Continuous and Discrete Algorithms for Modelling the Kessler Syndrome

Philip Soliman - 4945255

TU Delft
Faculty of Electrical Engineering, Mathematics and Computer Science
BSc program Applied Physics and Mathematics

Delft, February 15, 2022

Supervisors:
dr. P.M. Visser
prof. dr. J. Thijssen

Contents

Abstract

This thesis contains the development of *continuous* Kepler orbit- and a *discrete* numerical integration-based collision detection algorithms in a system of LEO satellites, which in combination with collision algorithm form a simplified space debris evolution model. This model is then used to study the Kessler syndrome.

The *continuous* and *discrete* algorithms get their names from the solutions of the Two Body Problem (TBP) and the methods for collision detection that they are based on; the analytical and continuous time solution of TBP resulting in the Kepler orbits and the numerical, discrete time Velocity Verlet integration of the TBP. The collision model consists of an algorithm for fragmentation collisions largely based on the NASA Standard Breakup Model and a method for elastic, random scattering collisions.

Comparison between the continuous and discrete algorithms shows that on average both predict the same time to the first collision in a system of homogeneously distributed satellites. The algorithms differ in their efficiency depending on the number and the radius of the satellites in and the geometry of the system. For relatively small satellite numbers in large systems, the continuous algorithm is computationally more efficient. However, as more satellites or fragments result from previous collision, the continuous algorithm is outperformed by the discrete algorithm. Consequentially, its time complexity appears to be $O(N^2)$.

Armed with this knowledge, the continuous algorithm is used to show that an initially small system of satellites is able to evolve into a large population of debris particles within several decades. Similarly, the discrete algorithm is used to show that an ordered collection of satellites in an homogeneously distributed system of debris-like particles exhibits the effect that a collision early on in the simulation can cause a cascade of collisions at a later stage. Hence Both the discrete and continuous algorithms predict a Kessler Syndrome and mimic predictions made by more advanced models from leading space agencies like NASA's LEGEND, ESA's DELTA and JAXA's LEODEEM [Lio+13].

Future research could focus on including atmospheric drag and gravitational perturbations to the continuous algorithm, thereby lengthening the time frame during which it can realistically simulate a system of satellites in LEO. To achieve this it is suggested that one execute the calculations inherent to the algorithm in parallel on a GPU, as these are independent of each other.

1. Introduction

> "*We have a full-on chain reaction it has been confirmed that the [the debris] is an unintentional side-effect of the Russians striking one of their own satellites*", excerpt from the movie *Gravity* [15].

In the movie *Gravity* the explosion of a satellite in Low Earth Orbit (LEO) generates a debris field that causes subsequent collisions with satellites and a "*full-on chain reaction*" of more debris and collisions. This debris is travelling at an altitude similar to that of Hubble, where a group of astronauts are performing a servicing mission. On the 15^{th} of November of 2021, Russians actually blew up one of their satellites in an anti-satellite missile test, forcing astronauts in the ISS to shelter in the Crew Dragon spacecraft [RA21].

Explosions of this scale act as seeder events, generating numerous small fragment and a few large ones. The generation of large numbers of debris particles is not limited to anti-satellite tests, as old rocket bodies have the potential to explode at any time and are essentially ticking time bombs. Nor is it merely limited to explosions. Given a large enough population of debris particles in LEO a cascade of subsequent collisions could follow, generating increasingly more fragments. This idea was originally conceived of by Donald J. Kessler in 1978 and has since become known as the 'Kessler Syndrome' [KC78]. The ultimate conclusion of a Kessler Syndrome is that certain space activities and the operation of satellites providing essential services like communication and location become complicated for several decades.

The cascading effect that one seeder effect has may be seen in Figure 1, in which the collision of the active commercial satellite Iridium 33 and the defunct Russian military satellite Kosmos 2251 resulted in over a thousand debris fragments larger than 10 cm [Nic09]. Afterwards, the number of payload fragmentation debris can be seen to starkly increase for multiple years after.

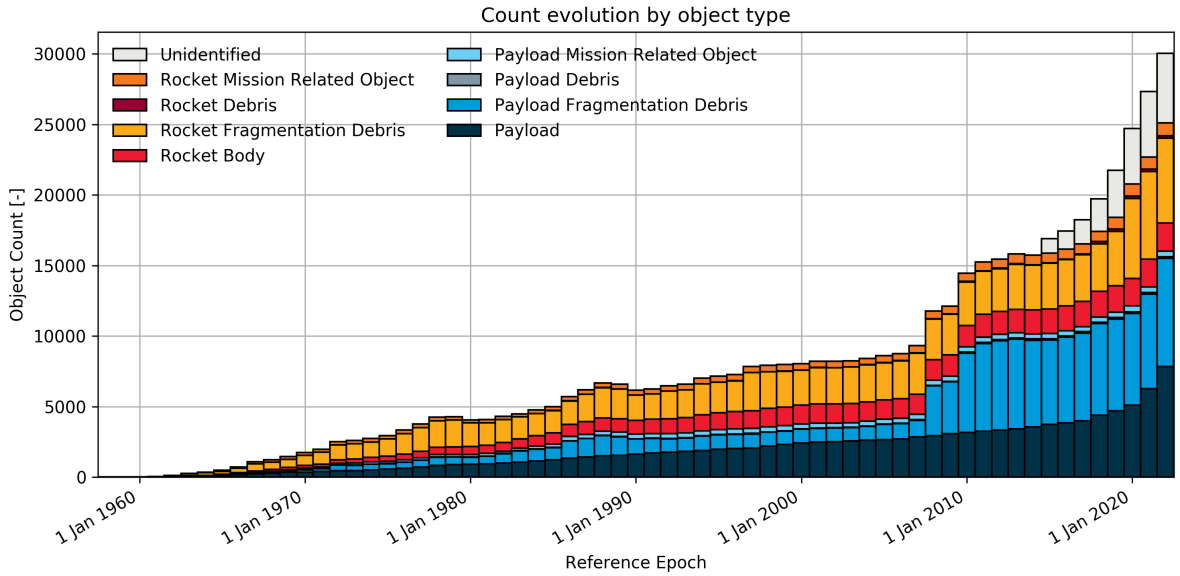


Figure 1: Bar chart showing the evolution of the number and types of objects in orbit around Earth. [ESO]

In modelling the Kessler syndrome several choices need to be made. One could opt for a statistical approach, which assumes a particular distribution of satellites in Earth's orbit and tries to compute the average time to the first or next collision. Kessler himself did exactly this in his original paper [KC78]. He observed that orbital perturbations, like the slightly oblate shape of the Earth, the atmospheric drag induced on satellites in Low Earth Orbit (LEO) or solar radiation pressure, caused the two orbital parameters to change considerably over short spans of time. These orbital elements, argument of pericentre ω and longitude of ascending node Ω, are constant in the idealised scenario of spherical gravitational potential. However, Kessler assumed them to be randomly distributed as the result of the perturbations mentioned. This allowed him to estimate the density of satellites at discrete volumes depending on the altitude, as measured from Earth's surface, ΔR and geocentric latitude β. This density, together with an estimation for the average impact velocity and cross sectional area of satellites in a certain altitude band then gives an average impact rate

A slightly more general approach involves the construction of a system dynamics model. This kind of model uses non-linear, coupled differential equations to determine the amount of debris and number of collisions. The strength of this approach lies in that it can easily be expanded to include complex effects like solar radiation pressure, atmospheric drag and de-orbiting of satellites, as it does not require the exact modelling of the orbit, mass, size or collision of any satellites or debris. In [DH18] this method is used to study different scenarios of orbital debris evolution, which include among others a 'conflict with a large-scale deployment of anti-satellite weapons', the instantaneous loss of control of satellites due to an Electromagnetic Pulse (EMP) and the 'cessation of LEO satellite launches'.

Another option is that of an entirely deterministic model. This approach aims to (approximately) model every satellite and collision event and is thereby able to give the state of the entire system at certain times. To achieve this, one usually makes several assumptions about the system to simplify calculations. One may for example assume the Earth to act as the sole source of gravitational attraction, hence leaving out the effect the satellites have on each other or the moon on the satellites. Another possibility is to assume that the Earth is be perfectly spherical, neglecting atmospheric drag and solar radiation pressure. Debris evolution models are especially sensitive to which of these assumptions and other assumptions are made. As concluded in [DRD15] for instance, the largest source of uncertainty in predicting the long term evolution of LEO space debris, aside from solar radiation pressure and the parameters of the breakup model, is 'rate of compliance with post-mission disposal'.

No matter from what context the equations in the model arise, be it statistical or deterministic, there exist two major ways of solving them; numerically and analytically. The former method approximates the equations in a model and solves them at discrete times with arbitrary accuracy. Numerical methods are able to solve very complex systems of equations at the cost long computation times. The latter, on the other hand, quickly gives the exact solution of the system at any given time. Which of these methods is used in a model ultimately determines what it is able to simulate. Numerical models can easily be adjusted to include orbital perturbations, but can only do so with a certain accuracy and at specific times. Analytical methods offer an exact solution at all times, though it needs to be completely re-derived when a new effect is added to the model.

Lastly, a mix of all these kinds of models is also an option. One could model the orbits *exactly* and use the orbital parameters to determine a collision *probability* for each satellite. A setup for a model like this using Kepler orbits is given in [JM17]. This paper uses Kepler orbits

as well, not to find the probability of a collisions, but to actually find the time and place of one.

The goal of this paper is to create a simplified space debris evolution model for a system of colliding satellites in Kepler orbits in order to study the Kessler Syndrome. To do so, it discusses the motivation for (Section 2), realisation of (Sections 3.1 and 3.2) and comparison between (Section 3.3) two deterministic approaches to collision detection between satellites. Section 4 describes the model that is used to perform collisions in the simulation, which incorporates scattering collisions (Section 4.1) and an implementation of the (statistical) NASA Standard Breakup Model for fragmentation collisions (Section 4.2) [Joh+01]. Section 5 then combines the two collision detection algorithms with the collision model to study the evolution of various systems of satellites.

2. The two-body problem

The dynamics of Earth's satellites are described by the two-body problem (TBP), if the mutual gravity of the satellites, the gravitational influence of other bodies (planets, the Moon, the Sun, close encounters with asteroids, etc.) and the effect of other perturbing forces (drag induced by Earth's atmosphere, solar radiation pressure, Earth's non-spherical shape) are neglected. In addition, the centre of mass (COM) frame may be assumed to coincide with the (reference) frame of the Earth, if the mass of the Earth is considerably larger than the total mass of the satellites. Under these assumptions each satellite's position must satisfy the equation of relative motion

$$\frac{d^2\vec{r}}{dt^2} + \mu\frac{\vec{r}}{r^3} = 0, \tag{2.1}$$

where $\vec{r}$ is the position vector of the satellite in Earth's reference frame with norm r and $\mu = \mathcal{G}(m_1 + m_2)$. In this report a solution to equation (2.1) is obtained in two ways. These will be discussed below.

2.1. Kepler orbits: an analytical solution to the two-body problem

The first of the two solutions of the TBP problem posed in section 2 starts by reducing the system of coupled, second order (homogeneous) differential equations defined by equation (2.1) to one linear second order inhomogeneous differential equation.[1] It does so through a coordinate transformation; from the Cartesian coordinate system — in which the problem is originally defined and the reference frame of the Earth — to a polar one (r, θ) in the orbital plane. By substituting $u = 1/r$ and using that the angular momentum (per unit mass) of the satellite $h = r^2\frac{d\theta}{dt}$ is constant under the influence of a central force, i.e. Earth's gravity, gives

$$\frac{d^2u}{d\theta^2} + u = \frac{\mu}{h^2}, \tag{2.2}$$

which may be solved to get the general equation of an ellipse

$$r(\theta - \varpi) = r(f) = \frac{h^2/\mu}{1 + e\cos f} = \frac{a(1-e^2)}{1 + e\cos f}. \tag{2.3}$$

Here, θ is the orbital angle of the body with respect to some reference direction, a is the semi-major axis, e is the eccentricity and f is the true anomaly. The latter is defined as the angle with respect to the longitude of periapsis ϖ, which itself is the angle that minimizes r

$$r(\varpi) = a(1-e) = r_p. \tag{2.4}$$

Similarly, r attains its maximum at the apoapse

$$r(\varpi + \pi) = a(1+e) = r_a. \tag{2.5}$$

Furthermore, let T be the period of a satellite's orbit, then its average orbital frequency or mean motion is given by $n = \frac{2\pi}{T}$ and is related to μ and a through Kepler's third law

$$n^2 = \frac{\mu}{a^3}. \tag{2.6}$$

[1] All the theory discussed is obtained from [MD00, pages 22-62]

Note that using equation (2.6) we can rewrite

$$h = \sqrt{\mu a(1-e^2)} = na^2\sqrt{1-e^2} = nab, \tag{2.7}$$

where $b = a\sqrt{1-e^2}$ is the semi-minor axis.

Time does not appear in equation (2.3), as all dependency on t is eliminated in the preceding derivation. In order to locate a satellite on its elliptical orbit at a particular time we need to know the eccentric anomaly E [2]

$$M_a = nt = E - e\sin E, \tag{2.8}$$

where M_a is the mean anomaly. Seeing as equation (2.8) is transcendental in E, there exists no closed form for E in terms of M_a. However, using fixed point iteration on the function $g(E) = M_a + e\sin E$ and trigonometric, angle sum identities gives, after three iterations, the following expression

$$E = M_a + (e - \frac{1}{8}e^3)\sin M_a + \frac{1}{2}e^2\sin 2M_a + \frac{3}{8}e^3\sin 3M_a, \tag{2.9}$$

which is valid for $e < 0.6627434$. Using the Newton-Raphson method to find the root of the function $h(E) = E - e\sin(E) - M_a$ offers a solution as well, even if $e \geqslant 0.6627434$. The Cartesian coordinates in the reference frame of the Earth are obtained using

$$\vec{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \mathcal{R}(\Omega, \omega, I) \begin{pmatrix} a(\cos E - e) \\ a\sqrt{1-e^2}\sin E \\ 0 \end{pmatrix}, \tag{2.10}$$

where $\mathcal{R}$ is a transformation matrix defined in equation (A1) and depends on the longitude of ascending node Ω, the argument of periapsis ω and the inclination I. Similarly the velocity is

$$\vec{v} = \dot{\vec{r}} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \frac{na}{|\vec{r}|}\mathcal{R} \begin{pmatrix} -a\sin E \\ a\sqrt{1-e^2}\cos E \\ 0 \end{pmatrix}, \tag{2.11}$$

A property of a Kepler orbit is that it is completely determined by its angular momentum $\vec{L}$ and eccentricity vector $\vec{e}$ (REF). The former is purely perpendicular to its orbital plane and given by

$$\vec{L} = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix} = \mathcal{R} \begin{pmatrix} 0 \\ 0 \\ mh \end{pmatrix} = L \begin{pmatrix} \sin\Omega\sin I \\ -\cos\Omega\sin I \\ \cos I \end{pmatrix}, \tag{2.12}$$

where m is the mass of a satellite and $L = mh = mnab$. The latter points from the center of the ellipse to the central body and, therefore,

$$\vec{e} = \begin{pmatrix} e_1 \\ e_2 \\ e_3 \end{pmatrix} = \mathcal{R} \begin{pmatrix} e \\ 0 \\ 0 \end{pmatrix} = e \begin{pmatrix} \cos\Omega\cos\omega - \sin\Omega\sin\omega\cos I \\ \sin\Omega\cos\omega + \cos\Omega\sin\omega\cos I \\ \sin\omega\sin I \end{pmatrix}. \tag{2.13}$$

An orbit may be specified by first determining $L = |\vec{L}|$ and $e = |\vec{e}|$, which give the value of a through equations (2.7) and (2.12). Then, l_3 gives I, e_3 fixes ω, l_1 yields Ω. $\vec{L}$ and $\vec{e}$ are themselves given in terms of $\vec{r}$ and $\vec{v}$ through equations A2 and A3.

[2] for a geometric interpretation of E see

2.2. Numerical integration of the two-body problem

From the equation of relative motion (2.1) we get that at any point in time the acceleration of a satellite due to Earth's gravity is completely determined by its position, $\frac{d^2\vec{r}}{dt^2} = \vec{a}(\vec{r}) = -\mu\frac{\vec{r}}{r^3}$. This makes the TBP well suited for the use of a numerical integration scheme. At this point, there are numerous options with regards to which particular scheme can be used. Considering that the aim of this report is to simulate the system of satellites for multiple periods and include collisions and/or breakups in doing so, its integration scheme should meet certain requirements. With regards to the former, it should at least i) preserve (angular) momentum, ii) be time-invariant/-symmetric and iii) have a bounded energy and momentum error. Requirement i) ensures that satellites in orbit around Earth remain in orbit, given they do not encounter any other satellites and collide with them. In other words, satellites do not, by themselves and after several orbits, escape from Earth's gravitational influence or crash into Earth itself. For similar reasons ii) is important; it should not matter for the position (velocity) of a satellite whether it is propagated for- or backward in time, as the laws of physics are time-invariant. Also related is iii), which allows the system to be simulated for multiple periods (long spans of time) without an increasing deviation from the exact solution[3]. Aside from the mentioned requirements, the scheme should provide the position and velocity of a satellite at the same time instant, in order to conveniently simulate collisions. Though this is also necessary for the calculation of (approximately) conserved quantities, like total energy and (angular) momentum.

All of the above are satisfied by the Velocity Verlet (VV) integration scheme, which is one of the simplest schemes that does so [HLW03]. At any time t one performs the following two operations to propagate the position and velocity a time-step Δt forward

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{\vec{a}(\vec{r}(t))}{2}\Delta t^2,$$
$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{a}(\vec{r}(t)) + \vec{a}(\vec{r}(t+\Delta t))}{2}\Delta t.$$

VV is a second order integrator, meaning that its global error is proportional to Δt^2. If we perform VV for all particles in the system we get Algorithm 1.

Algorithm 1 Velocity Verlet

procedure VELOCITYVERLET($R_t, V_t, \Delta t$)
  $R_{t+\Delta t} = \varnothing$
  $V_{t+\Delta t} = \varnothing$
  for $\vec{r}_i$ **in** $\vec{r}$ **and** $\vec{v}_i$ **in** $\vec{v}$ **do**
    $\vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{\vec{a}(\vec{r}_i(t))}{2}\Delta t^2$
    $\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{\vec{a}(\vec{r}_i(t)) + \vec{a}(\vec{r}_i(t+\Delta t))}{2}\Delta t$
    append $\vec{r}_i(t+\Delta t)$ to $R_{t+\Delta t}$ and $\vec{v}_i(t+\Delta t)$ to $V_{t+\Delta t}$
  end for
  return $R_{t+\Delta t}$ and $V_{t+\Delta t}$
end procedure

[3] like the Kepler orbits described in 2.1

3. Collision detection

No matter what type of modelling approach is used in order to determine the evolution of debris in Earth's orbit, the effect of collisions on the system must be included. They are the main source of debris, generating several hundreds or thousands of fragments per event and injecting those fragments into a range of lower and higher orbits.

The problem of modelling a collisional system of satellites is fourfold; we need to determine i) which satellites collide, ii) where and iii) when they do so and, lastly, iv) we need to simulate the collisions themselves. Problems i-iii) are the topic of this section, while iv) is addressed in Section 4.

How collisions are detected depends on the approach of the TBP; in Section 3.1 Kepler orbits are used while in Section 3.2 another method is described in the context of numerical integration of the TBP.

3.1. Collision detection using Kepler orbits

Given a system of satellites in Kepler orbits we can determine what satellites will collide. We denote the satellites as i and j, the collision time as $t_{i,j}^{\text{col}}$, their position and velocity as $\vec{r}_i$, $\vec{r}_j$, $\vec{v}_i$ and $\vec{v}_j$. The method described in this section was provided by dr. Visser in its entirety.

3.1.1. Colliding pairs

From Equations (2.4) and (2.5) we get that r_p and r_a define the lower and upper bound for the radial coordinate of any satellite. As an initial crude search for possible colliding pairs of satellites, we can therefore check whether the apoapse of i, $r_{a,i}$, exceeds the periapse of j, $r_{p,j}$. In doing so, we have to take the radii of satelllites s_i and s_j into account. Assuming without loss of generality $a, i < a, j$, this amounts to checking [Opi51]

$$r_{a,i} + s_i \geqslant r_{p,j} - s_j,$$

$$a_i(1 + e_i) + s_i \geqslant a_j(1 + e_j) - s_j. \tag{3.1}$$

If the above condition is satisfied for some pair (i, j), then these satellites could possibly collide. In any case, the orbits of these satellites will share some minimum distance, which is generally denoted as the Minimum Orbit Intersection Distance (MOID). For two random orbits and given $s_{i,j}$ is sufficiently small compared to $a_{i,j}$, the MOID will lie near the intersection of the two orbital planes, the nodal line. The direction of the nodal line is given as [MMB98]

$$\vec{K}_\pm = K\hat{K}_\pm = \pm\vec{L}_i \times \vec{L}_j,$$

where $\vec{L}_{i,j}$ is the angular momentum of the satellites from Equation (2.12) and the plus and minus signs refer to the two intersection points (FIG). The angle between $\hat{K}_\pm$ and $\vec{e}_{i,j}$ is the true anomaly of the intersection point, because the former necessarily lies in the orbital plane and the latter always points toward periapsis[4]. Taking the inner product with $\vec{e}_i$ therefore gives

$$\vec{e}_i \cdot \vec{K}_\pm = Ke\cos f_\pm,$$

[4]It must therefore lie along the argument of periapsis ϖ. Thus if we denote the angle between $\vec{e}_i$ and $\vec{K}_\pm$ as ν, then $\nu = \theta - \varpi = f$, where θ is the angle between $\vec{K}$ and some reference direction.

where $f_\pm$ is the true anomaly of the intersection points and a similar relationship holds for $\vec{e}_j$. Substituting this into Equation (2.3) yields

$$\vec{r}_{i,\pm} = \frac{a_i(1-e_i^2)}{1+e\cos f_\pm}\hat{K}_\pm = \frac{a_i(1-e_i^2)}{K+\vec{e}_i\cdot\vec{K}_\pm}\vec{K}_\pm = \frac{a_i(1-e_i^2)\vec{L}_i\times\vec{L}_j}{\pm|\vec{L}_i\times\vec{L}_j| + \vec{e}_i\cdot(\vec{L}_i\times\vec{L}_j)}. \tag{3.2}$$

There is an equivalent relation for j. In order to obtain an approximate MOID we linearise the motion of i and j around $\vec{r}_{i,\pm}$ and $\vec{r}_{j,\pm}$ as

$$\vec{\rho}_i(t) = \vec{r}_i + \vec{v}_i t \quad \text{and} \quad \vec{\rho}_j(s) = \vec{r}_j + \vec{v}_j s,$$

where the velocities $\vec{v}_{i,j}$ are obtained from Equation (A4) and s and t are independent parameters. All subscripts referring to the two intersection points are left out, since the following analysis is the same for both. Taking the squared norm of the relative distance $\delta^2(t,s) = |\rho_j(s) - \rho_i(t)|^2$ and differentiating to t and s gives

$$\begin{aligned}\frac{d\delta^2}{dt} &= \vec{v}_i\cdot\vec{v}_i t - \vec{v}_i\cdot\vec{v}_j s - (\vec{r}_j - \vec{r}_i)\cdot\vec{v}_i,\\ \frac{d\delta^2}{ds} &= -\vec{v}_i\cdot\vec{v}_j t + \vec{v}_j\cdot\vec{v}_j s + (\vec{r}_j - \vec{r}_i)\cdot\vec{v}_j.\end{aligned}$$

Equating these to zero for $t = t_{MOID}$ and $s = s_{MOID}$ results in the following system of equations

$$\begin{pmatrix} |\vec{v}_i|^2 & -\vec{v}_i\cdot\vec{v}_j \\ -\vec{v}_i\cdot\vec{v}_j & |\vec{v}_j|^2 \end{pmatrix}\begin{pmatrix} t_{MOID} \\ s_{MOID}\end{pmatrix} = \begin{pmatrix} (\vec{r}_j-\vec{r}_i)\cdot\vec{v}_i \\ -(\vec{r}_j-\vec{r}_i)\cdot\vec{v}_j\end{pmatrix},$$

which may be solved to yield[5]

$$\begin{aligned} t_{MOID} &= (\vec{r}_j-\vec{r}_i)\cdot\frac{\vec{v}_j\times(\vec{v}_i\times\vec{v}_j)}{|\vec{v}_i\times\vec{v}_j|^2},\\ s_{MOID} &= (\vec{r}_j-\vec{r}_i)\cdot\frac{\vec{v}_i\times(\vec{v}_i\times\vec{v}_j)}{|\vec{v}_i\times\vec{v}_j|^2}.\end{aligned}$$

This results in the following points at which the two satellite will have an approximate MOID

$$\vec{r}_i^{\,1} = \vec{\rho}_i(t_{MOID}) = \vec{r}_i + \left[(\vec{r}_j-\vec{r}_i)\cdot\frac{\vec{v}_j\times(\vec{v}_i\times\vec{v}_j)}{|\vec{v}_i\times\vec{v}_j|^2}\right]\vec{v}_i, \tag{3.3}$$

$$\vec{r}_j^{\,1} = \vec{\rho}_j(s_{MOID}) = \vec{r}_j + \left[(\vec{r}_j-\vec{r}_i)\cdot\frac{\vec{v}_i\times(\vec{v}_i\times\vec{v}_j)}{|\vec{v}_i\times\vec{v}_j|^2}\right]\vec{v}_j. \tag{3.4}$$

To determine whether (i,j) indeed form a colliding pair, their MOID should be smaller than the sum of their radii

$$\delta_{MOID} = |\vec{\rho}_j(s_{MOID}) - \vec{\rho}_i(t_{MOID})| < s_i + s_j. \tag{3.5}$$

[5]Here it has been used that for vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ we always have $|\vec{a}|^2|\vec{b}|^2 - (\vec{a}\cdot\vec{b})^2 = |\vec{a}\times\vec{b}|^2$ and $(\vec{a}\cdot\vec{c})\vec{b} - (\vec{a}\cdot\vec{b})\vec{c} = \vec{a}\times(\vec{b}\times\vec{c})$ (vector triple product).

3.1.2. First MOID passage time

Let $t^1_{i,j}$ be the time that satellites i, j first pass the MOID points defined in Equations (3.3) and (3.4). As is illustrated in Section 3.1.3, these times are needed to determine the collision time t^{col}_{ij}. More specifically, $t^1_{i,j}$ denote the time it takes satellites i, j to move from their creation points $\vec{r}^{\,0}_{i,j}$ to their collision points $\vec{r}^{\,1}_{i,j}$. In order to obtain these passage times, we use Kepler's second law, which says that a body's position vector sweeps out equal area in equal times. Put differently, the difference in time between two points on an orbit is proportional to the period T of the orbit

$$t^1 - t^0 = \frac{A}{\pi ab} T = \frac{2A}{nab},$$

where t^0 is the creation time[6] and $b = a\sqrt{1-e^2}$. The area A may be determined from figure (FIG) as

$$A = \frac{(E^1 - E^0)ab}{2} + \frac{a\vec{e} \times \vec{r}^{\,0}}{2} \cdot \hat{L} - \frac{a\vec{e} \times \vec{r}^{\,1}}{2} \cdot \hat{L}.$$

substituting this for A gives

$$t^1 - t^0 = \frac{\Delta E}{n} - \frac{\vec{e} \times (\vec{r}^{\,1} - \vec{r}^{\,0})}{nb} \cdot \hat{L}, \tag{3.6}$$

with $\Delta E = E^1 - E^0$, the difference in eccentric anomalies. As derived by Dr. Visser

$$\cos \Delta E = \frac{\vec{r}^{\,1} \cdot \vec{r}^{\,0}}{b^2} + \frac{(\vec{r}^{\,1} + \vec{r}^{\,0}) \cdot \vec{e}}{a} - \frac{(\vec{r}^{\,1} \cdot \vec{e})(\vec{e} \cdot \vec{r}^{\,0})}{b^2}. \tag{3.7}$$

Differentiating (3.7) with respect to t^1 gives

$$\sin \Delta E = -\frac{|\vec{r}^{\,1}|}{na} \left(\frac{\vec{v}^{\,1} \cdot \vec{r}^{\,0}}{b^2} + \frac{\vec{v}^{\,1} \cdot \vec{e}}{a} - \frac{(\vec{v}^{\,1} \cdot \vec{e})(\vec{e} \cdot \vec{r}^{\,0})}{b^2} \right). \tag{3.8}$$

This may be combined with Equation (3.7) to determine ΔE.

3.1.3. Collision time

So far we have described methods to determine colliding pairs and their corresponding positions and velocities at collision. Note that if we define $T_i = \frac{2\pi}{n_i}$ and $T_j = \frac{2\pi}{n_j}$ as the period of the satellites, then this collision time may be decomposed as

$$t^{\text{col}}_{ij} = kT_i + t^1_i + dt_i = lT_j + t^1_j + dt_j \quad \text{with} \quad k, l = 0, 1, 2, ..., \tag{3.9}$$

where $dt_{i,j}$ is small shift in time accounting for the fact that satellites are not required to exactly be at $\vec{r}^{\,1}_{i,j}$ for a collision to occur. We now linearise the motion of the satellites around t^{col}_{ij} as

$$\vec{\rho}_i(t) = \vec{r}^{\,1}_i + \vec{v}_i t \quad \text{and} \quad \vec{\rho}_j(t) = \vec{r}^{\,1}_j + \vec{v}_j t.$$

If we let $\vec{u} = \vec{v}_j - \vec{v}_i$ and $\vec{\delta} = \vec{\rho}_j(dt_i) - \vec{\rho}_i(dt_j)$, then the satellites are at their actual collision points and it must hold

$$\frac{|\vec{u} \times \vec{\delta}|}{|\vec{u}|} < s_i + s_j, \tag{3.10}$$

[6] the subscripts referring to the satellites are neglected here, as the analysis is the same for both

as derived in Section 3.2.3. In appendix B it is derived that Equation (3.10) implies

$$\left|kT_i + t_i^1 - lT_j - t_j^1\right| < \frac{\sqrt{(s_i + s_j)^2 - |\vec{r}_j^1 - \vec{r}_i^1|^2}|\vec{u}|}{|\vec{v}_i \times \vec{v}_j|}. \tag{3.11}$$

The problem of finding the exact collision time is therefore equivalent to finding the smallest integers $k \geqslant 0$ and $l \geqslant 0$ that satisfy inequality (3.11).

