Let's say I am want to write an access generation algorithm for satellites in python. Right now, what I have, is an access generation algorithm for ground telescope to satellite accesses. I want something similar, but for satellite to satellite accesses. This is what I'm working with so far:

I have some data classes such as:

```
@dataclass
class AccessPoint:
    epoch: datetime
    azimuth_radians: float
    elevation_radians: float
    range_meters: float

@dataclass
class Access:
    sat: EarthSatellite
    observer: Topos
    start: datetime
    end: datetime
    uuid: UUID
    culmination: Optional[datetime] = None
    access_points: Optional[list[AccessPoint]] = None
    generated_time: Optional[datetime] = None
```

and then the actual access generation code:

```
```

```
def _event_finder(
    cfg: Config,
    sat: EarthSatellite,
    start: datetime,
    end: datetime,
    gs: Entity #right now, this is a telescope
) -> Tuple[List[Access], List[Access]]:
    # Defaults for everything
    min_az_rad = 0.0
    max_az_rad = 2 * pi
    min_el_rad = radians(cfg.min_access_elevation_degrees)
    max_el_rad = pi / 2
    fan_params = FanGeometry()

    # Default to telescope geometry (most permissive)
    is_telescope = True

    if "mission_planning_constraints" in gs.prototype_extensions.extensions:
        constraints = PlanningConstraints()
        constraints.ParseFromString(
            gs.prototype_extensions.extensions["mission_planning_constraints"].value
        )

        if constraints.HasField("sensor_pointing_limits"):
            pointing_limits = constraints.sensor_pointing_limits
            min_az_rad = pointing_limits.minimum_azimuth_radians
            max_az_rad = pointing_limits.maximum_azimuth_radians
            min_el_rad = pointing_limits.minimum_elevation_radians
            max_el_rad = pointing_limits.maximum_elevation_radians

        if constraints.HasField("fan_geometry"):
            is_telescope = False
            fan_params = constraints.fan_geometry

    else:
        return [], []

    loc = gs.location.position
    lat = loc.latitude_degrees
    lon = loc.longitude_degrees
    alt = loc.altitude_hae_meters.value

    observer = Topos(
        latitude_degrees=lat,
        longitude_degrees=lon,
        elevation_m=alt
    )

    if cfg.accesses_calculate_day_night:
        night_periods = calculate_night_periods(observer, start, end)
    else:
        night_periods = [(start, end)]

    all_accesses: List[Access] = []
    illumination_filtered_accesses: List[Access] = []
    filtered_accesses: List[Access] = []

    for start_night, end_night in night_periods:
        events = sat.find_events(
        events = sat.find_events(
            observer,
            ts.from_datetime(start_night),
            ts.from_datetime(end_night),
            sensor_elevation_effective(degrees(min_el_rad))
        )

        # Convert events to accesses
        if events:
            accesses = events_to_accesses(
                _process_events(events),
                start_night,
                end_night,
                sat,
                observer
            )
            all_accesses.extend(accesses)
        else:
            return [], []

    # Filter on illumination geometry
    if cfg.accesses_calculate_sunlit_accesses:
        for access in all_accesses:
            sunlit_periods = calculate_sunlit_periods(
                sat, access.start, access.end, 0.3
            )
            adjusted_accesses = adjust_access_for_illumination(
                access, sunlit_periods
            )
            illumination_filtered_accesses.extend(adjusted_accesses)
    else:
        illumination_filtered_accesses = all_accesses[:]  #
```

