simulation` component can be described very similarly to the `Output` component with the addition of a relation to the `Util` component. This `Util` component only exists for code readability reasons. It comprises four modules consisting of utility functions like orbital anomaly conversions or helper functions for array operations. The last and most significant component is the `Model`, further described in Section 4.2. It contains structures to save satellite data, including orbital state, and provides an interface to all other components to access those storing capabilities.

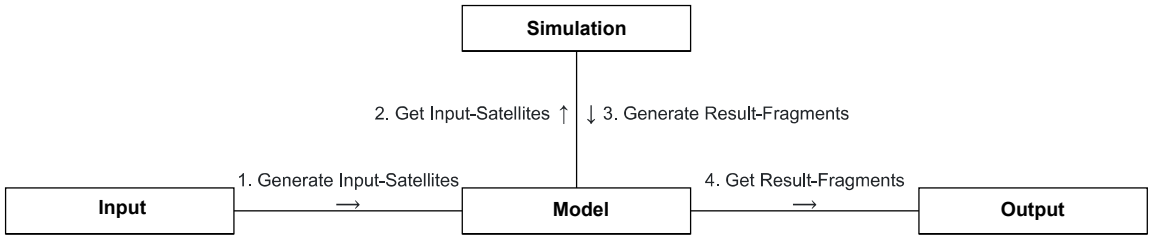


Figure 4.2.: Illustration of the Communication via the Model

The last point marks the most prominent advantage of the architecture visible. There is no direct linkage between `Input`, `Simulation`, and `Output`, other than the user specification, but instead, those three components communicate via the `Model`. The `Input` builds the existing satellites with utility from the `Model`. Following, the `Simulation` uses those given input satellites to generate satellite fragments within the same `Model`. Lastly, the `Output` uses the final iteration of the `Model` to create result files. This flow is illustrated in Figure 4.2. Hence, we have achieved an adaptable and progressive architecture with low coupling and high cohesion.

Furthermore, the achieved independence of components makes the reuse of only a fraction of components very easy. For instance, it is pretty straightforward just to use `Input` and `Model` if one only wants to parse satellites from a file without the intention of simulating a breakup event.

4.2. Model

This section describes the `Model` whose role is to store and provide data for the other components. Figure 4.3 shows the hierarchy among the different classes in this segment. We will describe this regime from left to right, beginning with the `OrbitalElement` class. It consists of the six Keplerian elements previously introduced in Subsection 2.4.1, including the Epoch and unit/ anomaly conversion functionality. Notably, it does not make use of the C++ Standard Library but rather has its own implementation. Memory efficiency is the reasoning behind this decision. Whereas `std::tm` consists of multiple integers, the here presented `Epoch` uses only one single double and integer. Considering the TLE Data contains the epoch in the UTC (Coordinated Universal Time), additional expenses for time zones are not required.

The `OrbitalElementsFactory` is linked with the `OrbitalElement` class and hides and simplifies the latter's construction behind some methods. These methods are adjusted to mirror possible needs like constructing an `OrbitalElement` from TLE Data or resolving

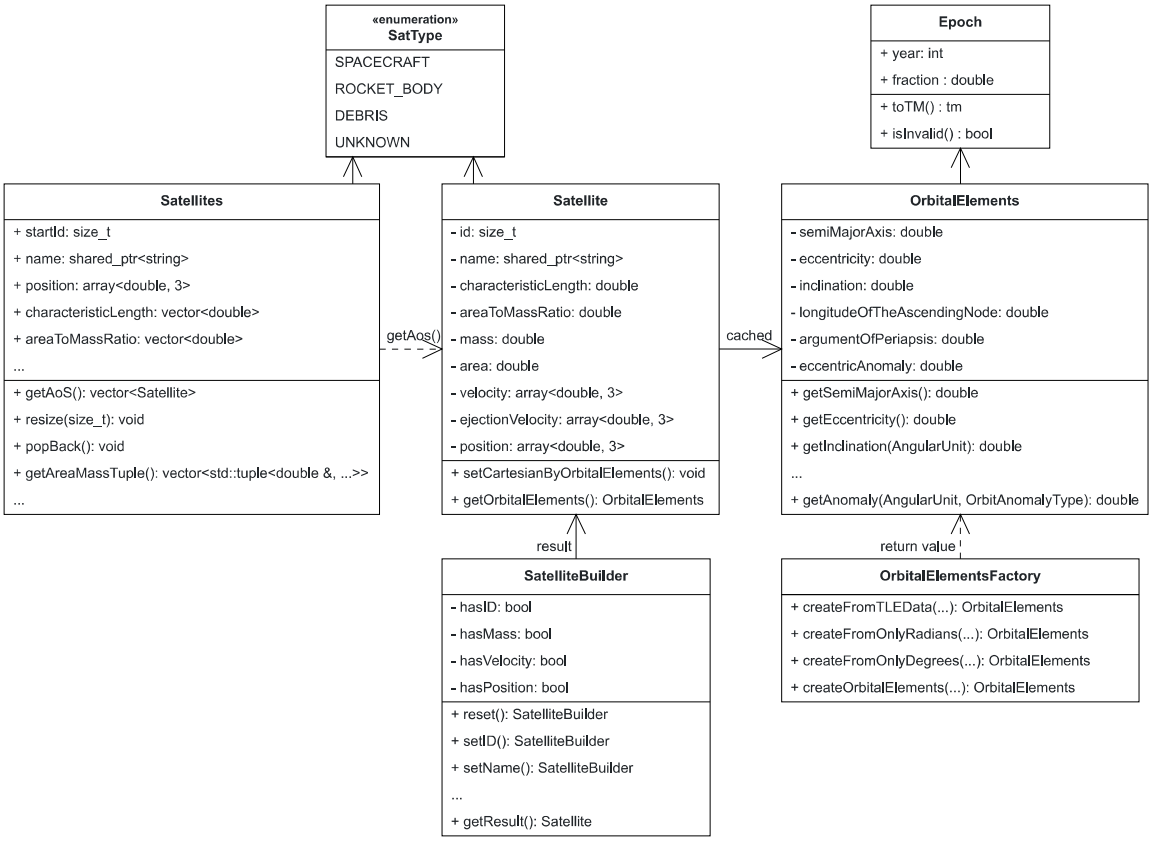


Figure 4.3.: Model visualized as UML Class Diagram, using `std` qualifier, Getter and Setter not shown

different units to an `OrbitalElement`.

Besides, the central part of the `Model` lies in the `Satellite` class, which unifies the core functionality and the memory capacities for the different required input and calculated result parameters. One has to point out that not every parameter is present as a value but rather as a pointer. With this, we can save memory, as we often reuse the same constant name for fragments. In addition, it provides cached methods for the conversion between cartesian state vectors and orbital elements, with the latter being cached for further reuse if they are once determined. These conversions methods are taken from the pykep library[2] and slightly refactored.

In the same fashion as the `OrbitalElementsFactory`, the `SatelliteBuilder` constructs `Satellite` classes and additionally checks the completeness and validity of the satellites under construction. An example use case will later be presented in Section 5.2.

Theoretically, that summarizes every vital aspect required to understand the whole scheme, but there are some more interesting facts about the `Model`, especially the `Satellites` class. The first version of the implementation used a vector of `Satellite` classes corresponding to the Array of Structures ansatz. Although this may be sufficient for pure functionality, we strived for an optimal runtime. By replacing the Array of Structures with a Structure

[2] `https://esa.github.io/pykep/` last accessed: 04.09.2021

of Arrays ansatz realized in the `Satellites` class, we have improved the runtime due to more efficient memory access patterns. In addition, we were able to improve code readability significantly by using the structured binding features of C++17 with different tuple-views over the `Satellites` class. A tuple-view in this context consists of a vector of tuples containing references to `Satellites`' members' elements.

4.3. Input

This section provides information about the extensive scope of the `Input` component. We can separate the `Input`'s purpose chiefly into three different zones of application.

First of all, breakups always consist of involved objects - either one satellite for an explosion event or two for the collision case. We can consider this as the first flow of data into the model, as Figure 4.2 illustrates. Thus the interface for this data source is encoded in the purely virtual `DataSource`. Secondly and thirdly, we need the user specification for the `Simulation` and the `Output`. This information is defined via either of the two purely virtual interfaces `InputConfigurationSource` and `OutputConfiguartionSource`.

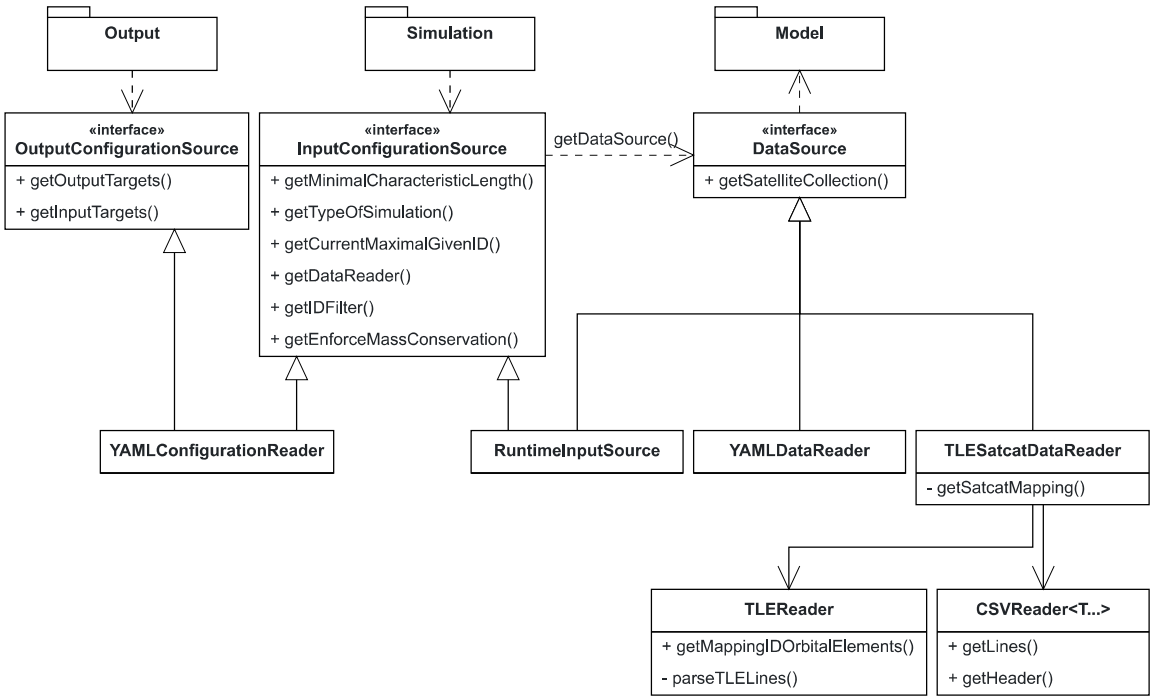


Figure 4.4.: Input visualized as UML Class Diagram, using `std` qualifier

Following the above theoretical introduction, the concrete implementation is presented in the paragraphs below. Something to consider before reading further, the below-presented YAMLReaders hide the dependency yaml-cpp,[3] which is utilized to parse the YAML files.

We realized the interface `DataSource` in three subclasses. These are `YAMLDataReader`, `TLESatcatReader`, and `RuntimeInputSource`. The `YAMLDataReader` reads satellite data

[3] `https://github.com/jbeder/yaml-cpp` last accessed: 05.09.2021

from a user-generated YAML file, whereas the second one joins data from the NORAD satellite catalog[4] and a TLE file, with the satellite catalog being a CSV file. A remarkable feature is that the `CSVReader<T...>` supports every thinkable CSV scheme through templates, making it ideal for reuse in other scenarios. As can be seen, the last-mentioned `RuntimeInputSource` is ideal for using the C++ implementation as a library, as its only purpose is to be a container given to the `BreakupBuilder`, which will be presented in Section 4.2. The interface `ConfigurationSource` is accomplished in two subclasses, with one being the already above-mentioned `RuntimeInputSource` class. With this in mind, the method parameter container idea opens up as the `RuntimeInputSource` provides the model data as well as the user specification for the simulation. The precise application is later demonstrated in Section 5.2. In contrast, the second option, `YAMLConfigurationReader`, reads the configuration from a user-generated YAML file similar to the aforementioned `YAMLDataReader`. Last but not least, the `YAMLConfigurationReader` implements the `OutputConfigurationSource` to define the desired output target files like VTK or CSV.

In conclusion, the here in Figure 4.4 presented class structure has multiple advantages since it is still possible to extend and swiftly adapt the implementation to personal needs on-demand without changing interfaces between the architecture's components.

4.4. Simulation

The `Simulation`, shown in Figure 4.5, consists of an ensemble of four classes, with one being purely virtual. First to mention is the `BreakupBuilder` which, analogously to the `SatelliteBuilder`, assembles `Breakup` classes. It provides methods to load a given user specification, respectively an `InputConfigurationSource`, or overwrite already defined settings from within the developer's context.

Centrally, the core of the implementation is made up of the `Breakup` class and its subclasses, which represent the two cases of explosion and collision. At first glance, one might assume the antipattern of a confusing blob-class [Kru20], but the opposite is true. By consequently splitting each step of the NASA Breakup Model of EVOLVE 4.0 into its own method, we get clean, well-separated implementation steps. We further use the Template Method pattern [Kru20] in order to implement shared functionality only once in the superclass. After the construction of the class, one can commence the simulation by the `run()` method and get the result via `getResult()`. Overall, the run method follows these steps:

1. Reset the state
2. Calculate the number of fragments with the correct Size Distribution (Equation 2.6 and Equation 2.7)
3. Allocate required memory
4. For each fragment: Calculate characteristic Length L_c
5. For each fragment: Calculate area-to-mass ratio A/M and area A and mass M
6. Enforce Mass Conservation: Delete fragments until mass maximum is reached or optionally generate more fragments until the threshold is reached

[4] `https://www.celestrak.com/pub/satcat.csv` last accessed: 01.09.2021

7. For each fragment: Assign a parent (implies assigning base velocity $\vec{v}_{parent}$)
8. For each fragment: Calculate scalar ejection velocity $v_{ejection}$ and cartesian ejection velocity $\vec{v}_{ejection}$ and total velocity vector $\vec{v}_{total}$

The eight protected methods of `Breakup` in Figure 4.5 follow exactly those eight steps in the order of appearance. The implementation precisely follows the equations introduced in Chapter 2 using only the components of the C++ standard library like `std::normal_distribution` or `std::uniform_real_distribution`. The random number generator beneath is always a `std::mersenne_twister_engine`. Nevertheless, we must consider that the L_c values follow a power-law distribution not predefined in the C++ standard. Hence, we transform a value

