EPiC Series in Computing

Volume 61, 2019, Pages 170–182

ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems

Efficient n-to-n Collision Detection for Space Debris using 4D AABB Trees*

Stanley Bak[1] and Kerianne Hobbs[2]

[1] Safe Sky Analytics
Manlius, NY, USA
stanleybak@gmail.com

[2] Air Force Research Laboratory, Dayton, OH, USA
kerianne.hobbs@us.af.mil
Georgia Institute of Technology
kerianne@gatech.edu

Abstract

Collision detection algorithms are used in aerospace, swarm robotics, automotive, video gaming, dynamics simulation and other domains. As many applications of collision detection run online, timing requirements are imposed on the algorithm runtime: algorithms must, at a minimum, keep up with the passage of time. Even offline reachability computation can be slowed down by the process of safety checking when n is large and the specification is n-to-n collision avoidance. In practice, this places a limit on the number of objects, n, that can be concurrently tracked or verified. In this paper, we present an improved method for efficient object tracking and collision detection, based on a modified version of the axis-aligned bounding-box (AABB) tree data structure. We consider 4D AABB Trees, where a time dimension is added to the usual three space dimensions, in order to enable per-object time steps when checking for collisions in space-time. We evaluate the approach on a space debris collision benchmark, demonstrating efficient checking beyond the full catalog of $n = 16848$ space objects made public by the U.S. Strategic Command on www.space-track.org.

1 Introduction

The prediction of collisions between large numbers of objects in 3D space is important in many domains such as interactive video games or the verification of cyber-physical systems. In set-based verification methods for dynamical systems, most effort and runtime typically consists of the reachability computation step. However, the process of safety checking can become a bottleneck when the specification is that no objects collide with any other objects (the n-to-n collision detection problem), and the number of objects, n, is large.

*DISTRIBUTION A. Approved for public release; Distribution unlimited. Approval AFRL PA case number 88ABW-2018-4991, 05 OCT 2018.

This was the focus of a recently-proposed benchmark on space debris collision detection [10]. In that work, a method based on axis-aligned bounding box (AABB) trees was proposed and could verify the absence of collisions faster than real-time for around 4000 objects. The key insight there was to perform checking using global variable time steps, versus performing a check at each multiple of a fixed time step. In this work, we modify the approach to also permit variable time steps on a per-object basis. This requires adding a time dimension to the AABB tree data structure, resulting in a method that works on 4D AABB trees. The new result is evaluated on real space debris data made public by the U.S. Strategic Command. In particular, we demonstrate faster-than-real-time tracking is possible with up to about 65000 objects.

Note that in this paper we focus on the discrete-time version of the problem, with a fixed minimum time step, although extensions to continuous time are possible. Further, although applications use collision detection online, we measure the offline performance of the algorithm, and compare the simulated time versus the computation time. Our algorithm is geared towards finding the earliest collision, as this is sufficient for the verification problem and applications may want to react to this event as it may impact future object trajectories. An extended technical report with additional details and a proof of algorithm correctness is available online [1].

2 AABB Trees

Axis-Aligned Bounding Box (AABB) trees [2] are a type of bounding volume hierarchy used in collision detection methods. Bounded volume hierarchies are trees where the leaf nodes represent volumes of individual objects, and inner nodes of the tree correspond to sets that contain every object and set below them in the tree. An AABB tree is a binary tree, where each object and set is represented by an axis-aligned (non-rotated) bounding box. Thus, the root of the tree will be a bounding box that contains every object, and the parent of two leaf nodes will simply be the bounding box containing the two objects. AABB trees can also maintain balance as new objects are inserted and updated using surface area heuristics, so that query operations remain efficient. We do not plan to review all the details of AABB trees here, although detailed introductions are available elsewhere[1].

The three operations on AABB trees we will use are:

- `insert` - AABB trees store a world object and an associated box, usually the object's occupancy region. Insert operations take such pairs and add them to the AABB tree, possibly performing tree rotations to maintain balance for efficient queries.
- `query` - Tree queries check if a given box intersects with any previously-inserted box in the tree, returning a list of colliding objects. Queries are performed by starting at the tree root and recursively checking if the query box intersects the left or right child. In the ideal case, half the objects can be discarded at each layer, leading to an efficient $\mathcal{O}(\log(n))$ lookup time. This may not be possible if the query box is large or the tree balance is poor.
- `update` - Objects in an AABB tree can have their bounds updated within an existing tree. This can prevent the need to construct a new tree at every time step.