copy

```
    # Filter on cutout geometry
    if len(illumination_filtered_accesses) > 0 and
cfg.accesses_calculate_sensor_pointing_limits:
        if is_telescope:
            for access in illumination_filtered_accesses:
                # Only perform azimuth filtering if not default
                if are_default_azimuths(min_az_rad,
max_az_rad):
                    az_filtered_accesses = [access]
                else:
                    az_filtered_accesses = apply_cutout_azimuth(
                        access, min_az_rad, max_az_rad
                    )

                # Elevation filtering
                el_filtered_accesses = []
                if round(max_el_rad, 2) < round(pi / 2, 2):
                    for a in az_filtered_accesses:
                        el_filtered_accesses.extend(
                            apply_cutout_elevation(a, min_el_rad,
max_el_rad)
                        )
                    filtered_accesses.extend(el_filtered_accesses)
                else:

filtered_accesses.extend(az_filtered_accesses)

        else:  # fan
            fan_beam = FanBeamMode1(

outer_radius=fan_params.maximum_range_meters /
1000.0,  # km
horizontal_beamwidth=np.radians(fan_params.horizontal_beamwidth_deg),

vertical_beamwidth=np.radians(fan_params.vertical_beamwidth_deg),

tilt_angle=np.radians(fan_params.boresight_elevation_deg),
                rotation_angle_z=np.radians(90) -
np.radians(fan_params.boresight_azimuth_deg),
            )

            for access in illumination_filtered_accesses:
                vertical_filtered_accesses =
apply_fan_limits_vertical(access, fan_beam)
                horizontal_filtered_accesses = []
                for va in vertical_filtered_accesses:
                    horizontal_filtered_accesses.extend(
                        apply_fan_limits_horizontal(va, fan_beam)
                    )

                range_filtered_accesses = []
                for ha in horizontal_filtered_accesses:
                    range_filtered_accesses.extend(
                        apply_fan_limits_maximum_range(ha,
fan_beam)
                    )


filtered_accesses.extend(range_filtered_accesses)

    else:
        filtered_accesses = illumination_filtered_accesses[:]
# copy
```

~~~
# copy

    # Populate access points
    for a in filtered_accesses:
        a.access_points = create_access_points(a)

    return all_accesses, filtered_accesses
```
~~~

Here is the problem space:

* Description
* At present access-generation only supports ground to satellite access calculation due to the fact that the access calculation relies on a binary search for astronomical "rise" and "set" events (satellite coming over the horizon and satellite dropping below the horizon). These rise/set events are a convenient way to determine occultation/obstruction of the sensor to the target.
* This will not support satellite to satellite accesses because satellites in follow orbits (same orbital elements, different true anomaly) will always have clear line of sight to each other without any occultation periods.
* A new access calculation method is required to determine visibility between two satellites which does not rely on "rise" and"set" events, but can still determine periods of visibility and non-visibility and determine transition states for each event (was not visible, is now visible).
* In addition, for ground to sat accesses distances between ground location and satellite were somewhat bounded, this condition is not well constrained in many sat to sat scenarios. The new mechanism must also account for a "maximum visible distance" parameter. If the distance between the satellites is greater than the

maximum visible distance than this is considered the same"non visible" as if the satellite were in occultation.

Implementation Details:

* The implementation for this feature must be efficient algorithmically (do not initially worry about computational inefficiencies due to underlying language limitations). Because this will run on a combination of hundreds and thousands of satellites it must be very efficient.

* Brute force mechanisms such as "propagate the position of both satellites ever 0.5 seconds over 3 hours and check if the earth is occulting the two satellites in each time step" will be unacceptable. However, I expect some level of brute force exploration will be required, but this should be mitigated as much as possible.

* Libraries such as skyfield and astropy should be considered in the development of this algorithm.

* On first implementation we will assume that the satellite has universal control of it's orientation such that you may use 360 degree visibility. We should not need to model any sensor geometries for this implementation.

Thought for 3m 25s

python

```python
import numpy as np
from dataclasses import dataclass
from datetime import datetime
from typing import List, Optional, Tuple
from uuid import UUID
from math import pi, radians
from skyfield.api import EarthSatellite, Topos, load
from skyfield.searchlib import find_discrete
from skyfield.constants import ERAD

# Assuming Config, Entity, PlanningConstraints, FanGeome
# Also assuming helper functions like calculate_night_pe
# apply_cutout_azimuth, apply_cutout_elevation, apply_fa
# apply_cutout_azimuth, apply_cutout_elevation, apply_fa
# apply_fan_limits_maximum_range, create_access_points,