To this end, let

$$p = \frac{T_i}{|t_i^1 - t_j^1|}, \quad q = \frac{T_j}{|t_i^1 - t_j^1|}, \quad \delta = \frac{\sqrt{(s_i + s_j)^2 - |\vec{r}_j - \vec{r}_i|^2}|\vec{u}|}{|\vec{v}_i \times \vec{v}_j|}, \tag{3.12}$$

then (3.11) is rewritten to

$$1 - \delta < kp - lq < 1 + \delta. \tag{3.13}$$

Equation (3.13) implies that the points k, l are the smallest integers points that lie between the lines

$$x = \left(\frac{q}{p}\right) y + \frac{1-\delta}{p} \quad \text{and} \quad x = \left(\frac{q}{p}\right) y + \frac{1+\delta}{p} \quad \text{for} \quad x, y \in \mathbb{R}^2 \quad \text{and} \quad y \geqslant 0,$$

We assume without loss of generality that $T_i > T_j$, which implies that $p > q$ and $p > 1$[7]. Now we describe an algorithm that finds integer points k_n and l_n such that their ratio $\frac{l_n}{k_n}$ converges to $\frac{p}{q}$. It determines the successive remainders q_n

$$q_0 = p, \quad q_1 = q, \quad q_{n+2} = q_n \bmod q_{n+1} = q_n - a_n q_{n-1}, \quad \text{for} \quad n = 0, 1, 2, ..., \tag{3.14}$$

where $a_n = \lfloor \frac{q_n}{q_{n+1}} \rfloor$. The corresponding sequence q_n goes to zero for n going to infinity. If $q_n = k_n p - l_n q$, then $\frac{l_n}{k_n}$ are the convergents of the continued fraction expansion (cfe) of $\frac{p}{q}$. In addition a_n are the coefficients of the same cfe and it holds that

$$l_n = a_n l_{n-1} + l_{n-2} \quad \text{and} \quad k_n = a_n k_{n-1} + k_{n-2}. \tag{3.15}$$

The algorithm will look for points that satisfy (3.13) in the basis $\{\vec{b}_n, \vec{b}_{n+1}\}$ defined by

$$\vec{b}_n = (-1)^n \begin{pmatrix} k_n \\ l_n \end{pmatrix}, \quad \vec{b}_{n+2} = \vec{b}_n + a_n \vec{b}_{n+1},$$

where $(-1)^n$ term assures the basis vectors lie in the first quadrant and the recursion relation is motivated by relations (3.15). Any coordinates in the n^{th} basis (ξ, η) are related to the original (x, y) coordinates as

$$\begin{pmatrix} x \\ y \end{pmatrix} = \xi \vec{b}_n + \eta \vec{b}_{n+1} = \begin{pmatrix} \xi k_n - \eta k_{n+1} \\ \xi l_n - \eta l_{n+1} \end{pmatrix}.$$

Hence, the lines defined in (3.14) are described as

$$\text{lower: } \xi = \left(\frac{q_{n+1}}{q_n}\right) \eta + \frac{1+\delta}{q_n}, \quad \text{upper: } \xi = \left(\frac{q_{n+1}}{q_n}\right) \eta + \frac{1-\delta}{q_n}$$

[7] Because $|t_i^1 - t_j^1| < T_j$

in the n^{th} basis. Note that the width of the band between these lines $w_n = \frac{2\delta}{q_n}$ gains a factor $\frac{q_n}{q_{n+1}} > \phi$ for each step of the algorithm, i.e.

$$w_{n+1} = \frac{2\delta}{q_{n+1}} = \frac{q_n}{q_{n+1}}\frac{2\delta}{q_n} = \frac{q_n}{q_{n+1}} w_n \geqslant \phi w_n, \tag{3.16}$$

where ϕ is the golden ratio. The upper bound of the range within which points are searched for in the n^{th} basis is determined by the intersection of $\vec{b}_{n+2}$ with the lower line

$$A : (\xi, \eta) = \left(\frac{1+\delta}{q_{n+2}}, \frac{(1+\delta)(q_n - q_{n+2})}{q_{n+1}q_{n+2}} \right) = \left(\frac{1+\delta}{q_{n+2}}, \frac{(1+\delta)a_n}{q_{n+2}} \right).$$

Thus the points that are checked, are all integer coordinates that lie below the η coordinate of A and on the upper line[8],

$$\eta = 0, 1, 2, 3, ..., \left\lceil \frac{(1+\delta)a_n}{q_{n+2}} \right\rceil \quad \text{and} \quad \xi = \left\lceil \frac{q_{n+1}\eta + 1 - \delta}{q_n} \right\rceil \tag{3.17}$$

If any of these satisfy

$$\begin{aligned} 1 - \delta > \xi q_n - \eta q_{n+1} &= \xi(k_n p - l_n q) - \eta(k_{n+1} p - l_{n+1} q) \\ &= (\xi k_n - \eta k_{n+1})p + (\xi l_n - \eta l_{n+1})q = xp - yq, \end{aligned}$$

then $k = x = \xi k_n - \eta k_{n+1}$ and $l = y = \xi l_n - \eta l_{n+1}$ are a solution and the collisions time is given by Equation (3.9)

$$t_{ij}^{\text{col}} \approx kT_i + t_i^1 \approx lT_j + t_j^1 \quad \text{with} \quad k, l = 0, 1, 2, ..., \tag{3.18}$$

where there is an approximate sign, as the small[9] shift in time $dt_{i,j}$ is neglected. If there are no such points, then the algorithm checks the next basis $\{\vec{b}_{n+2}, \vec{b}_{n+3}\}$. Algorithm 2 contains a full description of all the steps

[8] The upper line is closer to the origin than the lower line.

[9] on the order of the radii of the satellite

Algorithm 2 Collision time of two satellites

procedure TIMECOLLISION($t_i^1, t_j^1, T_i, T_j, \delta$)
  $\Delta t = |t_i^1 - t_j^1|$
  if $\Delta t = 0$ **do**
    return t_i^1 ▷ particles arrive at the collision point at exactly the same time
  end if
  $d = 2 + \delta$
  $q_0 = p$
  $q_1 = q$
  $k_0 = 1$
  $k_1 = 0$
  $n = 0$
  while $d > 1 + \delta$ **do**
    $a_{2n} = \lceil \frac{q_{2n}}{q_{2n+1}} \rceil$
    $q_{2n+2} = q_{2n} - a_{2n} q_{2n+1}$
    if $q_{2n+2} = 0$ **do**
      $\xi = \min\{\xi \mid \xi > \frac{1-\delta}{q_{2n}}, \xi \in \mathbb{N}\}$
      if $\xi q_{sn+1} < 1 + \delta$ **do**
        return ξk_{2n}
      else
        return ∞ ▷ no solution exists
      end else
    end if
    $H = \{\eta \mid 0 \leqslant \eta \leqslant \lceil \frac{(1+\delta)a_{2n}}{q_{2n+2}} \rceil \, \eta \in \mathbb{N}\}$
    for η **in** H **do**
      $\xi = \lceil \frac{q_{2n+1}\eta + 1 + \delta}{q_{2n}} \rceil$
      $d = \xi q_{2n} + \eta q_{2n+1}$
      if $d < 1 + \delta$ **do**
        $k = \xi k_{2n} + \eta k_{2n+1}$
      end if
    end for
    $k_{2n+2} = k_{2n} - a_{2n} k_{2n+1}$
    $a_{2n+1} = \lceil \frac{q_{2n+1}}{q_{2n+2}} \rceil$
    $q_{2n+3} = q_{2n+1} - a_{2n+1} q_{2n+3}$
    if $q_{2n+3} = 0$ **do**
      $\xi = \min\{\xi \mid \xi > \frac{1-\delta}{q_{2n+1}}, \xi \in \mathbb{N}\}$
      if $\xi q_{sn+1} < 1 + \delta$ **do**
        return ξk_{2n+1}
      else
        return ∞ ▷ no solution exists
      end else
    end if
    $k_{2n+3} = k_{2n+1} - a_{2n+1} k_{2n+2}$
    $n = n + 1$
  end while
  return $t_i^1 + kT_i$
end procedure

3.1.4. The continuous algorithm

If we combine the methods from Sections 3.1.1, 3.1.2 and 3.1.3 and apply these to all the entire collection of satellites, then we get Algorithm 3.

Algorithm 3 Collision lists

procedure LISTCOLLISION($R, V, M, S, T_0, t_{\max}$)
  $L_{col} = ()$ ▷ an empty, ordered list or tuple
  $R_{\text{col}} = ()$
  $V_{\text{col}} = ()$
  $T_{\text{col}} = ()$
  retrieve/calculate $\vec{L}$, $\vec{e}$, a, s, n, ω, Ω, I and E for all the satellites
  for $\vec{r}_i$ **in** R **do** ▷ only fragment indices i, if this is not the initial collision list
    $J = (j \mid j$ is not the index of a newly created fragment) ▷ $J = (j \mid \forall j > i)$, in case of the initial collision list
    $\vec{L}_i \in \vec{L}$, $\vec{e}_i \in \vec{e}$, $a_i \in a$, $s_i \in S$, $n_i \in n$, $\omega_i \in \omega$, $\Omega_i \in \Omega$, $I_i \in I$ and $E_i \in E$
    for j **in** J **do**
      $\vec{L}_j \in \vec{L}$, $\vec{e}_j \in \vec{e}$, $a_j \in a$, $s_i \in S$, $n_j \in n$, $\omega_j \in \omega$, $\Omega_j \in \Omega$, $I_j \in I$ and $E_j \in E$
      if $a_i \leqslant a_j$ **and** $r_{a,i} + s_i \leqslant r_{p,j} - s_j$ **do**
        remove j from J
      elif $a_j < a_i$ **and** $r_{a,j} + s_j \leqslant r_{p,i} - s_i$ **do**
        remove j from J
      end elif
      if J is empty **do**
        continue to next iteration
      end if
    end for
    calculate $\vec{r}_{i,\pm}$ from Equation (3.2)
    calculate $\vec{v}_{i,\pm}$ using $\vec{r}_{i,\pm}$ and Equation (A4)
    for j **in** J **do**
      calculate $\vec{r}_{j,\pm}$ and $\vec{v}_{j,\pm}$ in the same way
      calculate $\vec{r}^{\,1}_{i,\pm}$ using Equation (3.3)
      calculate $\vec{r}^{\,1}_{j,\pm}$ using Equation (3.4)
      $d_\pm = |\vec{r}^{\,1}_{j,\pm} - \vec{r}^{\,1}_{i,\pm}|$
      if $d_\pm < s_i + s_j$ **do** ▷ that is, do this for both d_+ and d_-
        append $\{i, j\}$ to L_{col}, $(\vec{r}^{\,1}_{i,\pm}, \vec{r}^{\,1}_{j,\pm})$ to R_{col} and $(\vec{v}_{i,\pm}, \vec{v}_{j,\pm})$ to V_{col}
        calculate $\Delta E_{i,\pm}$ and $\Delta E_{j,\pm}$ using Equations (3.7) and (3.8)
        $t^0_{i,\pm} \in T_0$, $t^0_{j,\pm} \in T_0$
        calculate $t^1_{i,\pm}$ and $t^1_{j,\pm}$ using Equation (3.6)
        $T_i = \frac{2\pi}{n_i}$, $T_j = \frac{2\pi}{n_j}$
        calculate $\delta_\pm$ using Equation (3.12)
        $t^{\text{col}}_{ij,\pm} = \text{TIMECOLLISION}(t^1_i, t^1_j, T_i, T_j, \delta)$
        append $t^{\text{col}}_{ij,\pm}$ to T_{col}
      end if
    end for
  end for
  sort T_{col} in increasing order and remove all $t^{\text{col}_{\text{ij}}} \in T_{\text{col}}$ for which $t^{\text{col}_{\text{ij}}} > t_{\max}$
  apply the same sorting to L_{col}, R_{col} and V_{col}
  return L_{col}, R_{col}, V_{col} and T_{col}
end procedure

Finally, we obtain Algorithm 4. This algorithm is deemed 'The continuous collision algorithm', as it is based upon the analytical Kepler orbits of all the satellites.

Algorithm 4 Continuous collision algorithm

Require: position R, velocity V, mass M, size S of all the satellites and a maximum simulation time t_{max}

1: $t = 0$
2: $L_{\text{col}}, R_{\text{col}}, V_{\text{col}}, T_{\text{col}} = \text{LISTCOLLISION}(R, V, M, S, T_0, t_{\text{max}})$ ▷ create initial collision list
3: **while** $t < t_{max}$ **do**
4: $\{i, j\} \in L_{col}$, $t_{ij}^{\text{col}} \in T_{\text{col}}$, $(\vec{r}_i, \vec{r}_j) \in R_{\text{col}}$, $(\vec{v}_i, \vec{v}_j) \in V_{\text{col}}$
5: $R_{fr}, V_{fr}, M_{fr}, S_{fr} = \text{SBM}(\vec{r}_i, \vec{r}_j, \vec{v}_i, \vec{v}_j, m_i, m_j, s_i, s_j, t_{ij}^{\text{col}})$ ▷ SBM(...) is defined in Section 4.2
6: merge R_{fr} with R, V_{fr} with V, M_{fr} with M and S_{fr} with S
7: delete all the i^{th} an j^{th} entries of R, V, M and S
8: delete all entries that contain an index i or j or value corresponding to i or j from $L_{\text{col}}, R_{col}, V_{\text{col}}, T_{\text{col}}$
9: $L_{\text{col,fr}}, R_{\text{col,fr}}, V_{\text{col,fr}}, T_{\text{col,fr}} = \text{LISTCOLLISION}(R, V, M, S, T_0, t_{\text{max}})$
10: merge $L_{\text{col,fr}}, R_{\text{col,fr}}, V_{\text{col,fr}}, T_{\text{col,fr}}$ with the existing collision lists
11: $t = t + t_{\text{ij}}^{\text{col}}$
12: **end while**

In line 2 of Algorithm 4, Algorithm 3 is called as it is described on the previous page. Fragments will be generated during the following iterations, which will have to be checked for collisions with other satellites and fragments from previous collisions as well.[10] Satellites that have not (yet) collided do not have to be checked for collisions again, since the collision lists for these satellites already exist. This will alter which indices should be checked for collisions, which is indicated by the comment on lines 7 and 8 in Algorithm 3.

Depending on how we store R, V, M and S, we may need to be careful to correct all the indices in the collision lists after line 7 in Algorithm 4. Deleting the i^{th} entry in R for example, causes all the entries that are stored at an index $j > i$ to shift down in their index by 1.

3.2. Collision detection for time-integrated system

This section describes how we can detect collisions without using the exact orbits of all the satellites, as opposed to Section 3.1. To do this we need an efficient way of searching for satellites that are close enough to each other such that a collision could occur within some interval Δt, the time step of the integration method. The information at hand is the position and velocity of all satellites. Take a particular satellite, say i, this satellite could potentially collide with any other satellite within a sphere of radius $d_i = |\vec{v}_i|\Delta t$. In the same way, any satellite j could collide with i as long as the latter is within a sphere of radius $d_j = |\vec{v}_j|\Delta t$ centered at j. Thus to find all possible collisions with i we need to check whether there is a non-empty intersection of the sphere of i with the one of j for all $j \neq i$. This amounts to

[10] Due to the nature of the collision model described in Section 4.2, fragments generated in the same collision are unlikely to collide.

checking $|\vec{r}_i - \vec{r}_j| \leqslant d_i + d_j = (|\vec{v}_i| + |\vec{v}_j|)\Delta t$, which may be simplified to

$$|\vec{r}_i - \vec{r}_j| \leqslant 2d_{max} = 2v_{max}\Delta t, \tag{3.19}$$

where $d_{max} = v_{max}\Delta t$ is the maximum radius, corresponding to the largest sphere, and v_{max} is the largest velocity. Checking condition (3.19) for all combinations of i and j and creating a list of potential collisions is certainly possible, but ultimately inefficient. Say our system has N satellites, then this brute-force method would have to check all $(N-1-i)$ values of j for the (N) values of i. The total number of comparisons is $\sum_{i=1}^{N} N - i = \frac{1}{2}N(N-1)$ and therefore of order N^2

3.2.1. K-d tree based algorithm for finding NN

The problem of efficiently searching for k nearby points is known as a k-Nearest Neighbour Search (kNNS), of which a variant is the Fixed Radius Nearest Neighbour Search (FRNNS). Both of these are relevant to detect potential colliding pairs. Using a special data structure known as a K-d tree it is not required to perform all distance comparisons [Ben75].

A K-d tree, or K-dimensional tree, organises data in a metric space along its K dimensions. In this case, the data are the positions of satellites in 3-D space and the metric is the Euclidean distance. Loosely stated, this data structure organises points that are close to each other by placing them in (K-dimensional) cells. The construction of the tree (and its cells) is done along each dimension in a binary way, as points are divided between two cells each time. Starting with the first dimension (X); all points with a horizontal coordinate smaller than the median point are grouped together in the left cell and all points greater than median point (and the median point itself) are put in the right cell. Then, the same procedure is applied to both of these cells in the second dimension (Y), splitting them up further into top and bottom cells. Now, again each of these cells is split along the remaining dimension (Z) into front and back ones. This results in 8 sub-cells that partition the entire domain of points. Repeatedly applying this entire procedure to each of these cells generates a structure of nested cells. Moreover, the dimension along which a cell is split is called its *splitting* dimension. Each of the cells is a node in the K-d tree. If a cell contains two (sub-)cells (left and right, up and down or front and back), it is called a *parent* node and its sub-cells *daughter* nodes. Cells that contain only points — i.e., cells that do not contain any other sub-cells — are called *leaf* nodes. The tree construction is complete, when each leaf node contains at most an integer m points, where $m \geqslant 1$ can be of our choosing.

We can now use the K-d tree structure to our advantage when performing a kNNS for satellite i. Firstly, we search for i's place in the tree. This is done by comparing its X coordinate to the median of the top node, which will place it either in its left or right daughter node. After the same comparison in the Y and Z coordinates for the next two layers of nodes, we again compare its X coordinate and so on. We repeat this until we reach a leaf node to place i in. Now we proceed to find the kNN. The idea is to keep track of a *nearest neighbor list* containing the k points with current smallest distance and alter the list as we walk through the tree. If we do this for all particles i in the system we get Algorithm 5.

Algorithm 5 kNNS for all particles in the system

1: **procedure** NNS($tree, R, k$)
2: $NN = \varnothing$
3: $D = \varnothing$
4: **for** $\vec{r}_i$ **in** R **do**
5: $NN_i = \varnothing$
6: $D_i = \varnothing$
7: walk down *tree* to i's leaf node, LN_i
8: **for** r_j **in** LN_i **do**
9: calculate the relative distance $d_{ij} = |\vec{r}_j - \vec{r}_i|$
10: **if** $d_{ij} < \max D_i$ **do**
11: append d_{ij} to D_i
12: **end if**
13: ensure D has no more than k elements, saving only the k smallest if necessary
14: **end for**
15: append all j that satisfy $d_{ij} \in D$ to NN_i and remove any that do not

We need to check if adjacent cells contain points that are closer than those in the current NN-list. Let d be the distance of the point in the current NN-list with the largest distance to i.

16: $d = \max D_i$
17: move up one level in the tree ▷ to the parent of the current node
18: d_{dn} = distance between $\vec{r}_i$ and the other daughter node along splitting dimension
19: **if** $d < d_{dn}$ **do**
20: walk down this daughter node until a leaf node LN is reached
21: **for** $\vec{r}_j$ **in** LN **do**
22: repeat lines 9 through 13
23: **end for**
24: go back to line 15
25: **else** ▷ 'prune' daughter node from the tree
26: **if** current node is the top node **do**
27: append NN_i to NN and D_i to D
28: continue to next iteration ▷ all the NN of i are found
29: **else**
30: go back to line 15
31: **end elif**
32: **end elif**
33: **end for**
34: **return** NNs and D
35: **end procedure**

The algorithm for a FRNNS is simpler than for kNNS in the sense that the radius within which we are looking for NN, i.e. $2d_{max}$, is fixed. So only the nodes that have a non-empty

intersection with the sphere of this radius centered at i will have to be checked.[11]

3.2.2. Time complexity of K-d tree and its algorithms

In the paper where he originally introduced the K-d tree, Bentley showed that it could be constructed in $O(N \log N)$ time, assuming the median at each cell splitting can be found in $O(N)$ time. The algorithms that finds the median in this time are complicated. However, if the data is presorted along the k dimensions, then a (balanced) K-d tree can in best case be built in $O(N \log N)$ and in worst case $O(kN \log N)$ ([Bro15]). The time to find k NN is at most $O(k \log N)$, which is simply the time of finding one nearest neighbor point multiplied by k. On the other hand, the time complexity of the FRNNS algorithm is $O(3^k k \log N)$ ([BSW77]). Due to this higher time complexity of the FRNNS, we will use the kNNS algorithm instead. As a consequence, k becomes a parameter in the model, representing the expected number of NN for each satellite at any given time. Given an approximately homogeneous distribution of N satellites in relatively thin shell of height h around the Earth with radius R [12], we can expect

$$\bar{k} = \left\lceil \frac{N(2v_{max}\Delta t)^3}{(R+h)^3 - R^3} \right\rceil \approx \left\lceil \frac{N(2v_{max}\Delta t)^3}{R^2(R+3h)} \right\rceil = \left\lceil N \frac{8d_{max}^3}{R^2(R+3h)} \right\rceil . \tag{3.20}$$

For $\Delta t = 10\text{s}$, $N = 10^5$ and $v_{max} = 8\text{km}\,\text{s}^{-1}$, we have $\bar{k} = 2$. It is better overestimate $\bar{k}$, as we can always delete NN whose relative distances do not satisfy requirement (3.19). This requires only a few extra distance comparisons, because the kNNS algorithm computes all distances anyway. Note that as more and more fragments enter the system, the value of $\bar{k}$ will increase in proportion to N.

As for the actual implementation of the K-d tree data structure and its kNNS algorithm in this model, the *Python* library scikit-learn was used ([Ped+11]).

3.2.3. Colliding pairs from NN

Once we have the NN of every satellite, we can determine which satellites will collide. To this end, suppose that we have found that i and j are NN of each other, let $\vec{v}_i$, $\vec{v}_j$, $\vec{r}_i$ and $\vec{r}_j$ be their respective positions and velocities. Then

$$\vec{d} = \vec{r}_j - \vec{r}_i, \quad \text{and} \quad \vec{u} = \vec{v}_j - \vec{v}_i,$$

are relative position and velocity. First, we linearise the motion of the satellites in the current time step. If

$$\vec{\rho}_i(t) = \vec{r}_i + \vec{v}_i t \quad \text{and} \quad \vec{\rho}_j(t) = \vec{r}_j + \vec{v}_j t,$$

then

$$\vec{\delta}(t) = \vec{\rho}_j(t) - \vec{\rho}_i(t) = \vec{d} + \vec{u}t, \tag{3.21}$$

is their relative position. We know that at the current time $t = t^*$ these satellites are close to each other, but we still need to check if their minimum relative distance $\delta = |\vec{\delta}|$ occurs within the time step $t = t^* + \Delta t$. This is the case, if δ is decreasing at $t = t^*$ and increasing $t = t^* + \Delta t$,

[11] There is ground to be gained here as well. Suppose an entire node is contained within the search sphere, then then there is no need to walk down it. We can in that case simply add all points in that node to the NN-list [BSC15].

[12] $\frac{h}{R} \ll 1$

or equivalently, if the component of $\vec{u}$ along $\vec{\delta}$ is positive at first and negative a time step later. Hence we require

$$\vec{u} \cdot \vec{\delta}(t^*) < 0 \quad \text{and} \quad \vec{u} \cdot \vec{\delta}(t^* + \Delta t) = \vec{u} \cdot (\vec{\delta}(t^*) + \vec{u}\Delta t) > 0, \tag{3.22}$$

where the equality follows from the linear motion of the satellites. Henceforth, to ease calculations and without loss of generality, we assume $t^* = 0$. If the conditions of (3.22) are satisfied, then we proceed to calculate the collision time $t_{i,j}^{\text{col}}$ by minimising d^2:

$$\begin{aligned}
\left.\frac{d\delta^2}{dt}\right|_{t=t_{i,j}^{\text{col}}} &= \left.\frac{d}{dt}\{\vec{d} \cdot \vec{d} + 2\vec{u} \cdot \vec{d}\,t + \vec{u} \cdot \vec{u}\,t^2\}\right|_{t=t_{i,j}^{\text{col}}}, \\
&= \vec{u} \cdot \vec{d}\,t_{i,j}^{\text{col}} + \vec{u} \cdot \vec{u}(t_{i,j}^{\text{col}})^2, \\
&= 0,
\end{aligned}$$

from which it follows that

$$t_{i,j}^{\text{col}} = -\frac{\vec{u} \cdot \vec{d}}{\vec{u} \cdot \vec{u}}. \tag{3.23}$$

Finally, we determine the minimum distance by substituting $t = t_{i,j}^{\text{col}}$ in Equation (3.21) and require that it ought to be smaller than the sum of the radii of the satellites, s_i and s_j

$$\delta^2(t_{i,j}^{\text{col}}) = \frac{(\vec{d} \cdot \vec{d})^2(\vec{u} \cdot \vec{u})^2 - (\vec{u} \cdot \vec{d})^2}{\vec{u} \cdot \vec{u}} = \frac{|\vec{u} \times \vec{d}|^2}{\vec{u} \cdot \vec{u}} < (s_i + s_j)^2,$$

giving the condition

$$\frac{|\vec{u} \times \vec{d}|}{|\vec{u}|} < s_i + s_j. \tag{3.24}$$

The procedure for checking these collision conditions for some satellite i with a set of other satellites Φ may is illustrated in Algorithm 6.

Algorithm 6 Checking collision conditions

procedure CHECKCOLLISION($R, V, i, \Phi, \Delta t$)
  $\vec{r}_i \in R$
  $\vec{v}_i \in V$
  $T_{\text{col}} = \varnothing$
  $J = \varnothing$
  For j **in** Φ **do**
    $\vec{r}_j \in R$
    $\vec{v}_j \in V$
    $\vec{d} = \vec{r}_j - \vec{r}_i$
    $\vec{u} = \vec{v}_j - \vec{v}_i$
    if $\vec{u} \cdot \vec{d} < 0$ **and** $\vec{u} \cdot (\vec{d} + \vec{u}\Delta t) > 0$ **do**
      if $|\vec{u} \times \vec{d}| < |\vec{u}|(s_i + s_j)$ **do**
        append j to J and $t_{\text{ij}} = -(\vec{u} \cdot \vec{d})/(\vec{u} \cdot \vec{u})$ to T_{col}
      else
        next ▷ minimum distance is too large
      end elif
    else
      next ▷ satellites share no minimum distance.
    end elif
  end for
  return J and T_{col}
end procedure

3.2.4. Collision ambiguity

Consider the two dimensional box of particles illustrated in Figure 2 a) and b).

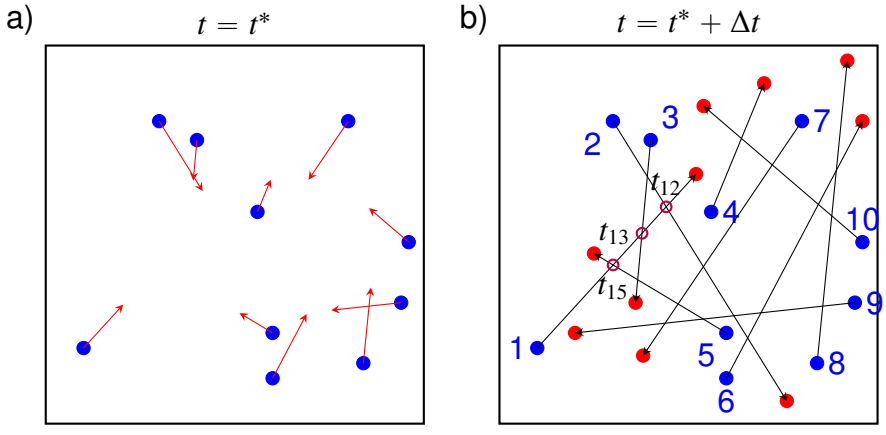


Figure 2: Collection of particles (satellites) situated in a two dimensional plane. The blue dots and red arrows in **a)** indicate the location and velocity of the particles at the beginning of the current time step $t = t^*$. In **b)** the same particles are now labelled by numbers 1 through 10 and their locations at $t = t^* + \Delta t$ are depicted as red dots. The linearised trajectory of a particle is represented by a black arrow. All the possible collision points of particle 1 are indicated with purple circles and labelled with the respective collisions times t_{12}, t_{13} and t_{15}.

Suppose these particles represent satellites and that each intersection between two trajectories depicted in Figure 2 b) is a possible collision point between the two satellites, i.e. conditions (3.22) and (3.24) are satisfied. For simplicity we assume that all these particles are all NNs of each other. The collision points for particle 1 are indicated in Figure 2 b). If we actually perform all these collisions in the current time step, we would run into a constraint of our model; a satellite can only collide once. After a collision, a satellite is deleted from the list of all satellites and replaced by its fragments. Since it is nonphysical for satellites to pass through each other, we must select only the earliest collision. To this end we produce following matrix

$$T^{\text{col}} = \begin{pmatrix} t_{12} & t_{13} & t_{15} & & & & & \\ & t_{23} & & t_{26} & t_{27} & & t_{29} & \\ & & t_{35} & & & & & t_{410} \\ & & & & t_{57} & & t_{59} & \\ & & \text{Sym.} & & & t_{68} & t_{69} & t_{610} \\ & & & & & & t_{79} & t_{710} \\ & & & & & & t_{89} & t_{810} \\ & & & & & & & t_{910} \end{pmatrix}.$$

Each row and column in T^{col} correspond to particle from Figure 2. Moreover, the entry t_{ij} on the i^{th} row and j^{th} column holds the time of the collision, should one occur between the respective particles. Note that

$$t_{\text{ij}} = t_{\text{ji}} \; \forall i, j. \tag{3.25}$$

This particular configuration of T^{col} allows us to find the earliest collisions. To do this, we set all the empty entries in T^{col} to a value $T \geqslant \Delta t$ and look for the minimum collision time along each of its rows and columns. In particular, if an entry t_{ij} satisfies

$$\text{for } n = i \; \forall j: \; t_{\text{ij}} < T^{\text{col}}_{nm} \; \wedge \; \text{for } m = j \; \forall i: \; t_{\text{ij}} < T^{\text{col}}_{nm}, \tag{3.26}$$

it will remain unchanged, otherwise it will be set to equal to T. Applying this to the above example of T^{col} will result in a new matrix

$$T^{col,0} = \begin{pmatrix} & t_{23} & & & & & \\ t_{32} & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & t_{89} & \\ & & & & t_{98} & & \end{pmatrix},$$

where all the blank spaces represent the value T. Observe that we can now extract the indices of two colliding pairs from this matrix

$$\text{iteration 0: } \{(2,3),(8.9)\}.$$

These particles must collide as the collision times are the minimal collision times for each of the involved particles, i.e. t_{98} and t_{89} satisfy condition (3.26), as do t_{23} and t_{32}. The reason the complementary pairs of collisions times are each the minimum of their respective row and column stems from Equation (3.25) and the symmetry of condition (3.26). One could argue this concludes the search for collisions in the current time step, as the earliest possible are now found. If we assume, however, that the fragments of these collisions do not greatly affect the trajectories of the remaining particles, then further collisions are possible after the first. In order to find these remaining collisions, we remove the rows and columns corresponding to the colliding pairs we have already found from the original T^{col} matrix. If we then apply the same procedure as above to this updated T^{col}, i.e. finding the minima along all rows and columns, we get

$$T^{col,1} = \begin{pmatrix} & & \\ & & t_{610} \\ & t_{106} & \end{pmatrix}.$$

This yields another colliding pair

$$\text{iteration 1: } \{(6,10)\}.$$

In the same way one may again determine $T^{col,2}$ and find the pair

$$\text{iteration 2: } \{(5,7)\},$$

after which $T^{col,3}$ is empty. This is because the remaining particles 1 and 4 share no collision point. In summary, we find the following set of colliding pairs

$$\{(2,3),(5,7),(6,10),(8,9)\}.$$

The above discussion yields Algorithm 7 for finding the first collisions.

Algorithm 7 Determining the first collision(s)

procedure FIRSTCOLLISION($P, T_{\text{col}}, \Delta t$)
  for $\{i, j\}$ **in** P **do**
    $T^{\text{col}}_{ij} = T^{\text{col}}_{ji} = t_{\text{ij}} \in T_{\text{col}}$ ▷ construct T^{col} matrix
  end for
  for $\{i, j\}$ **not in** P **do**
    $T^{\text{col}}_{ij} = \Delta t$
  end for
  $\Psi = \varnothing$
  $\tau = \varnothing$
  while $T^{\text{col}} \neq \varnothing$ **do**
    $t = \{t_{\text{ij}} \mid t_{\text{ij}}$ satisfies condition $(3.26)\}$
    $p = \{\{i, j\} \mid t_{\text{ij}} \in t\}$
    merge p with Ψ and t_{ij} with τ
    remove any row $r = i, j$ and column $c = i, j$ from T^{col} for all $\{i, j\} \in P$
  end while
  return Ψ and τ
end procedure

3.2.5. The discrete algorithm

Algorithm 8 outlines how the methods discussed in Sections 2.2, 3.2.1, 3.2.3 and 3.2.4 are combined into a model that simulates a system of colliding satellites. The algorithm lends its name to the discretisation of time that it is based on.