Tree balancing operations are used to ensure $\mathcal{O}(log(n))$ complexity for each of the operations. Checking all n objects at a fixed time step therefore takes $\mathcal{O}(n \log(n))$ time. With a time bound

[1] An accessible introduction to AABB trees is available online at `www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision-detection`.

of T and a time step of δ, the total runtime for constructing a tree at each step and checking for collisions is $\mathcal{O}(\frac{T}{\delta} n \log(n))$. In earlier work [10], we improved upon this by considering larger time steps, essentially reducing the $\frac{T}{\delta}$ term.

An AABB tree is used for collision detection, and so each object should have an associated 3D bounding box in space, at each point in time. We call this box an *occupancy region*. The occupancy region is all points within a cube with faces at a fixed distance r from a center point of $\mathsf{pos}(t)$, as defined by the the occupancy region function.

Definition 1 (**Occupancy Region**)**.** *An object's* occupancy region *at a fixed time is the set of 3D space the object takes up at that time. This set is defined by* $\mathit{occ}(\boldsymbol{w}, t) = \{x \in \mathbb{R}^3 : \|\boldsymbol{w}.\mathit{pos}(t) - x\|_\infty \leq \boldsymbol{w}.r\}$*, where* $\boldsymbol{w}.\mathit{pos}(t)$ *is the object's position over time, and* $\boldsymbol{w}.r$ *is the object's radius.*

3 Collision Detection with 4D AABB Trees

We propose the use of 4D AABB trees for efficiently solving the collision detection problem. Like traditional AABB trees, 4D AABB trees include the usual three space dimensions, but they also have an additional time dimension. Collisions are detected when two objects overlap in both space and time. The 4D nature of the tree allows time to be tracked per-object, so that variable time steps, computed on a per-object basis, can be performed.

3.1 Interval Occupancy Regions

A modified version of occupancy regions is needed that accepts intervals of time as an input, and returns a box which bounds the states at all times within the time interval. The function computing this region should be exact when the time interval is a single instant, but can otherwise provide an overapproximation.

Definition 2 (Interval Occupancy Region)**.** *An object's* interval occupancy region *at some interval of time* $\bar{\boldsymbol{t}} = [t_{\min}, t_{\max}]$ *(with* $t_{\min} \leq t_{\max}$*) is a superset of the 3D space the object occupies at all times within the interval. This set is defined by the function* $\mathit{occ\text{-}int}(\boldsymbol{w}, \bar{\boldsymbol{t}}) \supseteq \{x \in \mathbb{R}^3 : \; x \in \mathit{occ}(\boldsymbol{w}, t) \; \wedge \; t \in \bar{\boldsymbol{t}}\}$*.*

Interval occupancy functions have two additional properties which must hold:

- **Property 1**: occ-int should return the exact occupancy region when $\bar{\boldsymbol{t}}$ is a single instant in time (when $t_{\min} = t_{\max}$). Formally, $\mathsf{occ\text{-}int}(\mathbf{w}, [t, t]) = \mathsf{occ}(\mathbf{w}, t)$.
- **Property 2**: If a smaller time interval is used as an input, the output should also be smaller or equal. Formally, If $\bar{\boldsymbol{t}}_1 \subseteq \bar{\boldsymbol{t}}_2$ then $\mathsf{occ\text{-}int}(\mathbf{w}, \bar{\boldsymbol{t}}_1) \subseteq \mathsf{occ\text{-}int}(\mathbf{w}, \bar{\boldsymbol{t}}_2)$.

The proposed algorithm requires tracking time separately for each world object, and so we augment the state with this information. For each world object $\mathbf{w}$, we add a time interval $\bar{\boldsymbol{t}}$ which we refer to as a whole using $\mathbf{w}.\bar{\boldsymbol{t}}$, or by directly naming to the individual time values $\mathbf{w}.t_{\min}$ or $\mathbf{w}.t_{\max}$. This allows us to define a 4D occupancy region function.

Definition 3 (4D Occupancy Region)**.** *An object* $\boldsymbol{w}$ *has a* 4D occupancy region*, which is a 4D box constructed using the 3D space provided by its interval occupancy region function at its current time interval* $\boldsymbol{w}.\bar{\boldsymbol{t}}$*, along with the 1D interval defined by the time dimension* $\boldsymbol{w}.\bar{\boldsymbol{t}}$*. This box is defined by the function* $\mathit{occ\text{-}4d}(\boldsymbol{w}) = \mathit{occ\text{-}int}(\boldsymbol{w}, \boldsymbol{w}.\bar{\boldsymbol{t}}) \times \boldsymbol{w}.\bar{\boldsymbol{t}}$*, where* $\times$ *is the Cartesian product of two sets.*

The user must provide the occ-int from Definition 2, which takes in an *interval* of time and provides a box containing the occupancy regions for all times within the interval. This requires computing how objects move in a given time interval.