# Earth radius in km
EARTH_RADIUS_KM = ERAD / 1000.0

@dataclass
class AccessPoint:
    epoch: datetime
    azimuth_radians: float
    elevation_radians: float
    range_meters: float

@dataclass
class Access:
    sat: EarthSatellite
    observer: object  # Can be Topos or EarthSatellite
    start: datetime
    end: datetime
    uuid: UUID
    culmination: Optional[datetime] = None
    access_points: Optional[list[AccessPoint]] = None
    generated_time: Optional[datetime] = None

def _event_finder(
    cfg: Config,
    sat: EarthSatellite,
    start: datetime,
    end: datetime,
    gs: Entity  # Now can be ground station or satellite
) -> Tuple[List[Access], List[Access]]:
    # Load timescale if not global
    ts = load.timescale()

    # Defaults for everything
    min_az_rad = 0.0
    max_az_rad = 2 * pi
    min_el_rad = radians(cfg.min_access_elevation_degree
    max_el_rad = pi / 2
    fan_params = FanGeometry()

    # Determine if gs is ground or satellite
    is_ground = True  # Assume based on gs properties; a
    # Example: if gs.type == 'satellite': is_ground = Fa
    # For now, assume user modifies to detect
```

```
if is_ground:
    # Existing ground logic
    is_telescope = True

    if "mission_planning_constraints" in gs.prototyp
        constraints = PlanningConstraints()
        constraints.ParseFromString(
            gs.prototype_extensions.extensions["miss
        )

        if constraints.HasField("sensor_pointing_lim
            pointing_limits = constraints.sensor_poi
            min_az_rad = pointing_limits.minimum_azi
            max_az_rad = pointing_limits.maximum_azi
            min_el_rad = pointing_limits.minimum_ele
            max_el_rad = pointing_limits.maximum_ele

        if constraints.HasField("fan_geometry"):
            is_telescope = False
            fan_params = constraints.fan_geometry

    loc = gs.location.position
    lat = loc.latitude_degrees
    lon = loc.longitude_degrees
    alt = loc.altitude_hae_meters.value

    observer = Topos(
        latitude_degrees=lat,
        longitude_degrees=lon,
        elevation_m=alt
    )
else:
    # Satellite observer
    # Assume gs has TLE data to create EarthSatellit
    # Example: observer = EarthSatellite(gs.tle_line
    # User to fill in the creation
    observer = ...  # Create EarthSatellite for obse
    is_telescope = False  # No pointing limits for s

if cfg.accesses_calculate_day_night:
    if is_ground:
        night_periods = calculate_night_periods(obse
    else:
        # For sat-to-sat, perhaps calculate eclipse
        # For sat-to-sat, perhaps calculate eclipse
        # Here, skipping day/night for simplicity; a
        night_periods = [(start, end)]
else:
    night_periods = [(start, end)]

all_accesses: List[Access] = []
illumination_filtered_accesses: List[Access] = []
filtered_accesses: List[Access] = []

for start_night, end_night in night_periods:
    if is_ground:
        # Existing ground logic
        events = sat.find_events(
            observer,
            ts.from_datetime(start_night),
            ts.from_datetime(end_night),
            altitude_degrees=sensor_elevation_effect
        )

        if events:
            accesses = events_to_accesses(
                _process_events(events),
                start_night,
                end_night,
                sat,
                observer
            )
            all_accesses.extend(accesses)
        else:
            continue  # No accesses in this period
    else:
        # Satellite-to-satellite logic
        def visibility_func(t):
            # t can be scalar or array Time
            # Compute positions in km
            r1 = observer.at(t).position.km  # Obser
            r2 = sat.at(t).position.km  # Target pos
            d = r2 - r1
            dd = np.sum(d**2, axis=0)  # |d|^2, hand
            distance = np.sqrt(dd)  # |d|