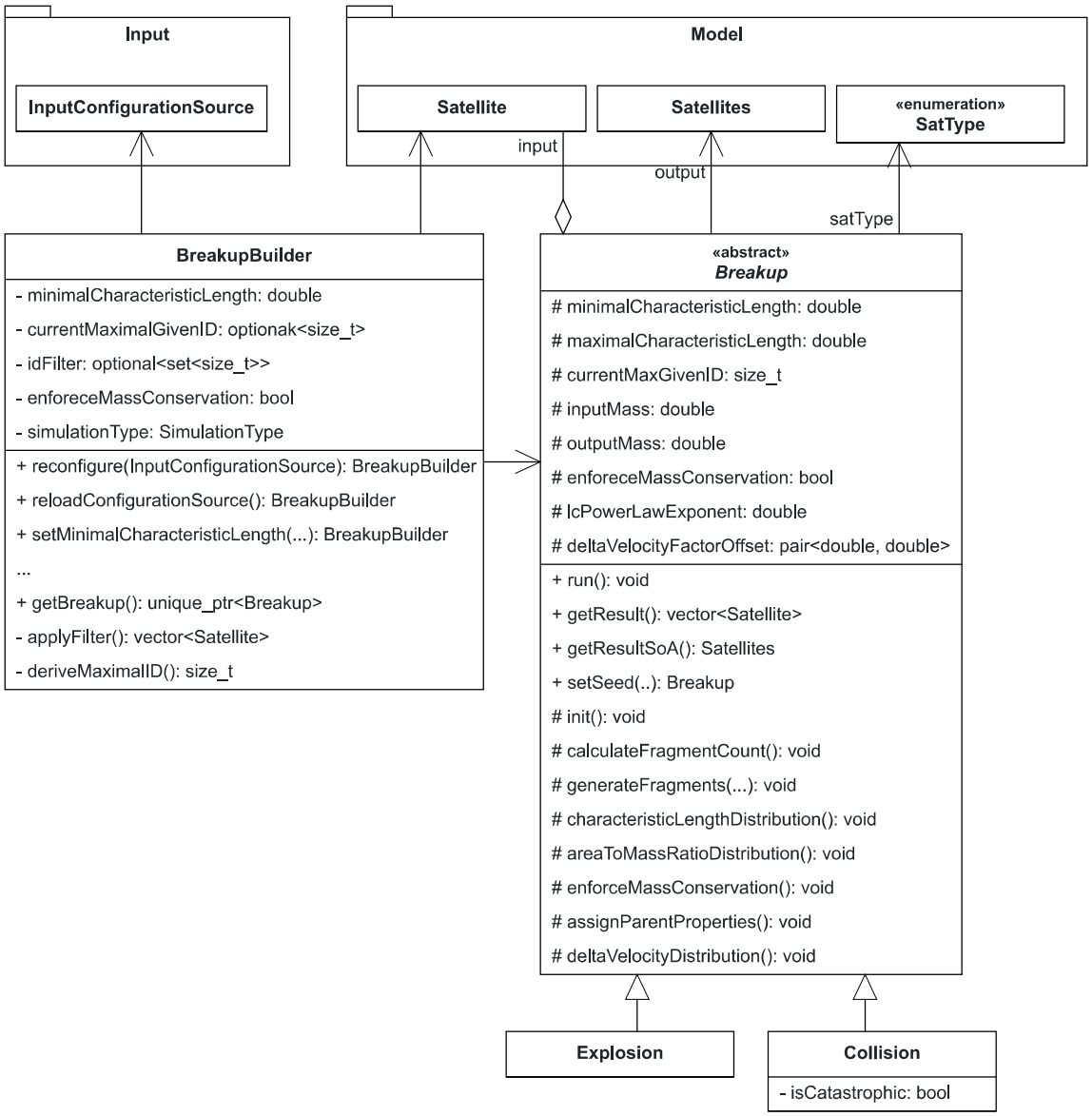


Figure 4.5.: Simulation visualized as UML Class Diagram, using `std` qualifier

y from a uniform distribution into a power-law distributed value x by applying Equation 4.1:

$$x = \left((a^{n+1} - b^{n+1}) \cdot y + b^{n+1}\right)^{\frac{1}{n+1}} \tag{4.1}$$
$$\text{where } y \in \mathcal{U}_{[0;1]} \text{ and } L_c \in [a, b]$$
$$n = \begin{cases} -2.6 & \text{if explosion} \\ -2.71 & \text{if collision} \end{cases}$$

The here presented values for n are respectively the exponents of the derivatives of Equation 2.6 and Equation 2.7. Equation 4.1 comes in handy because it already defines a lower and an upper bound for the resulting value range. The former fits because Equation 2.6 and Equation 2.7 describe the L_c distribution for values greater than a specific minimal L_c aka a lower bound. The latter is reasonable if used with the maximal parent L_c since we cannot produce fragments greater than the parents. With the help of Equation 4.1, we can fulfill Step 4.

Another problem to solve was the gap between 8 cm and 11 cm. Thus we use a linear approach to close the gap in Step 5 with a bridge function, which is shown in the following Equation 4.2:

$$D_{A/M}^{Bridge}(\lambda_c, \chi) = \beta \cdot D_{A/M}(\lambda_c, \chi) + (1 - \beta) \cdot D_{A/M}^{SOC}(\lambda_c, \chi) \tag{4.2}$$
$$\text{where } \beta = \frac{L_c - 0.08}{0.03} \text{ and } \lambda_c = \log_{10}(L_c)$$

Subsection 2.3.2 already introduced the mass conservation issues of the NASA Breakup Model. In this implementation, as shown in Step 6, we solve this issue by making the mass budget as upper bound an obligation, whereas the fulfillment remains optional via a configuration option of the `Input` component.

The assignment of parent bodies in Step 7 is trivial in the explosion case since there is only one parent. For the collision, we adopt the following from the Python implementation inspired approach in Algorithm 1:

This approach definitely fulfills its purpose for a catastrophic collision. However, one can argue that the approach is limited in the non-catastrophic case because depending on the collision characteristics, including the partaking satellites material, impact velocity, and location of impact, only the projectile is really fragmented. In contrast, only a tiny portion of fragments would originate from the target, yet Algorithm 1 would assign most fragments to the target origin. Nonetheless, we use this approach, as the above described does not hold for every non-catastrophic collision and the NASA Breakup Model lacks in specifying some characteristics like satellite material or the exact location of the impact required for a better, more complex assignment model.

The remaining steps of the enumeration are strictly implemented, as one might expect with the knowledge given in Chapter 2.

Furthermore, to optimize the time-to-solution Step 4, Step 5, and Step 8 run entirely in parallel with `std::execution::parallel_unsequenced_policy` whose implementation

Algorithm 1: Assignment of Parent Body in the Collision Case

Input: satellites: `Satellites`, target: `Satellite`, projectile: `Satellite`
Result: satellites containing assigned parents

$satellites \leftarrow shuffle(satellites)$;
$normedTargetMass \leftarrow target.mass \cdot \frac{outputMass}{inputMass}$;
$assignedTargetMass \leftarrow 0$;
foreach *satellite* **do**
    if $satellite.lc > projectile.lc$ **then**
        $satellite.parent \leftarrow target$;
        $assignedTargetMass \leftarrow assignedTargetMass + satellite.mass$;
foreach $satellite \in notAssigned$ **do**
    if $assignedTargetMass < normedTargetMass$ **then**
        $satellite.parent \leftarrow target$;
        $assignedTargetMass \leftarrow assignedTargetMass + satellite.mass$;
    else
        $satellite.parent \leftarrow projectile$;

depends on the utilized compiler.[5] The results of this optimization will later be presented in Chapter 8.

4.5. Output

The `Output` is the simplest component. It consists of a single interface linking the `Output` to the `Model`. In addition, which slightly contradicts Figure 4.6 UML class diagram, there is, for simplicity reasons, a method that takes a whole `Breakup` object and prints its result.

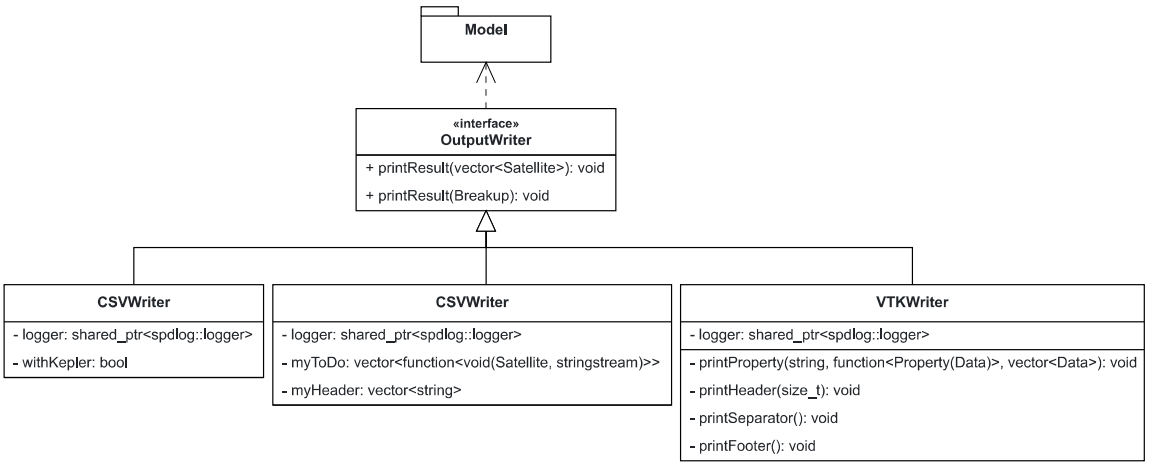


Figure 4.6.: Output visualized as UML Class Diagram, using `std` qualifier

[5] `https://en.cppreference.com/w/cpp/compiler_support` last accessed: 11.09.2021

Otherwise, we have three different implementing subclasses, with two of them producing CSV and the other one VTK files. They all have in common that they use the spdlog library[6] to print the results in an asynchronous fast way per default. The `CSVWriter` on the left is the most uncomplicated of the three and just prints the parameters of the satellites, and if chosen, the Kepler Elements. The `CSVPatternWriter` uses a customizable pattern to produce all possible parameter combinations in CSV output files. To avoid using many if statements, it consists of a vector of functions (`myToDo`)that each print one parameter. Logically the vector of functions is called for every satellite to fill a line. Lastly, the `VTKWriter` produces files with the extension ".vtu", where each satellite property is printed with a similar functional approach as in the `CSVPatternWriter`. These files can then be visualized with for example *ParaView*.[7]

[6] `https://github.com/gabime/spdlog` last accessed: 05.09.2021
[7] `https://www.paraview.org/` last accessed: 08.09.2021

5. User Guide

This section will neither replace the `README.md` on the GitHub Site[1] nor give detailed instructions about the use of the C++ implementation, but it will introduce the different available scenarios based on Chapter 2.

5.1. Input Scenarios

The user from outside a developer's context has three options for using the Breakup Model via the YAML configuration file. First of all, one can manually define all possible parameters in the YAML data file. Optionally one can use the Orbital Elements or define a collision scenario without any orbital relation by making the target fixed and giving the projectile directly the impact velocity. Internally both scenarios are the same since only the velocity difference is relevant for the calculation. However, if one has the Orbital Elements, it is more comfortable to use the provided option. On the other hand, one can use a semi-automatic input by giving in the YAML data file for each individual satellite the location to a TLE file from which only the Orbital Elements are parsed, whereas information about e. g. mass still has to be defined manually. The last option is fully automatic, and no YAML data file has to be specified. Here the NORAD Satcat[2] and an extensive TLE file[3] must be given. The program will then perform the mapping fully automatic. One thing to consider here is that one should use a filter in case of more than two input satellites. Otherwise, the partaking bodies are unclear, and the program aborts.

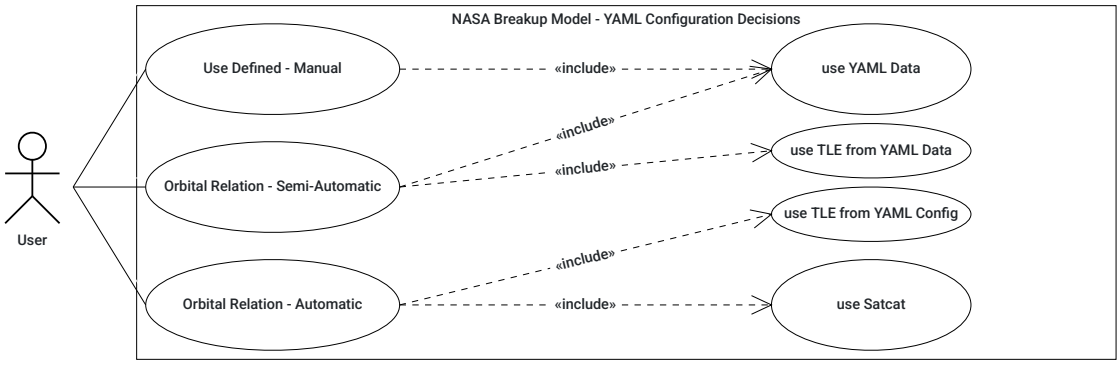


Figure 5.1.: YAML Configuration File Decisions visualized as UML Use Case Diagram

A key point to realize is that the C++ implementation does not check if there actually is

[1] `https://github.com/schuhmaj/nasa-breakup-model-cpp` last accessed: 01.09.2021
[2] `https://www.celestrak.com/pub/satcat.csv` last accessed: 01.09.2021
[3] `https://www.space-track.org/` last accessed: 01.09.2021

a collision probability between two satellites or whether they are at the same position. This check has to be conducted by the user.

5.2. Using the Project as Library

Listing 5.1 sketches how to create a breakup simulation within Section 4.3 introduced `RuntimeInputSource` container.

```
auto configSource = std::make_shared<RuntimeInputSource>(0.05, satellites);
BreakupBuilder breakupBuilder{configSource};
auto breakup = breakupBuilder.getBreakup();
breakup->run();
```

Listing 5.1: Constructing and running a breakup simulation from within C++

One `Satellite` in the vector of satellites can be built by the `SatelliteBuilder`, introduced in Section 4.3. The versatility and readability of this Builder ansatz result from Listing 5.2:

```
Satellite iss = satelliteBuilder
    .setID(25544)
    .setName("ISS")
    .setSatType(SatType::SPACECRAFT)
    .setMassByArea(399.05)
    .setOrbitalElements(factory.createFromOnlyRadians(
          { 6798505.86, 0.0002215, 0.9013735469,
            4.724103630312, 2.237100203348, 0.2405604761},
            OrbitalAnomalyType::MEAN))
    .getResult();
```

Listing 5.2: Source code demonstrating the qualities of the Builder approach

If in the above-presented Listing 5.2 some required value had been missing, the builder would have thrown an exception.

Part III.

Results and Conclusion

6. Verification and Validation

Section 3.4 indicated that the C++ implementation is well-tested and that the output is verified. This chapter details the methods used for testing and comparing the simulation's results.

6.1. Testing

The GoogleTest framework[1] provides the utility in order to verify the functionality of the previously presented implementation in C++. In general, each unit test targets a single class of the previously presented structure. The additional dependencies required for testing a class follow the structure presented in Figure 4.1's UML component view. Notably, every test case uses the `Model` in some way.

The first requirement for reproducible tests is to run the simulation with a fixed random seed. If the former requirement holds, we can then test the fragment generation, which number should always follow the in Subsection 2.3.1 presented equations under the constraint that no mass conservation is enforced. Moreover, we can test that the power-law distribution of L_c values follows these equations with some tolerance of 1%-2%. Lastly, since we parallelized the generation of fragments, we also check for duplicates in the L_c set, indicating issues like race conditions within the random number generation. If those checks pass, the L_c set fits the described requirements of the NASA Breakup Model.

In contrast, the A/M distribution is not that easy to verify since every fragment's A/M value follows a different bi-normal or normal distribution, with each having its own parameter set depending on the fragment's characteristic Length L_c. Thus, we cannot test if all A/M values follow some distribution like in the former case. For this reason, we check the values against the Python implementation. This comparison and the similar testing scenario for the velocity distribution are presented in the following Section 6.2.

6.2. Comparison with Reference Implementation

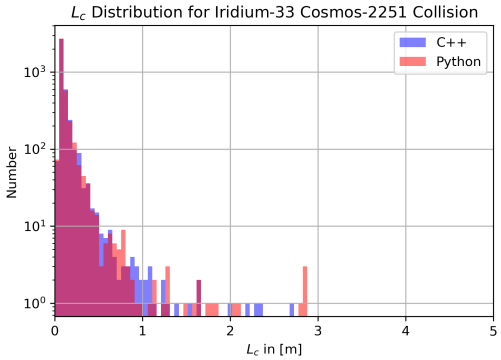
Without losing generality, this section contrasts the C++ implementation and the Python implementation, introduced in Subsection 3.4.3, at the Iridium-33 Cosmos-2251 collision scenario.

Since the NASA Breakup Model is empirical and not a deterministic physical law, we cannot validate our implementation against a physical formula. Instead, we have to compare the produced results with available data. A quantitative comparison is not possible, as only limited amounts of data are available for collisions and explosions. Therefore, our validation relies on the qualitative comparison.