The simplest way to compute interval occupancy regions in the discrete time setting would be to loop over all time instants and compute the smallest box that contains the occupancy region at every point in time. Unfortunately, this strategy would make the runtime of a call to occ-int depend on the length of the interval of time passed in, and would reduce the performance of the approach.

If a closed-form solution of the $\mathsf{pos}(t)$ function is available, interval analysis methods [11] can be used to compute interval occupancy regions. Interval arithmetic methods can be used to provide bounds on functions where the arguments are each intervals. For example, a function $f(x, y) = 2x + y$ can be used with an interval arithmetic library to compute that when $x \in [1, 2]$ and $y \in [2, 4]$, $f(x, y) \in [4, 8]$. In our case, if we have a formula for $\mathsf{pos}(t)$, we could provide it to an interval arithmetic library along with any time interval to produce a bound on $pos([t_{\min}, t_{\max}])$. Note that this approach may provide an overapproximation of the function's true minimum and maximum, due to the well known dependency problem with interval arithmetic. For example, directly evaluating $f(x) = x * x$ in interval arithmetic, with $x \in [-1, 1]$, gives the overapproximation $[-1, 1]$, whereas the true bounds are $[0, 1]$. Accuracy is improved when smaller input intervals are provided and, in most cases, in the limit the output will approach the true minimum and maximum. Interval arithmetic evaluation scales independently of the sizes of the input intervals, and so the computation time is $\mathcal{O}(1)$.

Many times, however, for physics simulations closed-form solutions may not be available. Instead the dynamics of the system may be expressed with ordinary differential equations (ODEs). These can be numerically simulated using a method such as Runge Kutta to provide the value of $\mathsf{pos}(t)$.

If the ODE describing the movement of the object is a function of a single variable, interval methods can still be used to compute the interval occupancy region. This is done by simulating the system for the amount of time at the beginning and start of the desired time interval to compute the minimum and maximum values of the single variable. Note that in this case, although the movement ODE involves a single variable, a conversion from the single variable to 3D space can be an arbitrary closed-form function using other constant variables or properties associated with each object. For example, in our evaluation, the single variable of an orbiting object will be the true anomaly (the angular position in orbit), which gets converted to 3D space using other, fixed, orbital elements that are unique to each object. Interval arithmetic is used to convert from the bounds on the single variable to bounds in 3D space.

Formally, if we have $\dot{x} = f(x)$ (where $f : \mathbb{R} \to \mathbb{R}$) for some continuous Lipschitz function f (in order to guarantee existence and uniqueness of solutions) with solution $g(t)$ (that can be obtained through numerical simulation), and $\mathsf{pos}(t) = h(g(x))$ (where $h : \mathbb{R} \to \mathbb{R}^3$), then we can compute bounds on $\mathsf{pos}([t_{\min}, t_{\max}])$ by (i) using a numerical simulation to simulate $\dot{x} = f(x)$ to get the values of $g(t_{\min})$ and $g(t_{\max})$, (ii) performing an interval evaluation of $h([g(x_{\min}), g(x_{\max})])$.

Although more complex than the closed-form solution method, if the numerical simulation time is fast, and $x_{\min}$ and $x_{\max}$ can be looked up efficiently, the interval evaluation part of the computation remains $\mathcal{O}(1)$. Note that if the ODE is a function of multiple variables, this approach is not applicable, and more general, reachability methods [4] may be necessary to provide the bounds computed by occ-int. The reason numerical simulation is permitted for single-variable systems is that $g(t_{\min})$ and $g(t_{\max})$ bound $g(t)$ at all intermediate times.

Algorithm 1 4D AABB Tree Collision Detection

Input: $\mathbf{w}_1 \dots \mathbf{w}_n$, T, δ
Output: First collision $(\mathbf{w}, \mathbf{v}, t)$ or None

```
1: tree ← AABBTree()                                  ▷ Creates empty AABB tree
2: ℓ ← initializeTree(tree, w_1 ... w_n)
3: if ℓ is not None then
4:     return (ℓ[0], ℓ[1], 0)
5: while true do
6:     v ← getSmallestMaxTimeObject(tree)
7:     if v.t_max ≥ T then
8:         break
9:     advanceTime(v, T, δ)
10:    tree.update(v, occ-4d(v))
11:    u = resolveCollisions(v, tree, δ)
12:    if u is not None then
13:        return (v, u, v.t_min)
14: return None
```