            # Max distance check (assume cfg.max_vis
            max_dist_km = cfg.max_visible_distance_m
            if np.any(dd == 0):  # Coincident, visib
                vis = (distance <= max_dist_km)
                return vis.astype(float)

            rd = np.sum(r1 * d, axis=0)  # r1 · d
            lambda_f = -rd / dd

            # dist = |r1 x r2| / |d|
            cross = np.cross(r1, r2)  # Shape (3,) o
            if cross.ndim == 1:
                cross_mag = np.linalg.norm(cross)
            else:
                cross_mag = np.linalg.norm(cross, ax
            dist = cross_mag / distance

            # Visibility
            occulted = (dist < EARTH_RADIUS_KM) & (l
            too_far = distance > max_dist_km
            vis = np.logical_not(occulted | too_far)

            return vis.astype(float)  # 1.0 if visib

        t0 = ts.from_datetime(start_night)
        t1 = ts.from_datetime(end_night)

        # Step size in days (e.g., 30 seconds)
        step_days = 30.0 / 86400.0

        times, y = find_discrete(t0, t1, visibility_

        # Convert to datetime
        transition_times = [t.utc_datetime() for t i

        # Initial state
        initial_vis = visibility_func(t0)

        accesses = []
        current_start = start_night
        current_state = 1 if initial_vis > 0.5 else

        idx = 0
        while idx < len(times):
            ti = transition_times[idx]
            yi = 1 if y[idx] > 0.5 else 0
```

```
            if current_state == 1:
                # End of access
                access = Access(
                    sat=sat,
                    observer=observer,
                    start=current_start,
                    end=ti,
                    uuid=UUID(...)  # Generate UUID
                )
                accesses.append(access)
            current_state = yi
            current_start = ti
            idx += 1

        # If ends visible
        if current_state == 1:
            access = Access(
                sat=sat,
                observer=observer,
                start=current_start,
                end=end_night,
                uuid=UUID(...)
            )
            accesses.append(access)

        # Optionally compute culmination for each ac
        for access in accesses:
            def neg_distance(t):
                r1 = observer.at(t).position.km
                r2 = sat.at(t).position.km
                dist = np.linalg.norm(r2 - r1, axis=
                return -dist

            t_start = ts.from_datetime(access.start)
            t_end = ts.from_datetime(access.end)
            max_times, max_y = find_maxima(t_start,
            if len(max_times) > 0:
                access.culmination = max_times[0].ut

        all_accesses.extend(accesses)

# Filter on illumination geometry (same for both)
if cfg.accesses_calculate_sunlit_accesses:
    for access in all_accesses:
        sunlit_periods = calculate_sunlit_periods(
            sat, access.start, access.end, 0.3
        )
        adjusted_accesses = adjust_access_for_illumi
            access, sunlit_periods
        )
        illumination_filtered_accesses.extend(adjust
else:
    illumination_filtered_accesses = all_accesses[:]

# Filter on cutout geometry
if len(illumination_filtered_accesses) > 0 and cfg.a
    if is_ground:
        if is_telescope:
            for access in illumination_filtered_acce
                # Only perform azimuth filtering if
                if are_default_azimuths(min_az_rad,
                    az_filtered_accesses = [access]
                else:
                    az_filtered_accesses = apply_cut
                        access, min_az_rad, max_az_r
                    )

                # Elevation filtering
                el_filtered_accesses = []
                if round(max_el_rad, 2) < round(pi /
                    for a in az_filtered_accesses:
                        el_filtered_accesses.extend(
                            apply_cutout_elevation(a
                        )
                    filtered_accesses.extend(el_filt
                else:
                    filtered_accesses.extend(az_filt

        else:  # fan
            fan_beam = FanBeamMode1(
                outer_radius=fan_params.maximum_rang
                horizontal_beamwidth=np.radians(fan_
                vertical_beamwidth=np.radians(fan_pa
                tilt_angle=np.radians(fan_params.bor
                rotation_angle_z=np.radians(90) - np
            )

            for access in illumination_filtered_acce
                vertical_filtered_accesses = apply_f
```