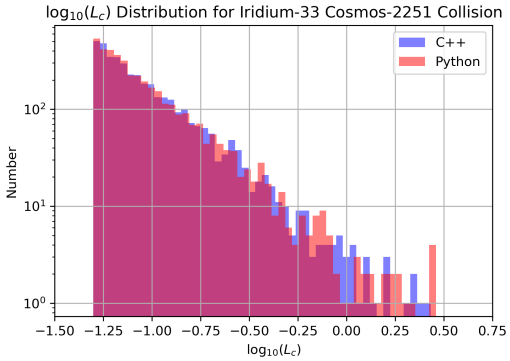
[1] `https://github.com/google/googletest` last accessed: 05.09.2021

As the Python implementation was the main reference with a similar execution order and strategy, a baseline comparison to its predictions is essential for the validation.

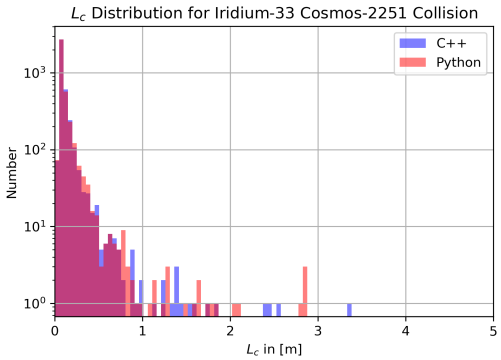
In the following, we investigate results regarding the correctness of our implementation but will neither classify nor interpret the results in detail or compare them to the actual event, as this will be done in Chapter 7. Furthermore, we only handle one collision result and its characteristic distributions. The explosion case will not be compared since explosions and collisions only differ in their size distributions but not in their corresponding area-to-mass ratio or velocity distributions if one disregards varying parameters for the latter. Besides, as Section 6.1 has shown, the size distributions have already been tested.



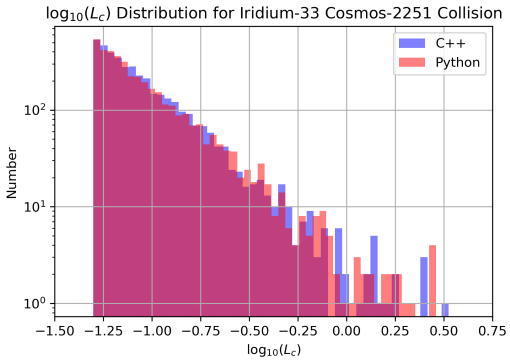
(a) First run
$D = 0.01820$ and $p = 0.52857$



(b) Logarithmic view of (a)



(c) Second run
$D = 0.01896$ and $p = 0.47546$



(d) Logarithmic view of (c)

Figure 6.1.: Comparison of characteristic Lengths L_c between Python and C++ implementation for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m. For each run, an Kolmogorov–Smirnov test has been performed, which results can be seen in (a) and (c). Here, with $\alpha = 0.05$, the equivalence is accepted for both.

For completeness, Figure 6.1 shows the already verified distribution of characteristic Lengths L_c; one time plotted logarithmically. As one can see, the distributions are nearly overlapping. Nevertheless, to be sure, we use a simple Kolmogorov–Smirnov test[2] to verify

[2] `https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test` last accessed: 12.09.2021

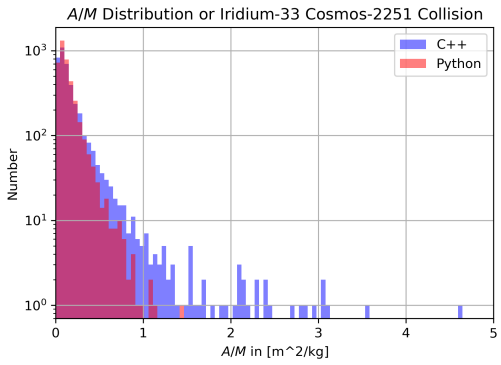
the observation. Equation 6.1 shows the rejection condition for the null hypothesis for a given significance level α, with the first-named being, in our case, the equivalence of two distributions. The Iridium-33 Cosmos-2251 collision distribution holds $n = m = 3954$, with both parameters being the number of fragments.

$$D_{n,m} > \sqrt{-\ln\left(\frac{\alpha}{2}\right) \cdot \frac{1 + \frac{m}{n}}{2m}} \tag{6.1}$$

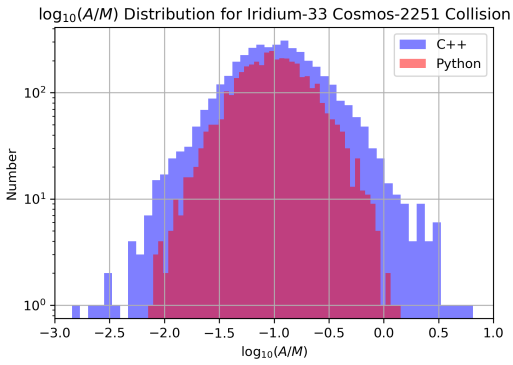
The application of Equation 6.1 with $\alpha = 0.05$ yields an acceptable result for both cases of Figure 6.1. A second possible criterion is the p-value, with which we can reject the hypothesis if $\alpha > p$. However, it is not a criterion for acceptance.

The following figures always base on the first simulation-run, which simulates the Iridium-33 (556 kg) Cosmos-2251 (900 kg) collision with a minimal characteristic Length $L_{c,min} = 5$ cm for generated fragments.

Figure 6.2 and Figure 6.3 continue with the area-to-mass ratio comparison for two scenarios. The first one shows the distribution of the area-to-mass ratio compared to the Python implementation. We can see an extreme deviation for both higher and lower area-to-mass ratios. This phenomenon results from the fact that the Python implementation only implements the area-to-mass distribution for fragments greater than 11 cm and reuses this distribution for smaller fragments. This decision contradicts the original NASA Breakup Model. However, to precisely check the equivalence of the implementations, we could remove the bridging function (Equation 4.2) and the distribution function for smaller fragments (Equation 2.11) from our C++ implementation. The potential effects of this (false) scenario are collected in Figure 6.3 and fulfill the expectation of equivalence.



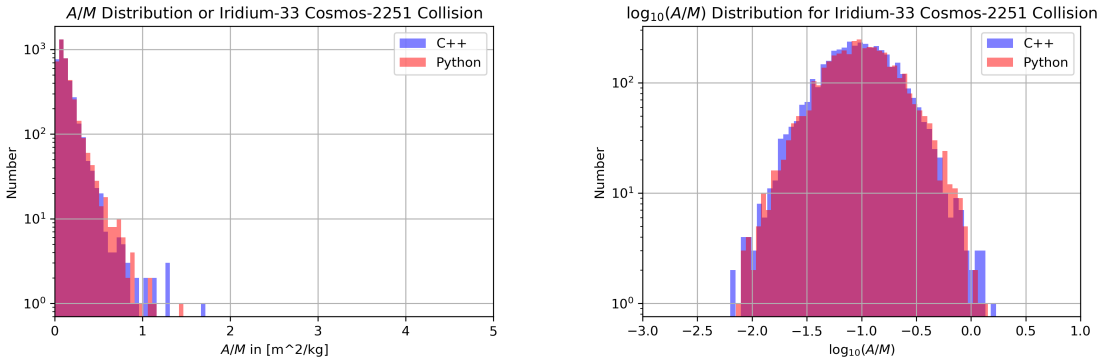
(a) $D = 0.06828$ and $p = 1.94407 \cdot 10^{-8}$



(b) Logarithmic view of (a)

Figure 6.2.: Comparison of area-to-mass ratios A/M between Python and C++ implementation in its as-is state for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m. The result of the Kolmogorov–Smirnov test can be seen in (a). Here, with $\alpha = 0.05$, the equivalence is rejected.

However, we can prove that our C++ implementation fits better with the case distinction between bi-normal distribution for big fragments, normal distribution for smaller fragments, and bridging function for in-between. Figure 6.4 shows the expected A/M probability distributions for a specific characteristic Length L_c. We consider the left Figure 6.4b for the



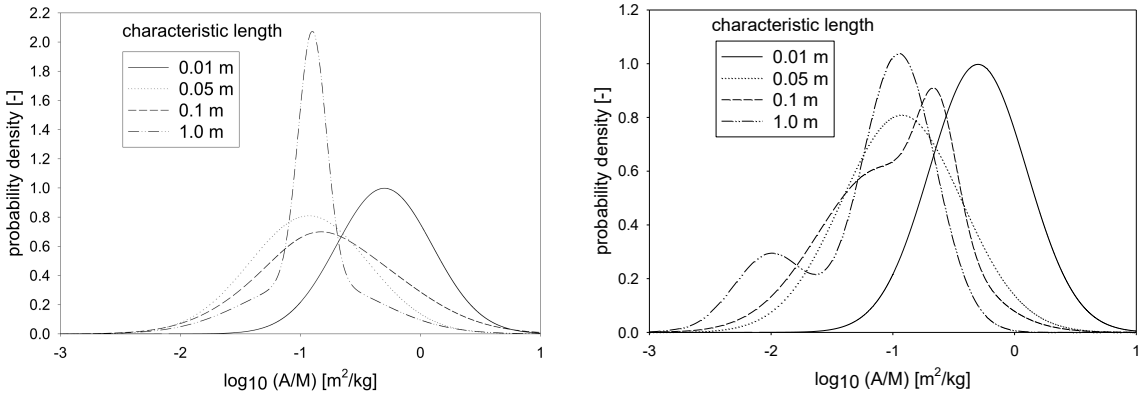
(a) $D = 0.02680$ and $p = 0.11663$

(b) Logarithmic view of (a)

Figure 6.3.: Comparison of area-to-mass ratios A/M between Python and C++ implementation in a modified state for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m. Here, the C++ implementation works without bridging function and small fragment distribution function. The result of the Kolmogorov–Smirnov test can be seen in (a). Here, with $\alpha = 0.05$, the equivalence is accepted

Iridium-33 Cosmos-2251 collision as we work with spacecraft fragments. First, we notice that our function's maximum should be at $\log_{10}(A/M) \approx -1$, as smaller fragments are more often generated and our minimal characteristic Length is 0.05 m. Moreover, $\log_{10}(A/M) > 0$ is still probable for $L_c \geq 0.05$ m sizes, yet the Python implementation does not produce those values in sufficient quantities, whereas the C++ does, as illustrated in Figure 6.2b. For even smaller values like 0.01 m, the function would be even more shifted to the right - a criterion which the Python implementation cannot satisfy as it lacks the equations for small fragments. The density function for $L_c = 0.1$ m is to be classified for the bridge function. Figure 6.4b shows that for fragments with $L_c \geq 0.1$ m, values of $\log_{10}(A/M) \leq -2$ are still probable. We even have a local maximum for fragments of the size 1.0 m there. Nevertheless, the Python implementation does not portray this behavior, whereas the C++ does, again as depicted in Figure 6.2b.

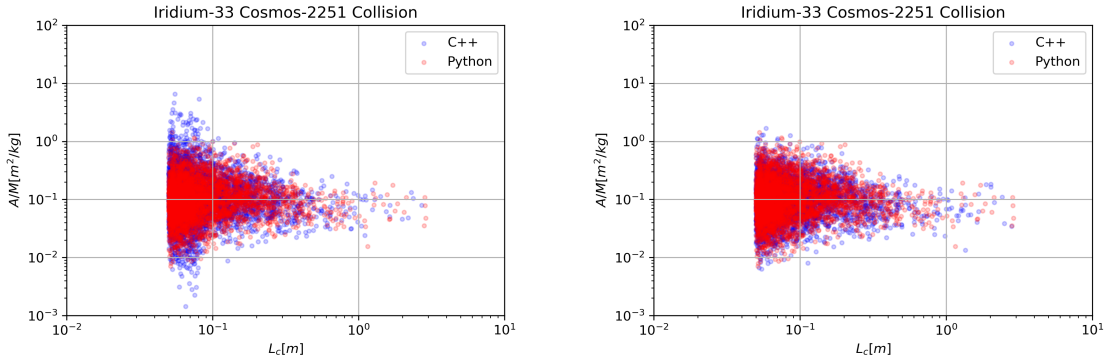
Figure 6.5a introduces another view of the given data. It plots the A/M values dependent on the L_c values. It is beneficial to illustrate the 0.05 m curve from Figure 6.4b and proves that the broadness of A/M values truly comes from the $L_c \approx 0.05$ m range. In contrast, Figure 6.5b shows the insufficiency of the Python implementation in this value range again.



(a) for R/B (rocket body) fragments

(b) for S/C (spacecraft) fragments

Figure 6.4.: Breakup probability density functions for A/M values constructed by using the NASA Breakup Model's Equation 2.8 and Equation 2.11; from [BJR+98]

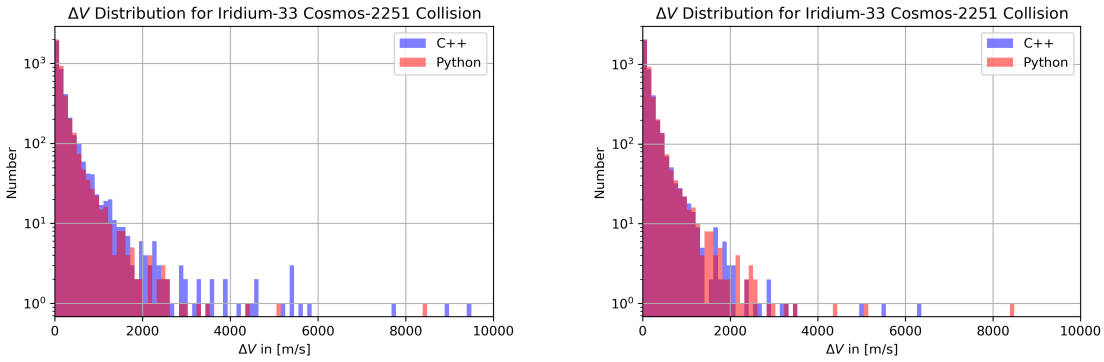


(a) C++ as it is

(b) C++ adapted to mirror insufficient Python

Figure 6.5.: Comparison of characteristic Lengths L_c to area-to-mass ratios A/M for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m.

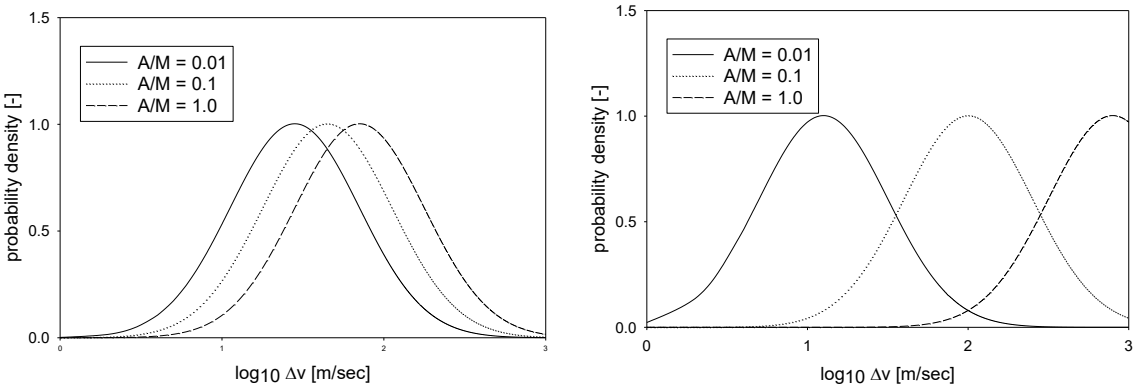
In the end, only the delta velocity distribution, aka the ejection velocity, remains. Since the NASA Breakup Model does not specify the direction of the ejected fragments, the following diagrams only show pure scalar velocities without any directional components. Again, Figure 6.6 and Figure 6.8 differentiate between the as-is state of the C++ and the modified version. Here, the C++ implementation noticeably produces more fragments with higher ejection velocities. This observation is not surprising since the ΔV distribution depends on the generated A/M values. And as the C++ implementation does not ignore the special demands for small fragments' A/M calculation, the distributions differ, and the result of a KS test ends in rejection of equivalence. However, we can analogously prove that our results better approximate the standards of the NASA Breakup Model by using Figure 6.7. It shows that higher ejection velocities values are more probable for higher A/M ratios, especially in the collision case depicted in Figure 6.7b. Due to the previously presented reasons that the Python implementation cannot produce high A/M, it generates fewer big scalar velocities. Figure 6.8 additionally depicts the A/M values and underlines the fact that higher A/M ratios lead to higher velocities. In conclusion, the broader A/M spectrum of the C++ implementation generates a broader velocity spectrum.