3.2 Main Algorithm

The 4D AABB tree collision detection procedure is described in Algorithm 1. The algorithm takes in n objects, a time bound T, minimum time step δ, and returns the earliest collision (a pair of objects and a time), or `None` if collisions are impossible. The algorithm first inserts all objects at the minimum time step into the ABBB tree using procedure `initializeTree`, and then the main loop on line 5 advances time for a single object at a time. As objects are advanced, their object-specific time-step grows exponentially, and then collision checking is done using the 4D AABB Tree. The time step may then be reduced to increase accuracy, if a potential collision is detected. For space reasons, a proof that the loop terminates as well as algorithm correctness is provided in the technical report [1].

First, however, we detail each of the procedures used by the high-level algorithm. The algorithm uses the auxiliary procedures `initializeTree`, `getSmallestTimeObject`, `advanceTime`, and `resolveCollisions`, which are described in Sections 3.3 to 3.6.

3.3 Procedure `initializeTree`

The `initializeTree` procedure is used to initially insert all world objects into the AABB tree. If every object has been inserted into the tree with minimum time $t_{\min} = 0$, such that no two boxes in the tree overlap, then None is returned. If this is impossible, because there are objects that initially collide, then a pair of colliding objects should be returned instead.

The simplest implementation, shown in Algorithm 2 sets each object's time interval to be exactly 0, and inserts the object into the tree. It is possible to improve the efficiency of the algorithm by instead inserting objects with a larger interval of time, even up to the full time range $[0, T]$. However, increasing the initial time interval could result in the detection of overlapping 4D regions due to overapproximation (false positives), and then have a need to reduce time bounds of some of the objects to eliminate the overlap. In addition to this extra complexity, the initial performance is a one-time cost, so efficiency improvements are not critical to the overall performance.

Algorithm 2 Procedure `initializeTree`

Input: `tree`, $\mathbf{w}_1 \ldots \mathbf{w}_n$
Output: Pair of colliding objects at time 0 $(\mathbf{w}, \mathbf{v})$ or None

```
1: for i in 1 to n do
2:     w_i.t̄ ← [0, 0]
3:     box = occ-4d(w_i)
4:     ℓ = tree.query(box)                    ▷ query returns a list
5:     if ℓ is not empty then
6:         return (w_i, ℓ[0])
7:     tree.insert(w_i, box)
8: return None
```

Algorithm 3 Procedure `advanceTime`

Input: $\mathbf{w}, T, \delta$

```
1: prev_steps ← (w.t_max − w.t_min)/δ
2: next_steps ← 1
3: if prev_steps > 0 then
4:     next_steps ← 2 ∗ prev_steps
5: w.t_min ← w.t_max + δ
6: w.t_max ← w.t_min + next_steps ∗ δ
7: if w.t_max > T then
8:     w.t_max ← T
```

3.4 Procedure `getSmallestMaxTimeObject`

The `getSmallestTimeObject` procedure returns the object with the smallest $t_{\max}$ of all the objects in the AABB tree, with ties broken arbitrarily. For efficiency, rather than iterating over all the objects in the tree, the implementation should use a priority queue implemented with something like a binary heap. This priority queue will need to be updated every time $t_{\max}$ is changed for any object, and whenever an object is removed from the AABB tree. In the implementation, this can be done elegantly by overriding the AABB tree methods `insert`, `update`, and `remove` to both update the 4D AABB tree and as well as update the object's $t_{\max}$ in the priority queue. The entire `getSmallestMaxTimeObject` procedure, then, consists of simply returning the object at the front of the priority queue. For this reason, we do not include its pseudocode here.

3.5 Procedure `advanceTime`

The `advanceTime` procedure updates the minimum time for a single object $\mathbf{v}$ to be one time step δ beyond its previous maximum time. This is the only place where $t_{\min}$ is changed.