```
                    horizontal_filtered_accesses = []
                    for va in vertical_filtered_accesses
                        horizontal_filtered_accesses.ext
                            apply_fan_limits_horizontal(
                        )

                    range_filtered_accesses = []
                    for ha in horizontal_filtered_access
                        range_filtered_accesses.extend(
                            apply_fan_limits_maximum_ran
                        )

                    filtered_accesses.extend(range_filte
        else:
            # For sat-to-sat, no pointing limits, skip f
            filtered_accesses = illumination_filtered_ac
    else:
        filtered_accesses = illumination_filtered_access

    # Populate access points
    for a in filtered_accesses:
        a.access_points = create_access_points(a)  # May

    return all_accesses, filtered_accesses
```

How does this actually work? Can you give me a rundown in simple terms?

Let's break down the satellite-to-satellite access generation algorithm in simple terms, focusing on how it works and what it's doing, especially since you're adapting a ground-to-satellite access algorithm to handle satellite-to-satellite visibility.

What's the Goal?

The algorithm determines when two satellites can "see" each other over a given time period. Two satellites can see each other if:

1. The Earth isn't blocking the line of sight between them (no occultation).
2. They're not too far apart (within a maximum visible distance,

defined in the config). Unlike ground-to-satellite access, which relies on "rise" and "set" events (when a satellite appears above or drops below the horizon), satellite-to-satellite access needs a different approach because satellites in similar orbits (e.g., following each other) might always have a clear line of sight without these horizon events.

How It Works (High-Level)

The algorithm:

1. **Checks Visibility Over Time**: It figures out periods when one satellite can see another by checking if the Earth is in the way or if they're too far apart.
2. **Finds Start and End Times**: It identifies when visibility begins and ends (transitions between visible and not visible).
3. **Handles Additional Filters**: It applies constraints like illumination (is the target satellite sunlit?) and optionally calculates access points (details like range and angles at specific times).
4. **Is Efficient**: It avoids checking every tiny time step by using a smart search method to find key transition points.

Step-by-Step Rundown

Here's how the code operates, simplified:

1. Setup

- **Inputs**: You provide the target satellite (`sat`), the observer (another satellite or ground station, stored in `gs`), a time window (`start` to `end`), and a configuration (`cfg`) with settings like minimum elevation angle or maximum visible distance.
- **Observer Type**: The algorithm checks if `gs` is a ground station or a satellite. If it's a ground station, it uses the original ground-to-satellite logic (finding rise/set events). If it's a satellite, it switches to the new satellite-to-satellite logic.

- **Time Periods**: If you care about day/night (e.g., for ground stations), it splits the time window into night periods. For satellite-to-satellite, it might skip this or use eclipse periods, but the example assumes the full time window for simplicity.

2. Ground vs. Satellite Logic

- **Ground-to-Satellite (Original Logic)**:
  - Uses Skyfield's `find_events` to detect when the satellite rises above or sets below the horizon (based on a minimum elevation angle, like 10 degrees).
  - These events mark the start and end of access periods when the satellite is visible from the ground.
  - It then filters these periods based on constraints like illumination (is the satellite sunlit?) or sensor pointing limits (azimuth/elevation ranges or fan geometry).
- **Satellite-to-Satellite (New Logic)**:
  - Since there's no horizon for satellites, it can't rely on rise/set events.
  - Instead, it checks visibility by calculating if:
    - The Earth blocks the line of sight (occultation).
    - The satellites are within the maximum visible distance (e.g., 10,000 km).
  - It uses a mathematical check to determine visibility at different times, avoiding brute-force checks every 0.5 seconds.

3. Visibility Check (Satellite-to-Satellite)