(a) C++ as it is
$D = 0.03920$ and $p = 0.00459$

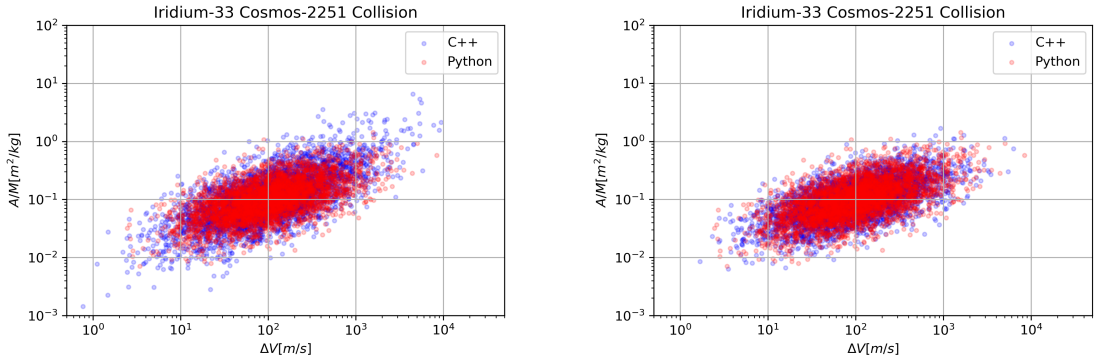
(b) C++ adapted to mirror insufficient Python
$D = 0.01947$ and $p = 0.44156$

Figure 6.6.: Comparison of scalar ejection Velocity Distribution ΔV between Python and C++ implementation for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m. One time with our accurate implementation and one time with the adapted version. Each scenario shows the results of a Kolmogorov–Smirnov test. Here, with $\alpha = 0.05$, the equivalence is rejected for (a) and accepted for (b).



(a) for explosions (b) for collisions

Figure 6.7.: Breakup probability density functions for ejection velocity values constructed by using the NASA Breakup Model's Equation 2.15; from [BJR+98]



(a) Python and C++ (C++ as it is)

(b) Python and modified C++ (C++ adapted to mirror insufficient Python)

Figure 6.8.: Comparison of scalar ejection Velocity Distribution ΔV to area-to-mass ratios A/M for the collision of Iridium-33 and Cosmos-2251 with $L_{c,min} = 0.05$ m

7. Results

After comparing and validating the results, this chapter will connect the calculated results to the actual fragmentation events. Therefore three famous events are portrayed: The collision of Iridium-33 and Cosmos-2251 in 2009, the deliberate destruction of Fengyun-1C with an anti-satellite missile in 2007, and lastly, the explosion of the Nimbus 6 rocket body in 1991. Asides from that, the data of the latter was utilized to derive the NASA Breakup Model. Thus, we can expect for this breakup event a relatively high precision in the model's replica.

Satellite	Total Cataloged Debris	with RCS Value	still in Orbit
Iridium-33	657	628	346
Cosmos-2251	1714	1668	1082
Fengyun-1C	3530	3438	2864

Table 7.1.: Numbers as of September 2021, determined with the NORAD Satellite Catalog[a]

[a] `https://www.celestrak.com/pub/satcat.csv` last accessed: 01.09.2021

Satellite	Total Cataloged Debris	with RCS Value	still in Orbit
Iridium-33	349	349	335
Cosmos-2251	809	809	785
Fengyun-1C	2680	2680	2630

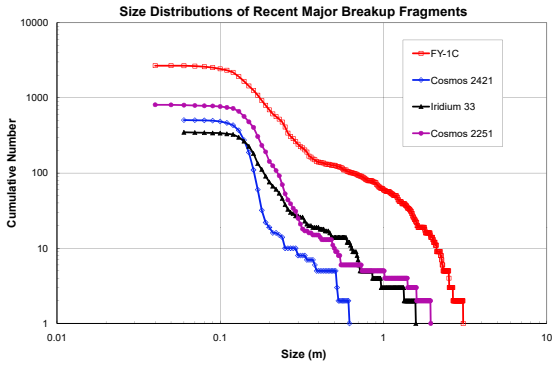
Table 7.2.: Numbers as of June 2009, from [LS09]

We will commence this section with the shared analysis of the characteristic Length L_c distributions for Iridium-33, Cosmos-2251, and Fengyun-1C. The Nimbus 6 R/B explosion will be examined later in its dedicated Section 7.4. Before we continue, one notes the following: Since 2007/ 2009, the number of cataloged objects for the fragmentation events has constantly been growing while radar systems detect new fragments. Table 7.1 and Table 7.2 clarify this circumstance. In order to make the comparison as comprehensive as possible, the following sections will not only present the original NASA references but also demonstrate derived statistics from the TLE and satellite data as of now. Moreover, one can note that radar capabilities limit fragment detection and that small fragments are more probable than big fragments, as presented in Subsection 2.3.1. As a result, small fragments are often overseen, yet they exist. Essentially there are only rare amounts of cataloged fragments of $L_c < 0.05$ m in size due to radar limitations. Therefore the following simulations are calibrated with a minimal characteristic Length of $L_c = 0.05$ m to compensate for the detection insufficiencies for small fragments. Nevertheless, this decision does not resolve unsatisfying radar detection in the size regime $0.05\text{ m} > L_c > 0.1$ m. This problem sharpens when looking at pure numbers. Table 7.3 shows the actual regularly tracked objects

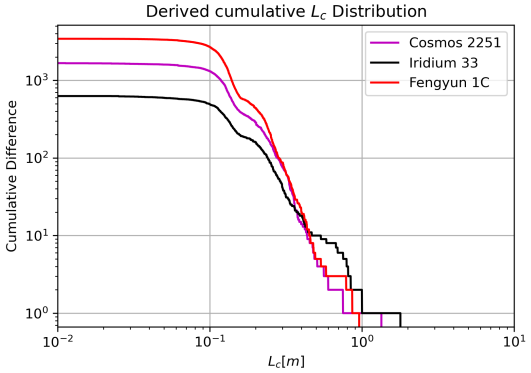
compared to the statistical estimated numbers for specific size regimes.

Regularly tracked objects	Estimated Numbers		
all size regimes	$L_c > 0.1$ m	$0.1\text{ m} \geq L_c > 0.01$ m	$0.01\text{ m} \geq L_c > 0.001$ m
$\approx 29.22 \cdot 10^3$	$\approx 36.5 \cdot 10^3$	$\approx 1 \cdot 10^6$	$\approx 330 \cdot 10^6$

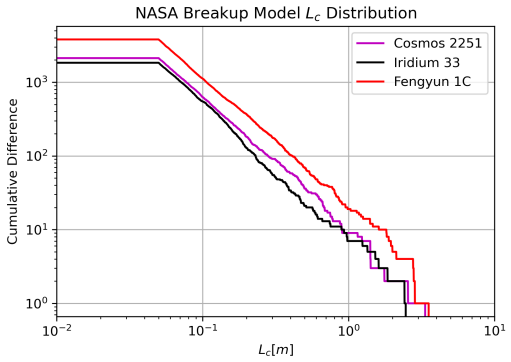
Table 7.3.: The table compares the actual tracked number by Space Surveillance Networks to the statistical estimated number; from [aE] in September 2021



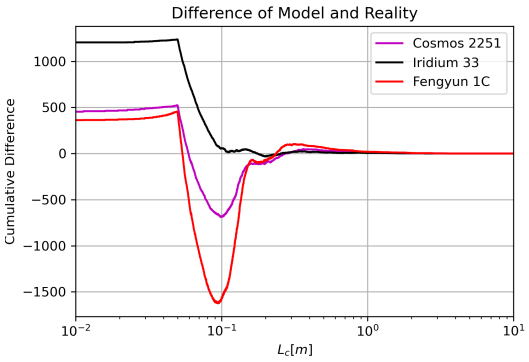
(a) NASA Reference from 2009; from [LS09]



(b) Derived cumulative L_c distribution based on the numbers with RCS value from Table 7.1. The L_c was derived by using Equation 2.1.



(c) Calculated cumulative L_c distribution based on the C++ NASA Breakup Model using $L_{c,min} = 0.05$ m



(d) The difference between the derived values from the satellite catalog (b) and the modeled data (c). A surplus means that the model generated more fragments than in reality, whereas the reverse holds for negative values.

Figure 7.1.: Comparison of cumulative L_c distributions

Figure 7.1c shows the results of our NASA Breakup Model's L_c distribution for the three fragmentations. It further presents in Figure 7.1a the original derived L_c distributions from 2009 and in Figure 7.1b the current L_c distributions derived from the radar cross section. We can see that the model precisely follows the power-law given in Equation 2.7 and that we still have some pitch for fragments smaller than 10^{-1}m, which is reasonable as our radar

detection for small fragments is limited. Since Cosmos-2251 is the bigger of the two satellites, the implementation assigns it the larger fragments, which holds for comparison with the results from Figure 7.1a, but interestingly not when regarding Figure 7.1b.

Figure 7.1d summarizes the statement of Figure 7.1b and Figure 7.1c by comparing their differences. One can note that the prediction for larger parts matches the expectations of reality. However, the model significantly underestimates the size range of 0.1 m, whereas the size regime around 0.5 m is overestimated. This fact is not surprising, as smaller fragments are more common and, consequently, it is implemented like that in the NSA Breakup Model, as shown in Subsection 2.3.1. Therefore, we can assume that the model is suitable, and the error lies in the limited radar data. On the other hand, we here follow the assumption of spherical geometry. Thus the derived size distributions for the actual events contain a modeling error, which could also explain the deviations for 0.05 m and 0.1 m.

The A/M and ΔV distribution will be shown individually in the following two sections: Section 7.2 and Section 7.3.

7.1. Data Methodology

Before discussing Figure 7.2 and continuing to Section 7.2, a brief exemplary introduction about the origin of Figure 7.2c and Figure 7.2d is given. This explanation also applies to subfigures (c) and (d) of Figure 7.4 and Figure 7.7. Generally, we strived for current data to compare results because the reference data from 2007/ 2009 is incomplete in terms of cataloged satellites. So in order to produce diagrams depicting A/M and L_c values, both of them must be independent. Therefore the L_c values were determined by Equation 2.1 by using the radar cross section. In contrast, the A/M value was derived by the starred ballistic coefficient B^* from the TLE data. These two independent sources enable us to plot diagrams that do not show any noticeable power-law relation between the values. Equation 7.1 shows the correlation between the mentioned B^* given in the TLE in $[R_{Earth}^{-1}]$, the ballistic coefficient B, the atmospheric density ρ_0, the drag coefficient C_d, and the area-to-mass ratio A/M:

$$B^* = \frac{\rho_0 \cdot B}{2} = \frac{\rho_0 \cdot C_D}{2} \cdot A/M \tag{7.1}$$

We used $\rho_0 = 0.1570 kg/m^2 \cdot R_{Earth}$ [wik] as the atmospheric density and as coefficient of drag $C_d = 2.2$ [PA11] for the given diagrams. Nevertheless, this model uses "a fixed density at any given height" [AP09], which "does not vary with solar activity" [AP09]. As a result, B^* is underestimated. Thereby the produced A/M ratios are not sufficient in their order of magnitude. So as a consequence, we applied the proposed scaling factors from [PA11] and [PA09] in order to produce fitting results. After all, our diagrams consisting of only current satellites on-orbit like in Figure 7.2c can be compared to the reference source like in Figure 7.2b. However, if we apply the presented strategy on all cataloged fragments, even those already decayed, we get higher A/M values than expected.

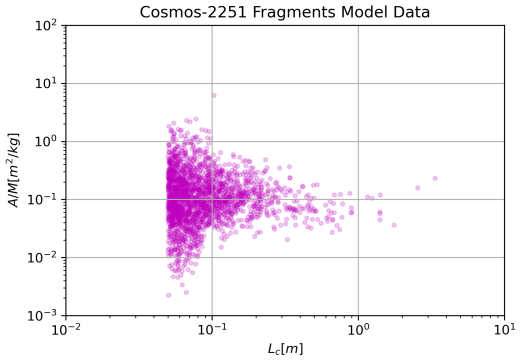
We assume that these deviations come from an overestimated B^*. As utilized in Figure 7.2d, the whole TLE set consists of a significant number of already decayed satellites' last created TLE data. So as they were decaying, they were in deeper atmospheric layers at these moments, resulting in higher aerodynamic drag, finally leading to a bigger B^*. Moreover,

one notices that these parameters were calculated in 2009 and correlated to atmospheric density and solar activity back then. So, in conclusion, these diagrams (c) and (d) are helpful for comparison, but they should be handled with caution as they mirror the simplifications above.

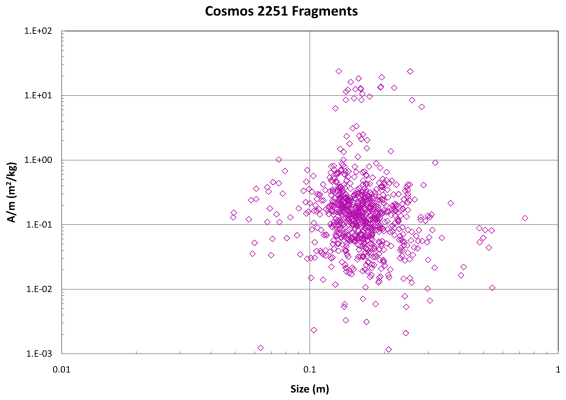
7.2. The Collision of Iridium-33 and Cosmos-2251

The here presented figures use the parent assignment feature of the C++ implementation and group the fragments of this collision by parent bodies. Figure 7.2 and Figure 7.4 portray the characteristic Length L_c to area-to-mass ratio A/M distribution. It is noticeable that the model fits well for the Cosmos-2251 fragments, whereas the prediction for the Iridium-33 satellite is inadequate. The model predicts the maximum frequency at $\log_{10}(A/M) \approx -1$, yet in reality, it is around $\log_{10}(A/M) \approx -0.5$. Regardless, we could expect this difference as the NASA Breakup Model is based on observations and experiments before Iridium-33's deployment, but the satellites' construction changed over time. Since the NASA Breakup Model does not consider lightweight composite materials like they were used for Iridium-33 or other modern vessels [LS09], it predicts too low A/M ratios. To summarize, the observed values of Figure 7.4b are reasonable because, as Subsection 2.3.2 mentioned, high A/M values indicate light materials. Figure 7.4d would confirm this observation of significantly higher A/M values in reality, although one has to be careful with Figure 7.4d as of its simplifications explained in Section 7.1. Otherwise, Figures (c) and (d) for both Iridium and Cosmos further reveal the limitations of radar tracking since most fragments are distinctly visible in the area around $L_c \approx 0.1$ m.

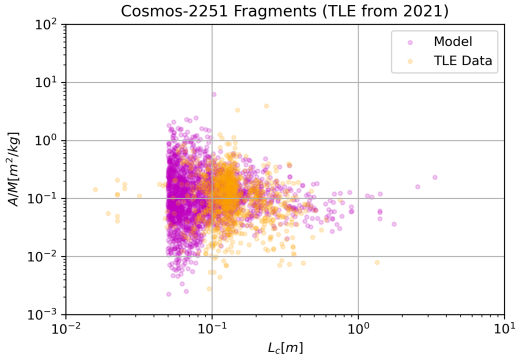
Figure 7.3 and Figure 7.5 complete this section by reviewing the fragments' velocity to their A/M ratio. Both diagrams do not show any significant deviation from the general behavior that bigger A/M values indicating smaller masses have greater velocities than more massive fragments, which is completely reasonable under consideration of energy conservation. Lastly, Figure 7.6 shows the Iridium-33 Cosmos-2251 collisions ejection velocities including their directions, one time logarithmically scaled.