There is a choice of what to use for $t_{\max}$. In our implementation, we double the length of the object's time interval when `advanceTime` is called, up to the time bound. The result is that if the time interval is never decreased in `resolveCollisions`, which happens when the current object's 4D box does not intersect with any other objects, then the object will only be iterated over a logarithmic number of times with respect to the number of time steps $\frac{T}{\delta}$, in the `while` loop on line 5 in the high-level algorithm. This is in contrast to the brute force method or basic AABB tree approach, where every time step requires some processing for every object, which

Algorithm 4 Procedure `resolveCollisions`

Input: Object with potential collisions $\mathbf{v}$, `tree`, δ
Output: Object colliding with $\mathbf{v}$ at time $\mathbf{v}.t_{\min}$ or None

```
1: ℓ ← tree.queryObject(v)
2: while ℓ is not empty do
3:     for y in ℓ do
4:         if occ-4d(y) ∩ occ-4d(v) then
5:             steps_y ← (y.t_max − y.t_min)/δ
6:             steps_v ← (v.t_max − v.t_min)/δ
7:             if steps_y = 0 and steps_v = 0 then
8:                 return y
9:             else if y.t_min < v.t_min then
10:                y.t_min ← v.t_min
11:                tree.update(y, occ-4d(y))
12:            else if steps_v ≤ steps_y then
13:                new_steps ← floor(steps_y / 2)
14:                y.t_max ← y.t_min + new_steps ∗ δ
15:                tree.update(y, occ-4d(y))
16:            else
17:                new_steps ← floor(steps_v / 2)
18:                v.t_max ← v.t_min + new_steps ∗ δ
19:                tree.update(v, occ-4d(v))
20:    ℓ ← tree.queryObject(v)
21: return None
```

results in a linear scaling with respect to the number of time steps $\frac{T}{\delta}$. In practice, the 4D boxes may overlap when large time intervals are used, and so the actual scalability will be somewhere between linear and logarithmic, depending on (i) the distances between world objects, (ii) how fast their position changes, and (iii) the accuracy of occ-int. Generally speaking, objects that are far from others will use larger time intervals, and therefore require less processing. The proposed procedure is shown in Algorithm 3.

3.6 Procedure `resolveCollisions`

The `resolveCollisions` procedure takes in a single world object $\mathbf{w}$ whose occupancy region may intersect with other objects in the AABB tree, and reduces the $t_{\max}$ of the passed-in object and/or the intersecting objects, in order to eliminate the intersection. When this function is called, only $\mathbf{w}$ may intersect with other objects; other pairs of objects in the tree do not intersect. If reducing $t_{\max}$ is impossible for both objects, because their time intervals are both a single instant in time, then a collision is detected and the colliding object is returned. If no collision is detected, then None is returned and we are certain no 4D boxes in the tree intersect.

The procedure uses a `queryObject` method on the AABB tree, which is shorthand for calling `query` on the tree using the object's interval occupancy region box, and returning a list of objects with an intersection, excluding the object passed as input to `queryObject`. The detailed procedure is shown in Algorithm 4.

The procedure first queries to AABB tree to see if any objects overlap with $\mathbf{v}$. If this is the case, the loop of the function essentially decreases the time intervals of either $\mathbf{v}$ or the colliding

object $\mathbf{y}$ and updates the tree with the 4D boxes corresponding to the new time bounds. At the end of each iteration of the loop, on line 20, the tree is queried again to see if all collisions with $\mathbf{v}$ have been resolved. Since the time bounds keep decreasing every time through the loop, either at some point collisions will no longer exist and the loop will exit, or the time intervals will be reduced to a single point and a collision still exists. In the latter case, a collision actually exists, and the colliding object is returned (line 8).

The time interval of one of the potentially colliding objects is always decreased when the 4D boxes intersect. There are three ways this can happen controlled by the three branches on lines 9, 12, and 16. These are, correspondingly, increasing $\mathbf{y}$'s minimum time, decreasing $\mathbf{y}$'s maximum time, and decreasing $\mathbf{v}$'s maximum time.

4 Space Debris Benchmark

We evaluate the proposed collision detection approach on a space debris collision detection application. The U.S. Space Surveillance Network tracks around 23000 objects larger than 10 cm in orbit around Earth, although it is estimated there are hundreds of thousands of objects between 1 cm and 10 cm, and possibly millions smaller than 1cm [9]. Due to the high velocities involved (an object in low-earth orbit moves at 7800 m/s or about 28000 km/hr), even collisions with small objects can cause catastrophic damage, and further magnify the problem by creating additional space debris. In February 2009, the Iridium 33 communications satellite collided with the defunct Russian military satellite Cosmos-2251, creating roughly 2100 new pieces of debris larger than 10 cm [13]. Although small amounts of atmospheric drag may eventually deorbit objects so they burn up in the Earth's atmosphere, this process can take dozens to hundreds of years, depending on the orbit. A further concern, popularized as "Kesler Syndrome" [14], is that debris-creating spacecraft collisions could cascade, resulting in an exponential increase in the amount of space debris, and threatening space access. In response to increased launches and interest in space based services, the White House released Space Policy Directive-3, National Space Traffic Management Policy [16], which discusses collision detection and avoidance extensively saying: "Timely warning of potential collisions is essential to preserving the safety of space activities for all."