The core of the new logic is the `visibility_func`, which determines if the satellites can see each other at a given time:

- **Positions**: It gets the 3D positions (in kilometers) of both satellites using Skyfield's `at` method.

- **Distance**: Calculates the distance between them (`|d|`).
- **Occultation Check**:
  - Computes the closest approach of the line between the satellites to the Earth's center using vector math (dot products and cross products).
  - If this distance is less than Earth's radius (about 6,371 km) **and** the closest point lies between the satellites, the Earth is blocking the line of sight (occultation).
- **Max Distance**: Checks if the distance between satellites is less than or equal to the maximum allowed (from `cfg.max_visible_distance_meters`).
- **Result**: Returns 1.0 (visible) if there's no occultation and the satellites are close enough, or 0.0 (not visible) otherwise.

4. Finding Visibility Transitions

- Uses Skyfield's `find_discrete` to efficiently find times when visibility changes (from visible to not visible or vice versa).
- It steps through time (e.g., every 30 seconds) and evaluates `visibility_func` .
- When the function switches from 0.0 to 1.0 or 1.0 to 0.0, it records these as transition times.
- These transitions define the start and end of access periods (e.g., a period starts when the satellites become visible and ends when they're occulted or too far).

5. Building Accesses

- Creates `Access` objects for each visibility period, storing:
  - The target satellite (`sat`).
  - The observer (the other satellite).
  - Start and end times of the access.
  - A unique ID (`uuid`).

- Optionally finds the **culmination** (the time of closest approach during the access) by searching for the minimum distance between satellites using Skyfield's `find_maxima` on the negative distance function.

6. Filtering

- **Illumination**: If configured (`cfg.accesses_calculate_sunlit_accesses`), it checks if the target satellite is sunlit during the access period and splits or filters accesses accordingly (using `calculate_sunlit_periods` and `adjust_access_for_illumination`).
- **Sensor Constraints**: For ground stations, it applies azimuth, elevation, or fan geometry filters. For satellite-to-satellite, it skips these since the example assumes 360-degree visibility (no sensor pointing limits).
- The result is a list of filtered accesses that meet all constraints.

7. Access Points

- For each access, it generates `AccessPoint` objects (using `create_access_points`), which include details like the time, azimuth, elevation, and range at specific points during the access. For satellite-to-satellite, this might need tweaking to compute angles relative to the observer satellite's frame.

Key Features

- **Efficiency**: Instead of checking every 0.5 seconds, it uses `find_discrete` to jump to transition points, reducing computations. The step size (e.g., 30 seconds) balances accuracy and speed.
- **Occultation**: Uses vector geometry to check if the Earth blocks the line of sight, avoiding brute-force ray tracing.
- **Max Distance**: Incorporates a maximum visible distance to handle cases where satellites are too far apart to communicate.

- **Flexibility**: Reuses much of the ground-to-satellite code, with a branch for satellite-to-satellite logic, making it maintainable.
- **Skyfield**: Leverages Skyfield for accurate orbital propagation and time handling, ensuring robust calculations.

Example Scenario

Imagine two satellites, Sat1 and Sat2, orbiting Earth:

- **Time Window**: 1 hour (e.g., 12:00 to 13:00).
- **Max Distance**: 10,000 km.
- At 12:00, Sat1 and Sat2 are 5,000 km apart with a clear line of sight (no Earth in the way). They're visible.
- At 12:15, the Earth moves into the line of sight (occultation). Visibility ends.
- At 12:30, they emerge from behind the Earth, and visibility resumes until 12:45 when they're 12,000 km apart (beyond max distance).
- The algorithm outputs two accesses: 12:00–12:15 and 12:30–12:45, with culmination times (closest approach) calculated.

Assumptions and Notes

- The code assumes 360-degree visibility for satellite-to-satellite (no sensor pointing limits), as specified.
- You need to define how the observer satellite is created from `gs` (e.g., using TLE data).
- The `create_access_points` function may need adaptation to compute azimuth/elevation for satellite-to-satellite (relative to the observer's velocity or body frame).