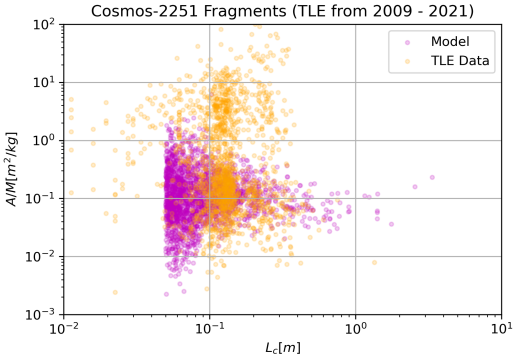
(a) The C++ NASA Breakup Model's prediction



(b) NASA Reference from 2009; from [LS09]



(c) Reconstructed distribution based on current TLE and Satellite Catalog data (only 2021 and still in orbit); see Section 7.1



(d) Reconstructed distribution based on all available TLE and Satellite Catalog data (2009-2021); see Section 7.1

Figure 7.2.: Cosmos-2251 fragments' L_c to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.

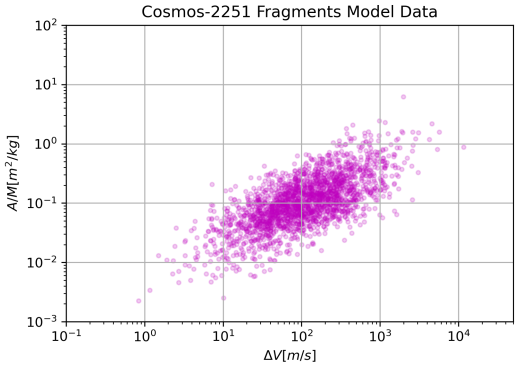
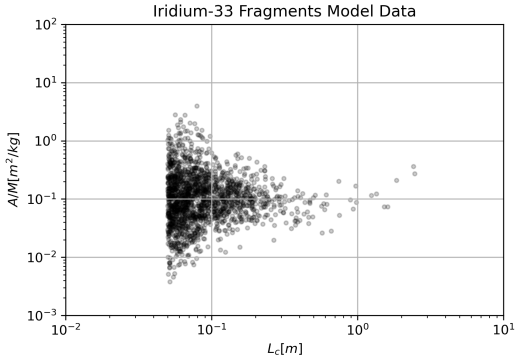
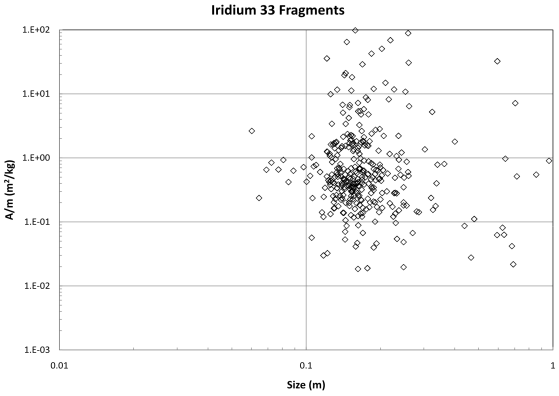


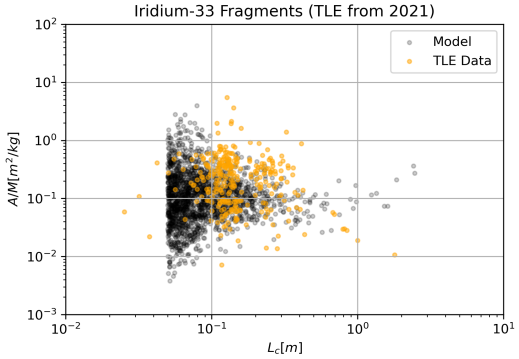
Figure 7.3.: Cosmos-2251 fragments' ΔV to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.



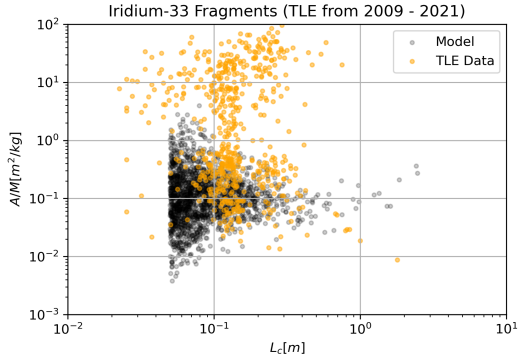
(a) The C++ NASA Breakup Model's prediction



(b) NASA Reference from 2009; from [LS09]



(c) Reconstructed distribution based on current TLE and Satellite Catalog data (only 2021 and still in orbit); see Section 7.1



(d) Reconstructed distribution based on all available TLE and Satellite Catalog data (2009-2021); see Section 7.1

Figure 7.4.: Iridium-33 fragments' L_c to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.

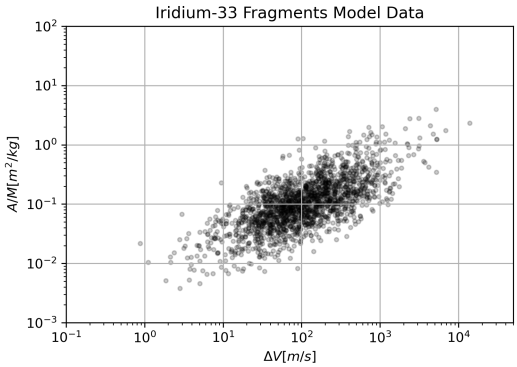
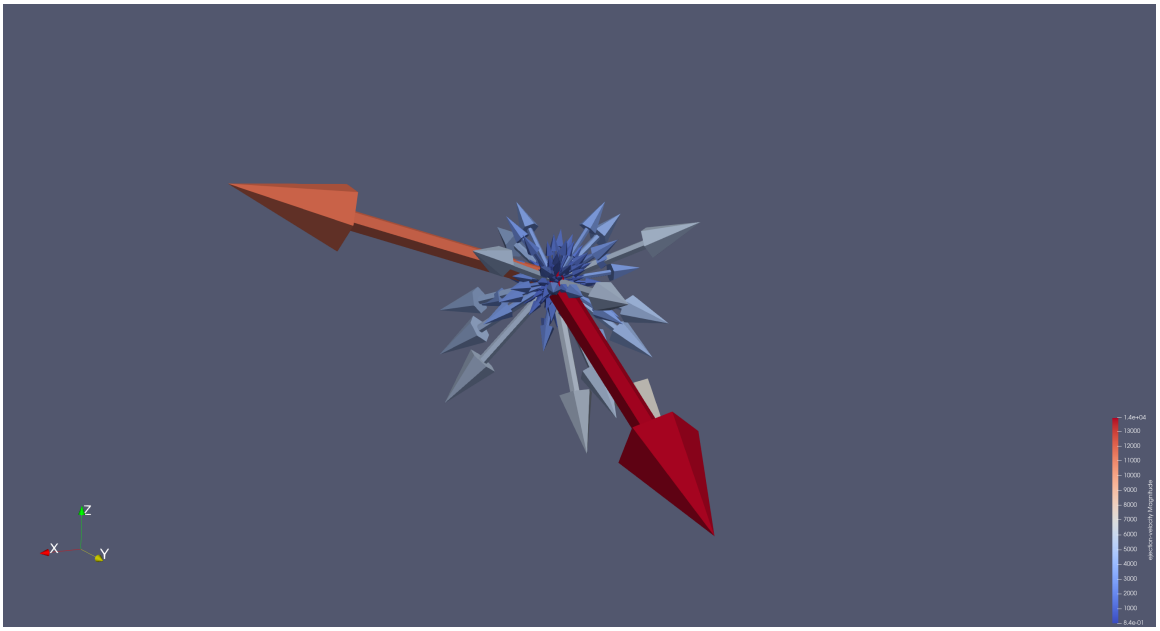


Figure 7.5.: Iridium-33 fragments' ΔV to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.



(a) Standard scale

(b) Logarithmic scale

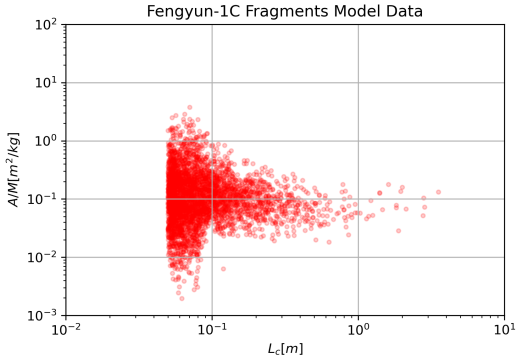
Figure 7.6.: Iridium-33 and Cosmos-2251 Collision fragments' $\vec{v}_{ejection}$ visualized with *ParaView*. In total, there are 3954 fragments as the simulation was run with $L_{c,min} = 0.05$ m

7.3. The Destruction of Fengyun-1C

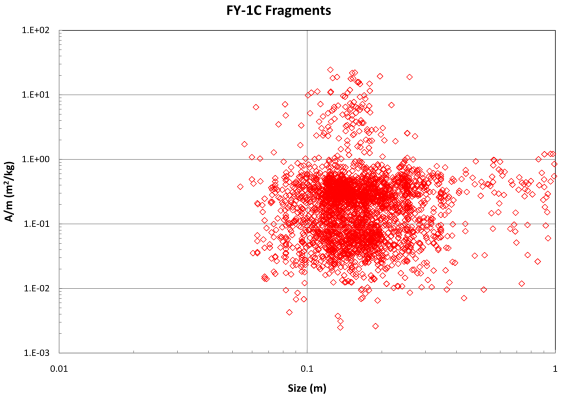
The People's Republic of China tested an anti-satellite missile against one of their dysfunctional weather satellites on January 11, 2007. This deliberate destruction of a satellite led to the breakup event with the most fragments cataloged until now, 2021. Since it was a kinetic kill vehicle, no explosives were involved, and therefore we modeled the event like a collision of a warhead with Fengyun-1C. The minimal characteristic Length has been set to 0.05 m and Fengyun-1C's mass to 880 kg. In contrast, there were no explicit data sets available on the used warhead, but only its type: SC-19. Therefore, we have chosen to model the warhead with 500 kg of mass. Lastly, the impact velocity was well-known and set to 9.36 km/s. [PA09]

First of, the here presented plots for the Fengyun-1C fragmentation do not differentiate between the former satellite's and warhead's fragments as in other literature. The motivation behind this decision is that the combined number of fragments models the actual event's amount better.

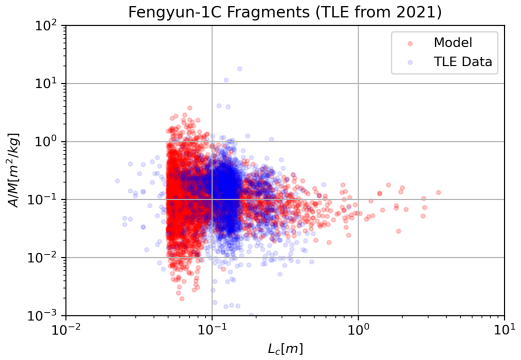
The resulting figures are similarly structured as in Section 7.2, with Figure 7.7 illustrating the A/M distribution and Figure 7.8 portraying the ejection velocities. Overall, it is noticeable that the simulation models the fragments decently, with most fragments having values in the corridor of $\log_{10}(A/M) \in [-2; 0]$. Overall, if just taking radar limitations and the current available TLE Data into consideration, the results of Figure 7.7c are highly fitting to the NASA Breakup Model's predictions. In contrast, Figure 7.7d shows more fragments with a high A/M ratio. We can interpret this in two ways: Either the NASA Breakup Model underestimates the fragment count of the 0.1 m size range. This hypothesis is strengthened by Figure 7.7b containing the original NASA Reference of 2009. Or alternatively, Figure 7.7d's derived TLE Data contains an error, as assumed by Section 7.1.



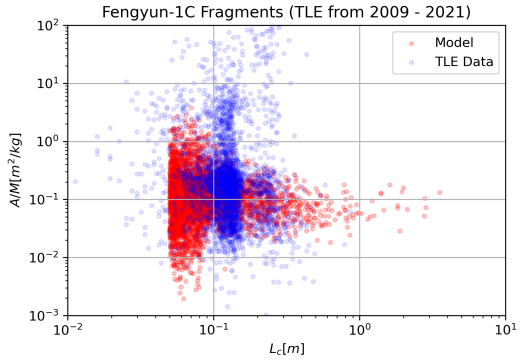
(a) The C++ NASA Breakup Model's prediction



(b) NASA Reference from 2009; from [LS09]



(c) Reconstructed distribution based on current TLE and Satellite Catalog data (only 2021 and still in orbit); see Section 7.1



(d) Reconstructed distribution based on all available TLE and Satellite Catalog data (2009-2021); see Section 7.1

Figure 7.7.: Fengyun-1C fragments' L_c to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.

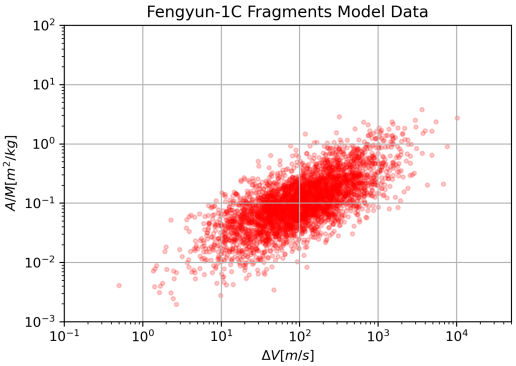
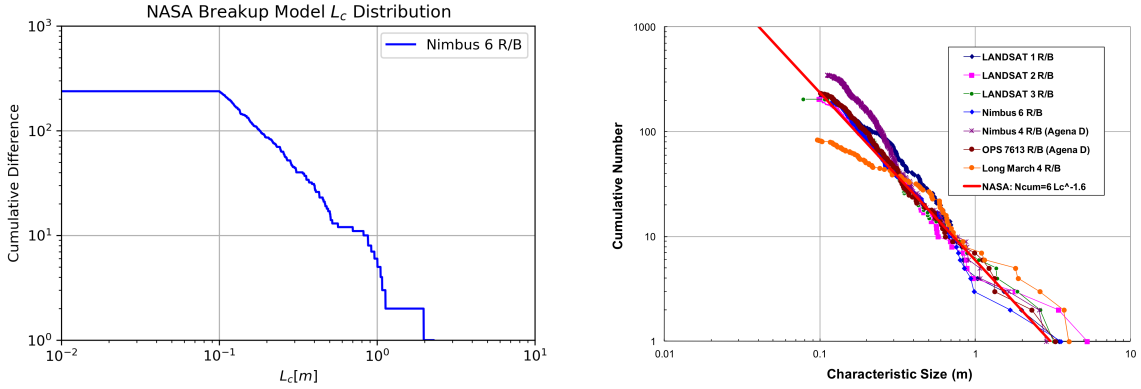


Figure 7.8.: Fengyun-1C fragments' ΔV to A/M value distribution. The simulation was run with $L_{c,min} = 0.05$ m.

7.4. The Explosion of the Nimbus 6 R/B

The last section of this chapter handles the Nimbus 6 upper stage explosion. It was a Delta 2nd stage with a mass of 839 kg, which exploded due to an error with the propulsion system on May 1, 1991. [BJR+98] Although it is given, the mass has no direct influence on the number of generated parts in case of an explosion, as mentioned in Subsection 2.3.1. The simulation has been run with a minimal characteristic Length L_c of 0.1 m, and as Figure 7.9 underlines, the size distribution of the simulation-run nearly overlaps the distribution of the actual event. This fact is not surprising since the Nimbus 6 R/B explosion was one event taken into account for the derivation of the NASA Breakup Model. The latter two, Figure 7.10 and Figure 7.11 show the A/M and the Delta Velocity distribution. One might ask why no other comparison figures are given. Overall there are two reasons. First of all, there are only rare figures showing the Nimbus 6 R/B explosion in literature. Secondly, the approach we took to create the formerly presented L_c to A/M figures needs scaling factors that have not been calculated for the Nimbus 6 R/B explosion.