Algorithm 8 Discrete collision algorithm

Require: position R, velocity V, mass M, size S of all the satellites, a time step Δt, a maximum simulation time t_{max} and the number of expected nearest neighbours k

$t = 0$
$N =$ The total number of satellites
$tree =$ K-DTREE(R) ▷ construct a K-d tree of initial positions
while $t < t_{max}$ **do**
  $NNs =$ NNS$(tree, k)$ ▷ determine NNs of each satellite
  $P = \varnothing$
  $T_{\text{col}} = \varnothing$
  for i:=0 **to** N **do**
    $J =$ the i^{th} list of NNs ▷ indices of satellite i's nearest neighbours
    if J is empty **then**
      continue to beginning of loop
    end if
    $J, T_{\text{col}} =$ CHECKCOLLISION$(R, V, i, J, \Delta t)$ ▷ check conditions (3.22) and (3.24)
    if $J = \varnothing$ **then**
      continue to beginning of loop
    end if
    append the set $\{\{i, j\} | \forall j \in J\}$ to P and T_{col} to T_{col}.
  end for
  $P, T_{\text{col}} =$ FIRSTCOLLISION$(P, T_{\text{col}}, \Delta t)$ ▷ pick out only the first collisions
  for $\{i, j\}$ **to** P **do**
    $t_{\text{ij}} \in T_{\text{col}}$
    $R_{fr}, V_{fr}, M_{fr}, S_{fr} = \text{SBM}(\vec{r}_i, \vec{r}_j, \vec{v}_i, \vec{v}_j, m_i, m_j, s_i, s_j, t_{\text{ij}})$ ▷ SBM$(\dots)$ is defined in Section 4.2
    append R_{fr} to R, V_{fr} to V, M_{fr} to M and S_{fr} to S
  end for
  append fragment parameters to $\vec{r}, \vec{v}, m, s$
  delete $\vec{r}_i, \vec{v}_i, m_i, s_i$ for all $i \in P$
  $tree =$ K-DTREE$(\vec{r})$ ▷ construct new tree
  $t = t + \Delta t$
end while

3.3. Comparison of the continuous discrete algorithms

In order to compare Algorithms 4 and 8, we generate a homogeneous distribution of N satellites within a spherical shell of inner radius r_{inner} and height h. To this end, let for $i = 1, 2, \dots, N$

$$\Omega_i, \omega_i, M_{a,i} \in \mathcal{U}[0, 2\pi) \quad \text{and} \quad I_i \in \mathcal{U}[0, \pi).$$

To ensure that the mean anomaly is zero in the periapsis, we redefine

$$M_{a,i} \leftarrow M_{a,i} - \omega_i.$$

Also, let

$$r_{i,1}, r_{i,2} \in \mathcal{U}[r_{\text{inner}}, r_{\text{inner}} + h], \tag{3.27}$$

then the periapsis and apoapsis are given as

$$r_{p,i} = \min\{r_{i,1}, r_{i,2}\}, \quad r_{a,i} = \max\{r_{i,1}, r_{i,2}\}.$$

The semi-major axis and eccentricity are

$$a_i = \frac{r_{a,i} + r_{p,i}}{2}, \quad e_i = \frac{r_{a,i} - r_{p,i}}{r_{a,i} + r_{p,i}} \tag{3.28}$$

The mean motion n_i may be obtained through Equation (2.6). The radius s and mass m is taken to be the same for all the satellites. A useful way of characterising the various orbits is via a Gabbard diagram, which plots the periapsis and apoapsis of a satellite against its orbital period. Figure 3 shows the Gabbard diagram for this homogeneous distribution of satellites.

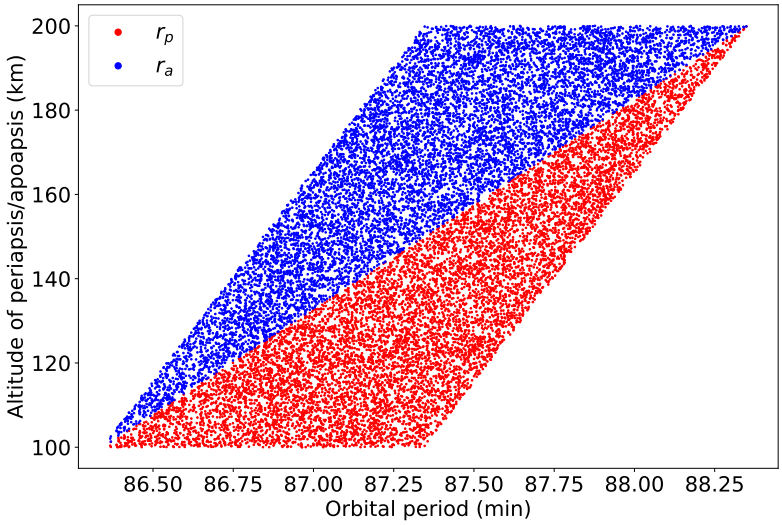


Figure 3: Gabbard diagram for a homogeneous distribution of $N = 10000$ satellites with $r_{\text{inner}} = R_\oplus + 100$ km and $h = 100$ km. The red and blue dots indicate the periapsis and apoapsis of each of the satellites, respectively. Note that at any particular height, the density of satellites is approximately constant. In addition, this distribution contains orbits with eccentricities ranging from $e = 1.06 \times 10^{-7}$ to $e = 7.63 \times 10^{-3}$. This is because the relatively low value of h does not allow for highly eccentric orbits.

Comparison of the continuous and discrete algorithms is done by analysing their predicted time of the first collision for this distribution. As derived by Dr. Visser, the expected time for a collision in this case is given by

$$\bar{t}_{\text{col}} = \frac{\bar{a} h \bar{T}}{\pi N^2 s^2}, \tag{3.29}$$

where $\bar{a}$ is the average semi-major axis and $\bar{T}$ is the average orbital period. Equation (3.29)

defines the following three relations for $\bar{t}_{\text{col}}$

$$\bar{t}_{\text{col}} = c_1 \frac{1}{N^2} \quad \text{for constant } s \text{ and } h, \tag{3.30}$$

$$\bar{t}_{\text{col}} = c_2 \frac{1}{s^2} \quad \text{for constant } N \text{ and } h, \tag{3.31}$$

$$\bar{t}_{\text{col}} = c_3 \frac{1}{h} \quad \text{for constant } N \text{ and } s. \tag{3.32}$$

The constants c_1, c_2 and c_3, are determined from Equation (3.29). For the case of the homogeneous distribution of Figure 3, we have $\bar{a} = 6.52 \times 10^3$km and $\bar{T} = 5.42 \times 10^3$s so that

$$c_1 = 1.12 \times 10^{13}\ \text{s}, \quad c_2 = 1.12 \times 10^{7}\ \text{s}\,\text{m}^{-2} \quad \text{and} \quad c_3 = 1.12\ \text{s}\,\text{m}^{-2}. \tag{3.33}$$

In case of Algorithm 4, the time of the first collision is simply the first entry of the initial collision list. In contrast, Algorithm 8 is executed until it finds a collision. The results of this analysis are presented in Figures 4, 5 and 6.

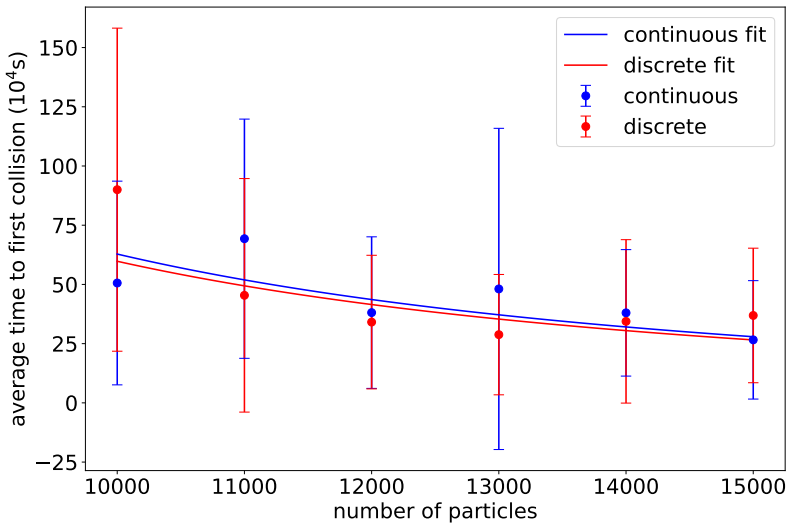


Figure 4: Error bar plot of the average time to the first collision against the particle number N over 10 runs, as predicted by Algorithm 4 in blue and Algorithm 8 in red and for $s = 10$ m and $h = 100$ km. The red and blue lines are a linear-least squares fit applied to the data for the relation between t_{col} and N given in Equation (3.30).

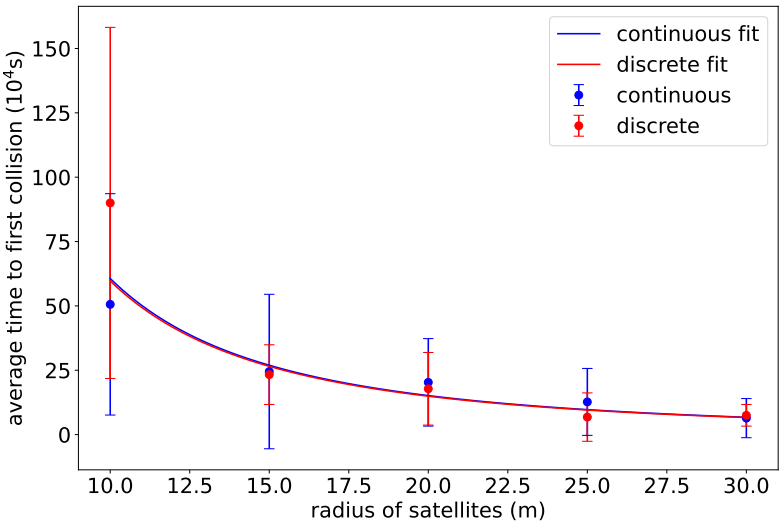


Figure 5: Same as Figure 4, but now the average time to first collision versus the radius of the satellites s is plotted. The linear-least squares fit is now applied for the relation between t_{col} and s given in (3.31).

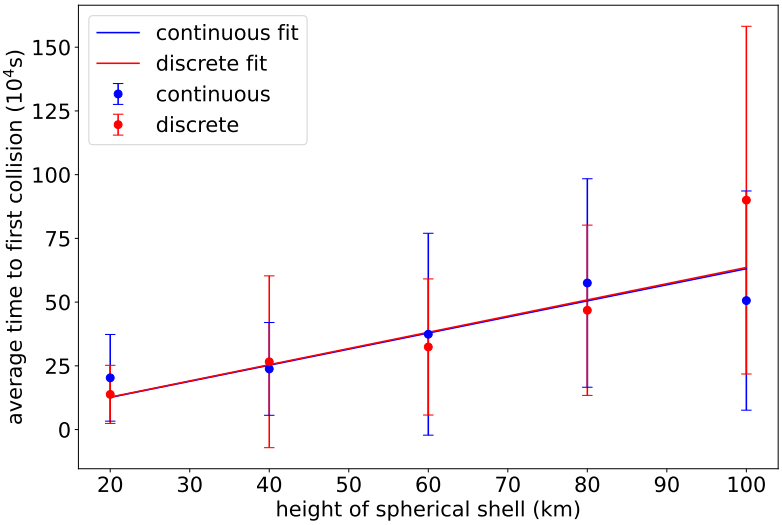


Figure 6: Same as Figure 4, but now the average time to first collision versus the height of the spherical shell h is plotted. The linear-least squares fit is now applied for the relation between t_{col} and h given in (3.32).

From the fits in Figures 4, 5 and 6 we find for the continuous algorithm

$$c_1 = (6.28 \pm 0.55) \times 10^{13}\ \text{s}, \quad c_2 = (6.06 \pm 0.62) \times 10^{7}\ \text{s}\,\text{m}^{-2} \quad \text{and} \quad c_3 = (6.31 \pm 0.68)\ \text{s}\,\text{m}^{-2}. \tag{3.34}$$

Similarly, for the discrete algorithm

$$c_1 = (5.98 \pm 0.68) \times 10^{13}\ \text{s}, \quad c_2 = (5.97 \pm 0.61) \times 10^{7}\ \text{s}\,\text{m}^{-2} \quad \text{and} \quad c_3 = (6.36 \pm 0.56)\ \text{s}\,\text{m}^{-2}. \tag{3.35}$$

From Figures 4, 5 and 6 we find that the averages from the the algorithms agree with each other. As a matter of fact, each separate data point in these figures was generated using 10 different instances of the homogeneous distribution of satellites. This means that the algorithms generate similar predictions for the averages.

Moreover, this verification is not limited to the algorithms themselves. Equation (3.29) is verified by results of the linear least-squares fits given in (3.34) and (3.35), which agree with (3.33) up to constant factor of about 6. In any case, the inverse-square relations between $\bar{t}_{\text{col}}$ and s, N given by (3.30) and (3.31) and the linear relation between $\bar{t}_{\text{col}}$ and h appear to properly represent the data.

Another point of investigation is the execution time of the both of the algorithms depending on the parameters N, s and h. To compare the two algorithms, we will now look at the required computation time until the first collision is found. This means that we will be comparing the time Algorithm 4 spends on creating the initial collision list to the time Algorithm 8 is executed until it finds a collision in a certain time step. However, instead of actually executing the Algorithm 8 until the occurrence of the first the collision, we estimate its projected total execution by determining the average execution time for a single time step and multiplying that with the number of expected time steps,

$$\bar{t}_{\text{exec,discrete}} \approx \bar{t}_{\text{exec},\Delta t} \frac{\bar{t}_{\text{col}}}{\Delta t}, \tag{3.36}$$

where $\bar{t}_{\text{exec},\Delta t}$ is the average execution time of one time step of Algorithm 8. The results of this analysis are presented in Figures 7, 8 and 9.[13]

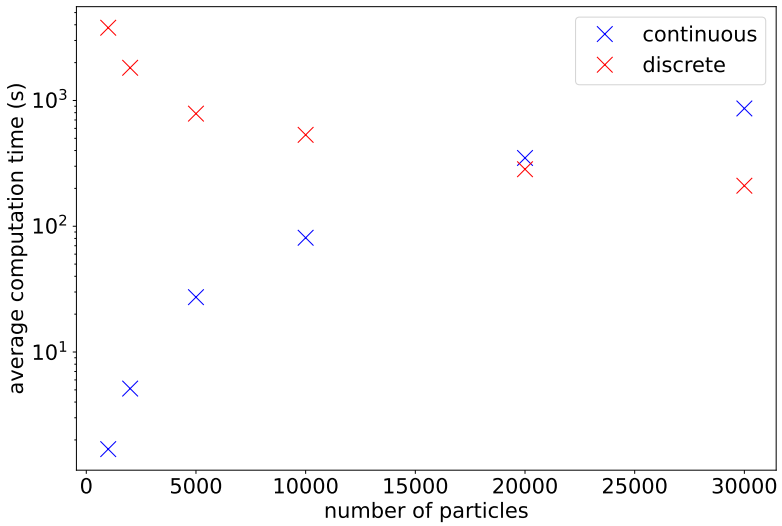


Figure 7: Semi-log plot of the average computation time until the first collision against the particle number N over 5 runs for Algorithm 4 in blue and Algorithm 8 in red. The algorithms have been applied to the same distribution with $s = 10$ m and $h = 100$ km.

[13] All execution times have been obtained using a **HP ZBook Studio G5 with Intel Core i7 processor (6×2.20 GHz) and 16 GB DDR4-SDRAM**

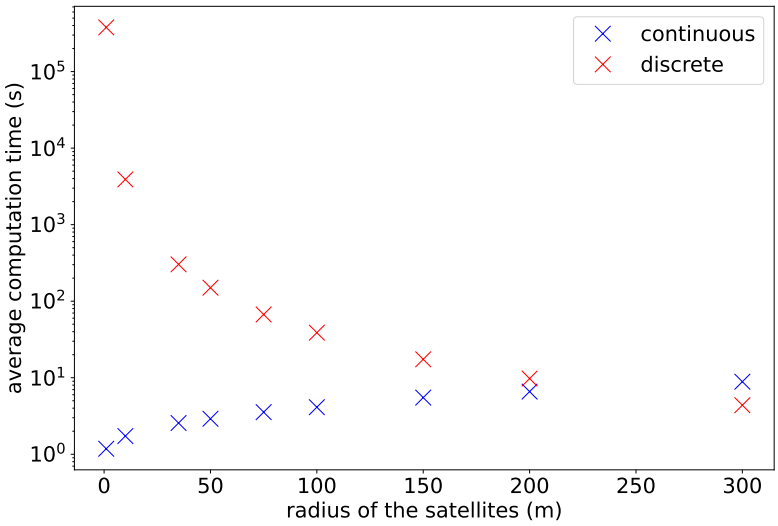


Figure 8: Same as Figure 7, but now the average computation time against the satellite radius s over 5 runs is shown. The algorithms have been applied to the same distribution with $N = 1000$ and $h = 100$ km.

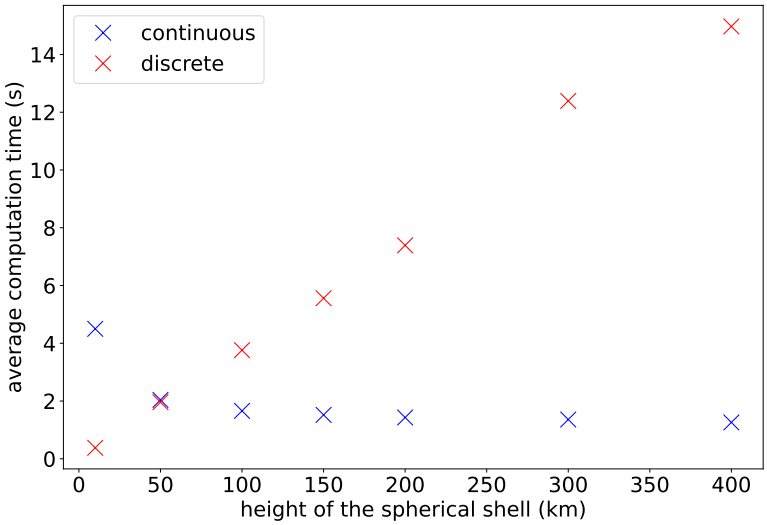


Figure 9: The average computation time until the first collision against the height of the spherical shell h over 5 runs. The algorithms have been applied to the same distribution with $N = 1000$ and $s = 10$ m.

From Figure 7 we see that the continuous algorithm is expected to perform better than the discrete algorithm for systems with low particle numbers. This can be attributed to the fact the creation of the initial collision list Algorithm 3 has a time-complexity of at least $O(N^2)$. Depending on the exact distribution of satellites, there will be some number of particles N above which the discrete algorithm performs better. In the case of this homogeneous distribution

of satellites with $s = 10$ m and $h = 100$ km this turning point in efficiency occurs around $N = 20000$.

We observe from Figure 8 that the continuous algorithm performs best for distributions of smaller satellites, that is for s up to about 200 m for $N = 1000$ and $h = 100$ km. For larger radii Algorithm 3 has to perform more computationally expensive calculations for each pair of satellites, because the condition (3.1) is now satisfied more often. In other words, more (possible) colliding pairs are formed for larger radii. Hence the continuous algorithm has to spend more time on more satellites; calculating the (location of the) MOID, checking the minimal distance and possibly calculating collision times.

Lastly, Figure 9 indicates that for a small shell the continuous algorithm faces the same hurdle as it did for increased particle number and larger satellite radius; more possible colliding pairs of satellites, which lead to more (expensive) calculations.

As for the average computation time of the discrete algorithm, it closely follows the relation (3.29), that is inverse quadratic in N and s and linear in h. This is to be expected, as we have used that relation to obtain these results. The reason for this correspondence of the discrete algorithm between the average time to first collision $\bar{t}_{\text{col}}$ and the average execution time thereof $\bar{t}_{\text{exec,discrete}}$ lies in that the average computation time of a time step $\bar{t}_{\text{exec},\Delta t}$ only grows linearly in N. This can be seen from the single for-loop on line 8 in Algorithm 8, as opposed to the double for-loops on lines 7 and 10 in Algorithm 3. In other words, for any particle that is added to the system the continuous algorithm has to perform part of or all its collisions checks for that particle against all other N particles. In contrast, the discrete algorithm merely needs to build the K-d tree for $N + 1$ particles and perform one extra numerical integration.

Another topic of similarity between the discrete and continuous is how they may be optimised. Next to the K-d tree method discussed in Section 3.2.2, the implementation of the discrete algorithm uses the *C/C++* extension to *Python*, *Numpy* to outsource among other things the for-loop in line 8 of Algorithm 8. Though this does not change the time-complexity of the discrete algorithm, it does give it a relative advantage over the continuous algorithm, which makes use of *Numpy* to a lesser extent. In any case, i) the construction of a K-d tree, ii) the numerical integration of the satellites and iii) the creation of the collision list all perform calculations that can be considered as independent of each other and are therefore able to be executed in parallel on a GPU ([Bro15]). This would allow for higher particle numbers in both the discrete and continuous algorithms.

Additionally, some of the assumptions made to model the system of satellites using idealised Kepler orbits, could be dropped in favor of realism. For instance, both algorithms do not incorporate atmospheric drag, even though this is one of the major sinks of debris in the LEO. What is more, the assumption that the gravitational potential field of the Earth is spherically symmetric, has as a consequence that the satellite orbits do not precess. This also has a major effect on the spread of debris in LEO. All these and the other assumptions allow the continuous algorithm to assume that the collisions found earlier in its execution to remain unaltered. Put differently, the only time a collision is removed from the existing collision list is when one or more of the satellites involved in that collision collide in the current iteration. This means that a future version of the continuous algorithm will at most have to create a new collision list after each collision to adjust for drag. In contrast, the discrete algorithm can easily be adjusted to include atmospheric drag terms, a non-homogeneous gravitational field and the influence of other perturbing bodies like the moon. Hence the most ground to be gained here is for the continuous algorithm.

4. Modelling collisions

As this report aims to simulate the exact dynamics of the entire satellite system, we require a recipe for performing collisions between satellites. Fortunately, the NASA's Standard Breakup Model (SBM) is able to distinguish between types of collisions and give key parameters of fragments, like the number produced, their characteristic length (scattering cross-section), masses and ejection velocities.

Before describing how NASA's SBM may be implemented in this context, let us first consider a simpler kind of collision; one where no fragmentation takes place and satellites merely 'bounce' off each other.

4.1. Elastic collisions

If the exact geometry of two masses involved in a collision as well as the fraction of conserved kinetic energy $\epsilon = \frac{E_{k,f}}{E_{k,i}}$ [14] are known, then the pre- and post-collision velocities are fully determined by the conservation of mass and momentum. Moreover, a fully elastic collision conserves all kinetic energy, which implies $\epsilon = 1$. However, to prevent modelling not only the exact shape and dimensions of all the satellites in Earth's orbit, but also those of all the fragments resulting from collisions, we instead choose to keep track of only one value, for satellite and fragment alike. This value is the *characteristic length*, which is related to the (radio) cross-section in the NASA SBM, as is discussed in Section 4.2. As a consequence, we wish to simulate an elastic collision of two bodies without assuming their exact geometry. We do this by introducing a small, random deflection to one of the velocity vectors and using conservation of momentum and kinetic energy to determine the other velocity vector.

More concretely, let $\vec{v_1}$, $\vec{v_2}$, m_1 and m_2 be the velocities and masses of two bodies on a collision trajectory. Also let $\vec{v_1}'$ and $\vec{v_2}'$ be the velocities after the collision. The collision may alter the magnitude. Conservation of momentum and kinetic energy give

$$\vec{v}'_2 = \vec{v}_2 + \frac{m_1}{m_2}\left(\vec{v}_1 - |\vec{v}'_1|\hat{v}'_1\right) \tag{4.1}$$

$$|\vec{v}'_2|^2 = |\vec{v}_2|^2 + \frac{m_1}{m_2}\left(|\vec{v}_1|^2 - |\vec{v}'_1|^2\right), \tag{4.2}$$

where the post-collision velocity of the first particle is written as its magnitude times its direction $\vec{v}'_1 = |\vec{v}'_1|\hat{v}'_1$. Taking the inner product of the first equation with itself and equating with the second gives, after some manipulation

$$\frac{m_1}{m_2}\left(|\vec{v_1}|^2 - 2|\vec{v}'_1|\hat{v}'_1 \cdot \vec{v}_1 + |\vec{v}'_1|^2\right) + 2\left(\vec{v}_1 \cdot \vec{v}_2 - |\vec{v}'_1|\hat{v}'_1 \cdot \vec{v}_2\right) = |\vec{v}_1|^2 - |\vec{v}'_1|^2$$

Rewriting further gives

$$|\vec{v}'_1|^2\left(\frac{m_1}{m_2}+1\right) - 2|\vec{v}'_1|\left(\frac{m_1}{m_2}\hat{v}'_1 \cdot \vec{v}_1 + \hat{v}'_1 \cdot \vec{v}_2\right) + |\vec{v}_1|^2\left(\frac{m_1}{m_2}-1\right) + 2\vec{v}_1 \cdot \vec{v}_2 = 0,$$

[14] Here, $E_{k,i}$ and $E_{k,f}$ are the total pre- and post-collision kinetic energy, respectively.

which is a quadratic equation in $|\vec{v}'_1|$ and may be solved using the quadratic formula. Setting

$$\begin{aligned}
\mathcal{A} &= \frac{m_1}{m_2} + 1 = \frac{m_1 + m_2}{m_2} \\
\mathcal{B} &= -2\left(\frac{m_1}{m_2}\hat{v}'_1 \cdot \vec{v}_1 + \hat{v}'_1 \cdot \vec{v}_2\right) = -2\hat{v}'_1 \cdot \left(\frac{m_1}{m_2}\vec{v}_1 + \vec{v}_2\right) \\
\mathcal{C} &= |\vec{v}_1|^2\left(\frac{m_1}{m_2} - 1\right) + 2\vec{v}_1 \cdot \vec{v}_2 = \vec{v}_1 \cdot \left(\frac{m_1 - m_2}{m_2}\vec{v}_1 + 2\vec{v}_2\right),
\end{aligned}$$

then gives the solution as

$$|\vec{v}'_1| = \frac{-\mathcal{B} \pm \sqrt{\mathcal{B}^2 - 4\mathcal{A}\mathcal{C}}}{2\mathcal{A}}$$

If $\mu_1 = \frac{m_1}{m_1+m_2}$ and $\mu_2 = \frac{m_2}{m_1+m_2}$, then

$$|\vec{v}'_1| = \hat{v}'_1 \cdot (\mu_1\vec{v}_1 + \mu_2\vec{v}_2) \pm \sqrt{\left[\hat{v}'_1 \cdot (\mu_1\vec{v}_1 + \mu_2\vec{v}_2)\right]^2 - \vec{v}_1 \cdot ((\mu_1 - \mu_2)\vec{v}_1 + 2\mu_2\vec{v}_2)} \quad (4.3)$$

In order to simplify equation (4.3), a transformation to the centre of mass (COM) frame is performed

$$\vec{v}_i \longrightarrow \vec{u}_i = \vec{v}_i - \vec{V}_{com} \qquad i = 1, 2 \quad ,$$

where $\vec{V}_{com} = \frac{m_1\vec{v}_1 + m_2\vec{v}_2}{m_1+m_2}$, which will remain unchanged, because of conservation of momentum and mass. We have the following identity,

$$\begin{aligned}
\mu_1\vec{u}_1 + \mu_2\vec{u}_2 &= \frac{m_1\vec{u}_1 + m_2\vec{u}_2}{m_1 + m_2} \\
&= \vec{V} - \vec{V} \\
&= 0
\end{aligned}$$

which cancels several terms in equation (4.3), leaving

$$|\vec{u}'_1|^2 = \mu_2\vec{u}_1 \cdot (\vec{u}_1 - \vec{u}_2) = \mu_2^2|\vec{u}|^2 = |\vec{u}_1|^2, \quad (4.4)$$

where $\vec{u} = \vec{v}_2 - \vec{v}_1$. The last identity follows by working out the inner products $\vec{u}_1 \cdot \vec{u}_1$ and $\vec{u}_1 \cdot \vec{u}_2$. By symmetry or by equation (4.2) it must also hold that

$$|\vec{u}'_2|^2 = \mu_1^2|\vec{u}|^2 = |\vec{u}_2|^2 \quad (4.5)$$

Note in addition that the total kinetic energy may be partitioned into

$$E_k = \frac{M|\vec{V}_{com}|^2}{2} + \frac{m_1m_2|\vec{u}|^2}{2M} = E_{k,com} + E_{k,int}, \quad (4.6)$$

in which $M = m_1 + m_2$. The former of the two contributions comes from the kinetic energy due to movement of the COM frame itself, $E_{k,com}$, and the latter stems from the inherent or internal kinetic energy of the particles in the COM frame, $E_{k,int}$. For the same reasons that V_{com} does

not change, $E_{k,com}$ does not either. Additionally, in an elastic collision $E_{k,int}$ does not change; calculating $E_{k,int}$ directly, using the just derived expressions for the scattered magnitudes, gives

$$E_{k,int} = \frac{m_1\mu_2^2|\vec{u}|^2}{2} + \frac{m_2\mu_1^2|\vec{u}|^2}{2} = \frac{m_1m_2|\vec{u}|^2}{2M}$$

So we recover the internal kinetic energy, as expected.

Equations (4.4) and (4.5) give expressions for the magnitude of the scattered velocities in the COM frame. It therefore remains to find expressions for the direction of those velocities. To that end, another conversion is used. This time to a more suited (spherical) coordinate system

$$\vec{u_1}' \longrightarrow |\vec{u}'_1|\hat{u}'_1(\theta_1, \phi_1)$$

where, as before, $|\vec{u}'_1|$ is the magnitude and $\hat{u}_1(\theta_1, \phi_1)$ or $\hat{u}_1$ the direction of the velocity vector. θ_1 and ϕ_1 are the polar and azimuthal angle, respectively. The direction of the first particle before collision is then given by,

$$\hat{u}_1 = \begin{pmatrix} \sin\theta_1\cos\phi_1 \\ \sin\theta_1\sin\phi_1 \\ \cos\theta_1 \end{pmatrix}$$

such that the change in direction may modelled by adjusting the polar and azimuthal angles,

$$\theta'_1 = \theta_1 + \Delta\theta_1 \qquad \phi'_1 = \phi_1 + \Delta\phi_1,$$

in which θ'_1 and ϕ'_1 are the angles after the collision. $\Delta\theta_1$ and $\Delta\phi_1$ are two free parameters of this problem. In fact, these are the only two such free parameters. In total there are six parameters to be determined; all three velocity components of each of the two particles. Conservation of momentum and kinetic energy fix four of these, which leaves two.

To slightly limit this freedom of choice and make this problem more relevant to modelling the collision (and possible fragmentation) of two satellites, we introduce a maximum scattering angle α. This, in combination with properly chosen $\Delta\theta_1$ and $\Delta\phi_1$, gives the adjusted direction of the first particle, $\hat{u}'_1(\theta'_1, \phi'_1) = \hat{u}'_1$.