Space debris collision detection is an on-going problem with real-time requirements. The prediction speed must exceed the time needed to run the computation, in order to be able to predict collisions and warn satellite operators to make orbital adjustments.

A description of Kepler orbital dynamics is available in previous work [10] and will only be briefly reviewed here. An orbiting object's position and velocity can be uniquely determined from a set of six *orbital elements*: the semi-major axis a, eccentricity e, true anomaly ν, inclination i, right ascension of the ascending node Ω, and argument of perigee ω. When objects are only under the influence of the gravity of Earth (Kepler dynamics) only one of these parameters changes, the true anomaly ν, which is like the angular position of the object in its elliptical orbit. The value of ν evolves according to the differential equation

$$\dot{\nu} = \sqrt{\frac{\mu}{(a(1-e^2))^3}}(1 + e\cos\nu)^2 \tag{1}$$

where μ is the geocentric gravitational parameter, and the other orbital parameters in the ODE are constant. A nonlinear transformation consisting of three rotations involving i, Ω and ω then convert ν to a point in 3D space in the ECI reference frame, where collisions can be checked.

This setup meets the requirements for computing occ-int described in Section 3.1. The value of differential equation of ν is a function of a single variable, and a nonlinear transformation of

ν provides the position. In the formalism of Section 3.1, Equation 1 is f, the solution to the ODE $\nu(t)$ is g, and the transformation to the ECI frame is h.

We also consider decomposition and parallelization of the orbital collision detection problem. We use the minimum altitude (perigee) and maximum altitude (apogee) as a way to statically check if collisions are possible. If the apogee of one object is less than the perigee of another, with small adjustments to take into account the radii of the objects, then no collision is possible. With this approach, we can partition the original problem with a large n into p smaller problems that can be analyzed in parallel. There is some replication in this approach, as certain satellites may be in multiple partitions. For space reasons, we omit the details of this approach here, and instead refer interested readers to the technical report [1].

We evaluate our approach using orbital elements taken from real objects, using two-line element (TLE) sets made public by the U.S. Strategic Command on `www.space-track.org`. We used the full catalog of objects larger than 10 cm taken from 3 April 2018, which initially had $n = 16840$ objects. In order to be able to scale up or down n for evaluation, when less objects were desired we simply dropped the remaining objects in the database. When we need to evaluate with more objects, we randomly combined the orbital elements from existing objects. Both of these approaches maintain the expected distributions of the full data set, which is clustered around low-earth orbit and not uniform across space.

Upon checking for collisions, we detected three objects in the database that seemed to be initially colliding, caused by identical TLE values. These were two Soyuz and one resupply spacecraft docked at the International Space Station, and thus in an identical orbit. We manually removed two of these from the database, in order to permit performance evaluation of the algorithms in the case of no collisions, making the full catalog for our evaluation $n = 16838$ objects.

We evaluate the overall performance of the 4D AABB method on three platforms: an *embedded* system with 1GB RAM and an Intel Atom CPU (1.33 GHz), a *laptop* computer with 16 GB RAM and an Intel i5-5300U CPU (2.30 GHz), and a more powerful *workstation* with 32 GB RAM and an Intel Xeon 8124M CPU (3.00 GHz). All measurements were performed on Ubuntu Linux 16.04. We also measured performance with and without partitioning. For the partitioned versions, we set the number of partitions p to be equal to the number of physical cores on the laptop and workstation platforms, and used $p = 2$ for the embedded processor, since memory was a limiting resource there.

Although applications would need to predict for hours to days of orbit time, the crucial factor is the *ratio* of the computed orbit time to the collision detection algorithm runtime. For ease of experimentation, we fixed the orbit time to a smaller 600 seconds (10 minutes). Since we were evaluating performance, during measurement we ensured no collisions occurred by using a sufficiently small object radius. However, because objects in LEO move at 7800 m/s, a small time step is necessary to prevent the tunneling problem with discrete time collision checking. We evaluated with a time step of $\delta = 10^{-4}$, so that LEO objects move about 0.78 m per step. This is reasonable, as we envision collision boxes would be at least 10m to account for sensor errors.