(a) The C++ NASA Breakup Model's prediction with $L_{c,min} = 0.1$ m

(b) NASA Reference, from [Lio12]

Figure 7.9.: Nimbus 6 R/B fragments' L_c distribution (both in blue). The NASA reference shows additional lines illustrating other upper stage explosions. The red line illustrates the power-law of Equation 2.6. The exact match is no coincidence because these are the explosions used to derive Equation 2.6 of the NASA Breakup Model.

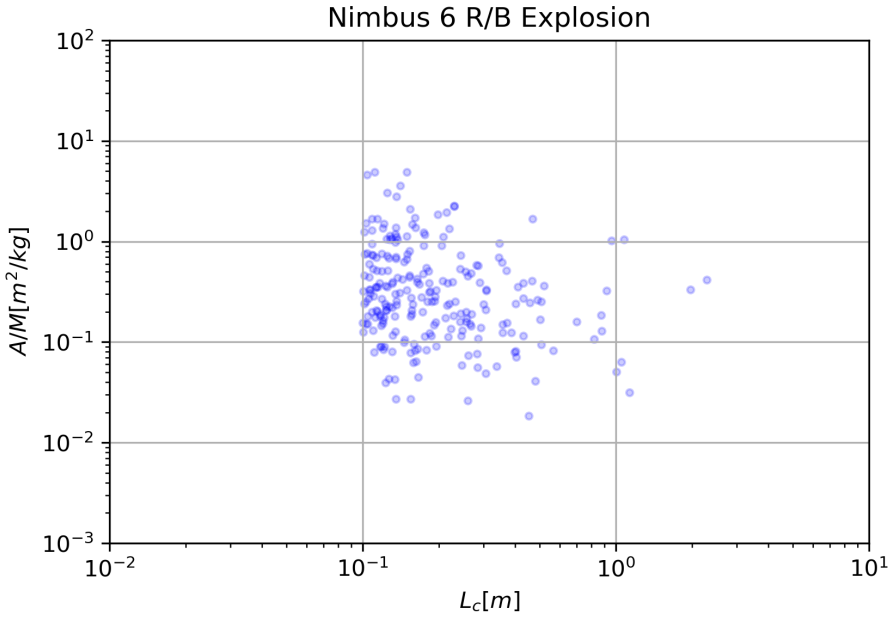


Figure 7.10.: Nimbus 6 R/B fragments' L_c to A/M value distribution. The simulation was run with $L_{c,min} = 0.1$ m.

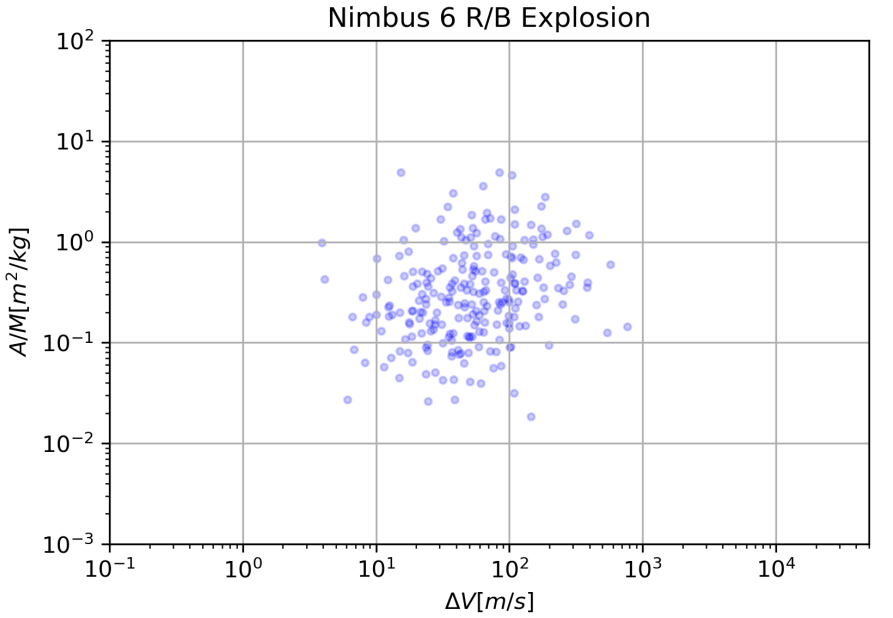


Figure 7.11.: Nimbus 6 R/B fragments' ΔV to A/M value distribution. The simulation was run with $L_{c,min} = 0.1$ m.

8. Runtime Measurements

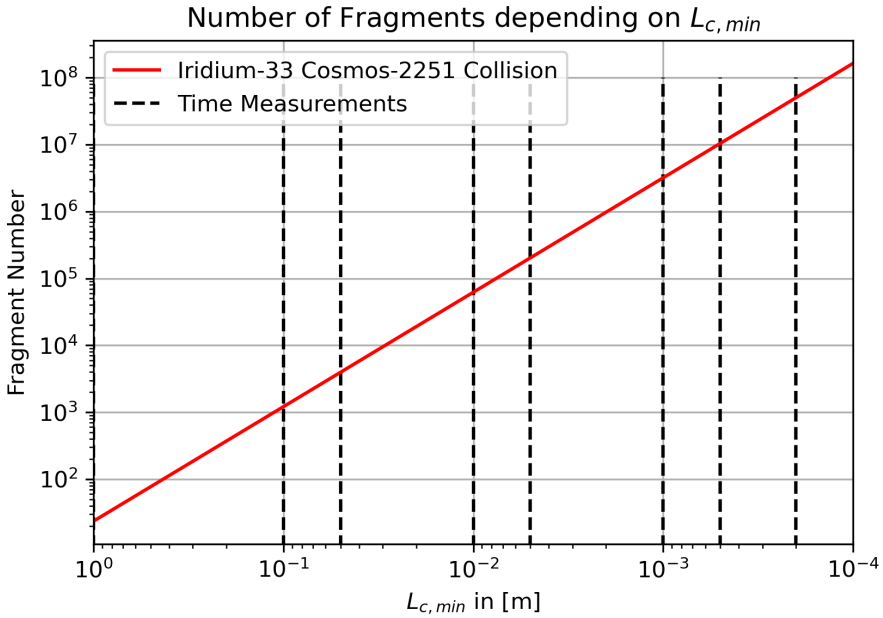


Figure 8.1.: Cumulative fragment number for a certain minimal characteristic Length $L_{c,min}$ for the collision of Iridium-33 and Cosmos-2251. Here, no mass conservation was enforced. The dashed lines illustrate the time measurement points of Chapter 8.

This chapter provides detailed runtime measurements for the Iridium-33 Cosmos-2251 Collision scenario. Since the formulas remain the same for other collisions or explosions, this chapter exemplifies any simulation with a specific output size depending on the minimal characteristic Length $L_{c,min}$. First, the most significant changes on the way to better runtime are portrayed, and later we will investigate the influence of the mass conservation option, which is alongside the characteristic Length $L_{c,min}$, the only variable able to change the output volume. All plots depicting runtime are made on a computer with the following hardware specifications: *Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz* and *16GB DDR4-3200*. Overall, the processor has in total eight physical and sixteen virtual cores. The program was compiled in *RELEASE* mode permitting the compilers *GCC 9.3* and *Clang 10.0* to make extensive optimizations.

The Figure 8.1 below shows the minimal characteristic Lengths utilized for the creation of the here presented figures. The last data point for time measurements is at $L_c = 0.0002$ m instead of 0.0001 m, because the testing computer failed to reserve the necessary memory for $L_c = 0.0001$ m, which corresponds to approximately 163 million fragments. For comparison reasons, the simulation allocates and deallocates for the scenario $L_c = 0.0002$ m around

9.0 GB of RAM as tracked by *Intel's VTune Profiler*.[1] However, this is not the peak memory consumption since it also contains temporary allocations. Therefore, the system's manager only recorded a maximum memory consumption of around 6.0 GB. The two last presented data points are more interesting for runtime analysis than actually using their results as the NASA Breakup Model was only validated for sizes between 1 mm and 1 m, as mentioned in Subsection 2.3.1.

8.1. Milestones of Runtime

The runtime of the simulation mainly depends on the simulation-run itself and the efficiency of the utilized model, granted that we can amortize the runtime of input and output. Accordingly, two components of Chapter 4 are relevant if we strive for the shortest possible runtime: The `Model` and the `Simulation`. The first optimization was accomplished for the `Model` in changing the fundamental programming paradigm. Initially, we started with an object-oriented approach modeling the fragments as satellites saved in a `std::vector`. This strategy corresponds to the Arrays of Structure (AoS) approach, which is generally excellent from a designer's viewpoint since it is intuitive but not from a computational perspective. By decreasing the rate of cache misses using neighborhoods, we can increase performance. Thereby, the successive data must lie side by side in memory. Consequently, we moved from an object-oriented approach to the data-oriented Structure of Arrays (SoA) ansatz, in which we concentrate all data into a single object. The SoA fixes the memory bottleneck as the next date can be prefetched before the CPU needs it.

Nonetheless, this ansatz is only applied for the simulation's output but not for the input because we amortize the input on the one hand. On the other hand, the arrays of structures' ansatz is more suitable for the input as it is flexible. This flexibility comes in handy when reading satellites from an input file, and those satellites are constructed from different means like TLE, YAML, or CSV using a dedicated uniform Builder interface.

In conclusion, the simulation works with an SoA layout which improves the runtime of the whole project. These changes become apparent in Figure 8.3 and Figure 8.4, which illustrate the speed gains absolute and relative. The blue curve represents the AoS, whereas the orange curve represents the SoA layout. Notably, the SoA seems to be as good as the AoS approach. However, comparing the relative speed, illustrated in Figure 8.4, shows that the AoS layout is slightly better than the SoA until a minimal characteristic Length $L_{c,min} \approx 0.01$ corresponding to around 62000 fragments. This observation seems reasonable since if we assume that the advantage of successive data is only of minor size for each fragment's generation, then this advantage would accumulate to more significant values with increasing fragment amounts.

Regardless one must add that the paradigm change happened in the mid of the implementation when not every feature was already implemented, e. g., the mass conservation was implemented later. Here, we speak of the always activated upper bound, not the later considered enforcement as lower bound. So the here plotted SoA implementation has slightly more work to do than the AoS layout, and despite this, it is faster than the AoS implementation for a higher number of generated debris. Therefore the SoA ansatz wins the

[1] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html` last accessed: 10.0.09.2021

duel of faster runtime even if the AoS is insignificantly better sometimes.

The transition to the SoA layout paved the way for further optimizations with the most significant being the introduction of parallelism. The parallelization has been accomplished using the C++ 17 features of `std::execution`, enabling us to efficiently transfer sequential `for_each` statements into parallel execution. As each fragment is independent of the other, this can be performed lock-free without risking correctness. We only need to be careful with generating random numbers to avoid race conditions when accessing the corresponding generator. This issue has been resolved using the storage specifier `thread_local`, creating one generator instance for each thread's lifetime. These changes particularly accelerated the more enormous scenarios with millions of fragments being generated.

The effects of the undertaken parallelization measures are presented in Figure 8.2, which illustrates the exemplary core utilization of one simulation-run. First of all, as profiled with *VTune*, the general performance bottlenecks are the generation of random numbers and the frequent unavoidable utilization of the pow function. The parallelization efforts directly counter these problems, even though the workload is only divided among processors and not decreased. Figure 8.2 shows the general core utilization, as measured by *VTune*, for the scenario of $L_c = 0.0005$ m. We cannot achieve full parallelization since, on the one hand, the input will always be only sequential executable, and on the other hand, time is lost due to thread creation and joining. According to *VTune*, the latter is the reason behind most registered waiting time, with only a few cores used in Figure 8.2. Despite the disadvantages, the presented simulation performs better with parallelization than without, as presented in Figure 8.3.

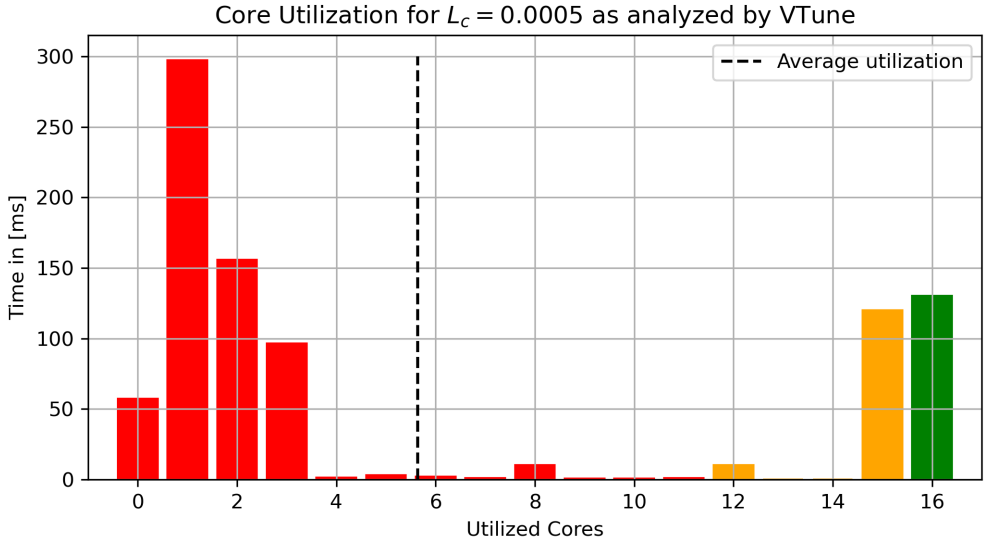


Figure 8.2.: The Breakup Simulation with $L_{c,min} = 0.0005$ analyzed with *VTune*

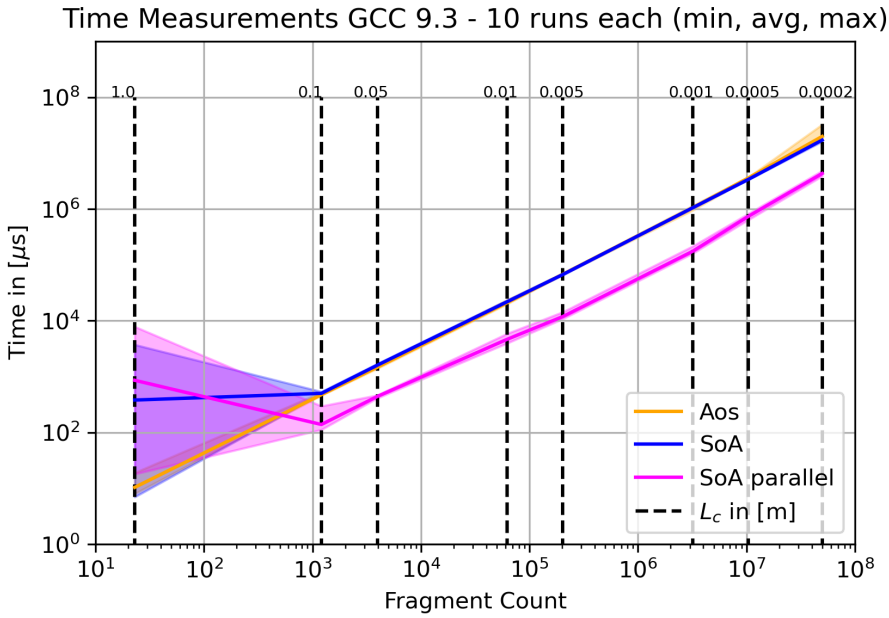
Furthermore, the SoA layout permits us not only to parallelize the calculation but also to vectorize operations. Thus, we do not use the `std::execution::parallel_policy` but rather `std::execution::parallel_unsequenced_policy`. Nevertheless, the main bottleneck methods generating characteristic Lengths, area-to-mass ratios, and ejection velocities

do not use vectorization, presumably due to synchronization issues with the random number generator. However, the simulation makes use of vectorization in the tuple-view methods (see Section 4.2) and in the parent assignment method (see Section 4.4) as profiled with *Intel Advisor*[2].