To prevent making any other assumptions about the direction of the particle's velocity after the collision, $\Delta\theta_1$ and $\Delta\phi_1$ should be chosen in such a way that the resulting $\hat{v}'_1$ is randomly picked from a spherical cap centered on $\hat{u}_1$, see figure 10.

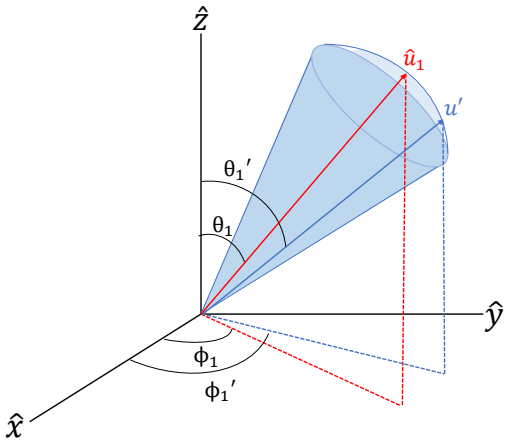


Figure 10: Depiction of how the adjusted direction $\hat{u}_1'$ may differ from the original direction $\hat{u}_1$. The blue shaded cone with spherical cap includes all the possibilities for $\hat{u}_1'$. The particular $\hat{u}_1'$ in the figure is one instance of these possibilities. The radius of the cone is related to the maximum scattering angle α.

Now, let $\Delta\xi = \sin\Delta\phi_1$ and $\Delta\eta = \sin\Delta\theta_1$, then we must have $\Delta\xi^2 + \Delta\eta^2 \leqslant \sin^2\alpha$, as can be seen from figure 11. In addition, let $\rho = \sqrt{\Delta\xi^2 + \Delta\eta^2}$ and $\nu = \arctan\Delta\eta/\Delta\xi$. In order to obtain an uniform distribution of vectors on the spherical cap we take,

$$\begin{aligned}\rho &\sim \sqrt{U[0, \sin^2\alpha]} \\ \nu &\sim U[0, 2\pi),\end{aligned}$$

in which $\sin^2\alpha$ represents the square of the (maximum) radius of the cone. Consequently the adjustment angles are determined as,

$$\begin{aligned}\Delta\phi_1 &= \pm\arcsin\left[\rho\cos\nu\right] \\ \Delta\theta_1 &= \arcsin\left[\sin\Delta\phi\tan(\nu)\right],\end{aligned}$$

where the sign of $\Delta\phi_1$ is chosen randomly.

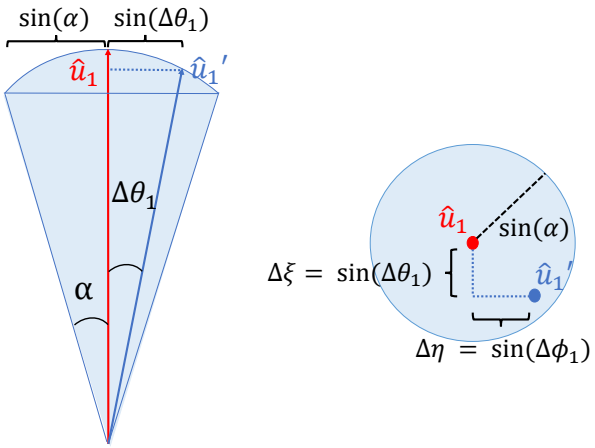


Figure 11: *left*: section of the cone in the polar plane. The relation of $\Delta\theta_1$ to its projected distance to the center of the cone is depicted here. A similar relationship holds in the azimuthal plane for $\Delta\phi_1$ and its projected distance to the center of the cone. *right*: a top down view of the cone showing that coordinates $\Delta\xi$ and $\Delta\eta$ must lie on a disk.

Finally, combining the magnitude and direction of the first scattered particle gives

$$\vec{v}'_1 = |\vec{u}'_1|\hat{u}'_1 + \vec{V}_{com}$$

$\vec{v}'_2$ is then given by equation (4.1).

4.2. NASA's SBM

In a fragmentation or breakup collision of two bodies the relative kinetic energy defined in equation (4.6) is not fully recovered in the sum of the kinetic energy of the fragments. Some of this kinetic energy is used in deformation or is dissipated as heat, as satellites are broken apart and deformed. In addition, the number of fragments, their mass, speed and direction depend on, among other things, the impact velocity (both magnitude and angle), mass, shape and composition of the satellites. This all greatly complicates the exact modelling of a breakup. Especially in the case of the Kessler syndrome; where any exact model aiming to describe it is then required to include the exact shape of each of the roughly 30,000 satellites in LEO and any fragments resulting from a collision.

However, NASA's Standard Breakup Model (SBM) is able to provide all the necessary information to describe a breakup, as stated at the beginning of this section [Joh+01]. The caveat lies in that the SBM is based on several probability distributions, which in turn are based on numerous observations of real breakup events and ground tests performed by NASA. Hence the SBM introduces a statistical component to the full model of this report.

The model requires only the masses of the satellites, which are discerned as the mass of the lighter projectile m_p and the heavier target satellite m_t, and the impact velocity $\vec{u} = \vec{v}_t - \vec{v}_p$. Collisions are then subdivided into two categories; non-catastrophic and catastrophic. These are distinguished as

$$e_{imp} = \frac{\frac{1}{2}m_p[\mathrm{kg}]|\vec{u}|^2[\frac{\mathrm{m}^2}{\mathrm{s}^2}]}{m_t[\mathrm{g}]} > 40\ [\mathrm{J/g}], \tag{4.7}$$

where e_{imp} is the kinetic energy of the projectile divided by the mass of the target. The greater-than sign holds for catastrophic collisions. For non-catastrophic collisions it is a less-than-or-equal-to sign. The fragmented mass of each breakup is given either as

$$m_{\text{frag}} = m_p \left(\frac{|\vec{u}|}{1\left[\frac{\text{km}}{\text{s}}\right]} \right)^2, \tag{4.8}$$

in the case of non-catastrophic collisions or as

$$m_{\text{frag}} = m_t + m_p, \tag{4.9}$$

for catastrophic collisions. This fragmented mass fixes the number of fragments of a certain size and larger.

$$N_{\geqslant L_c} = 0.1 \left(\frac{m_{\text{frag}}}{1[\text{kg}]} \right)^{0.75} \left(\frac{L_c}{1[\text{m}]} \right)^{-1.71}, \tag{4.10}$$

in which L_c is the characteristic length of a fragment in meters. The distribution of L_c is given by

$$L_c = \left((r_{\text{min}}^{-1.71} - r_{\text{max}}^{-1.71})x + r_{\text{max}}^{-1.71} \right)^{\frac{1}{-1.71}} \quad \text{where} \quad x \sim \mathcal{U}[0,1]. \tag{4.11}$$

Here, r_{min} and r_{max} are the minimum and maximum fragment radii. It remains to be determined what the mass and velocity of each individual fragment are. Both of these are given by (superposed) normal distributions. Firstly, the area-to-mass ratio of a fragment A/M for $L_c \geqslant 0.11$ is described by

$$D_{A/M}^{L_c \geqslant 0.11} = \alpha(\lambda_c)\text{N}(\mu_1(\lambda_c), \sigma_1(\lambda_c), \chi) + (1 - \alpha(\lambda_c))\text{N}(\mu_2(\lambda_c), \sigma_2(\lambda_c), \chi), \tag{4.12}$$

and for $L_c < 0.11$

$$D_{A/M}^{L_c < 0.11} = \text{N}(\mu_3(\lambda_c), \sigma_3(\lambda_c), \chi) \tag{4.13}$$

in which $\lambda_c = \log_{10}(L_c)$, N is the normal distribution with independent variable

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

α, μ_i and σ_i for $i = 1, 2, 3$ are all functions of λ_c, which may be found in the appendix equations (C2) and (C1). After obtaining the area-to-mass ratio form either equation (4.13) or (4.12) the mass of the fragment is obtained using the following relation between (cross-sectional) area A and L_c

$$A = 0.556945 L_c^{2.0047077} \tag{4.14}$$

Secondly, the difference in speed of the fragments as compared to the speed of their parent satellites, the Δv distribution, is

$$D_{\Delta v} = \text{N}(\mu(\chi), \sigma, \nu). \tag{4.15}$$

Note that in this distribution the independent variable is $\nu = \log_{10}(\Delta v)$ and that μ and σ are now functions of χ (appendix equation (C3)).

The SBM does not specify the direction of the resulting fragments. This can be done using the same method as in Section 4.1 to randomly generate vectors on a spherical cap and assigning these to all fragments. In doing so, conservation of momentum must be guaranteed. A relatively easy way to achieve this, is to apply the SBM to half the number of generated fragments;

assign mass, speed and direction to this first half and, finally, create an identical second half with reversed directions. Then, each pair of identical fragments will contribute nothing to the total momentum sum. A random factor in the fragments' point of origin should be added, as it is highly unrealistic for a breakup between two satellites to be symmetric. This does not affect momentum conservation, but does introduce a small error in the angular momentum of the system. Over multiple fragments or collisions, however, these errors can be expected to average out to zero.

A closer inspection of equation 4.10 reveals that for small enough fragmentation masses m_{frag} the number of fragments may drop below 1. This is of course, physically impossible. We therefore invoke a minimum number of fragments N_{min}, which implies a minimum of amount of fragmented mass in any collision

$$m_{\text{frag,min}} = \left(\frac{N_{\text{min}}}{0.1 L_{c,\text{min}}^{-1.71}} \right)^{1.25}, \tag{4.16}$$

where $L_{\text{c,min}}$ is the minimum characteristic length of the fragments that are generated by the SBM. If the fragmented mass m_{frag} calculated using either Equation (4.9) or (4.8) is lower than $m_{\text{frag.min}}$, then then a scattering is performed instead of a fragmentation[15]. N_{min} and $L_{\text{c,min}}$ thus become two parameters in the model, which together define the border between fragmentation and scattering collision. The minimum amount fragmented mass is

$$m_{\text{frag,min}} = 2.98 \times 10^{-1} \text{ kg for } N_{\text{min}} = 4 \text{ and } L_{\text{c,min}} = 0.05 \text{ m},$$
$$m_{\text{frag,min}} = 5.34 \times 10^{-3} \text{ kg for } N_{\text{min}} = 4 \text{ and } L_{\text{c,min}} = 0.01 \text{ m}.$$

Note that for a lower L_c, the minimum mass is smaller as well. This means that the inclusion of smaller fragments into the model relaxes the minimum mass constraint and allows for more fragmentation collisions to take place. This topic is further discussed in Section 5.

Algorithm 9 defines the procedure SBM(...), which is used in Algorithms 4 and 8 to generate fragments.

[15] See line 18 in Algorithm 9

Algorithm 9 Collision model

procedure SBM($\vec{r}_i, \vec{r}_j, \vec{v}_i, \vec{v}_j, m_i, m_j, s_i, s_j, t_{\text{ij}}, \Delta t$)
  $(\vec{r}_i, \vec{r}_j), (\vec{v}_i, \vec{v}_j) =$ VELOCITYVERLET$((\vec{r}_i, \vec{r}_j), (\vec{v}_i, \vec{v}_j), t_{\text{ij}})$ ▷ propagate to collision point
  $t_{\text{rest}} = \Delta t - t_{\text{ij}}$ ▷ time to next time step, in case of discrete algorithm
  $\vec{u} = \vec{r}_j - \vec{r}_i$
  $u = |\vec{u}|$
  $\vec{V}_{com} = \frac{m_i\vec{v}_i + m_j\vec{v}_j}{m_i + m_j}$
  $\vec{u}_i = \vec{v}_i - \vec{V}_{\text{com}}$
  $\vec{u}_j = \vec{v}_j - \vec{V}_{\text{com}}$
  $m_p = \min\{m_i, m_j\}$ ▷ projectile mass
  $m_t = \max\{m_i, m_j\}$ ▷ target mass
  $\vec{r}_p$ is the projectile position corresponding to m_p
  $\vec{r}_t$ is the target position corresponding to m_t
  $\vec{u}_p$ is the relative projectile velocity corresponding to m_p
  $\vec{u}_t$ is the relative target velocity corresponding to m_t
  determine the type c_{type} of collision using Equation (4.7)
  m_{frag} from either Equation (4.9) or (4.8)
  calculate $m_{\text{frag,min}}$ using Equation (4.16)
  if $m_{\text{frag}} < m_{\text{frag,min}}$ **do** ▷ fragmented mass is too small for a fragmentation
    use the methods described in Section 4.1 to obtain $\vec{v'}_i$ and $\vec{v'}_j$
    $(\vec{r'}_i, \vec{r'}_j), (\vec{v'}_i, \vec{v'}_j) =$ VELOCITYVERLET$((\vec{r}_i, \vec{r}_j), (\vec{v'}_i, \vec{v'}_j), t_{\text{rest}})$ ▷ propagate to next time step
    return $\vec{r'}_i, \vec{r'}_j, \vec{v'}_i, \vec{v'}_j, m_i, m_j, s_i, s_j$
  end if
  determine $N_{\geqslant L_c}$ from Equation (4.10)
  $N_{\text{frags}} = N_{\geqslant L_c}/2$ ▷ the other half is added later, round off to integer value if necessary
  $r_{\text{min}} = L_{c,\text{min}}$
  $r_{\text{max}} = \max\{s_i, s_j, 1\}$ ▷ NASA SBM is only valid for fragment sizes of up to 1 m
  sample N_{frags} number of L_c values from the distribution given in (4.11)
  calculate $\lambda_c = \log L_c$ for all values of L_c
  sample the corresponding A/M values using either distribution (4.13) or (4.12) and λ_c values
  obtain the mass of each fragment M_{frag} using Equation (4.14)
  if $c_{\text{type}} =$ catastrophic **do**
    ensure mass conservation by dividing M_{frags} by $\frac{\sum M_{\text{frags}}}{m_{\text{frag}}}$
  elif $c_{\text{type}} =$ non-catastrophic **do**
    ensure mass conservation by creating two fragments that have a similar mass $m_{\text{p,frag}}$ and $m_{\text{t,frag}}$ and size $L_{\text{c,p,frag}}$ and $L_{c,\text{t,frag}}$ as the projectile and target
  end elif
  calculate $\chi = \log A/M$ for all values of A/M
  sample the corresponding Δv values using distribution (4.15) and χ
  generate N_{frags} scattered fragment directions using $\vec{u}_p$ and the methods described in Section 4.1
  generate N_{frags} scattered fragment speeds by randomly adding Δv to or subtracting Δv from u.

multiply direction vectors and speeds to obtain the first half of relative fragment velocities $U_{\text{frag,1}}$

generate the other half of relative fragment velocities as $U_{\text{frag,2}} = -U_{\text{frag,1}}$

if $c_{\text{type}} =$ non-catastrophic **do**

calculate $A_{p,frag}$ and $A_{t,frag}$ from Equation (4.14)

calculate $\chi_{\text{p,frag}} = \log A_{p,frag}/m_{\text{p,frag}}$ and $\Xi_{\text{t,frag}} = \log A_{t,frag}/m_{\text{t,frag}}$

sample $\Delta v_{\text{p,frag}}$ and $\Delta v_{\text{t,frag}}$ from distribution (4.15)

generate $\vec{v}_{\text{p,frag}}$ and $\vec{v}_{\text{t,frag}}$ using the original directions $\vec{u}_i$ and $\vec{u}_j$ and alter magnitudes by adding or subtracting $\Delta v_{\text{p,frag}}$ and $\Delta v_{\text{t,frag}}$

append $m_{\text{p,frag}}$, $m_{\text{t,frag}}$, $L_{\text{c,p,frag}}$, $L_{c,\text{t,frag}}$, $\vec{v}_{\text{p,frag}}$ and $\vec{v}_{\text{t,frag}}$ to the fragment lists M_{frag}, L_c, $U_{\text{frag,1}}$ and $U_{\text{frag,2}}$ ▷ being careful to add the velocities in the proper direction

end if

$V_{\text{frags,1}} = U_{\text{frags,1}} + \vec{V}_{\text{com}}$ ▷ that is, add $\vec{V}_{\text{com}}$ for each $\vec{u}_{\text{frag,1}} \in U_{\text{frags,1}}$

$V_{\text{frags,2}} = U_{\text{frags,2}} + \vec{V}_{\text{com}}$

$E_{\text{frags}} = \frac{1}{2} M_{\text{frag}} \sum \left(V^2_{\text{frags,1}} + V^2_{\text{frags,2}} \right)$ ▷ shorthand for total kinetic energy of fragments

$E_{\text{initial}} = \frac{m_i |\vec{v}_i|^2}{2} + \frac{m_j |\vec{v}_j|^2}{2}$

$\epsilon = \frac{E_{\text{frags}}}{E_{\text{initial}}}$

if $\epsilon > 1$ **do** ▷ ensure kinetic energy conservation

$V_{\text{frags,1}} = \frac{V_{\text{frags,1}}}{\sqrt{\epsilon}}$

$V_{\text{frags,2}} = \frac{V_{\text{frags,2}}}{\sqrt{\epsilon}}$

end if

$\text{VELOCITYVERLET}(R_{\text{frags,1}}, V_{\text{frags,1}}, t_{\text{ij}})$ ▷ propogate to collision point

$R_{\text{frags,1}} = \{\vec{r}_p, \vec{r}_p, \ldots, \vec{r}_p\}$ ▷ with length N_{frags}

$R_{\text{frags,2}} = \{\vec{r}_t, \vec{r}_t, \ldots, \vec{r}_t\}$ ▷ idem

$R_{\text{random,1}} = \{\vec{r}_{\text{random,1}} \,|\, \vec{r}_{\text{random,1}}$ is random vector on the order of the radius of satellites$\}$

$R_{\text{random,2}} =$ idem

$R_{\text{frags,1}} = R_{\text{frags,1}} + R_{\text{random,1}}$ ▷ that is, add a random vector $\vec{r}_{\text{random}}$ to each fragment position vector $\vec{r}_{frag,1} \in R_{\text{frags,1}}$

$R_{\text{frags,2}} = R_{\text{frags,2}} + R_{\text{random,2}}$

$R_{\text{frags,1}}, V_{\text{frags,1}} = \text{VELOCITYVERLET}(R_{\text{frags,1}}, V_{\text{frags,1}}, t_{\text{rest}})$ ▷ propagate to next time step

$R_{\text{frags,2}}, V_{\text{frags,2}} = \text{VELOCITYVERLET}(R_{\text{frags,2}}, V_{\text{frags,2}}, t_{\text{rest}})$

append $R_{\text{frags,1}}$ to $R_{\text{frags,2}}$ to get R_{frags} , $V_{\text{frags,1}}$ to $V_{\text{frags,2}}$ to get v_{frags}, M_{frag} to itself to get M_{frags}, L_c to itself to get $L_{\text{c,frags}}$ ▷ we have identical halves of particles

return R_{frags}, V_{frags}, M_{frags}, $L_{\text{c,frags}}$

end procedure

4.3. Analysis of SBM

In order to investigate the behaviour of Algorithm 9, we force a collisions between a prograde satellite $i = 1$ and retrograde satellite $j = 2$. We have

$$a_1 = a_2, \quad e_1 = e_2, \quad I_1 = 60^\circ \text{ and } I_2 = 120^\circ,$$
$$\Omega_1 = 0^\circ \text{ and } \Omega_2 = 10^\circ, \quad M_{a,1} = M_{a,2} = \omega_1 = \omega_2 = 0.$$

These satellites have an impact velocity $|\vec{u}| = 7.976\text{km}\,\text{s}^{-1}$. Using Equation (4.7) we derive that the ratio of projectile mass to target mass above which the collision is considered catastrophic is

$$\mu_{\text{critical}} = 1.26 \times 10^{-6}. \tag{4.17}$$

We now vary the mass $m_{1,2}$ and radius of the satellites $s_{1,2}$ and plot the collision fragment characteristic length, mass, area-to-mass and $\Delta - v$ distributions. The results for several of these collision scenarios are shown in Figures 12, 13 and 14.

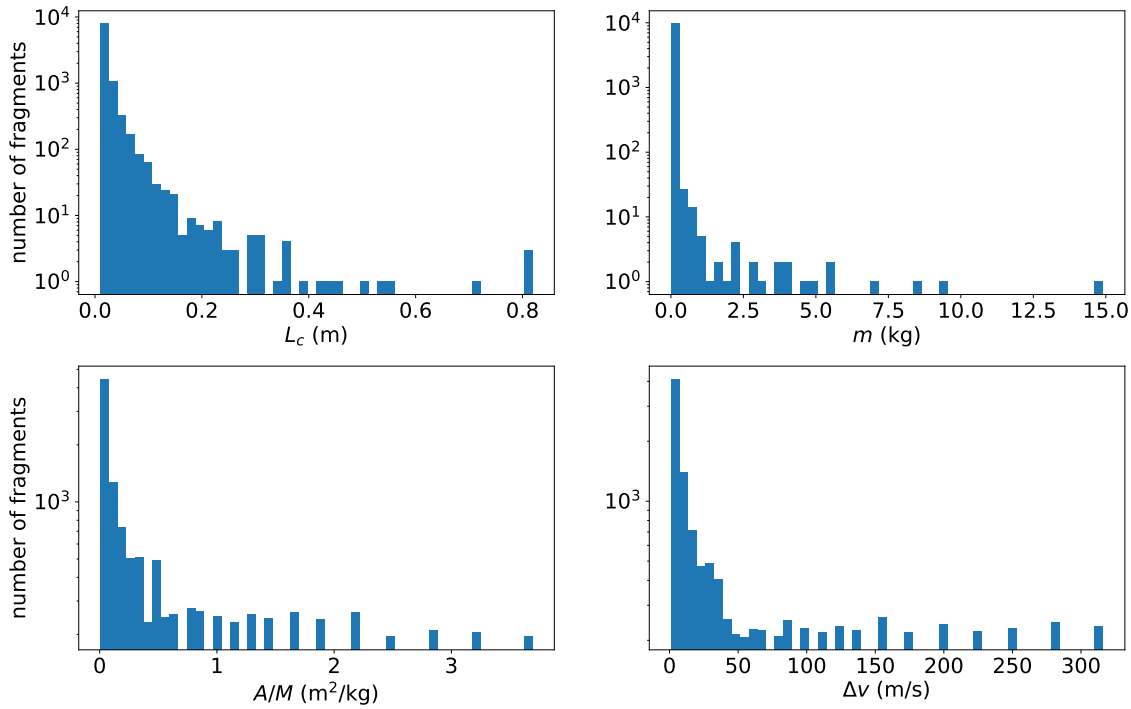


Figure 12: Fragment distributions for a catastrophic collision between two large satellites with $m_1 = m_2 = 200$ kg and $s_1 = s_2 = 2$ m as a function of L_c, m, A/M and Δv.

From Figure 12 we see that for a (catastrophic) collision between similarly large and massive satellites fragments form on the order of a few centimeters up to about a meter. This is because distribution (4.12) is limited to generating fragments of up to a meter. Additionally, most of the fragments are smaller than 10cm, which reflects the power law given in Equation (4.10).

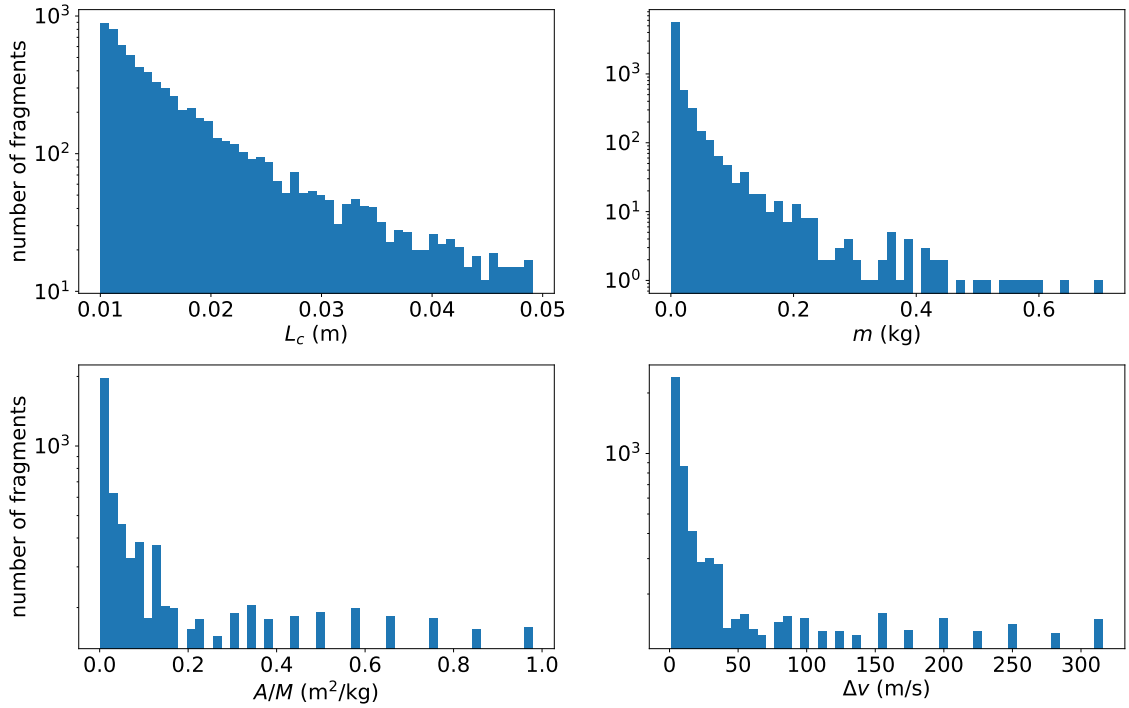


Figure 13: Fragment distributions for a catastrophic collision between a large $m_1 = 200$ kg, $s_1 = 2$ m and small satellite $m_2 = 2$ kg, $s_2 = 0.05$ m versus L_c, m, A/M and Δv as in Figure 12.

Figure 13 indicates another catastrophic collision. In this case satellite 2 is reduced in size and mass. We have

$$\frac{m_p}{m_t} = 0.01 > \mu_{\text{critical}}.$$

The reduced size of s_1 has as a consequence that only fragments with characteristic lengths smaller than $s_2 = 0.05$m are generated. This constraint on fragment size stems from line 26 in Algorithm 9 and is necessary in order to maintain the symmetry of the fragmentation. In particular, one half of the generated fragments velocities are scattered around the direction of the target velocity $\vec{v}_t$ and other half around the projectile velocity $\vec{v}_p$. The former group of fragments may thus be considered as originating from the target satellite and the latter from the projectile. The fragment size must be limited to the size of the smallest satellite, because both halves of fragments are identical in size and fragments can not be larger than the satellites they originate from. Additionally, the small mass of satellite 2 results in less fragments compared to the previous scenario in Figure 13.

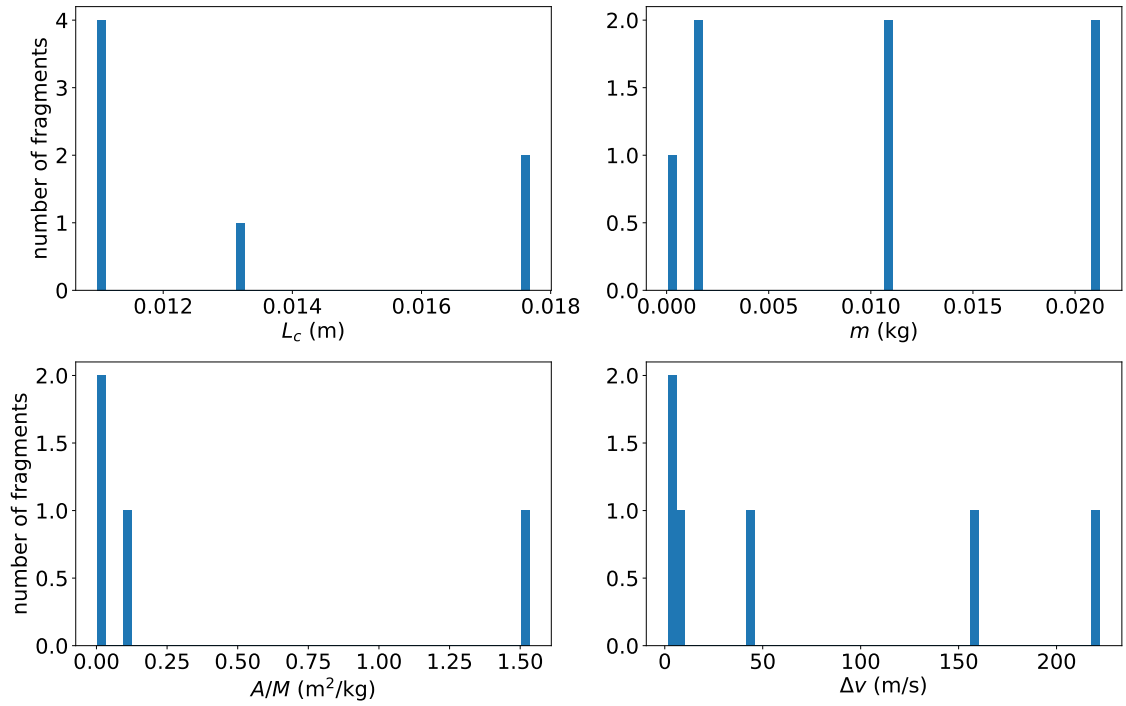


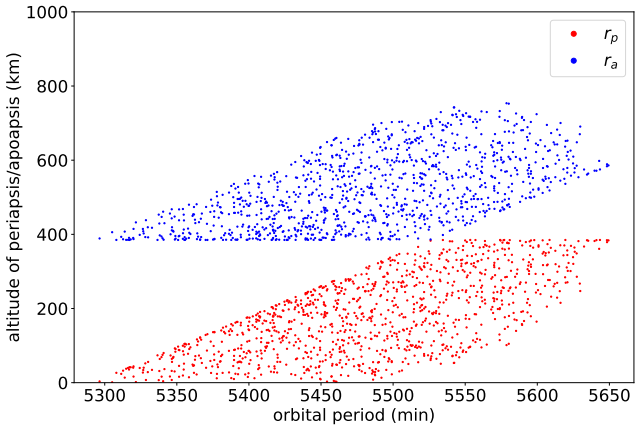
Figure 14: Fragment distributions for a non-catastrophic collision between a large $m_1 = 200$ kg, $s_1 = 2$ m and small satellite $m_2 = 2 \times 10^{-4}$ kg, $s_2 = 0.05$ m versus L_c, m, A/M and Δv as in Figure 12.

In contrast to Figures 12 and 13, Figure 14 shows the output of the SBM for a non-catastrophic collision. In this case we have

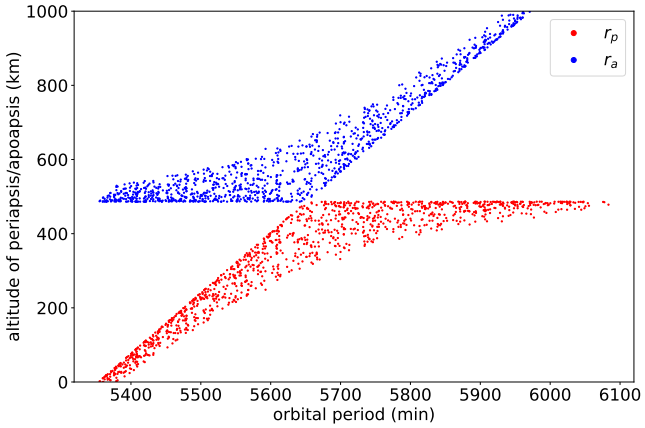
$$\frac{m_p}{m_t} = 1 \times 10^{-6} < \mu_{\text{critical}}.$$

Several small and light particles are generated, which resemble the projectile and the fragments originating from the target. One large fragment with a size and mass similar to the original target is generated as well. The velocities of the fragments after a non-catastrophic collision are therefore not symmetric for target and projectile, which stands in contrast to the case of catastrophic collisions.