A log-log plot of the results is shown in Figure 1, with the full table of results given in Table 1. Both the laptop and workstation platforms are able to propagate the full catalog of objects and perform collision detection faster than real-time. The partitioned workstation version is about 6x faster than real-time, and can scale to about 65000 objects while remaining faster than real-time. Even the embedded platform works well with smaller numbers of objects, with $n < 1000$ being tens of times faster than real-time. This is encouraging since embedded platforms like swarm robotics could use the approach for on-board collision maneuver prediction.

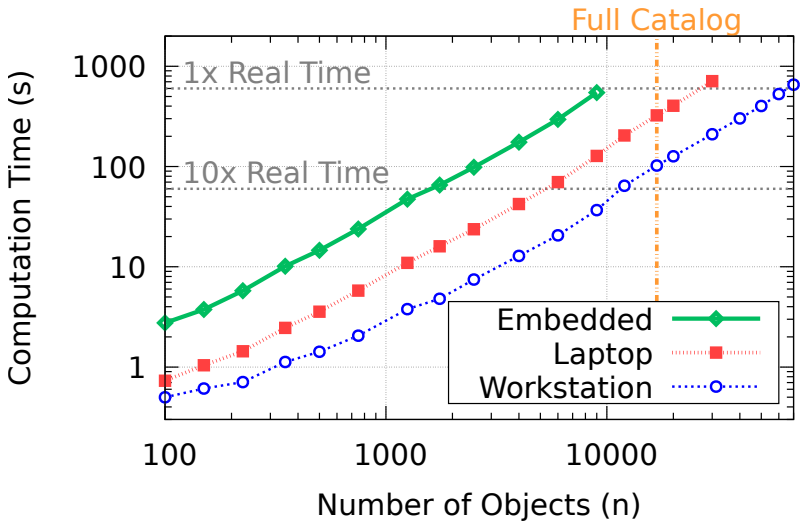

Figure 1: Runtimes for the partitioned versions on each platform from Table 1.

Due to the large number of steps (10^4 steps for each second of orbit time), using the brute force method and even the basic AABB tree method is completely infeasible for real-time collision detection for this problem. On the laptop platform with $n = 100$, the basic AABB approach would need about 12 hours to finish checking all 10 minutes of orbit time.

5 Related Work

Extensive surveys are available of collision detection methods for graphics and physics applications [12]. This work falls in the category of space-time intersection methods, but rather than extruding volumes in 4D, we compute 4D bounding boxes of the space-time paths of objects. Splitting time to improve accuracy has been used before in swept volume collision detection [5].

We compare our work with static interference detection methods, such as the original AABB approach [2]. Modifications of the basic AABB method attempt to reuse boxes across time steps, usually by fattening the boxes by some percentage. The amount of bloating is a problem-specific parameter, and such methods would be unlikely to work well for the orbit debris scenario, where velocities are high relative to the object radius. We are similar in a sense to adaptive time-step methods [6], except that we have per-object time steps. Other than AABB trees, other data structures are available for collision detection, such as oriented bounding-box (OBB) trees [8]. In this case, boxes can be arbitrarily rotated, and fast collision checking is done using the separating axis theorem [7]. This can be advantageous since rotated boxes may have less overapproximation than axis-aligned boxes, so that tree query operations will become more efficient. Since we use interval arithmetic [11] to reason between time steps, AABB trees seem better suited for storing the regions of space occupied by objects in intervals of time. Interval arithmetic methods have also been used in combination with OBB trees to provide continuous-time collision detection [15]. Sphere hierarchies have also been considered [3].

In this work, we used Kepler dynamics to propagate orbits. Although the focus of this application was to evaluate the collision detection methods presented, more accurate methods such as SGP4 [17] also exist for orbits which take into account J2 perturbation due to the Earth

Table 1: Runtime (seconds) to check 600 seconds of orbit time.

Objs n	Embedded ($p = 1$)	Laptop ($p = 1$)	Workstation ($p = 1$)	Embedded ($p = 2$)	Laptop ($p = 2$)	Workstation ($p = 8$)
100	3.0	0.7	0.5	2.8	0.7	0.5
150	4.7	1.2	0.7	3.7	1.0	0.6
225	7.4	1.8	1.1	5.8	1.4	0.7
350	12.6	3.1	1.9	10.1	2.5	1.1
500	19.0	4.5	2.9	14.6	3.6	1.4
750	30.8	7.3	4.6	23.8	5.8	2.1
1250	61.7	14.6	9.4	47.3	10.9	3.8
1750	91.4	21.9	14.0	65.4	16.0	4.8
2500	143.2	33.8	21.8	98.4	23.7	7.5
4000	250.2	64.4	38.1	175.0	42.2	12.8
6000	425.4	98.9	64.8	295.6	69.9	20.6
9000	739.3	171.7	113.8	545.7	127.7	36.7
12000	1185.6	268.2	178.7	-	204.2	64.3
16838	-	422.9	281.1	-	323.4	102.2
30000	-	967.7	622.2	-	710.5	209.8
50000	-	-	-	-	-	401.9
70000	-	-	-	-	-	654.7