- The step size (30 seconds) can be adjusted for accuracy vs. performance.
- Some helper functions (e.g., `calculate_sunlit_periods`) are

assumed to exist from your original code.

Why It’s Efficient

- **Avoids Brute Force**: Instead of checking every 0.5 seconds, it uses `find_discrete` to find visibility changes, typically requiring far fewer evaluations.
- **Vector Math**: The occultation check uses simple dot and cross products, which are fast even for arrays of times.
- **Reuses Code**: Shares filtering logic with ground-to-satellite, reducing code duplication.

If you need clarification on any part (e.g., the math behind occultation or how to adapt `create_access_points`), let me know!

how does find_discrete work?

In simple terms, Skyfield’s `find_discrete` function is a tool that efficiently finds times when a condition changes, like when two satellites go from being visible to not visible (or vice versa) in your satellite-to-satellite access algorithm. It’s designed to avoid checking every tiny time step, making it much faster than brute-force methods. Here’s a clear, concise explanation of how it works, tailored to your use case.

What `find_discrete` Does

`find_discrete` searches for points in time within a given window (from `t0` to `t1`) where a function switches between discrete states, such as 0.0 (not visible) and 1.0 (visible). In your code, this is used to detect when the visibility between two satellites changes due to the Earth blocking the line of sight or the distance exceeding the maximum allowed.

How It Works (Step-by-Step)

1. **Inputs**:
   - **Start and End Times** (`t0`, `t1`): These are Skyfield `Time`

   - **Start and End Times** (`t0` , `t1`): These are Skyfield `Time` objects defining the time window (e.g., 12:00 to 13:00).
   - **Function** (`visibility_func`): A function that takes a time (or array of times) and returns a float (e.g., 1.0 for visible, 0.0 for not visible).
   - **Step Size** (`stepsize`): A time interval (in days, e.g., 30 seconds = `30.0 / 86400.0`) that controls how finely it initially samples the time window.

2. **Initial Sampling**:
   - `find_discrete` evaluates the `visibility_func` at regular intervals (every `stepsize`) across the time window. For example, if `stepsize` is 30 seconds, it checks visibility at 12:00:00, 12:00:30, 12:01:00, etc.
   - This creates a sequence of values, like `[1.0, 1.0, 0.0, 0.0, 1.0]` , showing visibility at each step.
3. **Detecting Transitions**:
   - It looks for places where the function's output changes (e.g., from 1.0 to 0.0 or 0.0 to 1.0). These are potential visibility transitions (e.g., when the Earth starts or stops blocking the satellites).
   - For each change, it narrows down the exact time of the transition using a **binary search** (or similar root-finding method).
4. **Binary Search for Precision**:
   - When it finds a change (e.g., 1.0 at 12:00:00 and 0.0 at 12:00:30), it zooms in on the interval (12:00:00 to 12:00:30) and repeatedly evaluates `visibility_func` at smaller time steps to pinpoint the exact moment of the switch (e.g., 12:00:14.237).
   - This is much faster than checking every second, as it halves the search interval with each step.

5. **Output**:
    - Returns two lists:
        - `times` : A list of Skyfield `Time` objects where transitions occur (e.g., `[t1, t2, t3]` for times when visibility changes).
        - `values` : The corresponding function values at those times (e.g., `[1.0, 0.0, 1.0]`).
    - In your code, these are converted to `datetime` objects and used to define the start and end of access periods.

Example in Your Context

In your satellite-to-satellite access code:

- **Function**: `visibility_func` checks if two satellites are visible by calculating:
    - If the Earth blocks the line of sight (using vector math to check occultation).
    - If the distance between satellites is within the maximum allowed (e.g., 10,000 km).
    - It returns 1.0 (visible) or 0.0 (not visible).
- **Time Window**: Say, 12:00 to 13:00.
- **Step Size**: 30 seconds (`30.0 / 86400.0` days).
- **Process**:
    - `find_discrete` checks visibility every 30 seconds.
    - If it sees a change (e.g., 1.0 to 0.0 between 12:15:00 and 12:15:30), it uses binary search to find the precise time (e.g., 12:15:14.237).