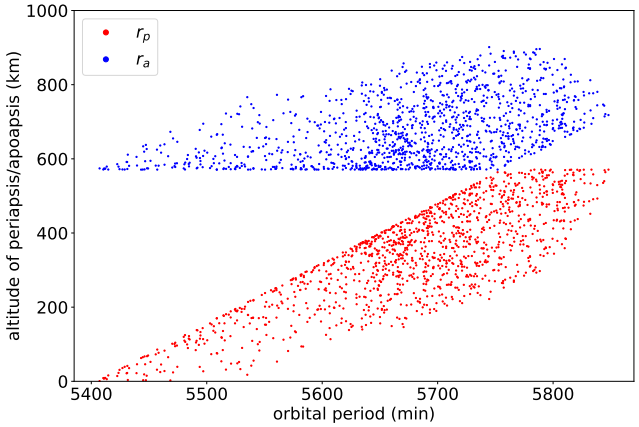
To observe the effect of the random scattering of the fragment velocity in line 36 of Algorithm 9 and the random addition or subtraction of Δv in line 37, Figure 15 shows Gabbard diagrams for collisions at several altitudes.



(a)



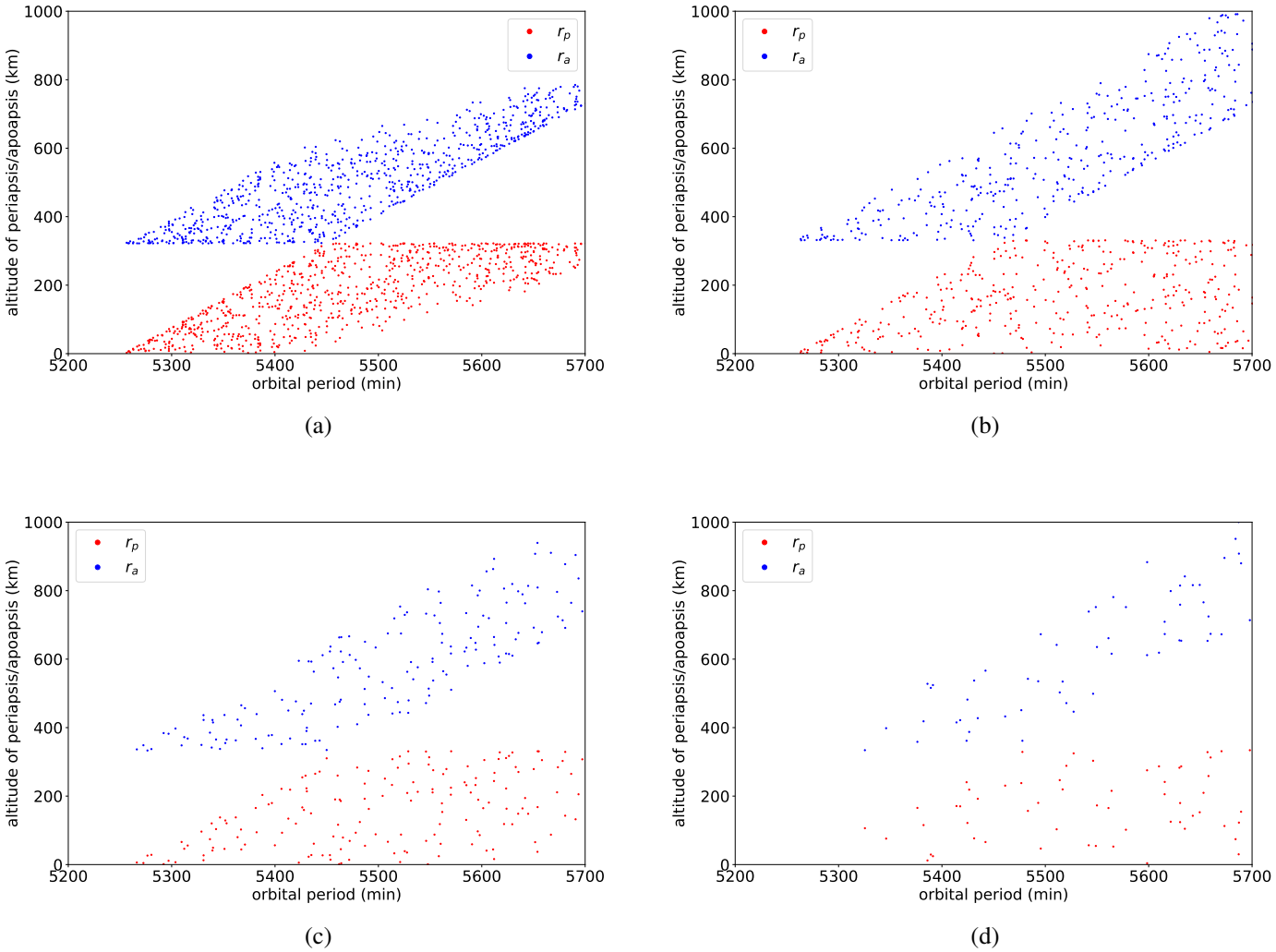
(b)



(c)

Figure 15: Gabbard diagrams of catastrophic collisions with $\alpha = 3^\circ$ and $L_{\mathrm{c,min}} = 0.05$ m at altitudes **(a)** 381km, **(b)** 485km and **(c)** 557km. The total number of fragments in each case was about 1500, of which 406, 275 and 212 collided with the Earth, respectively. The spread of fragments after all three collisions ranges over altitudes from 0 up to 1000km.

Moreover, Figure 16 shows that the maximum scattering angle is a major factor in determining how many fragment stay in orbit following a collision. This shows that the entire of this paper model is very sensitive to the parameters in the SBM((...)).



(a) (b)

(c) (d)

Figure 16: Gabbard diagrams of the same catastrophic collision with $L_{\text{c,min}} = 0.05$ m, **(a)** $\alpha = 3^\circ$, **(b)** $\alpha = 6^\circ$, **(c)** $\alpha = 10^\circ$ and **(d)** $\alpha = 15^\circ$. The total number of fragments is approximately 1500, of which 616, 977, 1121 and 1203 collided with Earth, respectively.

5. Kessler syndrome

Using the discrete and continuous algorithms to detect collisions and the SBM to perform them, we can simulate the evolution of a system of satellites. Firstly, let us apply the models to the homogeneous system introduced in Section 3.3. In the same section we saw that for low particle numbers and small satellite radii, the continuous algorithm performs best. Figures 17 and 18 shows the evolution of the number of satellites over time as predicted by Algorithm 4, from an initial configuration of $N = 100$ and $N = 500$ satellites, chosen independently from a homogeneous distribution in the spherical shell.

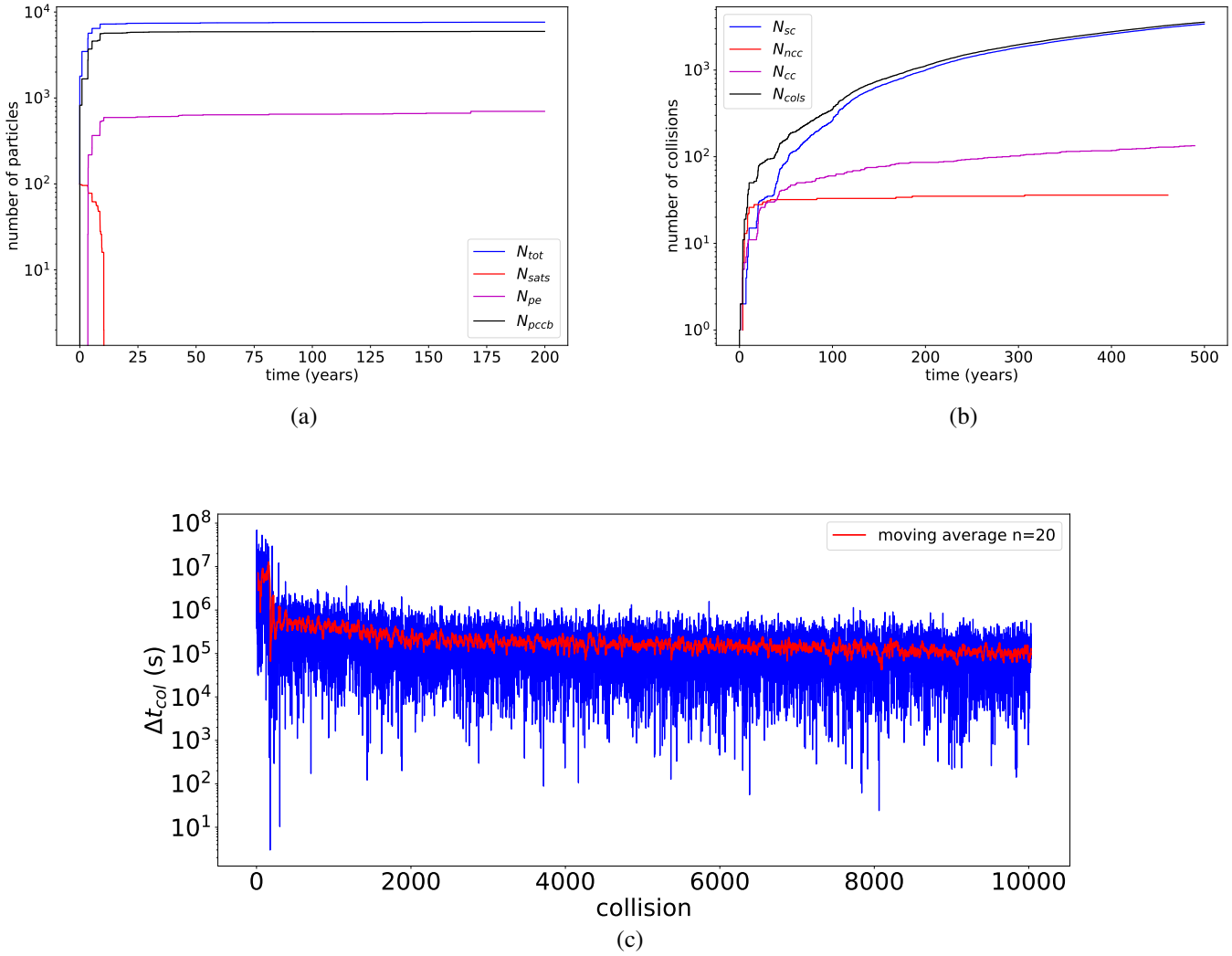


Figure 17: 500 year evolution of $N = 100$ homogeneous satellites with $s = 5$ m, $r_{\text{inner}} = 300$ km, $h = 50$ km $L_{\text{c,min}} = 0.05$ m and $\alpha = 3$. **a)** shows the total number of particles N_{tot} in blue, the number of remaining original satellites N_{sats} in red, the number of escaped particles N_{pe} in magenta and the number of particles collided with the central body N_{pccb} in black durinh the first 200 years. **b)** plots the total number of collisions N_{cols}, which is composed of the number of scatterings N_{sc} in blue, the number of non- and catastrophic collisions N_{ncc} and N_{cc} in red and magenta. **c)** is a plot of the time between successive collisions in blue with a 20-term moving average in red.

We see from Figure 17 that an initially small number of random medium sized satellites evolves into a large number of debris particles in just a few decades. Moreover, by the end of the first decade all the original satellites have collided and are either fully or partly fragmented. This is followed by a period where the total particle number does not increase significantly, but the number of collisions does.

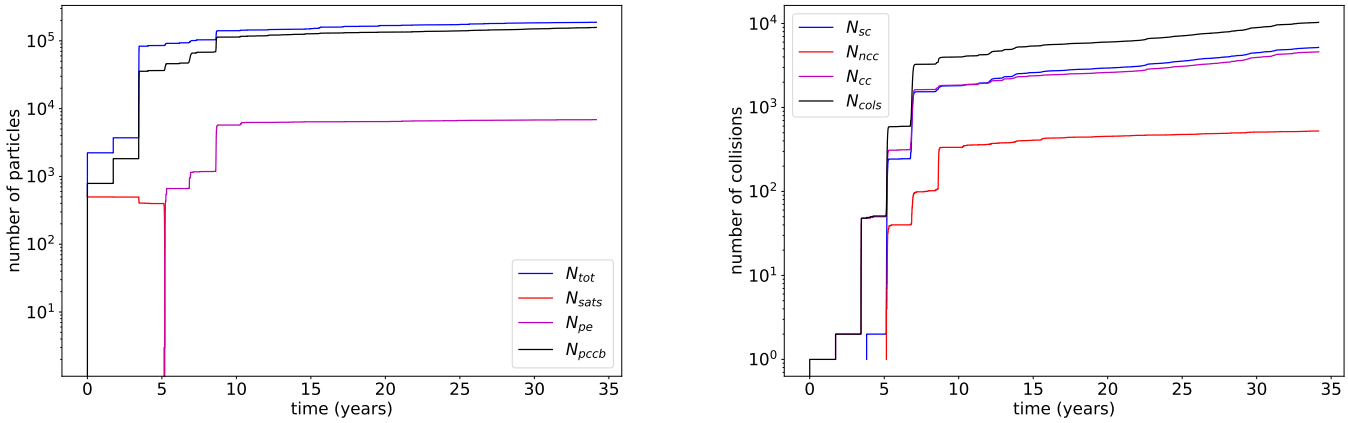
The composition of the collisions in 17.b) changes in the same period as well. From being mostly catastrophic collisions to all scatterings. This is because at this stage there is only debris present in the system, which is either too small or too light for Algorithm 9 to classify a collision as catastrophic. As is discussed in Section 4.3, most of the generated debris' size is

equal to or slightly larger than the minimum characteristic length $L_{\text{c,min}}$, which means that any collision between two debris particles can not produce more, smaller particles. Moreover, the low mass of the fragments causes $m_{\text{frag}} < m_{\text{frag,min}}$ for most collisions.

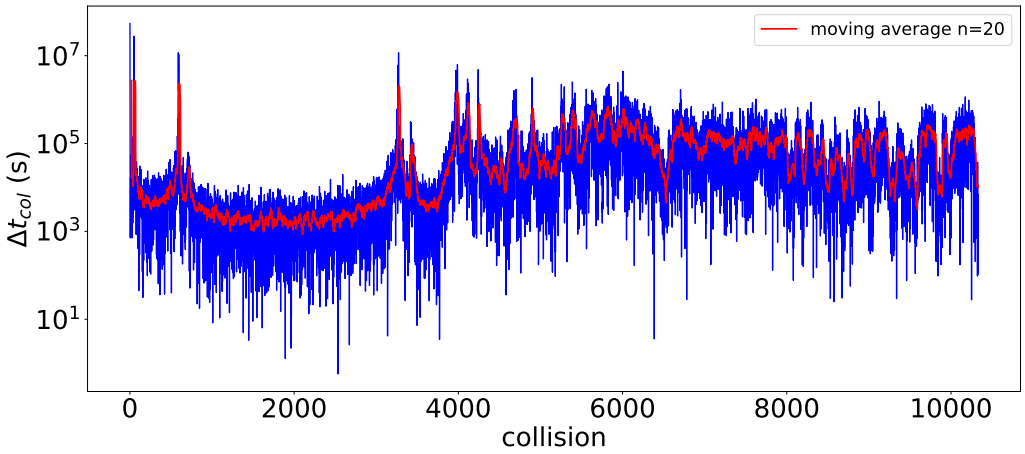
Lastly, the time between Δt_{col} collisions in Figure 18.c) is high at first, due to initially small number of satellites. However, after the fragmentation of these original satellites, Δt_{col} decreases two orders of magnitude. This means that even though the debris particles have small radii, the number of debris is large enough so that the time between collisions is decreased. Taking Equation (3.29) into consideration, we see that this must mean that the product of the number of satellites and the satellite radius Ns, becomes larger as time progresses. Due to the numerous fragments present in the system at later times, the radii of the satellites is no longer constant. Therefore we take the average radius $\bar{s}$, which gives

$$(Ns)_{\text{initial}} = 500 \text{ m} \quad (N\bar{s})_{\text{final}} = \sum_{i=1}^{N_{\text{final}}} s_i \approx 572 \text{ m},$$

where N_{final} is the number of particles at the end of the simulation.



(a) (b)



(c)

Figure 18: Same as Figure 17, but now for approximately 35 years of $N = 500$ homogeneously distributed satellites.

In Figure 18.a) we see that if the number of satellites in the beginning is increased by a factor of 5 with respect to the example of Figure 18, the number of resulting debris particles is about 10 to 20 times larger. Also, the number of original satellites present in the system vanishes more quickly, within 5 years to be precise. Both of these differences can be attributed to the increased likelihood that two original satellites collide with each other, which generates more fragments and causes these satellites to decrease in number more quickly. This also happened in the simulation for $N = 100$, but to a lesser extent; most of the original satellites collided with a fragment resulting from an earlier collision.

In comparison to the case of $N = 100$, we see in 18.b) that within a tenth of the simulated time the same total number of collisions occur, $N_{\text{cols}} \approx 10000$. Also, these collisions are mostly (non-)catastrophic, which means fragments are still being generated from collisions of larger sized debris. Frankly, only the first 35 years of this system could be simulated within the time frame of the writing of this paper. The reason for this was the high total particle number at the

end of the simulation, for which the continuous algorithm is not well suited in its current state. It would be interesting to simulate at least a hundred more years to investigate whether or not the average collision time stabilises like it did for $N = 100$ satellites.

In both Figures 17.a) and 18.a) one can also observe that in the first few year of the simulations a reasonable proportion of all the fragments generated in collisions collide with the Earth (central body) N_{pccb} and a smaller fraction escape from the system entirely N_{pe}. After about a decade these numbers remain approximately constant. In particular,

$$\left(\frac{N_{\text{pe}}}{N_{\text{final}}}\right)_{N=100} = 9.33\% \quad \text{and} \quad \left(\frac{N_{\text{pccb}}}{N_{\text{final}}}\right)_{N=100} = 77.35\%,$$
$$\left(\frac{N_{\text{pe}}}{N_{\text{final}}}\right)_{N=500} = 3.66\% \quad \text{and} \quad \left(\frac{N_{\text{pccb}}}{N_{\text{final}}}\right)_{N=500} = 83.92\%.$$

This can again be attributed to the decrease in large fragments and, thereby, the decrease in the number of generated fragments in any collision. In the beginning the original medium sized satellites collide forming large numbers of fragments, of which many will collide with the central body or escape.

Now that we understand how a system with a small number of randomly distributed satellites can evolve into one with many fragments, we investigate how a system of ordered satellites in combination with a high number of debris-like particles behaves over time. The motivation for this lies in the definition of the Kessler syndrome. That is, we want to see if usually stable configuration of satellites, can become destabilised due to a collision cascade. To generate a system of N ordered, non-colliding satellites we will divide the satellites over closely spaced non-intersecting layers. Now let

$$N_{\text{layer}} = \frac{N}{\#\text{layers}},$$

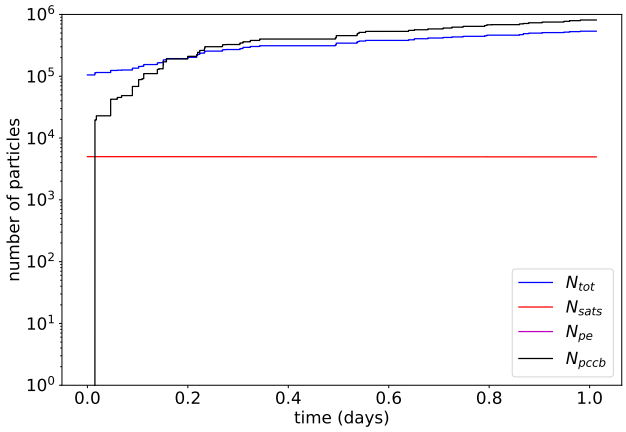
be the number of satellites in any layer, then choose for each satellite i

$$\Omega_i = i\frac{360^\circ}{N},\ \omega_i = 0,\ M_{a,i} = \gamma_1 + \frac{i \mod N_{\text{layer}}}{N_{\text{layer}}}(\gamma_2 - \gamma_1) \quad \text{and} \quad I_i = 60^\circ,$$

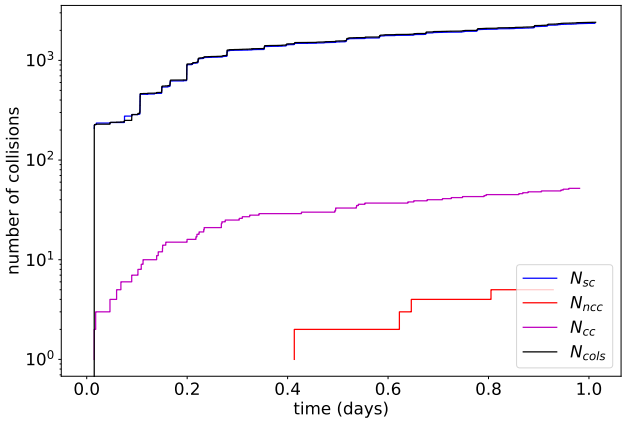
where γ_1 and γ_2 are the minimum and maximum mean anomaly between which the satellites in any given layer are situated. Also, the mean anomaly is now automatically zero in the periapsis. Furthermore, the semi-major axes and eccentricities are given as

$$a_i = h_1 + \frac{i \mod N_{\text{layer}}}{N_{\text{layer}}}(h_2 - h_1), \quad e_i = 1.00 \times 10^{-5}. \tag{5.1}$$

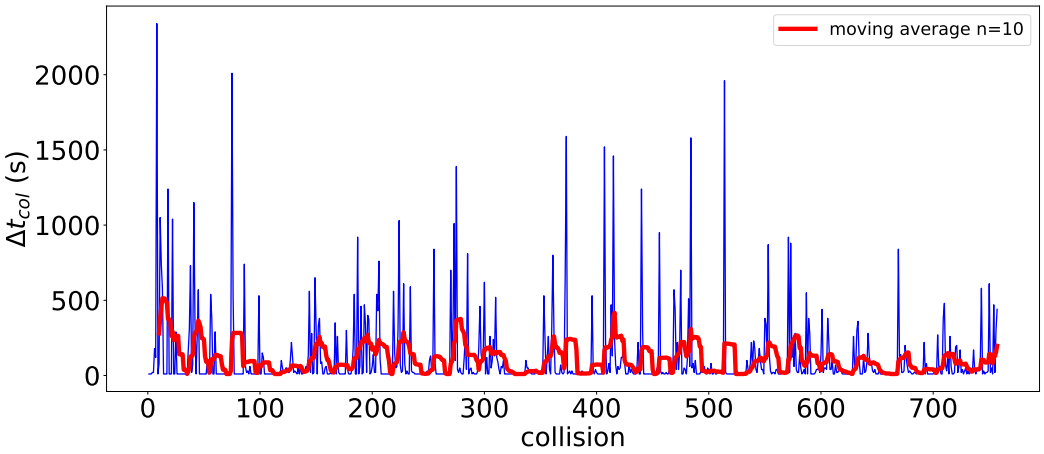
This configuration ensures that the layers are spaced evenly over a minimum and maximum height h_1 and h_2 and remain separated due to the low eccentricity of the satellites. Figure 19 shows the evolution of this ordered system of satellites combined with an instance of the homogeneous distribution of satellites.



(a)



(b)



(c)

Figure 19: One day evolution of a system of large ordered satellites within a homogeneous distribution of relatively small debris-like satellites. The ordered distribution consists of $N_{\text{ord}} = 5000$ satellites with radius $s_{\text{ord}} = 20$ m of mass $m_{\text{ord}} = 500$ kg. The layers are spaced within the spherical shell starting at an altitude of $h_1 = 100$ km up to $h_2 = 130$ km. The $N_{\text{hom}} = 10^5$ homogeneously distributed satellites have a radius $s_{\text{hom}} = 1$ m and a mass of $m_{\text{hom}} = 20$ kg. These were situated in the same shell with $r_{\text{inner}} = R_{\oplus} + 100$ km and $h = 30$ km. Moreover, the minimum characteristic length is lowered to $L_{\text{c,min}} = 0.01$ m to incorporate the effect of smaller collisions. The plots **a)**, **b)** and **c)** display the same information as Figures 17 and 18.

The discrete algorithm was used to generate the results in Figure 19, as this system involves many particles with relatively large radii situated within a spherical shell of a small height. Within a day 5 ordered satellites in this system have collided. These satellites must have collided with one of the debris-like particles from the homogeneous distribution, since the ordered satellites do not collide with each other. $N_{\text{cols}} = 758$ collisions generated a total of 535619 fragments. From equation (4.10) and using the values of given for the mass of the debris-like particles we find that the number fragments that can possibly be generated by the collision between these particles is 442. In the same way we can determine that the number of fragments

from a collision between a debris-like particle and a larger ordered satellite is 27895. Multiplying the former by $N_{\text{cols}} - 5$ gives 317766 and multiplying the latter with 5 gives 139475. Adding these fragment numbers up gives $N_{\text{frags,max}} = 457241$ which is the number of fragments generated in the first five satellite-debris collisions added to the maximum number of fragments that could have been generated by debris-debris collisions alone. However, The total number of fragments generated is over a million, because 812858 fragments have collided with the Earth. Since this number is larger than $N_{\text{frags,max}}$, this must mean that the fragments generated in the first five satellite-debris collisions have collided with more debris particles, which would generate more fragments than debris-debris collisions alone.

If anything, all the figures in this section show the tendency of one or a few fragmentation events early on in the evolution of a debris model to cause a cascade of collisions and debris generation right after. In this sense, both the discrete and continuous algorithms predict a Kessler Syndrome. Therefore, this thesis' simple collision model slightly mirrors predictions made by more advanced models from leading space agencies like NASA's LEGEND, ESA's DELTA and JAXA's LEODEEM [Lio+13].

6. Conclusion

The aim of this paper was to create a simplified space debris evolution model for a system of colliding satellites in Kepler orbits in order to study the Kessler Syndrome. To enable the use of Kepler orbits we neglected i) the mutual gravity between any two satellites, ii) the oblate shape of the Earth (J2 perturbation), iii) disturbing gravity of the moon and planets, iv) the atmosphere and v) solar radiation pressure. This allowed for a simple Two Body Problem (TBP) formulation for the equations of motion of the satellites. To check and compare this Kepler orbit-based collision detection method, another method was developed based on the numerical integration of the TBP. As the former is an analytical solution of the TBP valid at all times and the latter only at multiples of the time step Δt, these algorithms were aptly named *The continuous algorithm* and *The discrete algorithm*, respectively.

The comparison between the discrete and continuous algorithms focused on the average time to the first collision $\bar{t}_{col}$ in a system of N homogeneously distributed satellites with radius s and within a spherical shell of height h (in LEO). $\bar{t}_{col}$ was computed for several values of N, s and h, keeping the other two constant. The predictions of both algorithms were then successfully fitted to a theoretically determined relation for the value of $\bar{t}_{col}$ based on the same parameters. Further analysis of the computation time suggested that the continuous algorithm has a time-complexity of $O(N^2)$, as it was outperformed by the discrete algorithm for systems with high satellite numbers and densely populated with orbits. However, for systems with a relatively low number of small satellites in non-crossing orbits, the continuous algorithm performed better. This was because it calculates the time of the next collision, which was large for systems of this kind. As a consequence, it only needed to propagate the two colliding satellites to this time and it could do so instantly using the equation of a Kepler orbit. In contrast, the discrete algorithm had to propagate the entire system of satellites and could only perform collisions in the current time step. This advantage of the continuous algorithm is presumed to be entirely lost however, once one or more of the assumptions i-v) are removed from the model. Though both algorithms amenable to and could greatly benefit from parallel execution of their inherent calculations, it is for this reason that the continuous algorithm could be improved the most.

An implementation of the NASA Standard Breakup Model was used to perform general two-body collisions, which in combination with the continuous algorithm was used to predict long term LEO debris evolution of a small number of homogeneously distributed satellites. Two simulations were discussed, one with $N = 100$ and other with $N = 500$ satellites initially. Both predicted that all the original satellites collide and break apart within the first decade, causing further collision and fragmentation later on. In at least the first simulation, the average time between collisions was shown to decrease approximately two orders of magnitude over a period of 500 years. A third simulation of the short term evolution of 5000 ordered satellites in combination with 10^5 homogeneously distributed debris-like particles using the discrete algorithm showed a similar decrease in average collision time. Both algorithms showed the cascading effect a fragmentation collision early on in the simulation has on the the number of fragments and collisions after it. It was therefore concluded that this thesis' simple space debris evolution model already encompasses the Kessler Syndrome similar to more advanced from leading space agencies.

Future research could focus on expanding the continuous algorithm to include basic effects like ii) and iii) so that it can be applied more realistically to larger time frames. Especially, to

investigate if it is still able to bridge large time gaps more efficiently than the discrete algorithm or any other time integration based methods can.

References

[Opi51] E. J. Opik. "Collision probability with the planets and the distribution of planetary matter". In: *Proc. R. Irish Acad. Sect. A* 54 (Jan. 1951), pp. 165–199.

[Ben75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18 (9 Sept. 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[BSW77] Jon L. Bentley, Donald F. Stanat, and E.Hollins Williams. "The complexity of finding fixed-radius near neighbors". In: *Information Processing Letters* 6 (6 Dec. 1977), pp. 209–212. ISSN: 00200190. DOI: 10.1016/0020-0190(77)90070-9.

[KC78] Donald J. Kessler and Burton G. Cour-Palais. "Collision frequency of artificial satellites: The creation of a debris belt". In: *Journal of Geophysical Research* 83 (A6 1978), p. 2637. ISSN: 0148-0227. DOI: 10.1029/JA083iA06p02637.

[MMB98] S. P. Manley, F. Migliorini, and M. E. Bailey. "An algorithm for determining collision probabilities between small solar system bodies". In: *Astronomy and Astrophysics Supplement Series* 133 (3 Dec. 1998), pp. 437–444. ISSN: 0365-0138. DOI: 10.1051/aas:1998334.

[MD00] C. Murray and S. Dermott. *The Two-Body Problem*. Cambridge University Press, Feb. 2000, pp. 22–62. DOI: 10.1017/CBO9781139174817.003.

[Joh+01] N L Johnson' et al. *NASA'S NEW BREAKUP MODEL OF EVOLVE 4.0*. COSPAR, 2001, pp. 1377–1384. URL: www.elsevier.com/locate/asr.

[HLW03] Ernst Hairer, Christian Lubich, and Gerhard Wanner. "Geometric numerical integration illustrated by the Störmer-Verlet method". In: *Acta Numerica* 12 (2003). Includes all necessary proofs for (Velocity) Verlet: symplecticity, time-reversibility, conservation of (angular) momentum and energy, pp. 399–450. ISSN: 09624929. DOI: 10.1017/S0962492902000144.

[Nic09] J. Nicholas. *The Collision of Iridium 33 and Cosmos 2251: The Shape of Things to Come*. 2009. URL: https://ntrs.nasa.gov/citations/20100002023.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Lio+13] J.-C. Liou et al. "Stability of the future LEO environment - An IADC Comparison Study". In: ed. by L Ouwehand. Apr. 2013.

[BSC15] Jens Behley, Volker Steinhage, and Armin B. Cremers. "Efficient radius neighbor search in three-dimensional point clouds". In: IEEE, May 2015, pp. 3625–3630. ISBN: 978-1-4799-6923-4. DOI: 10.1109/ICRA.2015.7139702.

[Bro15] Russell A. Brown. "Building a Balanced k-d Tree in $O(kn \log n)$ Time". In: *Journal of Computer Graphics Techniques (JCGT)* 4.1 (Mar. 2015), pp. 50–68. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/01/03/.