not being a perfect sphere, atmospheric drag, the gravity of the Moon and Sun, solar radiation, and other effects. Since these will modify more than one orbital element, other methods would be needed to compute occ-int, such as those based on reachability [4]. Still, even our Kepler approach could be considered as a broad-phase pass to detect potentially-colliding objects for further analysis with the more accurate propagation methods.

In the original benchmark proposal [10], satellites were initialized using uniform random values for their orbital elements rather than TLE sets, and only a regular 3D AABB tree approach was evaluated, although global variable time steps were considered. The 4D AABB approach is more efficient because it permits each object to have an individual time step.

6 Conclusion

In this work we presented a 4D AABB tree and search space decomposition approach to improve the efficiency of the collision detection, which can be used directly or as part of the safety checking step in a verification engine. Using 4D AABB trees is more efficient because it can attempt to take large time steps on an individual object basis, and only reduce to smaller ones when potential collisions are detected. We evaluated the approach on a space debris collision detection benchmark from ARCH 2018, and demonstrated the possibility of online performance for up to about 65000 orbiting objects, about an order of magnitude improvement from the method suggested in the benchmark proposal.

Acknowledgments

Effort sponsored in whole or in part by the Air Force Research Laboratory, USAF, under Memorandum of Understanding/Partnership Intermediary Agreement No. FA8650-18-3-9325. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory.

References

[1] Stanley Bak and Kerianne Hobbs. Efficient n-to-n collision detection for space debris using 4D AABB trees (extended report). *arXiv:1901.10475*, 2019. https://arxiv.org/abs/1901.10475.

[2] Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of graphics tools*, 2(4):1–13, 1997.

[3] Angel P Del Pobil, Miguel A Serna, and Juan Llovet. A new representation for collision avoidance and detection. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 246–251. IEEE, 1992.

[4] Parasara Sridhar Duggirala, Chuchu Fan, Matthew Potok, Bolun Qi, Sayan Mitra, Mahesh Viswanathan, Stanley Bak, Sergiy Bogomolov, Taylor T. Johnson, Luan Viet Nguyen, Christian Schilling, Andrew Sogokon, Hoang-Dung Tran, and Weiming Xiang. Tutorial: Software tools for hybrid systems verification, transformation, and synthesis: C2e2, hyst, and tulip. In *Proceedings of the Multi-Conference on Systems and Control*. IEEE, 2016.

[5] A Foisy and V Hayward. A safe swept volume method for collision detection. In *International Symposium of Robotics Research*, pages 62–68, 1994.

[6] Elmer G Gilbert and SM Hong. A new algorithm for detecting the collision of moving objects. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 8–14. IEEE, 1989.

[7] Stefan Gottschalk. Separating axis theorem. Technical report, Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[8] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM, 1996.

[9] Steven A. Hildreth and Allison Arnold. Threats to u.s. national security interests in space: Orbital debris mitigation and removal. Technical report, Congressional Research Service, 101 Independence Ave, SE,Washington,DC,20540, 1 2014.

[10] Kerianne Hobbs, Peter Heidlauf, Alexander Collins, and Stanley Bak. Space debris collision detection using reachability. In *5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, EPiC Series in Computing. EasyChair, 2018.

[11] Luc Jaulin. *Applied interval analysis: with examples in parameter and state estimation, robust control and robotics*, volume 1. Springer Science & Business Media, 2001.

[12] Pablo Jiménez, Federico Thomas, and Carme Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[13] TS Kelso et al. Analysis of the iridium 33-cosmos 2251 collision. *Advances in the Astronautical Sciences*, 135(2):1099–1112, 2009.

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

[15] Stéphane Redon, Abderrahmane Kheddar, and Sabine Coquillart. Fast continuous collision detection between rigid bodies. In *Computer graphics forum*, volume 21, pages 279–287. Wiley Online Library, 2002.

[16] Donald J. Trump. Space Policy Directive-3, National Space Traffic Management Policy. https://www.whitehouse.gov/presidential-actions/space-policy-directive-3-national-space-traffic-management-policy/, June 18, 2018. Presidential Memoranda.

[17] David Vallado and Paul Crawford. Sgp4 orbit determination. In *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, page 6770, 2008.