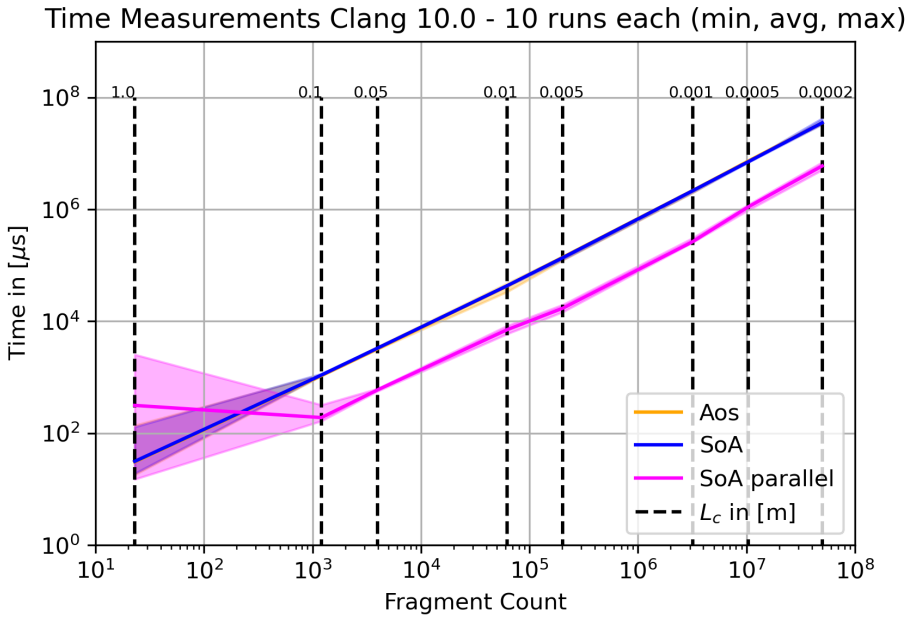
In summary, the last step, which parallelized the simulation-run, has improved runtime significantly, as Figure 8.3 and Figure 8.4 depict. Here, the latest change is plotted in magenta. It is easily overseen that the improved parallel implementation is on average six to eight times faster than the sequential executions with exception of few generated fragments correlating to great minimal characteristic Lengths $L_{c,min}$. However, this circumstance is not surprising as thread creation takes some time, and it is inefficient to create a thread just for little work.

Figure 8.3 and Figure 8.4 show the results twice, one time for the *GCC 9.3* and the other time for *Clang 10.0*. The proportions of the presented data mainly remain the same in both versions. However, the *GCC 9.3* is on average faster than *Clang 10.0*. This circumstance becomes especially apparent at a fragment count of 10^7. In the end, this comparison should emphasize the robustness of the implementation to different compilers since the speed differences are only of low magnitudes. For completeness, Section 8.2 continues with the dual presentation of its results.

[2] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html` last accessed: 14.09.2021

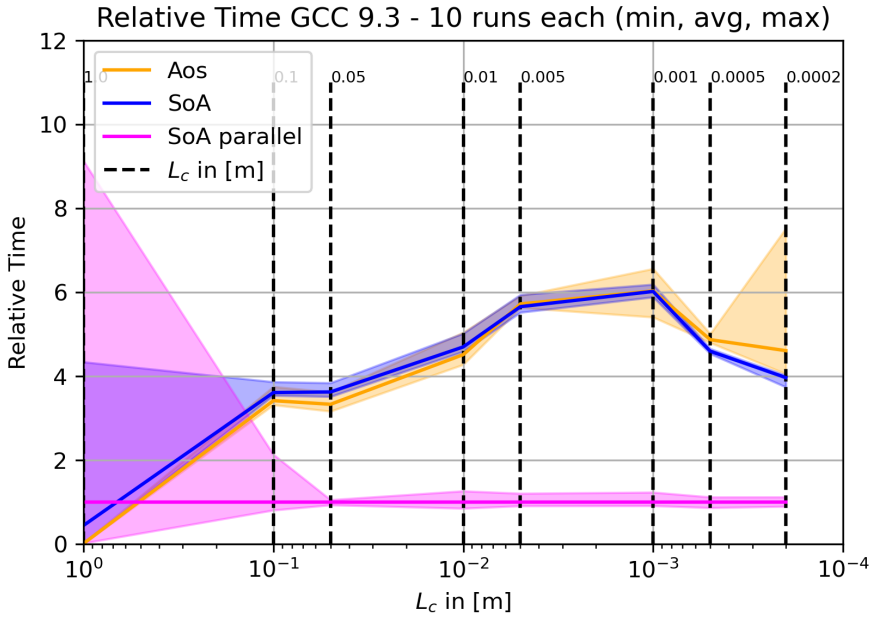


(a) Compiled with GCC 9.3

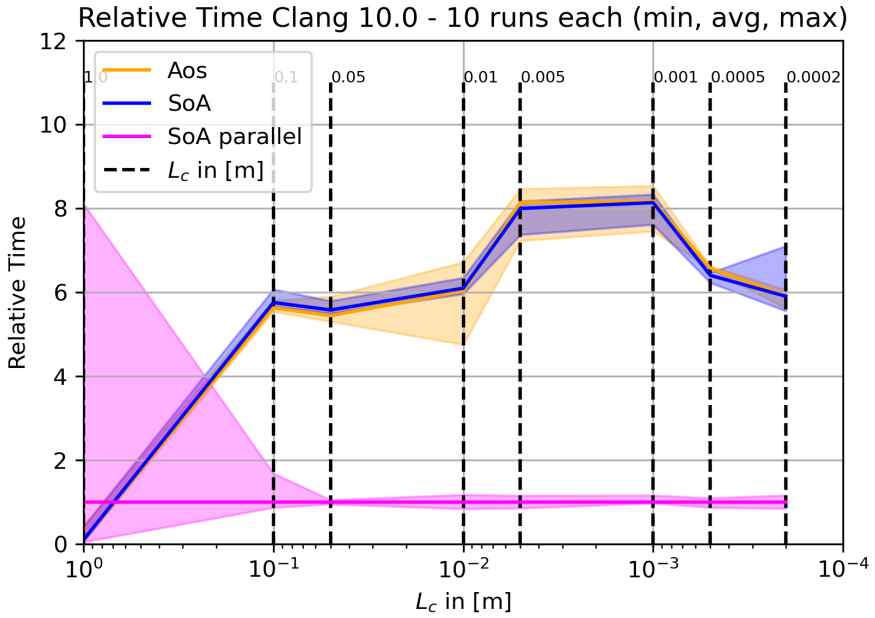


(b) Compiled with Clang 10.0

Figure 8.3.: Required calculation time [μs] compared to the number of fragments. The solid line marks the average time of ten sequential executed simulation-runs, whereas the colored zone around marks respectively the slowest and fastest simulation-run. Different implementation paradigms are colored differently. The dashed vertical lines illustrate the measurement points as presented in Figure 8.1. Figure 8.4 portrays a relative comparison since AoS and SoA approaches are nearly overlapping in these subfigures.



(a) Compiled with GCC 9.3



(b) Compiled with Clang 10.0

Figure 8.4.: Relative required calculation time for a specific minimal characteristic Length L_c. The solid line marks the average time of ten sequential executed simulation-runs, whereas the colored zone around respectively marks the slowest and fastest simulation-run. Different implementation paradigms are colored differently, and the parallel Structure of Arrays has been normed as one. The dashed vertical lines illustrate the measurement points as presented in Figure 8.1.

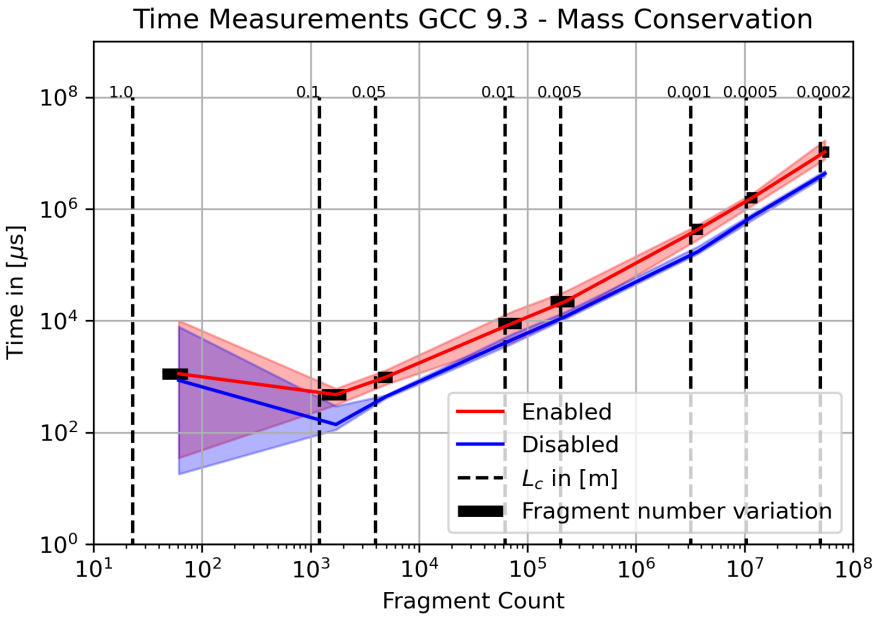
8.2. Influence of Mass Conservation

After introducing the three milestones of runtime optimization, this section advances with the option of mass conservation enforcement. Subsection 2.3.2 already proposed this procedure as controversial. Thus the filling of the mass budget, so that input mass equals output mass, is implemented optionally. If enabled, the simulation generates more fragments in case of an unfilled mass budget. Figure 8.5 still contains the vertical dashed lines illustrating the number of fragments created for a specific $L_{c,min}$ as proposed by the collision size distribution of the NASA Breakup Model. Notably, those lines are not overlapping anymore with the actual data points, each marked with a horizontal black line. Instead, these horizontal black lines begin at the minimal generated amount of fragments over ten simulation-runs and end at the maximum generated amount. The enforcement of the mass budget frequently generates more fragments because the horizontal black line is mainly on the right of the first estimation made by the Breakup Model's size distribution.

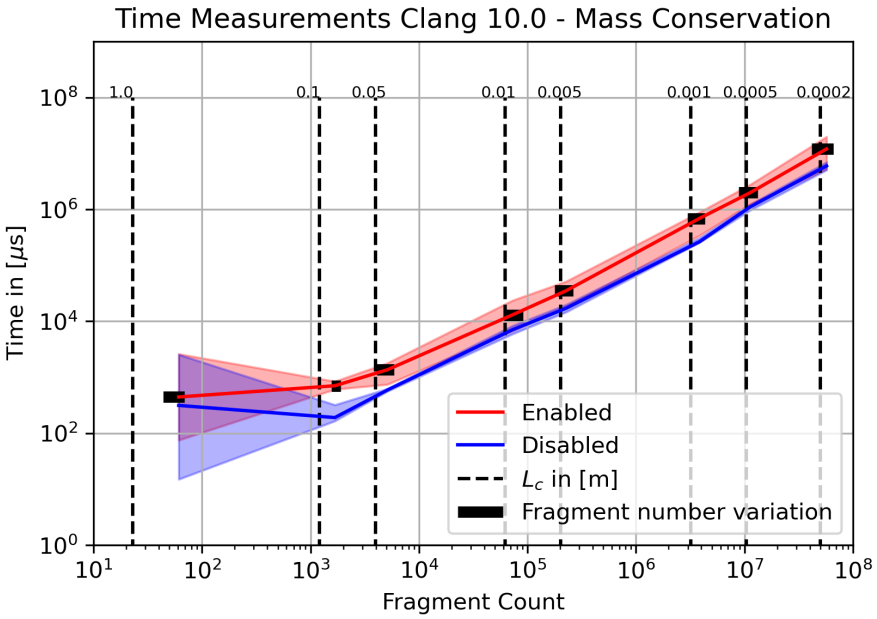
Consequently, the runtime usually is higher than with disabled mass budget fulfillment. Besides the triviality of more fragments being generated, there is another reason why the runtime is slower than when the option is turned off: The simulation cannot predict how many fragments it has to generate until the mass budget is fulfilled. Thus, it cannot foresight the memory requirements and has to work greedy by just allocating as much as it is best at the moment. Hence, the strategy generates and allocates space for only one fragment at a time since this fragment would always be the last if the best case applies.

Another observation of Figure 8.6 is that the runtime is similar to a normal execution flow in the best case. In view of the fact that in the best case, no additional fragments have to be produced or fragments only have to be removed because of exceedance, the simulation behaves likewise as with disabled enforcement.

The essence of this section is the evidence that the option of mass enforcement costs runtime and makes the simulation, on average, approximately two times slower as illustrated in Figure 8.6.

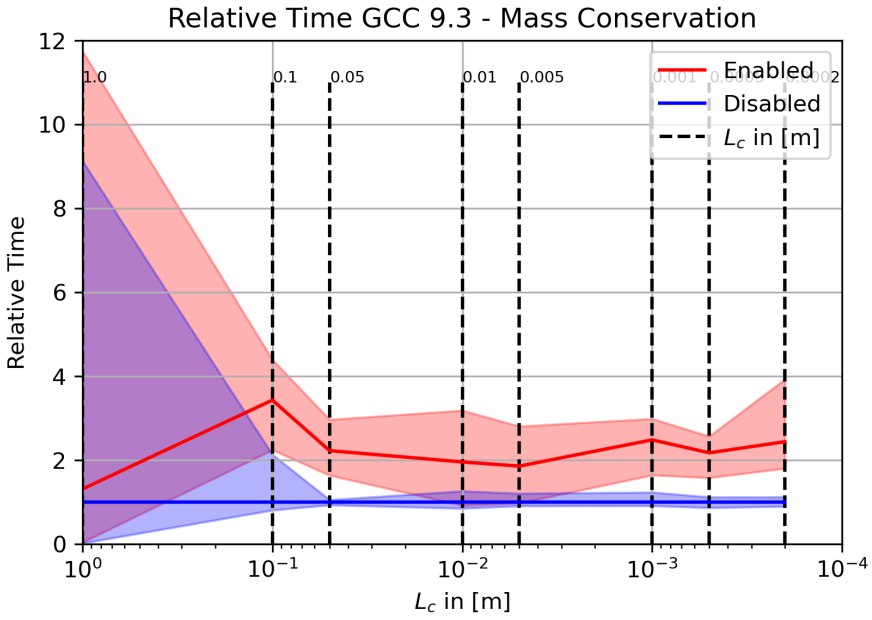


(a) Compiled with GCC 9.3

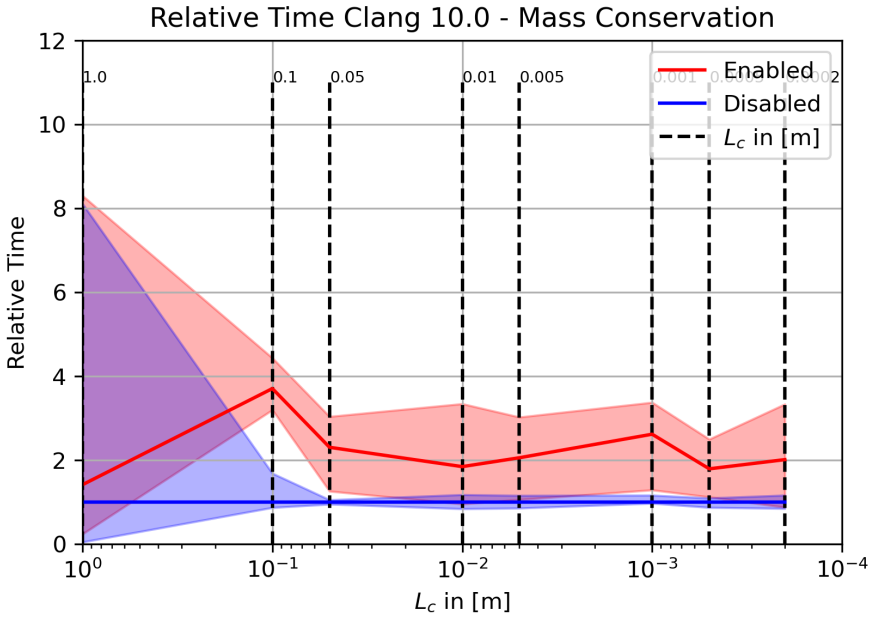


(b) Compiled with Clang 10.0

Figure 8.5.: Required calculation time [μs] compared to the number of fragments for enabled and disabled option mass conservation. The solid line marks the average time of ten sequential executed simulation-runs, whereas the colored zone around respectively marks the slowest and fastest simulation-run. Different implementation paradigms are colored differently. The dashed vertical lines illustrate the measurement points as presented in Figure 8.1. The horizontal black bars mark the lowest and highest produced fragment number over the ten simulation-runs. Notably, the fragment count is mainly greater than with disabled mass conservation marked by the dashed vertical lines.