[DRD15] J C Dolado-Perez, Bruno Revelin, and Romain Di-Costanzo. “SENSITIVITY ANALYSIS OF THE LONG-TERM EVOLUTION OF THE SPACE DEBRIS POPULATION IN LEO”. In: *Journal of Space Safety Engineering* 2 (1 June 2015), pp. 12–22.

[15] *Gravity*. 2015.

[JM17] Youngmin JeongAhn and Renu Malhotra. “Simplified Derivation of the Collision Probability of Two Objects in Independent Keplerian Orbits”. In: *The Astronomical Journal* 153 (5 Apr. 2017), p. 235. ISSN: 0004-6256. DOI: 10.3847/1538-3881/aa6aa7.

[DH18] Jakub Drmola and Tomas Hubik. “Kessler Syndrome: System Dynamics Model”. In: *Space Policy* 44-45 (Aug. 2018), pp. 29–39. ISSN: 1879338X. DOI: 10.1016/j.spacepol.2018.03.003.

[RA21] P. Rincon and J. Amos. *Russian anti-satellite test adds to worsening problem of space debris*. 2021. URL: https://www.bbc.com/news/science-environment-59307862.

[ESO] Space Debris Office ESOC/ESA. *Space environment statistics*. URL: https://sdup.esoc.esa.int/discosweb/statistics/.

Appendix

A: Additional equations for orbital elements

The coordinate transformation from the orbital to the reference plane is mediated by the following rotation matrix

$$\mathcal{R} = \begin{pmatrix} \cos\Omega\cos\omega - \sin\Omega\sin\omega\cos I & -\cos\Omega\sin\omega - \sin\Omega\cos\omega\cos I & \sin\Omega\sin I \\ \sin\Omega\cos\omega + \cos\Omega\sin\omega\cos I & -\sin\Omega\sin\omega + \cos\Omega\cos\omega\cos I & -\cos\Omega\sin I \\ \sin\omega\sin I & \cos\omega\sin I & \cos I \end{pmatrix}, \tag{A1}$$

with the longitude of ascending node Ω, argument of periapsis ω and inclination I.

The angular momentum of a satellite is given by

$$\vec{L} = m\vec{r} \times \vec{v}. \tag{A2}$$

The eccentricity vector of an orbit is related to the Laplace-Runge-Lenz vector $\overrightarrow{LRL}$ as

$$\vec{e} = \frac{\overrightarrow{LRL}}{GMm^2} = \frac{\vec{v} \times \vec{L}}{GMm} - \frac{\vec{r}}{|\vec{r}|}, \tag{A3}$$

where M is the mass of the central body and m and $\vec{r}$ are the mass and position vector of the satellite.

The velocity of a satellite can be determined from $\vec{L}$ and $\vec{e}$ through

$$\vec{v} = \frac{GMm}{|\vec{L}|}\vec{L} \times \left(\vec{e} + \frac{\vec{r}}{|\vec{r}|}\right). \tag{A4}$$

B: Derivations for Kepler orbit collision methods

Here follows the full derivation of equation 3.11 Squaring and expanding equation 3.10 gives

$|\vec{\delta}| < (s_i + s_j)|\vec{u}|$

$|(\vec{r}_j - \vec{r}_i + \vec{v}_j dt_j - \vec{v}_i dt_i) \times \vec{u}|^2 < (s_i + s_j)^2|\vec{u}|^2,$

$|(\vec{r}_j - \vec{r}_i) \times \vec{u}|^2 + 2(\vec{r}_j - \vec{r}_j) \times \vec{u} \cdot (\vec{v}_j dt_j - \vec{v}_i dt_i) \times \vec{u} + |(\vec{v}_j dt_j - \vec{v}_i dt_i) \times \vec{u}|^2 < (s_i + s_j)^2|\vec{u}|^2.$

The second term on the l.h.s. of the inequality can be shown to be zero. Where it can be used that for $\vec{d} = \vec{r}_j - \vec{r}_i$, $\vec{d} \cdot \vec{v}_i = 0$ and $\vec{d} \cdot \vec{v}_j = 0$, since $\vec{d}$ is the minimum distance vector. This in combination with the vector identity

$$(\vec{a} \times \vec{b}) \cdot (\vec{c} \times \vec{b}) = (\vec{a} \cdot \vec{c})|\vec{b}|^2 - (\vec{a} \cdot \vec{b})(\vec{c} \cdot \vec{b}).$$

As for the first term, we have

$$|(\vec{r}_j - \vec{r}_i) \times \vec{u}|^2 = |\vec{d} \times \vec{u}|^2 = |\vec{d}|^2|\vec{u}|^2 - (\vec{d} \cdot \vec{u})^2 = |\vec{d}|^2|\vec{u}|^2,$$

using that $\vec{d} \cdot \vec{u} = 0$. Hence this simplifies to

$$|(\vec{v}_j(t_{ij}^{col} - kT_j - t_j^1) - \vec{v}_i(t_{ij}^{col} - kT_i - t_i^1)) \times \vec{u}| < \sqrt{(s_i + s_j)^2 - |\vec{d}|^2}|\vec{u}|,$$

where $dt_{i,j}$ is expanded using equation 3.9. Now, using that $\vec{u} = \vec{v}_j - \vec{v}_i$, we get

$$\left|kT_i + t_i^1 - lT_j - t_j^1\right| < \frac{\sqrt{(s_i + s_j)^2 - |\vec{d}|^2}|\vec{u}|}{|\vec{v}_i \times \vec{v}_j|},$$

as required.

C: NASA SBM distributions

Area-to-mass ratio of fragments with $L_c \geqslant 11$ cm satisfies the following distribution

$$D_{A/M}^{L_c \geqslant 0.11} = \alpha(\lambda_c)\mathrm{N}(\mu_1(\lambda_c), \sigma_1(\lambda_c), \chi) + (1 - \alpha(\lambda_c))\mathrm{N}(\mu_2(\lambda_c), \sigma_2(\lambda_c), \chi), \tag{C1}$$

where

- $\lambda_c = \log_{10}(L_c)$
- $\chi = \log_{10}(A/M)$
- N is the normal distribution function with pre-factor α, mean $\mu_{1,2}$ and standard deviation $\sigma_{1,2}$. These are in turn given by

$$\alpha = \begin{cases} 0 & \lambda_c \leqslant -0.95 \\ 0.3 + 0.4(\lambda_c + 1.2) & -0.95 < \lambda_c < 0.55 \\ 1 & \lambda_c \geqslant 0.55 \end{cases}$$

$$\mu_1 = \begin{cases} -0.6 & \lambda_c \leqslant -1.1 \\ -0.6 + 0.318(\lambda_c + 1.1) & -1.1 < \lambda_c < 0 \\ -0.95 & \lambda_c \geqslant 0 \end{cases}$$

$$\sigma_1 = \begin{cases} 0.1 & \lambda_c \leqslant -1.3 \\ 0.1 + 0.2(\lambda_c + 1.3) & -1.3 < \lambda_c < -0.3 \\ 0.3 & \lambda_c \geqslant -0.3 \end{cases}$$

$$\mu_2 = \begin{cases} -1.2 & \lambda_c \leqslant -0.7 \\ -1.2 + 1.333(\lambda_c + 0.7) & -0.7 < \lambda_c < -0.1 \\ -2.0 & \lambda_c \geqslant -0.1 \end{cases}$$

$$\sigma_2 = \begin{cases} 0.5 & \lambda_c \leqslant -0.5 \\ -\lambda_c & -0.5 < \lambda_c < -0.3 \\ 0.3 & \lambda_c \geqslant -0.3 \end{cases}$$

Area-to-mass ratio of fragments with $L_c < 11$ cm satisfies

$$D_{A/M}^{L_c < 0.11} = \mathrm{N}(\mu(\lambda_c), \sigma(\lambda_c), \chi), \tag{C2}$$

where

$$\mu = \begin{cases} -0.3 & \lambda_c \leqslant -1.75 \\ -0.3 + 1.4(\lambda_c + 1.75) & -1.75 < \lambda_c < -1.25 \\ -1.0 & \lambda_c \geqslant -1.25 \end{cases}$$

$$\sigma = \begin{cases} 0.2 & \lambda_c \leqslant -3.5 \\ 0.2 + 0.1333(\lambda_c + 3.5) & \lambda_c > -3.5 \end{cases}$$

The delta-velocity distribution for fragments is distributed as

$$D_{\Delta v} = \mathrm{N}(\mu(\chi), \sigma, \nu). \tag{C3}$$

where

- $\nu = \log_{10}(\Delta v)$
- $\mu = 0.6\chi + 2.9$
- $\sigma = 0.4$

Code for the simulation

```
#from dataclass import datasets
import numpy as np
from scipy import special
import os

#%% datasets class
class datasets():
    """Contains a collection of datasets to experiment with the Kepler and
    k-d tree SCM's."""
    ############## constants ############
    # mass & radius of the earth
    earthM = 5.972e24               #kg
    earthR = 6371e3                 #m
    # The gravitational constant
    G = 6.67428e-11
    mu = G*earthM
    # Astronomical unit
    AU = (149.6e6 * 1000)           #149.6 million km, in meters.
    #time steps
    minute = 60
    hour = 60*minute
    day = 24*hour
    year = 365*day
    century = 100*year

    def __init__(self,data_type,args=None):
        if args != None:
            datasets.__dict__[data_type](self,args)
```

```
        else:
            datasets.__dict__[data_type](self)

    def kep(self):
        """load parameters for satellites (100 starlink satellites) at approximately same epoch"""
        import openpyxl
        filepath = r"C:\\Users\\TUDelftSID\\OneDrive - Delft University of Technology\\Documenten\\TUD\\BEP\\realsatdata\\"
        filename = 'starlink-track(1 meting 100 sats)'
        wb_obj = openpyxl.load_workbook(filepath+filename+'.xlsx')
        sheet = wb_obj.active
        col_names = []
        for column in sheet.iter_cols(1, sheet.max_column):
            col_names.append(column[0].value)
        data = {}
        for i, row in enumerate(sheet.iter_rows(values_only=True)):
            for j in range(16):
                if i == 0:
                    data[col_names[j]] = []
                else:
                    data[col_names[j]].append(row[j])
        for key in list(data.keys()):
            data[key] = np.array(data[key])

        #preparing data
        Nsats_k = len(data['NORAD_CAT_ID'])
        self.Epoch = data['EPOCH']
        Inc = data['Inc'] #degrees
        self.Ecc = data['Ecc']
        MnM = data['MnM'] #revolutions per day
        LAN = data['LAN'] #degrees
        AgP = data['AgP'] #degrees
        MnA = data['MnA'] #degrees
        SMA = data['SMa']
        #typical mass and size of starlink sats
        self.Nsats = Nsats_k
        self.S =  np.array([1]*Nsats_k) #'roughly the size of a table': https://skyandtelescope.org/astronomy-news/spacex-launches-latest-starlink-satellite-batch/'
        self.M = np.array([280]*Nsats_k)
        #converting units
        self.MnM = 2*np.pi/(self.day/MnM) #rads per second
        self.Inc = 2*np.pi*(Inc/360)
        self.LAN = 2*np.pi*(LAN/360)
        self.AgP = 2*np.pi*(AgP/360)
        self.MnA = 2*np.pi*(MnA/360)
        self.SMA = SMA*1e3

    def eph(self):
        """load ephemeris of all current starlink satellites (as of 27 September 2021).
        The correspoding ephemeris file should be placed in the same directory as this one."""
        #set file location
```

```
77          filepath = r"C:\Users\TUDelftSID\OneDrive - Delft University of
    Technology\Documenten\TUD\BEP\ephemeris starlink 2021270"
78          files = os.listdir(filepath)
79          Epoch = []
80          Nsats_e = len(files)
81          R = np.zeros((Nsats_e,3))
82          V = np.zeros((Nsats_e,3))
83          for i,file in enumerate(files):
84              n = os.path.join(filepath,file)
85              with open(n) as fi:
86                  eph = fi.readlines()[4].split(" ")
87                  eph[-1] = eph[-1][:-2]
88                  params = np.array(eph[1:]).astype(np.float64) #Epoch, XYZ,
    VxVyVz
89              #time of measurement given in seconds starting from the
    beginning of 2021
90              day = float(eph[0][4:7])
91              hour = float(eph[0][7:9])
92              minute = float(eph[0][9:11])
93              sec = float(eph[0][11:])
94              Epoch.append(day*24*3600+hour*3600+minute*60+sec)
95              R[i,:] = params[0:3]*1e3
96              V[i,:] = params[3:]*1e3
97          #typical mass and size of starlink sats
98          self.Nsats = Nsats_e
99          self.R = R
100         self.V = V
101         self.S =  np.array([1]*Nsats_e) #'roughly the size of a table':
    https://skyandtelescope.org/astronomy-news/spacex-launches-latest-
    starlink-satellite-batch/'
102         self.M = np.array([280]*Nsats_e)
103         #correct unsimultaneous measurement of parameters
104         #(still needs to be done...)
105
106
107     def col(self,N):
108         """ take one starlink sat and randomly change its LAN, Inc and AgP
109             thereby generating multiple sats that will likely collide"""
110         N = int(N)
111         self.Inc = np.random.random(N)*np.pi
112         self.Ecc = np.array([0]*N)
113         self.LAN = np.random.random(N)*2*np.pi
114         self.AgP = np.random.random(N)*2*np.pi
115         self.MnA = np.random.random(N)*2*np.pi-self.AgP #mean anomaly is
    zero at periapsis
116         self.SMA = np.array([6.5258*1e6]*N)
117         self.MnM = np.sqrt(self.mu/self.SMA**3)
118         self.S = np.array([3]*N) #increase chance of collisions
119         self.M = np.array([400]*N)
120         self.Nsats = N
121
122     def sim(self,args):
123         """a random system of satellites used to study/look for the Kessler
     syndrome"""
124         N = args[0]
```

```
        s = args[1]
        m = args[2]
        h = args[3]
        self.Nsats = N
        self.Inc = np.random.random(N)*np.pi
        self.LAN = np.random.random(N)*2*np.pi
        self.AgP = np.random.random(N)*2*np.pi
        self.MnA = np.random.random(N)*2*np.pi-self.AgP #mean anomaly is
    zero at periapsis
        R_i = self.earthR+h[0]*1e3 #minimum of ... km height
        R_o = self.earthR+h[1]*1e3 #maximum of ... km height: #LEO: 0<a
    <2000

        #completely homogeneous (no lower bound for eccentricity)
        a1 = np.random.uniform(low=R_i,high=R_o,size=N)
        a2 = np.random.uniform(low=R_i,high=R_o,size=N)
        A = np.array([a1,a2])
        per = np.min(A,axis=0)
        aper = np.max(A,axis=0)

        # #minimum eccentricity (favors higher orbits)
        # l = 2*R_o*e_min/(1+e_min)
        # per = np.random.uniform(low=R_i,high=R_o-l,size=N)
        # aper = np.random.uniform(low=per+l,high=R_o,size=(1,N))[0]
        self.SMA = (1/2)*(aper+per)
        self.Ecc = (aper-per)/(aper+per)
        self.MnM = np.sqrt(self.mu/self.SMA**3)
        self.S = np.array([s]*N)
        self.M = np.array([m]*N)

    def ordd(self,args):
        """system of sats that do not collide"""
        Nsats_o = args[0]
        self.Nsats = Nsats_o
        layers = 5
        s = args[1]
        m = args[2]
        h = args[3]
        band = (h[1]-h[0])*1e3
        lb = datasets.earthR + h[0]*1e3
        ub = datasets.earthR + h[1]*1e3
        self.Inc = np.array([(53/360)*2*np.pi]*Nsats_o)
        self.Ecc = np.array([0.00001]*Nsats_o)#0.001*(1 + np.random.random(
    Nsats_o))
        self.SMA = np.tile(np.arange(lb,ub,band/layers),(int(Nsats_o/layers
    ,)))
        self.MnM = np.sqrt(self.mu/self.SMA**3)
        self.LAN = np.linspace(0,2*np.pi,Nsats_o)
        self.AgP = np.array([(80/360)*2*np.pi]*Nsats_o)
        self.MnA = np.tile(np.arange(-70,70,140/layers),(int(Nsats_o/layers
    ,)))-self.AgP #mean anomaly is zero at periapsis
        self.S = np.array([s]*Nsats_o)
        self.M = np.array([m]*Nsats_o)

    def rog(self,args):
```

```
        """same as above, but with one rogue satellite"""
        self.ordd(args)
        Nsats_r= self.Nsats+1
        self.Nsats = Nsats_r
        self.Inc = np.append(self.Inc,np.array([((180-53)/360)*2*np.pi])) #
    retrograde sat
        self.Ecc = np.append(self.Ecc,self.Ecc[-1])
        self.SMA = np.append(self.SMA,self.SMA[8]) #same as 9th layer
        self.MnM = np.sqrt(self.mu/self.SMA**3)
        self.LAN = np.append(self.LAN,self.LAN[8]+(2/360)*2*np.pi) #aprox
    same as 9th layer.
        self.AgP = np.append(self.AgP,np.array([(80/360)*2*np.pi])) # same
    as before
        self.MnA = np.append(self.MnA,self.MnA[8])-self.AgP #mean anomaly
    is zero at periapsis
        self.S = np.append(self.S,np.array([2]))
        self.M = np.append(self.M,np.array([200]))

    def mist(self,args):
        """combination of ordered and homogeneously distributed satellites
    """
        ordd = datasets('ordd',args[0])
        mist = datasets('sim',args[1])
        self.Nsats = mist.Nsats + ordd.Nsats
        self.Inc = np.append(ordd.Inc,mist.Inc)
        self.LAN = np.append(ordd.LAN,mist.LAN)
        self.AgP = np.append(ordd.AgP,mist.AgP)
        self.MnA = np.append(ordd.MnA,mist.MnA)
        self.Ecc = np.append(ordd.Ecc,mist.Ecc)
        self.SMA = np.append(ordd.SMA,mist.SMA)
        self.MnM = np.append(ordd.MnM,mist.MnM)
        self.S = np.append(ordd.S,mist.S)
        self.M = np.append(ordd.M,mist.M)

    def sc(self,args):
        """one pair of colliding satellites"""
        Nsats = 2
        m0 = args[0]
        m1 = args[1]
        s0 = args[2]
        s1 = args[3]
        self.Nsats = Nsats
        self.Inc = np.array([60.0,120.0])*2*np.pi/360
        self.Ecc = np.array([0.005]*Nsats)
        self.SMA = np.array([100e3+self.earthR]*2)
        self.MnM = np.sqrt(self.mu/self.SMA**3)
        self.LAN = np.array([0,10.0])*2*np.pi/360
        self.AgP = np.array([0.0]*Nsats)
        self.MnA = np.array([0.0]*Nsats)-self.AgP #mean anomaly is zero at
    periapsis
        self.S = np.array([s0,s1])
        self.M = np.array([m0,m1])

    def ze(self,N):
        """random system of satellites in circular orbits"""
```

```
        self.Nsats = N
        h_l = 100
        h_u = 200
        self.Inc = np.random.random(N)*2*np.pi
        self.LAN = np.random.random(N)*2*np.pi
        self.AgP = np.random.random(N)*2*np.pi
        self.MnA = np.random.random(N)*2*np.pi-self.AgP #mean anomaly is zero at periapsis
        self.Ecc = np.zeros(N)
        R_i = self.earthR+h_l*1e3 #minimum of ... km height
        R_o = self.earthR+h_u*1e3 #maximum of ... km height: #LEO: 0<a<2000
        self.SMA = (np.random.random(N)*(R_o-R_i)+R_i)
        self.MnM = np.sqrt(self.mu/self.SMA**3)
        self.S = np.array([10.0]*N)
        self.M = np.array([200.0]*N)

    def fc(self):
        """OMM of all satellites (active and debris) in LEO (yet to be implemented)"""
        pass


#%% Main Kessler class
class Kessler(datasets):
    """Main Kessler class. Imports data from the datasets clas, contains an implementation
    of NASA's SBM and key methods (coordinate transformation from orbital plane to reference
    plane, angular momentum-, eccentricity vector, the Verlet algorithm and more)."""

    def __init__(self,Lcmin=None,alfa=None,Nbins=None):
        """Some NASA SBM parameters: Lcmin is the minimum characteristic length,
        alfa is the maxmimum scattering angle, 'nu' contains the to be sampled
        values of the log of delta velocities (nu = log(dV)) and minNF is lower
        bound for the number of fragments generated in a collision.
        """
        ### NASA SBM parameters
        if Lcmin == None:
            self.Lcmin = 0.05 #m
        else:
            self.Lcmin = Lcmin
        if alfa == None:
            self.alfa = 3 #degrees
        else:
            self.alfa = alfa
        if Nbins==None: #resolution of Lc and nu arrays
            self.Nbins = 50
        else:
            self.Nbins = int(Nbins)
        self.nu = np.linspace(0,2.5,self.Nbins) #10^0 = 1 to 10^(2.7) = 500 m/s
```

```
        self.alfa = 3 #maximum scattering and breakup angle
        self.minNF = 4 #minimum number of fragments
        #total number of collisions
        self.Ncols = np.array([0],dtype=np.int32)
        #arrays keeping track of number of collisions per type
        self.Ncc,self.Nncc,self.Nsc = np.array([],dtype=np.int32),\
            np.array([],dtype=np.int32),np.array([],dtype=np.int32)
        self.Ncct,self.Nncct,self.Nsct =np.array([],dtype=np.int32),\
            np.array([],dtype=np.int32),np.array([],dtype=np.int32)
        #arrays keeping track of number of escaped or decayed fragments
        self.Npe,self.Npccb = np.array([0],dtype=np.int32),\
            np.array([0],dtype=np.int32)

    def load_data(self,dtype,max_t,args=None):
        datasets.__init__(self,dtype,args=args)
        self.max_t = max_t

    ######################################################
    ############# NASA BREAKUP MODEL ####################
    ######################################################
    def collision(self,other,tcol,dt,alg=None):
        """Root collision function. Determines what kind of collision
    should take place
        (elastic/scatter, breakup) and calls the appropriate collision
    function. t_col
        is an np array of shape (n,) or list containing the time of all the
     n collisions
        (in case of kepler algorithm tcol = [0]).
        alfa is the maximum scattering angle in degrees.
        """
        #initialise fragment arrays
        other.R = np.zeros((0,3))
        other.V = np.zeros((0,3))
        other.M = np.zeros(0)
        other.S = np.zeros(0)
        #determine which particles scatter and which breakup
        if alg == 'kep':
            pi = np.array([self.si[0]])
            pj = np.array([self.sj[0]])
        else:
            pi = self.si
            pj = self.sj
        rad_i = self.S[pi]
        rad_j = self.S[pj]
        a1 = rad_i<=self.Lcmin#below Lcmin only scattering will take place
        a2 = rad_j<=self.Lcmin
        a = a1|a2
        scat_idx = np.arange(len(pi))[a]
        break_idx = np.arange(len(pj))[np.logical_not(a)]

        U = self.V[pi[break_idx]] - self.V[pj[break_idx]]
        M_i = self.M[pi[break_idx]]
        M_j = self.M[pj[break_idx]]
        Mp_idx = (M_i<M_j).astype(int)
        Mp = np.array([M_j,M_i])[Mp_idx,np.arange(len(break_idx))]
```

```
        U_normsq = np.sum(U**2,axis=1)
        limit = (self.minNF/(0.1*self.Lcmin**(-1.71)))**(4/3)
        b1 = Mp*U_normsq/1e6<=limit
        Mtot = M_i+M_j
        b2 = Mtot<=limit
        b = b1&b2
        scat_idx = np.append(scat_idx,break_idx[b])
        break_idx = break_idx[np.logical_not(b)]
        pi_br = pi[break_idx]
        pj_br = pj[break_idx]
        pi_sc = pi[scat_idx]
        pj_sc = pj[scat_idx]

        tcol_br = tcol[break_idx]
        tcol_sc = tcol[scat_idx]

        ncc,cc = self.colkind(pi_br,pj_br,tcol_br)
        U = self.V[ncc[0]] - self.V[ncc[1]]
        U_normsq = np.sum(U**2,axis=1)
        M_i_ncc = self.M[ncc[0]]
        M_j_ncc = self.M[ncc[1]]
        M_i_cc = self.M[cc[0]]
        M_j_cc = self.M[cc[1]]
        Mp_idx = (M_i_ncc<M_j_ncc).astype(int)
        Mp = np.array([M_j_ncc,M_i_ncc])[Mp_idx,np.arange(len(ncc[0]))]
        c1 = Mp*U_normsq/1e6<=limit
        c2 = M_i_cc+M_j_cc<=limit
        idxi = np.append(ncc[0][c1], cc[0][c2])
        idxj = np.append(ncc[1][c1], cc[1][c2])
        c1 = np.logical_not(c1)
        c2 = np.logical_not(c2)
        pi_br_ncc = ncc[0][c1]
        pj_br_ncc = ncc[1][c1]
        N_ncc = ncc[2][c1]
        ncc = (pi_br_ncc,pj_br_ncc,N_ncc,ncc[3])
        pi_br_cc = cc[0][c2]
        pj_br_cc = cc[1][c2]
        N_cc = cc[2][c2]
        cc = (pi_br_cc,pj_br_cc,N_cc,cc[3])

        sorter = np.argsort(pi)
        tcoldiff = tcol[sorter[np.searchsorted(pi, idxi, sorter=sorter)]]
        tcol_br = np.delete(tcol_br,np.where(tcol_br==tcoldiff)[0])
        tcol_sc = np.append(tcol_sc,tcoldiff)

        pi_sc = np.append(pi_sc,idxi)
        pj_sc = np.append(pj_sc,idxj)
        p_sc = (pi_sc,pj_sc)

        N_sc = len(pi_sc)
        N_ncc = len(pi_br_ncc)
        N_cc = len(pi_br_cc)
        self.N_l = [N_sc,N_ncc,N_cc]

        if len(pi_br)>0:
```

```
            self.breakup(other,tcol_br,ncc,cc,dt)
        if len(pi_sc)>0:
            self.scatter(other,tcol_sc,p_sc,dt)

        self.delarr = np.concatenate(cc[:2]+ncc[:2]+p_sc,axis=0)

    def colkind(self,pi,pj,tcol):
        """Determines what kind of collision occurs between two satellites.
        Returns indices of (non-)catastrophic collisions and gives the total
        number of fragments.
        """
        proj_i = np.where(self.M[pi]<=self.M[pj])[0] #i is projectile if its mass is less massive
        proj_j = np.where(self.M[pj]<self.M[pi])[0] #idem
        #relative kinetic energy of projectile divided by mass of larger sat
        #[J/g]
        U = self.V[pi] - self.V[pj]
        U_normsq = np.sum(U**2,axis=1)
        Er_i = (1/2)*self.M[pi][proj_i]*U_normsq[proj_i]/(1e3*self.M[pj][proj_i])
        Er_j = (1/2)*self.M[pj][proj_j]*U_normsq[proj_j]/(1e3*self.M[pi][proj_j])
        #determine indices of catastrophic collisions (Er>40J/g)
        cat_i = proj_i[np.where(Er_i>40)]
        ncat_i = proj_i[np.where(Er_i<=40)]
        cat_j = proj_j[np.where(Er_j>40)]
        ncat_j = proj_j[np.where(Er_j<=40)]
        cat = np.append(cat_i,cat_j)
        pi_c = pi[cat]
        pj_c = pj[cat]
        M_ncat = np.append(self.M[pi][ncat_i]*(U_normsq[ncat_i]/1e6),
                           self.M[pj][ncat_j]*(U_normsq[ncat_j]/1e6))
        ncat = np.append(ncat_i,ncat_j)
        pi = pi[ncat]
        pj = pj[ncat]
        # M_cat = (self.M[pi_c]+self.M[pj_c])[:,None]
        M_cat = (self.M[pi_c]+self.M[pj_c])
        N_ncat = 0.1*M_ncat**(0.75)*self.Lcmin**(-1.71)/2 #other half is added later
        N_cat = 0.1*M_cat**(0.75)*self.Lcmin**(-1.71)/2 #idem
        ncc = (pi,pj,N_ncat.astype(int),tcol[ncat])
        cc = (pi_c,pj_c,N_cat.astype(int),tcol[cat])
        return ncc,cc

    def breakup(self,other,tcol,ncc,cc,dt):
        """calls the appropriate version of the 'Kessler.fragment' method
        """
        #assign masses, sizes and velocities to fragments
        if len(ncc[2])>0:
            self.fragment(other,ncc[0],ncc[1],ncc[2],tcol,'ncat',dt)
        if len(cc[2])>0:
            self.fragment(other,cc[0],cc[1],cc[2],tcol,'cat',dt)

```

```
425     def fragment(self,other,pi,pj,N,tcol,kind,dt):
426         """calculates size, mass position and velocity of fragments and
    appends
427         these to the fragment arrays. Distinguishes between catastrophic (
    cc) and
428         non-catastrophic collisions (ncc). In the case of cc, both
    satellites are
429         fragmented entirely. As for ncc, the fragmented mass is calculated
    as the
430         product of the mass of the lighter projectile and the square of the
431         relative velocity (km/s). The remaining mass is deposited into to
    two
432         additional parent fragments (a small and big one reminiscent of the
     projectile
433         and target), both of which are given a velocity sampled from the
    same delta V
434         distributions as the fragments (using their AM-ratios). Both mass
    and
435         kinetic energy (kE) conservation are ensured using a simple scaling
    , where
436         the velocities are scaled only if the kE_final is greater than
    kE_initial
437         and left as they are otherwise.
438         """
439         Ncols=len(pi)
440         for i in range(Ncols):
441             Nfrags_tot = N[i] #total number of frags
442             min_r = self.Lcmin
443             max_r = min([self.S[pi[i]],self.S[pj[i]]])
444             if max_r>1.0:
445                 max_r = 1.0
446             unif = np.random.uniform(0,1,size=Nfrags_tot)
447             a = -1.71
448             n = self.Nbins
449             Lc = ((min_r**a-max_r**a)*unif+max_r**a)**(1/a)
450             Nfrags,Lc = np.histogram(Lc,bins=n)
451             chi = np.linspace(-2.5,0.3,n)
452             D = self.D_AM(Lc[:-1],chi)
453             AMarr = np.zeros((n,np.max(Nfrags)))
454             for j in range(n):
455                 samples = np.random.random((Nfrags[j],n))
456                 D_tiled = np.tile(D[j],(Nfrags[j],1))
457                 chi_idx = self.find_nearest(D_tiled,samples)
458                 AMarr[j,:Nfrags[j]] = 10**(chi[chi_idx])
459             #obtain mass of each fragment
460             Lcarr = np.transpose(np.tile(Lc[:-1],(np.max(Nfrags),1)))
461             A = 0.556945*Lcarr**2
462             zero_id = np.where(np.ndarray.flatten(AMarr)==0)
463             AMarr[np.where(AMarr==0)]=1 #to prevent division by zero
464             Marr = np.ndarray.flatten(A/AMarr)
465             Marr = np.delete(Marr,zero_id)
466             Lcarr = np.delete(np.ndarray.flatten(Lcarr),zero_id)
467             AMarr = np.delete(np.ndarray.flatten(AMarr),zero_id)
468             #assure mass conservation adding parent sats
469             if kind == 'ncat':
```

```
                Ml = np.array([self.M[pi[i]],self.M[pj[i]]])
                Sl = np.array([self.S[pi[i]],self.S[pj[i]]])
                pidx = np.where(Ml==min(Ml))[0]
                tidx = np.where(Ml==max(Ml))[0]
                if np.all(pidx==tidx):
                    pidx = pidx[0]
                    tidx = tidx[1]
                Mp = Ml[pidx]
                Mt = Ml[tidx]
                Sp = Sl[pidx]
                St = Sl[tidx]
                Ui = np.array(self.V[pj[i]]-self.V[pi[i]])
                consM = Mp*np.sum(Ui**2,axis=0)/1e6
                if consM<Mp:
                    dmt = dmp = (1/2)*consM
                else:
                    dm = (consM-Mp)/Mp
                    dmp = (dm%1)*Mp
                    dmt = dm*Mp-dmp
                Mt_n = Mt-dmt
                Mp_n = Mp-dmp
                Marr_par = np.append(Mt_n,Mp_n) #parent array
                Lc_t = St*(1-dmt/Mt)**(1/3)
                Lc_p = Sp*(1-dmp/Mp)**(1/3)
                Lcarr_par = np.append(Lc_t,Lc_p)
                AMt = 0.556945*St**2/Mt_n
                AMp = 0.556945*Sp**2/Mp_n
                AMarr_par = np.append(AMt,AMp)
                print('\nncc between {0} and {1}'.format(pi[i],pj[i]))
                print('Mtot = {0}, Mfrag {1}'.format(Mp+Mt,consM))
            elif kind=='cat':
                fragM = np.sum(Marr)
                consM = self.M[pi[i]]+self.M[pj[i]]
                #check mass conservation, remove/add particles if necessary
                effconsM = 0.5*consM # we double everything later
                Mdiff = fragM/effconsM
                print('\ncc between {0} and {1}'.format(pi[i],pj[i]))
                print('Mfrag {0}'.format(consM))
                Marr = Marr/Mdiff
            else:
                raise ValueError('invalid collision kind in assignAM. kind
is'+
                                 '"cat" or "ncat"')
            other.S = np.append(other.S,Lcarr)
            other.S = np.append(other.S,Lcarr)
            other.M = np.append(other.M,Marr)
            other.M = np.append(other.M,Marr)
            AMarr = 0.556945*Lcarr**2/Marr
            other.AM = AMarr
            #assign velocities (magnitude and direction)
            dV = self.DeltaV(AMarr)
            Vcom = (self.M[pi[i]]*self.V[pi[i]]+self.M[pj[i]]*self.V[pj[i
]])/\
                (self.M[pi[i]]+self.M[pj[i]])
            U_i = self.V[pi[i]]-Vcom
```

```
            U_i_norm = np.sqrt(np.sum(U_i**2))
            U_j = self.V[pj[i]]-Vcom
            U_j_norm = np.sqrt(np.sum(U_j**2))
            #calculate internal kinetic energy
            U = self.V[pj[i]]-self.V[pi[i]]
            U_norm = np.linalg.norm(U)
            other.u = U_norm
            U_sq = U_norm**2
            Mtot = self.M[pi[i]]+self.M[pj[i]]
            E_1_int = (1/2)*self.M[pi[i]]*self.M[pj[i]]*U_sq/Mtot
            i_auvec = self.avec(np.tile(U,(len(dV),1))/U_i_norm)
            #n = np.random.randint(2,size=len(dV))
            U_i_new = np.transpose(np.tile(-dV+U_i_norm,(3,1)))*i_auvec
            U_j_new = -np.copy(U_i_new)
            if kind == 'ncat':
                dV_par = self.DeltaV(AMarr_par)
                other.dV = np.append(other.dV,dV_par)
                i_auvec = self.avec(np.array([U_i/U_i_norm]))
                j_auvec = self.avec(np.array([U_j/U_j_norm]))
                U_par_new_i = (dV_par[0]+U_i_norm)*i_auvec
                U_par_new_j = (dV_par[1]+U_j_norm)*j_auvec
                Ei_2_tot = np.sum(Marr*(1/2)*
                                  np.sum((U_i_new+np.tile(Vcom,(Nfrags_tot
,1)))**2,axis=1))
                Ej_2_tot = np.sum(Marr*(1/2)*
                                  np.sum((U_j_new+np.tile(Vcom,(Nfrags_tot
,1)))**2,axis=1))
                Epar_2_tot = (1/2)*(Marr_par[0]*np.sum((U_par_new_i+Vcom)
**2)
                                    +Marr_par[1]*np.sum((U_par_new_j+Vcom)
**2))
                E_2_tot = Ei_2_tot + Ej_2_tot + Epar_2_tot #- (1/2)*Mtot*np
.sum(Vcom**2)
                E_1_tot = (1/2)*self.M[pi[i]]*np.sum(self.V[pi[i]]**2)+\
                    (1/2)*self.M[pj[i]]*np.sum(self.V[pj[i]]**2)
                E_frac = E_2_tot/E_1_tot
                #assign parent velocity, mass and size
                U_i_new = np.append(U_i_new,U_par_new_i,axis=0)
                U_j_new = np.append(U_j_new,U_par_new_j,axis=0)
                other.M = np.append(other.M,Marr_par,axis=0)
                other.S = np.append(other.S,Lcarr_par,axis=0)
                Nfrags_tot += 1 #add parent count
                print('fraction of conserved kinetic energy in ncc = ',
E_frac,end='\n')
            elif kind == 'cat':
                Ei_2_tot = (1/2)*np.sum(Marr*np.sum((U_i_new+np.tile(Vcom,(
len(Marr),1)))**2,axis=1))
                Ej_2_tot = (1/2)*np.sum(Marr*np.sum((U_j_new+np.tile(Vcom,(
len(Marr),1)))**2,axis=1))
                E_2_tot = Ei_2_tot + Ej_2_tot
                E_1_tot = E_1_int + Mtot*np.sum(Vcom**2)/2
                E_frac = E_2_tot/E_1_tot
                print('fraction of conserved kinetic energy in cc = {0:e}'
                      .format(E_frac),'\n')
            if E_frac<=1.0:
```

```
                E_scaling = 1
            else:
                E_scaling = E_frac
            #append fragment data
            U_i_norm = np.sqrt(np.sum(U_i_new**2,axis=1))
            Vfrags_i = (U_i_new + np.tile(Vcom,(Nfrags_tot,1)))/np.sqrt(
E_scaling)
            Vfrags_j = (U_j_new + np.tile(Vcom,(Nfrags_tot,1)))/np.sqrt(
E_scaling)
            Vfrags = np.append(Vfrags_i,Vfrags_j,axis=0)
            Vp_i = np.tile(self.V[pi[i]],(Nfrags_tot,1)) #'p' = parent
satellite
            Vp_j = np.tile(self.V[pj[i]],(Nfrags_tot,1))
            Rp_i = np.tile(self.R[pi[i]],(Nfrags_tot,1))
            Rp_j = np.tile(self.R[pj[i]],(Nfrags_tot,1))
            Vp = np.append(Vp_i,Vp_j,axis=0)
            Rp = np.append(Rp_i,Rp_j,axis=0)
            phi = 2*np.pi*np.random.random(2*Nfrags_tot)
            theta = np.arccos(1-2*np.random.random(2*Nfrags_tot))
            offset = (self.S[pi[i]]+self.S[pj[i]])/2 #average of the sats'
radii
            rand_vec = offset*self.spher_uvec(theta,phi)
            #in case of discrete alg: propagate particles to collision
point and add random starting position
            Rcp = Rp+Vp*tcol[i]+(1/2)*self.grav(Rp)*tcol[i]**2+rand_vec
            Vcp = Vfrags
            #in case of discrete alg: propagate to beginning of next
timestep
            time_iv = dt-tcol[i]
            Rdt,Vdt = self.Verlet(time_iv,R=Rcp,V=Vcp)
            other.R = np.append(other.R,Rdt,axis=0)
            other.V = np.append(other.V,Vdt,axis=0)


    def scatter(self,other,tcol,p_index,dt):
        """Models an elastic collision. Appends the parameters of particles
 involved
        in the collision to the corresponding (position, velocity, radius,
mass)
        fragment arrays. Typically used on relatively small (~Lcmin) and
light
        colliding particles.
        """
        pi = p_index[0].astype(int)
        pj = p_index[1].astype(int)
        n = len(pi)
        print('scattering occurs between {0} and {1}'.format(pi,pj),end='\n
')
        #transforming to center of mass frame
        M_i = np.transpose(np.tile(self.M[pi],(3,1)))
        M_j = np.transpose(np.tile(self.M[pj],(3,1)))
        Vcom = (M_i*self.V[pi]+M_j*self.V[pj])/(M_i+M_j)
        U_i = self.V[pi]-Vcom
        U_j = self.V[pj]-Vcom
        U_i_norm = np.transpose(np.tile(np.sqrt(np.sum(U_i**2,axis=1))
```

```
,(3,1)))
        i_auvec = self.avec(U_i/U_i_norm)
        #determine scattered particle vector
        U_i_new = U_i_norm*i_auvec
        U_j_new = U_j+(M_i/M_j)*(U_i - U_i_new) #momentum conservation
        #calculate new params
        Vfrags_i = U_i_new + Vcom
        Vfrags_j = U_j_new + Vcom
        Vfrags = np.append(Vfrags_i,Vfrags_j,axis=0)
        #other.V = np.append(other.V,Vfrags,axis=0)
        tcol = np.tile(tcol,(2)).T.reshape(2*n,1)
        Vp = np.append(self.V[pi],self.V[pj],axis=0) #'p' for parent satellite
        Rp = np.append(self.R[pi],self.R[pj],axis=0)
        tstep = dt-tcol
        Rcp = Rp + Vp*tcol + self.grav(Rp)*tcol**2 #to collision point
        Rdt = Rcp + Vfrags*tstep + self.grav(Rcp)*tstep**2 #to next timestep
        other.R = np.append(other.R, Rdt,axis=0)
        Vdt = Vfrags + (self.grav(Rcp)+self.grav(Rdt))*tstep/2
        other.V = np.append(other.V, Vdt,axis=0)
        other.M = np.append(other.M,np.append(self.M[pi],self.M[pj]))
        other.S = np.append(other.S,np.append(self.S[pi],self.S[pj]))

    ####### supporting functions ########
    def D_AM(self,Lc,chi):
        """AM distribution for fragments with the characteristic lentgth as
        independent variable. It must be noted that this distribution is only
        valid for Lc-values from 0.01m to 0.08m and from 0.11 to 1m. Hence
        a linear briding function remains to be implemented for the gap
        0.08-0.11m. """
        lc = np.log10(Lc)
        n = len(lc)
        d = len(chi)
        lc0 = lc[lc<-0.959] #<11cm
        n0 = len(lc0)
        lc1 = lc[lc>=-0.959] #>11cm
        n1 = len(lc1)

        #mu0
        mu0 = np.zeros(n0)
        mu0[np.where(lc0<=-1.75)] = -0.3
        mu0[np.where(lc0>-1.25)] = -1.0
        mask1 = np.zeros(n0,np.bool)
        mask2 = np.ones(n0,np.bool)
        mask1[np.where(lc0<-1.0)] = 1
        mask2[np.where(lc0>-1.75)] = 1
        mask= mask1==mask2
        mu0[mask] = -0.3-1.4*(lc0[mask]+1.75)

        #sig0
        sig0 = np.zeros(n0)
        sig0[np.where(lc0<=-3.5)] = 0.2
        sig0[np.where(lc0>=-3.5)] = 0.2+0.1333*(lc0[lc0>-3.5]+3.5)
```

```
        #alfa
        alfa = np.zeros(n1)
        alfa[np.where(lc1<=-0.95)] = 0
        alfa[np.where(lc1>=0.55)] = 1
        mask1 = np.zeros(n1,np.bool)
        mask2 = np.ones(n1,np.bool)
        mask1[np.where(lc1<0.55)] = 1
        mask2[np.where(lc1>-0.95)] = 1
        mask = mask1==mask2
        alfa[mask] = 0.3+0.4*(lc1[mask]+1.2)

        #mu1
        mu1 = np.zeros(n1)
        mu1[np.where(lc1<=-1.1)] = -0.6
        mu1[np.where(lc1>=0)] = -0.95
        mask1 = np.zeros(n1,np.bool)
        mask2 = np.ones(n1,np.bool)
        mask1[np.where(lc1<0)] = 1
        mask2[np.where(lc1>-1.1)] = 1
        mask=mask1==mask2
        mu1[mask] = -0.6-0.318*(lc1[mask]+1.1)

        #sig1
        sig1 = np.zeros(n1)
        sig1[np.where(lc1<=-1.3)] = 0.1
        sig1[np.where(lc1>=-0.3)] = 0.3
        mask1 = np.zeros(n1,np.bool)
        mask2 = np.ones(n1,np.bool)
        mask1[np.where(lc1<-0.3)] = 1
        mask2[np.where(lc1>-1.3)] = 1
        mask=mask1==mask2
        sig1[mask] = 0.1+0.2*(lc1[mask]+1.3)

        #mu2
        mu2 = np.zeros(n1)
        mu2[np.where(lc1<=-0.7)] = -1.2
        mu2[np.where(lc1>=-0.1)] = -2.0
        mask1 = np.zeros(n1,np.bool)
        mask2 = np.ones(n1,np.bool)
        mask1[np.where(lc1<-0.1)] = 1
        mask2[np.where(lc1>-0.7)] = 1
        mask=mask1==mask2
        mu2[mask] = -1.2-1.333*(lc1[mask]+0.7)

        #sig2
        sig2 = np.zeros(n1)
        sig2[np.where(lc1<=-0.5)] = 0.5
        sig2[np.where(lc1>=-0.3)] = 0.3
        mask1 = np.zeros(n1,np.bool)
        mask2 = np.ones(n1,np.bool)
        mask1[np.where(lc1<-0.3)] = 1
        mask2[np.where(lc1>-0.5)] = 1
        mask=mask1==mask2
        sig2[mask] = 0.5-0.2*(lc1[mask]+0.5)
```

```
        #distribution
        D = np.zeros((n,d))
        for i in range(n):
            if i<n0:
                D[i,:] = self.normalcum(mu0[i],sig0[i],chi)
            else:
                k = i-n0
                D[k,:]=alfa[k]*self.normalcum(mu1[k],sig1[k],chi)+\
                    (1-alfa[k])*self.normalcum(mu2[k],sig2[k],chi)
        return D

    def DeltaV(self,AM):
        """Delta V distribution for fragments with the log(AM) as the
    independent
        variable."""
        n = len(AM)
        nu = self.nu
        d = len(nu)
        nu_t = np.tile(nu,(n,1))
        mu = 0.9*np.log(AM)+2.9
        mu = np.transpose(np.tile(mu,(d,1)))
        sig = 0.4
        D = self.normalcum(mu,sig,nu_t)
        samples = np.random.random((n,d))
        idx = self.find_nearest(D,samples)
        dV = 10**nu[idx]
        return dV

    def normalcum(self,mu,sig,x):
        """Normal cumulative distribution function"""
        #N = (1/(sig*(2*np.pi)**0.5))*np.exp(-(1/2)*((x-mu)/sig)**2)
        Ncum = (1/2)*(1+special.erf((x-mu)/(sig*np.sqrt(2))))
        return Ncum

    def find_nearest(self,array, value):
        """finds the values that are closest together in two arrays
        of the same shape (along their second dimension). Used to
        sample AM and Delta-V distributions"""
        idx = np.argmin(np.abs(array - value),axis=1)
        return idx

    def spher_co(self,V):
        """gives the polar and azimuthal angles of a (unit) vector"""
        r = np.sqrt(np.sum(V**2,axis=1))
        phi = np.zeros(len(V[:,0]))
        xg0 = np.where(V[:,0]>0)
        xs0 = np.where(V[:,0]<0)
        xeq0 = np.where(V[:,0]==0)
        phi[xg0] = np.arctan(V[xg0,1]/V[xg0,0])
        phi[xs0] = np.arctan(V[xs0,1]/V[xs0,0])+np.pi
        phi[xeq0] = np.arctan(np.inf)
        theta = np.arccos(V[:,2]/r)
        spherical_params = np.transpose(np.array([theta,phi])) # sats x (r,
    theta,phi)
```

```
774         return spherical_params
775
776     def avec(self,v):
777         """adjusts a unit vector's direction randomly over a spherical cap
    with a
778         maximum (scattering) angle given by alfa"""
779         #convert velocity i particle to spherical coordinates
780         i_spher = self.spher_co(v)
781         #introduce small adjustment to polar and azimuthal angles to model
    scattering
782         rho = np.sqrt(np.random.random(len(i_spher[:,0]))*np.sin(self.alfa
    *2*np.pi/360)**2)
783         nu = np.random.random(len(i_spher[:,0]))*2*np.pi
784         n = np.random.randint(2,size=len(i_spher[:,0]))
785         Dphi = (-1)**(1-n)*np.arcsin(rho*np.cos(nu)) #'randomly' chooses
    sign for phi
786         Dtheta = np.arcsin(np.tan(nu)*np.sin(Dphi))
787         i_auvec = self.spher_uvec(i_spher[:,0]+Dtheta,i_spher[:,1]+Dphi)
788         return i_auvec
789
790     def spher_uvec(self,theta,phi):
791         """calculates a cartesion unit vector from the polar and azimuthal
    angle"""
792         spherical_vec = np.transpose(np.array([np.sin(theta)*np.cos(phi),
793                                                np.sin(theta)*np.sin(phi),
794                                                np.cos(theta)]))
795         return spherical_vec
796
797     ########################################################
798     ################### key methods ########################
799     ########################################################
800
801     @staticmethod
802     def cotrans(Omega,omega,I):
803         """orbital to reference plane coordinate transformation matrix.
    Produces
804         stacked matrices if the arguments are arrays of values."""
805         cO,co,cI = np.cos(Omega),np.cos(omega),np.cos(I)
806         sO,so,sI = np.sin(Omega),np.sin(omega),np.sin(I)
807         if type(Omega) == np.ndarray:
808             n = len(Omega)
809             P = np.zeros((n,3,3))
810             P[:,0,0],P[:,0,1],P[:,0,2] = cO*co-sO*so*cI,-cO*so-sO*co*cI,sO*
    sI
811             P[:,1,0],P[:,1,1],P[:,1,2] = sO*co+cO*so*cI,-sO*so+cO*co*cI,-cO
    *sI
812             P[:,2,0],P[:,2,1],P[:,2,2] = so*sI,co*sI,cI
813         else:
814             P = np.zeros((3,3))
815             P[0,0],P[0,1],P[0,2] = cO*co-sO*so*cI,-cO*so-sO*co*cI,sO*sI
816             P[1,0],P[1,1],P[1,2] = sO*co+cO*so*cI,-sO*so+cO*co*cI,-cO*sI
817             P[2,0],P[2,1],P[2,2] = so*sI,co*sI,cI
818         return P
819
820     def L_vec(self,valtype):
```

```
        """valtype is 'rvm' for cartesian and 'kep' for keplerian."""
        if valtype=='rvm':
            return np.cross(self.R,self.M[:,None]*self.V,axis=1)
        elif valtype=='kep':
            return (self.SMA**2*np.sqrt(1-self.Ecc**2)*self.M*self.MnM)[:,
    None]*\
                np.transpose(np.array([np.sin(self.LAN)*np.sin(self.Inc),
                                       -np.cos(self.LAN)*np.sin(self.Inc),
                                       np.cos(self.Inc)]))

    def RfromE(self,idx=None):
        """returns the reference position vector from the eccentric anomaly
    ."""
        if np.any(idx)==None:
            E = self.E
            a = self.SMA
            e = self.Ecc
            Omega = self.LAN
            omega = self.AgP
            I = self.Inc
        else:
            E = self.E[idx]
            a = self.SMA[idx]
            e = self.Ecc[idx]
            Omega = self.LAN[idx]
            omega = self.AgP[idx]
            I = self.Inc[idx]
        try:
            n = np.shape(E)[0]
            coM = Kessler.cotrans(Omega,omega,I)
            r_orb = np.array([a*(np.cos(E)-e),a*np.sqrt(1-e**2)*np.sin(E),
    np.zeros(n)]).T
            R = np.matmul(coM,r_orb.reshape(n,3,1)).reshape((n,3))
        except IndexError:
            coM = Kessler.cotrans(Omega,omega,I)
            r_orb = np.array([a*(np.cos(E)-e),a*np.sqrt(1-e**2)*np.sin(E)
    ,0]).T
            R = np.matmul(coM,r_orb)
        return R

    def VfromE(self,idx=None):
        """returns the reference velocity vector from the eccentric anomaly
    ."""
        if np.any(idx)==None:
            E = self.E
            a = self.SMA
            e = self.Ecc
            mnm = self.MnM
            Omega = self.LAN
            omega = self.AgP
            I = self.Inc
        else:
            E = self.E[idx]
            a = self.SMA[idx]
            e = self.Ecc[idx]
```

```
            mnm = self.MnM[idx]
            Omega = self.LAN[idx]
            omega = self.AgP[idx]
            I = self.Inc[idx]
        try:
            n = np.shape(E)[0]
            pf = (mnm/(1-e*np.cos(E))).reshape((n,1))
            coM = Kessler.cotrans(Omega,omega,I)
            v_orb = np.array([-a*np.sin(E),a*np.sqrt(1-e**2)*np.cos(E),np.
zeros(n)]).T
            V = np.matmul(coM,(v_orb*pf).reshape(n,3,1)).reshape((n,3))
        except IndexError:
            pf = (mnm/(1-e*np.cos(E)))
            coM = Kessler.cotrans(Omega,omega,I)
            v_orb = np.array([-a*np.sin(E),a*np.sqrt(1-e**2)*np.cos(E),0]).
T
            V = np.matmul(coM,v_orb*pf)
        return V

    def E_series(self):
        """returns (an approximation of) the eccentric anomaly from the
mean anomaly
        and eccentricity."""
        M = self.MnA
        e = self.Ecc
        return M + e*np.sin(M) + (e**2)*(1/2)*np.sin(2*M) +\
                (e**3)*((3/8)*np.sin(3*M)-(1/8)*np.sin(M)) +\
                (e**4)*((1/3)*np.sin(4*M)-(1/6)*np.sin(2*M))

    def e_vec(self,valtype):
        """returns the eccentricity vector. valtype is 'rvm' for cartesian
and
        'kep' for keplerian."""
        if valtype=='rvm':
            L = self.L_vec(valtype)
            r = np.linalg.norm(self.R,axis=1)
            return np.cross(self.V,L,axis=1)/(self.mu*self.M[:,None])- self
.R/r[:,None]
        elif valtype=='kep':
            return self.Ecc[:,None]*Kessler.cotrans(self.LAN, self.AgP,self
.Inc)[:,:,0]

    def excludeFrags(self):
        """returns boolean indices specifing which fragments will stay in
orbit,
        escape or collide with the central mass. Additionally returns the
norm
        of the angular momentum and the eccentricity (vector). This was
done
        to prevent unnecessary calls to Kessler.e_vec and Kessler.L_vec
methods."""
        L = self.L_vec('rvm')
        L_norm = np.linalg.norm(L,axis=1)
        l = L_norm**2/(self.mu*self.M**2)
        Ecc_vec = self.e_vec('rvm')
```

```
916         Ecc = np.linalg.norm(Ecc_vec,axis=1)
917         pncb = l>(1+Ecc)*self.earthR #frags' orbit does not cross central
    body
918         peo = Ecc<1 #frag is in an elliptical orbit
919         pma = np.sum(self.R*self.V,axis=1)>0 #frag is moving awayfrom
    central body
920         ps = peo&pncb
921         pe = np.logical_not(peo)&(pncb|(np.logical_not(pncb)&pma))
922         pccb = np.logical_not(pncb)&(peo|(np.logical_not(peo)&np.
    logical_not(pma)))
923         L_norm = L_norm[ps]
924         self.Ecc_vec = Ecc_vec[ps]
925         self.Ecc = Ecc[ps]
926         self.L = L[ps]
927         self.R = self.R[ps]
928         self.V = self.V[ps]
929         self.M = self.M[ps]
930         self.S = self.S[ps]
931         Npe = len(np.where(pe==True)[0])
932         Npccb = len(np.where(pccb==True)[0])
933         return Npe,Npccb
934
935     def grav(self,R=None):
936         if isinstance(R,type(None)):
937             R = self.R
938         r = np.sum(R**2,axis=1)
939         accel = -self.mu*R/(r[:,None])**(3/2)
940
941         return accel
942
943     def Verlet(self,dt=0,R=None,V=None):
944         """propagates R and V dt seconds forward. If dt is a single value
945         the entire system is propagated using this value. If dt is a list/
946         array of values with the same size as R and V along their first
947         dimension, then each individual pair R_i and V_i is propagated with
948         dt_i."""
949         if isinstance(R,type(None)):
950             R = self.R
951             V = self.V
952             accel_i = self.grav()
953         else:
954             accel_i = self.grav(R=R)
955         if isinstance(dt,np.ndarray) or isinstance(dt,list):
956             dt = dt[:,None]
957         R = R + V*dt + (1/2)*accel_i*dt**2
958         accel_iplus1 = self.grav(R)
959         V = V + (1/2)*(accel_i + accel_iplus1)*dt
960         return R,V
961
962     def physQuant(self):
963         """calculates conserved quantities"""
964         #net angular momentum
965         L_net = np.sum(np.cross(self.R,self.M[:,None]*self.V,axis=1),axis
    =0)
966         #total energy: kinetic + grav.pot.
```

```
        r = np.linalg.norm(self.R,axis=1)
        v2 = np.sum(self.V**2,axis=1)
        E_tot = (1/2)*np.sum(self.M*v2) - self.mu*np.sum(self.M/r)
        return L_net,E_tot

    def updateColCnt(self,Npe,Npccb):
        """keeps track of what collisions happen at what time"""
        try:
            Nsc = self.N_l[0]
            Nncc = self.N_l[1]
            Ncc = self.N_l[2]
            self.N_l = [0,0,0]
        except AttributeError:
            Nsc = Nncc = Ncc = 0
        if Nsc>0:
            self.Nsc = np.append(self.Nsc,Nsc)
            self.Nsct = np.append(self.Nsct,self.t)
        if Nncc>0:
            self.Nncc = np.append(self.Nncc,Nncc)
            self.Nncct = np.append(self.Nncct,self.t)
        if Ncc>0:
            self.Ncc = np.append(self.Ncc,Ncc)
            self.Ncct = np.append(self.Ncct,self.t)
        self.Npe = np.append(self.Npe,Npe)
        self.Npccb = np.append(self.Npccb,Npccb)
        self.Ncols = np.append(self.Ncols,Nsc+Nncc+Ncc)

#%% KEP subclass
from math import floor,ceil
class kepSCM(Kessler):
    """subclass of the Kessler class. Contains the method for collision
    detection
    based on kepler orbits of the satellites."""
    #parameter used to in the method to find the exact collision time of
    two sats
    batchsize = int(1e4)#np.iinfo(np.int32).max

    def __init__(self,dtype=None,args=None,max_t=None,
                 Lcmin=None,alfa=None,col=False,newdata=False,
                 data=None,inclFragCols=True):
        """Either initiliases a satellite class instance (col=False) or a
        fragment instance (col=True)."""
        Kessler.__init__(self,Lcmin,alfa)
        if not col: #initialising sats (col==False) or fragments (True)
            if newdata: #data specifies whether or not data should be
    generated
                if dtype==None: #if sats are initialised, a data type must
    be given
                    raise TypeError("specify data type: ['eph','kep','col
    ','sim','ord','rog','sc','ze']")
                self.load_data(dtype,max_t,args)
                if max_t==None:
                    raise TypeError("specify maximum simulation time: max_t
    ")
            else:
```

```
                if data==None: #if data is already generated it should be
    passed as an argument
                    raise TypeError("no data provided")
                latr = ['Nsats', 'Inc', 'LAN', 'AgP', 'MnA', 'SMA',
                        'Ecc', 'MnM', 'S', 'M', 'max_t']
                for atr in latr:
                    data_atr = getattr(data,atr)
                    setattr(self,atr,data_atr)
            self.t = 0 #current time
            self.t0 = np.zeros(self.Nsats)#creation time
            self.E = self.E_series()
            self.L = self.L_vec('kep')
            self.R = self.RfromE()
            self.V = self.VfromE()
            self.Ecc_vec = self.e_vec('kep')
            self.Nfrags = 0 #number of fragments present in the system
            self.N = self.Nsats
            kepSCM.inclFragCols = inclFragCols
            self.Nfragstot = np.array([0]).astype(np.int32)

    @classmethod
    def fromDataSet(cls,data):
        sats = cls(Lcmin=data.Lcmin,alfa=data.alfa,max_t=data.max_t,data=
    data)
        return sats

    def colList(self,other=None):
        """produces collision list. If other=None, then the collision list
        is made for the entire set of satellites contained within the 'self
    '
        kepSCM class instance. Otherwise, only collisions between the '
    other'
        and 'self' instances are checked, i.e. between frags and satellites
    """
        sat_i = []
        sat_j = []
        R_i = np.zeros((0,3))
        R_j = np.zeros((0,3))
        V_i = np.zeros((0,3))
        V_j = np.zeros((0,3))
        DE_i = np.zeros((0,3))
        DE_j = np.zeros((0,3))
        tcol_ij = []
        if other==None:
            satidx = np.arange(self.Nsats)
            t0i = t0j = self.t0
            Li = Lj = self.L_vec('kep')
            Ecc_veci = Ecc_vecj = self.Ecc_vec
            inst = self
            max_index = self.Nsats-1
        else:#other=sats
            satidx = np.arange(other.Nsats)
            t0i, t0j = self.t0, other.t0
            Li, Lj = self.L, other.L
            Ecc_veci, Ecc_vecj = self.Ecc_vec, other.Ecc_vec
```

```
            inst = other
            max_index = self.Nfrags #only loop over generated frags
        print('\ncreating collision list')
        for i in range(max_index):
            e_i = self.Ecc[i]
            s_i = self.S[i]
            a_i = self.SMA[i]
            if other==None:
                idx = satidx[i+1:]
            else:
                idx = satidx
            e_j = inst.Ecc[idx]
            s_j = inst.S[idx]
            a_j = inst.SMA[idx]
            idx_aj_ge = a_j>=a_i
            idx_aj_s = a_j<a_i
            a_j_ge = a_j[idx_aj_ge]
            a_j_s = a_j[idx_aj_s]
            idx1 = idx[idx_aj_ge]
            idx2 = idx[idx_aj_s]
            per_i = (1-e_i)*a_i
            apo_i  = (1+e_i)*a_i
            per_j = (1-e_j[idx_aj_ge])*a_j_ge
            apo_j = (1+e_j[idx_aj_s])*a_j_s
            c_aj_ge = apo_i-per_j+s_i+s_j[idx_aj_ge]>=0
            c_aj_s = apo_j-per_i+s_i+s_j[idx_aj_s]>=0
            idx = np.append(idx1[c_aj_ge],idx2[c_aj_s])
            if len(idx)==0:
                continue

            #MOID
            rest = len(idx)
            ei_vec = np.tile(Ecc_veci[i],(rest,1))
            ej_vec = Ecc_vecj[idx]
            Kvec = np.cross(np.tile(Li[i],(rest,1)),Lj[idx])