(a) Compiled with GCC 9.3



(b) Compiled with Clang 10.0

Figure 8.6.: Relative required time of enabled mass conservation compared to a disabled option (normed to one) for a specific minimal characteristic Length L_c. The solid line marks the average time of ten sequential executed simulation-runs, whereas the colored zone around respectively marks the slowest and fastest simulation-run. The dashed vertical lines illustrate the measurement points as presented in Figure 8.1.

9. Conclusion

This chapter sums up the whole thesis and provides an overview of opportunities building on top of this work.

9.1. Summary

We started the journey by introducing the importance of reliable models to predict outcomes of collisions and explosions in an ever-growing environment of on-orbit objects for the purpose of understanding the consequences of anthropological behavior in space. Chapter 2 further introduced the NASA Breakup Model as an often utilized way to predict these events empirically. It defined the significant fragment properties of interest like characteristic Length L_c, area-to-mass ratio A/M, and the ejection velocity $v_{ejection}$. The subsequent Chapter 3 tied on these fundamentals and created the context around this work. It explored different available implementations and underlined the necessity of an open-source, fast, reliable, and extensive implementation of the NASA Breakup Model in modern C++. Nevertheless, it also uncovered some ongoing improvement efforts for future breakup modeling.

The elegance of the involved architectural design was further traversed in Chapter 4, including the major principle of low coupled components and their highly cohesive interiors, which enables the reuse of only certain components like the `Model` as stand-alone. In addition, Section 4.4 defined relevant decisions made in order to create a complete implementation. These decisions included the definition of a bridge function by using linear interpolation to solve the A/M gap between fragments' sizes of 0.08 m and 0.11 m or the parent satellite assignment algorithm. Finally, Chapter 5 presented the different input use cases and syntactic sugar for the library application. It introduced the possibilities of automatic, semi-automatic, or manual determination of input parameters by using YAML files, TLE Data and the satellite catalog.

The last part included the qualitative comparison with the Python implementation, which served as the primary reference source. Moreover, Chapter 6 contained a description of the utilized testing frameworks. As the qualitative comparison with the Python Implementation was successful, we further illustrated its insufficiencies in the boundary areas of A/M ratio determination. Finally, we presented the results of our implementation able to approximate the NASA Breakup Model's characteristics better than the Python implementation due to the consideration of every functional use case.

Chapter 7 depicted the modeled results of three major events, including the Iridium-33 Cosmos-2251 collision, the Fengyun-1C anti-satellite test, and the Nimbus 6 R/B explosion. Additionally, it illustrated that the original model is partly insufficient for modern satellites like Iridium-33 since it does not consider the material composition of modern satellites. It thereby underestimates fragments with high A/M values indicating lighter materials. Nevertheless, the most results were fitting under the constraint of humankind's limited radar capabilities.

Lastly, Chapter 8 introduced the efforts undertaken to improve the performance of the presented C++ implementation, which essentially are the utilization of parallelism and a data-oriented design approach. These changes significantly improved runtime by factor six to eight on the testing system. Further, it investigated the effects on runtime by enabling the option enforced mass conservation, which resulted in a doubled runtime on average.

9.2. Outlook

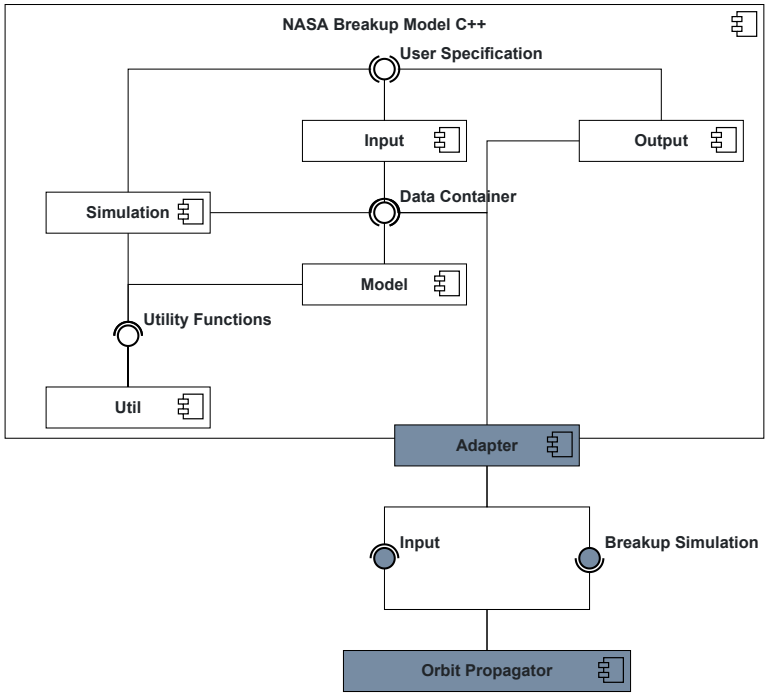


Figure 9.1.: Potential linkage of an orbit propagator to the C++ Breakup Model Implementation via an adapter

Overall, the implementation results are satisfying. Nevertheless, improvements are always possible as there is no final state, and the requirements constantly change. One primary goal was the creation of an open-source, well-documented C++ implementation of the NASA Breakup Model. As we have achieved this goal, it is possible to use this existing work as the foundation for future experiments around breakup events due to the variety of additionally provided universal tools like TLE input or Keplerian output.

As was presented before, some considerations are about improving the physical correctness of the Breakup Model. Since the here presented strategy of assigning parent bodies and ensuring the conservation of mass is rather simple, some improvements in future work would presumably advance the precise fulfillment of significant physical properties like energy conservation. The mentioned is neither fulfilled by this implementation nor the NASA Breakup Model, regardless of its justified presence. In fact, there are fascinating strategies out there to patch those properties into the model, with only a tiny portion covered in Chapter 3.

Another area to improve is the implementation itself. We currently solely depend on the C++ standard library in functional terms. Thus, third-party dependencies are just utilized for input, testing, and output. In contrast, the simulation and the model are the pure standards of C++ 17. This decision brings the advantage of reliability and correctness for the fundamentals of the implementation and enables parallelization and random number generation. However, third-party libraries could provide more performance in time-intensive areas like random number generation through, for instance, better implemented vectorization. Therefore, they are suitable for future improvements to this work, but further guarantees about their correctness have to be investigated.

As the Breakup Model's different fragmentation events are implemented via inheritance, it would be a valuable future addition to integrate other breakup models into the implementation to form a uniform breakup interface. Undoubtedly, these additions could include breakup models for other scenarios like reentry or breakup models with higher accuracy. Alternatively, it is also thinkable to integrate the architecture into a long-term model via an adapter, as illustrated in Figure 9.1. In conclusion, further reuse of the components is pretty straightforward, and since it is open-source, there are no restrictions on possible deployment scenarios nor thoughts.

Part IV.

Appendix

List of Figures

List of Tables

Bibliography

[ACH+17] Erick Ausay, A Cornejo, A Horn, K Palma, T Sato, B Blake, Frank Pistella, C Boyle, Naromi Todd, Jeffrey Zimmerman, et al. A comparison of the socit and debrisat experiments. In *European Conference on Space Debris*, 2017.

[aE] Space Debris Office at ESOC/ESA. Space environment statistics. `https://sdup.esoc.esa.int/discosweb/statistics/`. accessed: 03.09.2021.

[AIG+17] Roxana Larisa Andrișan, Alina Georgia Ioniță, Raúl Domínguez González, Noelia Sánchez Ortiz, Fernando Pina Caballero, and Holger Krag. Fragmentation event model and assessment tool (fremat) supporting on-orbit fragmentation analysis. In *7th European Conference on Space Debris*, 2017. `https://conference.sdo.esoc.esa.int/proceedings/sdc7/paper/678/SDC7-paper678.pdf` accessed: 02.09.2021.

[AP09] Luciano Anselmo and Carmen Pardini. Analysis of the consequences in low earth orbit of the collision between cosmos 2251 and iridium 33. In *Proceedings of the 21st International Symposium on Space Flight Dynamics*, pages 1–15. Centre nationale d'etudes spatiales Paris, France, 2009.

[Bar96] Simon Barrows. *Evolution of artificial space debris clouds*. PhD thesis, University of Southampton, 1996. `https://eprints.soton.ac.uk/192401/1/96069550.pdf` accessed: 03.09.2021.

[BJR+98] Anette Bade, Albert A Jackson, Robert C Reynolds, Peter Eichler, Paula Krisko, Mark Matney, Phillip D Anz-Meador, and Nicholas L Johnson. Breakup model update at nasa/jsc. In *49th International Astronautical Congress*, 1998.

[BLR+17] Vitali Braun, S Lemmens, B Reihs, H Krag, and A Horstmann. Analysis of breakup events. In *Proceedings of the 7th European Conference on Space Debris*, volume 7, 2017. `https://conference.sdo.esoc.esa.int/proceedings/sdc7/paper/1005/SDC7-paper1005.pdf` accessed: 03.09.2021.

[Bra19] Vitali Braun. Impact of debris model updates on risk assessments. In *Proceedings of the 1st NEO and Space Debris Detection Conference*, 2019. `https://conference.sdo.esoc.esa.int/proceedings/neosst1/paper/500/NEOSST1-paper500.pdf` accessed: 03.09.2021.

[CIO+21] Nicola Cimmino, Giorgio Isoletta, Roberto Opromolla, Giancarmine Fasano, Aniello Basile, Antonio Romano, Moreno Peroni, Alessandro Panico, and Andrea Cecchini. Tuning of nasa standard breakup model for fragmentation events modelling. *Aerospace*, 8(7):185, 2021.

[FBT+21] E. Farahvashi, K. D. Bunte, M. Traud, M. Millinger, Y. Li, and R. Srama. Scattering model for grazing impact of micro-particles on mirror surfaces. In *8th European Conference on Space Debris*, volume 8. ESA Space Debris Office, 2021. `https://conference.sdo.esoc.esa.int/proceedings/sdc8/paper/174/SDC8-paper174.pdf` accessed: 02.09.2021.

[JKLAM01] Nicholas L Johnson, Paula H Krisko, J-C Liou, and Phillip D Anz-Meador. Nasa's new breakup model of evolve 4.0. *Advances in Space Research*, 28(9):1377–1384, 2001.

[Joh10] Nicholas L Johnson. Orbital debris: the growing threat to space operations. In *33rd Annual Guidance and Control Conference*, 2010.

[JT21] Wynand Joubert and Steven Tingay. Simulations of orbital debris clouds due to breakup events and their characterisation using the murchison widefield array radio telescope. *Experimental Astronomy*, 51(1):61–75, 2021.

[KBS+16] Richard Klima, Daan Bloembergen, Rahul Savani, Karl Tuyls, Daniel Hennes, Dario Izzo, Karl Tuyls, and Leopold Summerer. Game theoretic analysis of the space debris dilemma. *Tech. rep., Final Report, ESA Ariadna Study 15/*, 8401, 2016.

[KJLM10] Donald J Kessler, Nicholas L Johnson, JC Liou, and Mark Matney. The kessler syndrome: implications to future space operations. *Advances in the Astronautical Sciences*, 137(8):2010, 2010.

[Kli06] Heiner Klinkrad. *Space debris: models and risk analysis*. Springer Science & Business Media, 2006.

[KOK01] Paula H Krisko, John N Opiela, and Donald J Kessler. The critical density theory in leo as analyzed by evolve 4.0. In *Space Debris*, volume 473, pages 273–278, 2001.

[Kra02] Herbert J Kramer. Debrisat - a destructive laboratory test of a satellite. In *Observation of the Earth and its Environment: Survey of Missions and Sensors*. Springer Science & Business Media, 2002. `https://directory.eoportal.org/web/eoportal/satellite-missions/d/debrisat` accessed: 01.09.2021.

[Kru20] Stephan Krusche. Lecture material: Patterns in software engineering, 2020.

[LHKO04] J-C Liou, DT Hall, PH Krisko, and JN Opiela. Legend–a three-dimensional leo-to-geo debris evolutionary model. *Advances in Space Research*, 34(5):981–986, 2004.

[Lio11] J-C Liou. An active debris removal parametric study for leo environment remediation. *Advances in space research*, 47(11):1865–1876, 2011.

[Lio12] JC Liou. Orbital debris modeling. *Presentation to Canadian Space Agency, Quebec, Canada*, 2012.

[LJ06] J.-C. Liou and N. L. Johnson. Risks in space from orbiting debris. *Science*, 311(5759):340–341, 2006.

[LJKAM02] J-C Liou, NL Johnson, PH Krisko, and PD Anz-Meador. The new nasa orbital debris breakup model. In *COSPAR Colloquia Series*, volume 15, pages 363–367. Elsevier, 2002.

[LKMS03] J-C Liou, DJ Kessler, M Matney, and G Stansbery. A new approach to evaluate collision probabilities among asteroids, comets, and kuiper belt objects. In *Lunar and Planetary Science Conference*, page 1828, 2003.

[LS09] JC Liou and D Shoots. Orbital debris quarterly news. *NASA Orbital Debris Program Office*, 2009.

[LSWG01] Hugh G Lewis, Graham Swinerd, Neil Williams, and Gavin Gittins. Damage: a dedicated geo debris model framework. *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP*, 473:373–378, 2001. `https://conference.sdo.esoc.esa.int/proceedings/sdc3/paper/80/SDC3-paper80.pdf` accessed: 03.09.2021.

[MPFC+15] Mathew Moraguez, Kunal Patankar, Norman Fitz-Coy, J-C Liou, Marlon Sorge, Heather Cowardin, John Opiela, and Paula H Krisko. An imaging system for automated characteristic length measurement of debrisat fragments. In *International Astronautical Congress Meetong*, 2015.

[PA09] Carmen Pardini and Luciano Anselmo. Assessment of the consequences of the fengyun-1c breakup in low earth orbit. *Advances in Space Research*, 44(5):545–557, 2009.

[PA11] Carmen Pardini and Luciano Anselmo. Physical properties and long-term evolution of the debris clouds produced by two catastrophic collisions in earth orbit. *Advances in Space Research*, 48(3):557–569, 2011.

[PCDL18] Alexis Petit, Daniel Casanova, Morgane Dumont, and Anne Lemaître. Creation of a synthetic population of space debris to reduce discrepancies between simulation and observations. *Celestial Mechanics and Dynamical Astronomy*, 130(12):1–19, 2018.

[RMSS17] Jonas Radtke, Sven Mueller, Volker Schaus, and Enrico Stoll. Luca2-an enhanced long-term utility for collision analysis. In *Proceedings of the 7th European Conference on Space Debris*, 2017.

[Sch20] Hilmar Schmundt. Putzkommando im all. *Der SPIEGEL*, 52:112–114, December 2020.

[SM21] Nicholas Sia and Piyush M Mehta. Next-generation re-entry aerothermodynamic modeling for space debris using deep learning. In *8th European Conference on Space Debris*, volume 8. ESA Space Debris Office, 2021. `https://conference.sdo.esoc.esa.int/proceedings/sdc8/paper/26/SDC8-paper26.pdf` accessed: 02.09.2021.

[st] space track.org. Basic description of the two line element (tle) format. `https://www.space-track.org/documentation#/tle-alpha5`. accessed: 26.08.2021.

[VO17] David A Vallado and Daniel L Oltrogge. Fragmentation event debris field evolution using 3d volumetric risk assessment. *7th ECSD, Darmstadt*, 2017.

[Wal18] Ulrich Walter. *Astronautics - The Physics of Space Flight*. Springer, third edition, 2018.

[wik] wikipedia.org. Bstar. `https://en.wikipedia.org/wiki/BSTAR`. accessed: 08.09.2021.