            K = np.linalg.norm(Kvec,axis=1)
            li = np.sum(Li[i]**2)/(self.mu*self.M[i]**2)
            ri_1 = Kvec*li/((K+np.sum(Kvec*ei_vec,axis=1))[:,None])
            ri_2 = Kvec*li/((-K+np.sum(Kvec*ei_vec,axis=1))[:,None])
            vi_1 = self.MOID_vel(Li[i],self.M[i],ei_vec,li,ri_1)
            vi_2 = self.MOID_vel(Li[i],self.M[i],ei_vec,li,ri_2)

            lj = np.sum(Lj[idx]**2,axis=1)/(inst.mu*inst.M[idx]**2)
            rj_1 = Kvec*lj[:,None]/((K+np.sum(Kvec*ej_vec,axis=1))[:,None])
            rj_2 = Kvec*lj[:,None]/((-K+np.sum(Kvec*ej_vec,axis=1))[:,None
    ])
            vj_1 = inst.MOID_vel(Lj[idx],inst.M[idx,None],ej_vec,lj[:,None
    ],rj_1)
            vj_2 = inst.MOID_vel(Lj[idx],inst.M[idx,None],ej_vec,lj[:,None
    ],rj_2)

            #approximate minimal distance
            d1 = rj_1-ri_1
            d2 = rj_2-ri_2
            w1 = np.cross(vi_1,vj_1)
```

```
            w2 = np.cross(vi_2,vj_2)
            w1_norm = np.sum(w1**2,axis=1)
            w2_norm = np.sum(w2**2,axis=1)
            ri_1 = ri_1 + np.sum(d1*np.cross(vj_1,w1)/w1_norm[:,None],axis
    =1)[:,None]*vi_1
            ri_2 = ri_2 + np.sum(d2*np.cross(vj_2,w2)/w2_norm[:,None],axis
    =1)[:,None]*vi_2
            rj_1 = rj_1 + np.sum(d1*np.cross(vi_1,w1)/w1_norm[:,None],axis
    =1)[:,None]*vj_1
            rj_2 = rj_2 + np.sum(d2*np.cross(vi_2,w2)/w2_norm[:,None],axis
    =1)[:,None]*vj_2
            d1 = np.linalg.norm(rj_1-ri_1,axis=1)
            d2 = np.linalg.norm(rj_2-ri_2,axis=1)
            s_j = inst.S[idx]
            idxidx1 = d1<s_i+s_j
            idxidx2 = d2<s_i+s_j
            idx1 = idx[idxidx1]
            idx2 = idx[idxidx2]
            idx = np.append(idx1,idx2)
            numcols = len(idx)
            if numcols==0:
                continue

            #organising data
            ri_1f = np.ndarray.flatten(ri_1[idxidx1])
            ri_2f = np.ndarray.flatten(ri_2[idxidx2])
            rj_1f = np.ndarray.flatten(rj_1[idxidx1])
            rj_2f = np.ndarray.flatten(rj_2[idxidx2])
            vi_1f = np.ndarray.flatten(vi_1[idxidx1])
            vi_2f = np.ndarray.flatten(vi_2[idxidx2])
            vj_1f = np.ndarray.flatten(vj_1[idxidx1])
            vj_2f = np.ndarray.flatten(vj_2[idxidx2])
            w1f = np.ndarray.flatten(w1[idxidx1])
            w2f = np.ndarray.flatten(w2[idxidx2])
            ri = np.append(ri_1f,ri_2f)
            rj = np.append(rj_1f,rj_2f)
            vi = np.append(vi_1f,vi_2f)
            vj = np.append(vj_1f,vj_2f)
            w = np.append(w1f,w2f)
            ri = ri.reshape(numcols,3)
            rj = rj.reshape(numcols,3)
            vi = vi.reshape(numcols,3)
            vj = vj.reshape(numcols,3)
            w = w.reshape(numcols,3)
            d = np.append(d1[idxidx1],d2[idxidx2])

            #time of first crossing
            rest = len(idx)
            ei_vec = np.tile(Ecc_veci[i],(rest,1))
            ej_vec = Ecc_vecj[idx]
            a_j = inst.SMA[idx]
            e_j = inst.Ecc[idx]
            b_i = a_i*np.sqrt(1-e_i**2)
            b_j = a_j*np.sqrt(1-e_j**2)
            omega_i = self.MnM[i]
```

```
            omega_j = inst.MnM[idx]
            ri0 = np.tile(self.RfromE(i),(rest,1))
            rj0 = inst.RfromE(idx)
            ri_norm = np.linalg.norm(ri,axis=1)
            rj_norm = np.linalg.norm(rj,axis=1)
            ri_dot_ri0 = np.sum(ri*ri0,axis=1)
            rj_dot_rj0 = np.sum(rj*rj0,axis=1)
            ri0_dot_ei = np.sum(ri0*ei_vec,axis=1)
            rj0_dot_ej = np.sum(rj0*ej_vec,axis=1)
            ri_dot_ei = np.sum(ri*ei_vec,axis=1)
            rj_dot_ej = np.sum(rj*ej_vec,axis=1)
            ripri0_dot_ei = np.sum((ri+ri0)*ei_vec,axis=1)
            rjprj0_dot_ej = np.sum((rj+rj0)*ej_vec,axis=1)
            vi_dot_ri0 = np.sum(vi*ri0,axis=1)
            vj_dot_rj0 = np.sum(vj*rj0,axis=1)
            vi_dot_ei = np.sum(vi*ei_vec,axis=1)
            vj_dot_ej = np.sum(vj*ej_vec,axis=1)
            x_i = ri_dot_ri0/b_i**2+ripri0_dot_ei/a_i-ri_dot_ei*ri0_dot_ei/
    b_i**2+e_i**2
            y_i = -(ri_norm/(a_i*omega_i))*(vi_dot_ri0/b_i**2+vi_dot_ei/a_i
    -vi_dot_ei*ri0_dot_ei/b_i**2)
            x_j = rj_dot_rj0/b_j**2+rjprj0_dot_ej/a_j-rj_dot_ej*rj0_dot_ej/
    b_j**2+e_j**2
            y_j = -(rj_norm/(a_j*omega_j))*(vj_dot_rj0/b_j**2+vj_dot_ej/a_j
    -vj_dot_ej*rj0_dot_ej/b_j**2)
            dE_i = kepSCM.arctan2(y_i,x_i)
            dE_j = kepSCM.arctan2(y_j,x_j)
            L_i = np.tile(Li[i],(rest,1))
            tcross_i = t0i[i] + dE_i/omega_i - np.sum((np.cross(ei_vec,ri-
    ri0,axis=1)/\
                                                       (1-e_i**2))*L_i/(self.
    mu*self.M[i]),axis=1)
            tcross_j = t0j[idx] + dE_j/omega_j - np.sum((np.cross(ej_vec,rj
    -rj0,axis=1)/\
                                                         (1-e_j[:,None]**2))*
    Lj[idx]/\
                                                         (inst.mu*inst.M[idx,
    None]),axis=1)

            #deterministic collision time
            N = len(idx)
            s_j = inst.S[idx]
            T_i = 2*np.pi/self.MnM[i]
            T_j = 2*np.pi/inst.MnM[idx]
            R_i = np.append(R_i,ri,axis=0)
            R_j = np.append(R_j,rj,axis=0)
            V_i = np.append(V_i,vi,axis=0)
            V_j = np.append(V_j,vj,axis=0)
            DE_i = np.append(DE_i,dE_i)
            DE_j = np.append(DE_j,dE_j)
            usq = np.sum((vj-vi)**2,axis=1)
            wsq = np.sum(w**2,axis=1)
            Dt = np.abs(tcross_i-tcross_j)
            delta = np.sqrt(usq*((s_i+s_j)**2-d**2)/wsq)/Dt
            for n,j in enumerate(idx):
```

```
                delt = delta[n]
                if Dt[n]==0:
                    tcol_ij.append(tcross_i[n])
                    sat_i.append(i)
                    sat_j.append(j)
                    continue
                q0,q1 = (T_i/Dt)[n],(T_j/Dt)[n]
                k0= 1
                k1 = 0
                it = 0
                while True and it<=5:
                    print('\rKEP-->tot: {0:.2f}%, '.format((i+1)*100/
    max_index)+
                          'sat i {0}: {1:.2f}%, '.format(i,(n+1)*100/N)+
                          'sat j {0}: iter = {1}'.format(j,it),end=' ')
                    a0 = floor(q0/q1)
                    q2 = q0-a0*q1
                    if q2==0:
                        y = 0
                        x = np.arange(int((1-delt)/q0),floor((1+delt)/q0)
    +1)[0]
                        if x>=0 and x*q0-y*q1 < delt+1:
                            k = k0*x-k1*y
                        else:
                            k = np.inf
                        break
                    k2 = k0-a0*k1
                    a1 = floor(q1/q2)
                    q3 = q1-a1*q2
                    if q3==0:
                        y = 0
                        x = np.arange(int((1-delt)/q1),floor((1+delt)/q1)
    +1)[0]
                        if x>=0 and x*q1-y*q2 < delt+1:
                            k = k1*x-k2*y
                        else:
                            k = np.inf
                        break
                    k3 = k1-a1*k2
                    ub = ceil((1+delt)*a0/q2)+1
                    batchn = ub//self.batchsize
                    residual = ub%self.batchsize
                    for i1 in range(batchn+1):
                        if i1<batchn:
                            y = np.arange(0,self.batchsize,dtype=np.int32)
                        else:
                            y = np.arange(0,residual,dtype=np.int32)
                        x = np.ceil((q1*y+1-delt)/q0)
                        difs = x*q0-y*q1
                        sol = difs<delt+1+i1*self.batchsize*(q0-q1)
                        solbool = np.any(sol)
                        if solbool:
                            sol = np.min(np.where(sol==True)[0])
                            y = y[sol].astype(np.int64)
                            x = x[sol].astype(np.int64)
```

```
                            k = (x+i1*self.batchsize)*k0-(y+i1*self.
batchsize)*k1
                            break
                    if solbool:
                        break
                    q0,q1 = q2,q3
                    k0 = k2
                    k1 = k3
                    it += 1
                    if it == 5:
                        k = np.inf
                        break
                tc = tcross_i[n]+k*T_i
                tcol_ij.append(tc)
                sat_i.append(i)
                sat_j.append(j)

        #sorting lists
        inf_idx = np.where(np.array(tcol_ij)==np.float('+inf'))[0]
        self.tcol = np.delete(tcol_ij,inf_idx,axis=0)
        self.si = np.delete(sat_i,inf_idx,axis=0).astype(np.int32)
        self.sj = np.delete(sat_j,inf_idx,axis=0).astype(np.int32)
        self.Ri = np.delete(R_i,inf_idx,axis=0)
        self.Rj = np.delete(R_j,inf_idx,axis=0)
        self.Vi = np.delete(V_i,inf_idx,axis=0)
        self.Vj = np.delete(V_j,inf_idx,axis=0)
        self.DEi = np.delete(DE_i,inf_idx,axis=0)
        self.DEj = np.delete(DE_j,inf_idx,axis=0)
        self.sortList()
        print('\nfinished collision list\n')

    def sortList(self,merge=False,other0=None,other1=None):
        """sorts collision list as well as corresponding colliding sat.
index,
        -position and velocity lists. Either sorts one list or mergesorts
        two sorted lists"""
        if merge==False:
            s_idx = self.tcol.argsort()
        else:
            self.tcol = np.append(other0.tcol,other1.tcol)
            self.si = np.append(other0.si,other1.si)
            self.sj = np.append(other0.sj,other1.sj)
            self.Ri = np.append(other0.Ri,other1.Ri,axis=0)
            self.Rj = np.append(other0.Rj,other1.Rj,axis=0)
            self.Vi = np.append(other0.Vi,other1.Vi,axis=0)
            self.Vj = np.append(other0.Vj,other1.Vj,axis=0)
            self.DEi = np.append(other0.DEi,other1.DEi)
            self.DEj = np.append(other0.DEj,other1.DEj)
            s_idx = self.tcol.argsort(kind='mergesort')

        self.tcol = self.tcol[s_idx]
        self.si = self.si[s_idx]
        self.sj = self.sj[s_idx]
        self.Ri = self.Ri[s_idx]
        self.Rj = self.Rj[s_idx]
```

```
        self.Vi = self.Vi[s_idx]
        self.Vj = self.Vj[s_idx]
        self.DEi = self.DEi[s_idx]
        self.DEj = self.DEj[s_idx]
        max_idx = self.tcol<self.max_t
        self.tcol = self.tcol[max_idx]
        self.si = self.si[max_idx].astype(np.int64)
        self.sj = self.sj[max_idx].astype(np.int64)
        self.Ri = self.Ri[max_idx]
        self.Rj = self.Rj[max_idx]
        self.Vi = self.Vi[max_idx]
        self.Vj = self.Vj[max_idx]
        self.DEi = self.DEi[max_idx]
        self.DEj = self.DEj[max_idx]

    def updateSats(self):
        """propagates system to desired time t"""
        self.t = self.tcol[0] #next collision
        idxi_i = self.si[0]
        idxi_j = self.sj[0]
        self.MnA[idxi_i] += self.MnM[idxi_i]*self.tcol[0]
        self.MnA[idxi_j] += self.MnM[idxi_j]*self.tcol[0]
        self.E[idxi_i] += self.DEi[0]
        self.E[idxi_j] += self.DEj[0]
        #even though calculating actual orbit positions is more realistic
    ...
        # self.R[idxi_i] = self.RfromE(idxi_i)
        # self.R[idxi_j] = self.RfromE(idxi_j)
        #their approximate collision points will satisfy:
        self.R[idxi_i] = self.Ri[0]
        self.R[idxi_j] = self.Rj[0]
        self.V[idxi_i] = self.VfromE(idxi_i)
        self.V[idxi_j] = self.VfromE(idxi_j)

    @classmethod
    def cols(cls,self):
        """creates new instance of kepSCM class for the fragments of a
    collision,
        excludes frags that escape or collide with central mass, calculates
        fragments' orbital elements and creates the fragment collision list
    ."""
        #create fragment instance of kepSCM class
        frags = cls(col=True,Lcmin=self.Lcmin,alfa=self.alfa)
        frags.max_t = self.max_t
        #this algorithm is 'continuous' and only 1 collision occurs at a
    time
        #so tcol=[0] (still has to be an array or list) and dt=0
        dr_i = np.linalg.norm(self.Ri[0]-self.R[self.si[0]])
        dr_j = np.linalg.norm(self.Rj[0]-self.R[self.sj[0]])
        if dr_i>20 or dr_j>20:
            raise ValueError('dr_i, dr_j = {0:e}, {1:e}'
                             .format(dr_i,dr_j))
        #generate fragments (scattered particles are also considered
    fragments)
        self.collision(frags,np.array([0]),0,alg='kep')
```

```
1370        #exclude fragments that escape (hyperbolic orbit) and/or collide
    with central mass
1371        Npe,Npccb = frags.excludeFrags()
1372        self.updateColCnt(Npe,Npccb)
1373        L_norm = np.linalg.norm(frags.L,axis=1)
1374        Nfrags = len(frags.S)
1375        h_sq = (L_norm/frags.M)**2 #angular momentum per unit mass
1376        frags.SMA = h_sq/(self.mu*(1-frags.Ecc**2))
1377        #b = frags.SMA*np.sqrt(1-frags.Ecc**2) #semi-minor axis
1378        frags.MnM = np.sqrt(frags.mu/frags.SMA**3)
1379        frags.Inc = np.arccos(frags.L[:,2]/L_norm) # acos(l3/|L|)
1380        frags.AgP = np.arcsin(frags.Ecc_vec[:,2]/(frags.Ecc*np.sin(frags.
    Inc))) #asin(e3/(e*sin(I)))
1381        frags.LAN = np.arcsin(frags.L[:,0]/(L_norm*np.sin(frags.Inc))) #
    asin(l1/(|L|*sin(I)))
1382        frags.E = frags.EfromR()
1383        frags.MnA = frags.E-frags.Ecc*np.sin(frags.E) #Kepler's equation
1384        frags.t0 = np.array([self.t]*Nfrags)#creation time of frags
1385        frags.Nfrags = Nfrags
1386        if self.inclFragCols:
1387            self.Nfrags = Nfrags
1388        else:
1389            self.Nfrags += Nfrags
1390        #delete collided particles from collision lists and other arrays
1391        #and correct particle counts
1392        self.deleteIndices()
1393        #create collision list for fragments
1394        frags.colList(self)
1395        return frags,Npe,Npccb
1396
1397    def deleteIndices(self):
1398        """deletes (indices of) collided satellites from all data arrays
1399        (collision list) and adjusts satellite and fragment count."""
1400        #remove performed collision and any future collision involving
    collided sats
1401        didx_i,didx_j = self.si[0],self.sj[0]
1402        didxs_i = np.append(np.where(self.si==didx_i)[0],
1403                            np.where(self.si==didx_j)[0])
1404        didxs_j = np.append(np.where(self.sj==didx_j)[0],
1405                            np.where(self.sj==didx_i)[0])
1406        didxs = np.union1d(didxs_i,didxs_j)
1407        #since the two colliding particles are deleted
1408        #every collision index larger than the collided indices
1409        #should be shifted too (downwards, following the smaller list of
    partciles)
1410        pili = self.si>didx_i
1411        pilj = self.si>didx_j
1412        pjli = self.sj>didx_i
1413        pjlj = self.sj>didx_j
1414        self.si[pili] = self.si[pili]-1
1415        self.si[pilj] = self.si[pilj]-1
1416        self.sj[pjli] = self.sj[pjli]-1
1417        self.sj[pjlj] = self.sj[pjlj]-1
1418        latr = ['si','sj','Ri','Rj','Vi','Vj','DEi','DEj','tcol']
1419        for atr in latr:
```

```
            atrval = getattr(self, atr)
            setattr(self,atr,np.delete(atrval,didxs,axis=0))
        #remove collided particles from all arrays
        didx = np.append(didx_i,didx_j)
        latr = ['L','R','V','Inc','LAN','AgP','MnA','SMA','Ecc','MnM','S','M',
                't0','E','Ecc_vec']
        for atr in latr:
            atrval = getattr(self, atr)
            setattr(self,atr,np.delete(atrval,didx,axis=0))
        #correct number of particles
        c1 = didx_i < self.Nsats
        c2 = didx_j < self.Nsats
        #substract only collided sats from total sat count
        self.Nsats = self.Nsats-c1*1-c2*1 #True*number=number & False*number=0
        #idem for frags
        self.Nfragstot[-1] = self.Nfragstot[-1]-(not c1)*1-(not c2)*1
        if not self.inclFragCols:
            self.Nfrags = self.Nfrags-(not c1)*1-(not c2)*1

    @classmethod
    def mergeFrags(cls,self,other): #self=fragments,other=satellites
        """merges satellite and fragment data arrays and collision lists.
        returns new merged class object"""
        merge = cls(col=True,Lcmin=other.Lcmin,alfa=other.alfa,max_t=other.max_t)
        merge.t = other.t
        merge.max_t = other.max_t
        #correct colliding indices of frags
        if other.inclFragCols:
            merge.Nsats = other.Nsats + other.Nfrags #include frag-frag cols
            merge.N = other.Nsats + other.Nfrags
            merge.Nfrags = 0 #reset frag count
            self.si = (self.si + other.Nsats).astype(np.int32)
            merge.Nfragstot = np.append(other.Nfragstot,other.Nfragstot[-1]
                                        + other.Nfrags)
        else:
            merge.Nsats = other.Nsats #update sat count
            merge.Nfrags = self.Nfrags + other.Nfrags #increase frag count
            merge.N = other.Nsats + other.Nfrags + self.Nfrags
            self.si = (self.si + other.Nsats + other.Nfrags).astype(np.int32)
        merge.sortList(merge=True,other0=self,other1=other)
        latr = ['L','R','V','Inc','LAN','AgP','MnA','SMA','Ecc','MnM','S','M',
                't0','E','Ecc_vec']
        for atr in latr:
            fragms = getattr(self,atr)
            satls = getattr(other,atr)
            atrval = np.append(satls,fragms,axis=0)
            setattr(merge,atr,atrval)
        merge.Nsc = other.Nsc
        merge.Nncc = other.Nncc
```

```
        merge.Ncc = other.Ncc
        merge.Nsct = other.Nsct
        merge.Nncct = other.Nncct
        merge.Ncct = other.Ncct
        merge.Npe = other.Npe
        merge.Npccb = other.Npccb
        merge.Ncols = other.Ncols
        return merge

    def MOID_vel(self,L,M,E_vec,l,r):
        r_norm = np.linalg.norm(r,axis=1)
        return np.cross(L/(M*l),E_vec+r/r_norm[:,None])

    @staticmethod
    def arctan2(y,x):
        """wrapper of the numpy.arctan2 method, which returns angles in the range [0,2pi)."""
        if len(np.shape(y))==0:
            if y>=0:
                return np.arctan2(y,x)
            else:
                return 2*np.pi+np.arctan2(y,x)
        else:
            out = np.arctan2(y,x)
            out[y<0] += 2*np.pi
            return out

    def EfromR(self):
        """returns the eccentric anomaly of a satellite given its position
        vector, eccentricity longitude of ascending node, argument of periapsis
        and inclination."""
        coM = Kessler.cotrans(self.LAN,self.AgP,self.Inc)
        coM_inv = np.linalg.inv(coM)
        n = np.shape(coM_inv)[0]
        r_orb = np.matmul(coM_inv,self.R.reshape(n,3,1)).reshape((n,3))
        F = kepSCM.arctan2(r_orb[:,1],r_orb[:,0]) #true anomaly
        #we have:
        #cosE =  (self.Ecc+np.cos(F))/(1+self.Ecc*np.cos(F))
        #sinE = (np.sqrt(1-self.Ecc**2)*np.sin(F))/(1+self.Ecc*np.cos(F))
        #But we leave out the denominator, as we devide it out in the arctangent anyway
        cosE = self.Ecc+np.cos(F)
        sinE = np.sqrt(1-self.Ecc**2)*np.sin(F)
        E = kepSCM.arctan2(sinE,cosE)
        return E

#%% KDT subclass
from sklearn.neighbors import KDTree
class kdtSCM(Kessler):
    """Subclass of the Kessler class. Contains a method for collision detection
    in an arbitrary system of satellites using a k-d tree"""

    def __init__(self,dtype=None,args=None,max_t=None,k=None,
```

```
                Lcmin=None,alfa=None,dt=None,col=False,
                newdata=False,data=None):
        Kessler.__init__(self,Lcmin,alfa)
        if not col:
            if newdata: #data specifies whether or not data should be
    generated
                if dtype==None: #if sats are initialised, a data type must
    be given
                    raise TypeError("specify data type: ['eph','kep','col
    ','sim','ord','rog','sc','ze']")
                self.load_data(dtype,max_t,args)
            else:
                if data==None: #if data is already generated it should be
    passed as an argument
                    raise TypeError("no data provided")
                latr = ['Nsats', 'Inc', 'LAN', 'AgP', 'MnA', 'SMA',
                        'Ecc', 'MnM', 'S', 'M', 'max_t', 'Lcmin', 'alfa']
                for atr in latr:
                    data_atr = getattr(data,atr)
                    setattr(self,atr,data_atr)
            self.t = 0 #current time
            self.E = self.E_series()
            self.R = self.RfromE()
            self.V = self.VfromE()
            del self.E # we have no other use for E in this algorithm
            if dt == None:
                self.dt = 10
            else:
                self.dt = dt
            if k == None:
                self.k = 6
            else:
                self.k = k
            self.tree = KDTree(self.R,leaf_size=30)
            self.Li,self.Ei = self.physQuant()
            self.Li_normsq = np.sum(self.Li**2)

    @classmethod
    def fromDataSet(cls,data):
        sats = cls(Lcmin=data.Lcmin,alfa=data.alfa,data=data)
        return sats

    @classmethod
    def simulate(cls,self,single=False):
        """main method of the kdtSCM class. Consists of a discrete
    algorithm
        with timestep dt. During each iteration the k-d tree is queried for
     NNs
        usingthe kepSCM.getNNs method. Motion of these NNss is then
    linearised
        in order to calcute their collision time and check if a collision
    occurs
        in the current timestep. Only the earliest collisions are performed
     in
        case any satellite occurs in multiple collisions (or mulitple
```

```
    collisions
        involve the same satellite). Then, all the colliding sats are
    passed to
        the Kessler.collision method, after whichthe entire system of
    satellites
        is propagated to the next timestep using the Kessler.Verlet method.
        Only now the fragments are appended to the existing set of
    satellites.
        If no satellites collide during any timestep, the system is
    propagated
        to the next timestep without performing any additional steps. After
        propagation, a new k-d tree is constructed using the
        sklearn.neighbors.KDTree method."""
        while self.t<self.max_t:#cumcol<num_cols:
            NNs = self.getNNs()
            n = len(NNs[:,0])
            k = self.k
            index = np.transpose(np.tile(NNs[:,0],(k-1,1)))
            S_i = np.transpose(np.tile(self.S[NNs[:,0]],(k-1,1)))
            R_i = np.transpose(np.tile(self.R[NNs[:,0]],(k-1,1,1)),(1,0,2))
            V_i = np.transpose(np.tile(self.V[NNs[:,0]],(k-1,1,1)),(1,0,2))
            S_j = self.S[NNs[:,1]]
            R_j = self.R[NNs[:,1]]
            V_j = self.V[NNs[:,1]]
            for i in range(2,self.k):
                S_j = np.append(S_j,self.S[NNs[:,i]],axis=0)
                R_j = np.append(R_j,self.R[NNs[:,i]],axis=1)
                V_j = np.append(V_j,self.V[NNs[:,i]],axis=1)
            S_j = np.reshape(S_j,(n,k-1))
            R_j = np.reshape(R_j,(n,k-1,3))
            V_j = np.reshape(V_j,(n,k-1,3))
            Ssum = (S_i+S_j)**2
            D = R_j-R_i
            U = V_j-V_i
            u_dot_d = np.sum(U*D,axis=2)
            u_norm = np.sum(U**2,axis=2)
            zidx = u_norm==0
            self_idx = np.where(NNs[:,1:k]==index)
            u_norm[zidx] = 1
            Tcol = -u_dot_d/u_norm
            Tcol[zidx] = np.inf
            Tcol[self_idx] = np.inf
            a1 = Tcol>0
            a2 = Tcol<=self.dt
            a = a1&a2
            if not np.any(a):
                self.updateColCnt(0,0)
                self.t += self.dt
                self.tree = KDTree(self.R,leaf_size=30)
                self.R,self.V = self.Verlet(dt=self.dt)
                self.printProgress()
                if not single: continue
                else: break
            else:
                u_norm = np.sum(U**2,axis=2)
```

```
                d_cross_u_norm = np.sum(np.cross(U,D,axis=2)**2,axis=2)
                b = d_cross_u_norm < Ssum*u_norm
                c = a&b
                if not np.any(c):
                    self.updateColCnt(0,0)
                    self.t += self.dt
                    self.R,self.V = self.Verlet(dt=self.dt)
                    self.tree = KDTree(self.R,leaf_size=30)
                    self.printProgress()
                    if not single: continue
                    else: break
                else:
                    idxi,idxj = np.where(c==True)
                    tcol = Tcol[idxi,idxj]
                    pi = NNs[idxi,0]
                    pj = NNs[idxi,idxj+1]
                    pi_s,i_inv,counts_i = np.unique(pi,return_inverse=True,
                                                    return_counts=True)
                    pj_s,j_inv,counts_j = np.unique(pj,return_inverse=True,
                                                    return_counts=True)
                    n,m = len(pi_s),len(pj_s)
                    Tcolmat = np.zeros((n,m)).astype(np.float32) #saves
    memory
                    Tcolmat[i_inv,j_inv] = Tcol[idxi,idxj]
                    Tcolmat[np.where(Tcolmat==0)] = np.inf
                    #ensure earliest collision for any pair of colliding
    sats
                    pi,pj,tcol = np.array([]),np.array([]),np.array([])
                    while np.any(Tcolmat!=np.inf):
                        Tcolmin = np.transpose(np.tile(np.min(Tcolmat,axis
    =1),(m,1)))
                        Tcolmat[np.where(Tcolmat>Tcolmin)] = np.inf
                        Tcolmin = np.tile(np.min(Tcolmat,axis=0),(n,1))
                        Tcolmat[np.where(Tcolmat>Tcolmin)] = np.inf
                        i_inv,j_inv = np.where((Tcolmat>=0)&(Tcolmat<self.
    dt))
                        pi = np.append(pi,pi_s[i_inv]).astype(np.int32)
                        pj = np.append(pj,pj_s[j_inv]).astype(np.int32)
                        self.tcol = np.append(tcol,Tcolmat[i_inv,j_inv])
                        Tcolmat = np.delete(Tcolmat,i_inv,axis=0)
                        Tcolmat = np.delete(Tcolmat,j_inv,axis=1)
                    self.si,self.sj = pi,pj
                    frags = cls(col=True)
                    self.collision(frags,self.tcol,self.dt)
                    #exclude fragments that escape (hyperbolic orbit) and/
    or collide with central mass
                    Npe,Npccb = frags.excludeFrags()
                    self.updateColCnt(Npe,Npccb)
                    self.t += self.dt
                    self.R,self.V = self.Verlet(dt=self.dt)
                    self.mergeFrags(frags)
                    self.Nsats = len(self.S)
                    self.tree = KDTree(self.R,leaf_size=30)
                    self.printProgress()
                    if not single: continue
```

```
                    else: break

    def getNNs(self):
        """retrieves the k-nearest neighbours (NNs) of all the satellites from the current
        k-d tree. The corresponding distances are retrieved as well. These are used to
        exclude any pair of satellites between which no collision is possible."""
        dmax = 2*np.max(np.linalg.norm(self.V,axis=1))*self.dt
        dist,nns = self.tree.query(self.R,k=self.k,return_distance=True, dualtree=False)
        N = self.Nsats
        nns[np.where(dist>dmax)] = N #set particles larger than dmax to 'infinity' index
        nns = np.delete(nns,np.where(nns[:,1]==N),axis=0) #delete particles with no nns
        nns[np.where(nns==N)] = nns[np.where(nns==N)[0],0] #set N+1 idx to own idx
        index = np.transpose(np.tile(nns[:,0],(self.k,1)))
        nns[np.where(nns<index)] = nns[np.where(nns<index)[0],0] #avoid double counting
        return nns

    def mergeFrags(self,other): #other=frags
        """deletes collided satellites and appends fragment data and
        checks for collisions with the earth."""
        R = np.delete(self.R,self.delarr,axis=0)
        V = np.delete(self.V,self.delarr,axis=0)
        M = np.delete(self.M,self.delarr)
        S = np.delete(self.S,self.delarr)
        R = np.append(R,other.R,axis=0)
        V = np.append(V,other.V,axis=0)
        M = np.append(M,other.M)
        S = np.append(S,other.S)
        learthR = np.linalg.norm(R,axis=1)>self.earthR
        self.R = R[learthR]
        self.V = V[learthR]
        self.M = M[learthR]
        self.S = S[learthR]

    def printProgress(self):
        """print progress and conserved quantities"""
        L,E = self.physQuant()
        L_dot_Li = np.sum(L*self.Li)
        print(str('\rKDT--> t = {0:.3f} min, L/L_init = {1:.12f},'
                  +'E/E_init = {2:.12f}, k = {3:.0f} '
                  +'#sats: {4} '
                  +'#cc: {5} '
                  +'#ncc: {6} '
                  +'#sc: {7} ')
              .format(self.t/self.minute,L_dot_Li/self.Li_normsq,E/self.Ei,
                      self.k,self.Nsats,np.sum(self.Ncc),np.sum(self.Nncc)
                      ,np.sum(self.Nsc)),end='')
```

```
1713
1714 #####old method for selecting earliest collisions (too strict)#########
1715 # #ensure only one collision occurs for any sat
1716 # intersect,comi,comj = np.intersect1d(pi,pj,assume_unique=True,
1717 #                                      return_indices=True)
1718 # if intersect.size > 0:
1719 # #at least one sat occurs in two distinct collisions
1720 # #which is impossible (as far as this model is concerned)
1721 #     #occurence of the satellite in the i'th index is the
1722 #     #collision that happens first if
1723 #     c = (tcol[comi]<tcol[comj]).astype(int)
1724 #     #picks out the correct indices
1725 #     col_idx = np.array([comj,comi])[c,np.arange(len(comi))]
1726 #     com = np.append(comi,comj)s
1727 #     #delete all common indices and append only the correct ones
1728 #     pi = np.append(np.delete(pi,com),pi[col_idx])
1729 #     pj = np.append(np.delete(pj,com),pj[col_idx])
1730 #     tcol = np.append(np.delete(tcol,com),tcol[col_idx])
```