    - Outputs times like `[12:15:14.237, 12:30:22.891]` with values `[0.0, 1.0]` , meaning visibility ended at 12:15:14.237 and started again at 12:30:22.891.

- **Access Creation**: Your code uses these transitions to build `Access` objects (e.g., an access from 12:00 to 12:15:14.237, another from 12:30:22.891 to 13:00).

Why It's Efficient

- **Avoids Brute Force**: Instead of checking every 0.5 seconds (7,200 checks for 1 hour), it starts with coarse steps (e.g., 120 checks for 30-second steps) and only refines where changes occur.
- **Binary Search**: Pinpoints transitions in just a few evaluations (e.g., 10–20 per transition) rather than checking every possible time.
- **Handles Complex Functions**: Works with any function that returns discrete values (like 0.0/1.0), making it perfect for visibility checks.

Key Notes

- **Step Size Matters**: A smaller `stepsize` (e.g., 10 seconds) is more accurate but slower; a larger one (e.g., 60 seconds) is faster but might miss brief transitions. Your code uses 30 seconds, which is a good balance for typical satellite orbits.
- **Function Requirements**: `visibility_func` must be vectorized (handle arrays of times) and return floats (e.g., 1.0/0.0). Your code does this correctly with NumPy.
- **Skyfield Integration**: `find_discrete` uses Skyfield's precise time handling, ensuring accurate orbital calculations.

Analogy

Think of `find_discrete` as a smart hiker looking for places where a trail crosses a river (visibility changes). Instead of checking every step along the path, they check every 100 meters, note where the trail goes from dry to wet, and then backtrack to find the exact crossing point. This saves time while still finding the key spots.

crossing point. This saves time while still finding the key spots.

If you want a deeper dive into the math (e.g., how binary search works in `find_discrete`) or help tuning the step size, let me know!

> Sweet. Can you give me the calculation for the occultation in mathematical terms? like use vectors and shit

Let’s dive into the occultation calculation used in your satellite-to-satellite access algorithm, expressed in mathematical terms with vectors. The goal is to determine if the Earth blocks the line of sight between two satellites, which is a key part of the `visibility_func` in your code. I’ll break it down clearly, using vector notation and equations, and tie it directly to how the code implements it.

Problem Setup

- **Satellites**: You have two satellites, Sat1 (observer) and Sat2 (target).
- **Positions**: At a given time t, Sat1 is at position r_1 and Sat2 is at position r_2, both in 3D Cartesian coordinates (e.g., in km, from Skyfield’s `position.km`).
- **Earth**: Modeled as a sphere centered at the origin $(0, 0, 0)$ with radius $R_E \approx 6371\,\mathrm{km}$ (Earth’s radius, `EARTH_RADIUS_KM` in the code).
- **Occultation**: The Earth blocks the line of sight if the straight-line path from r_1 to r_2 passes through the Earth (i.e., comes closer than R_E to the origin) and the intersection point lies between the satellites.

Mathematical Approach

To check for occultation, we need to find the **closest distance** from the origin (Earth’s center) to the line segment connecting r_1 and r_2. If this distance is less than R_E **and** the closest point lies on the segment between the satellites, the Earth is in the way.

Step 1: Define the Line Segment

- The vector from Sat1 to Sat2 is:

  $$d = r_2 - r_1$$

  This is the direction vector of the line from Sat1 to Sat2.

- The distance between the satellites is:

  $$d = |d| = \quad d \cdot d$$

  (In the code: `d = r2 - r1`, `dd = np.sum(d**2, axis=0)`, `distance = np.sqrt(dd)`.)

Step 2: Find the Closest Point to the Origin

The line from r_1 to r_2 can be parameterized as:

$$r(t) = r_1 + td, \quad t \in [0, 1]$$

- $t = 0$: Position at Sat1 (r_1).
- $t = 1$: Position at Sat2 (r_2).
- $t \in [0, 1]$: Points on the line segment between the satellites.

We want the point on this line closest to the origin. The distance from the origin to a point $r(t)$ is $|r(t)|$. To find the minimum distance, we minimize:

$$|r(t)|^2 = (r_1 + td) \cdot (r_1 + td)$$

This is a quadratic function in t. The minimum occurs where the derivative is zero. Expanding:

$$|r(t)|^2 = r_1 \cdot r_1 + 2t(r_1 \cdot d) + t^2(d \cdot d)$$

Differentiate with respect to t:

$$\frac{d}{dt}|r(t)|^2 = 2(r_1 \cdot d) + 2t(d \cdot d) = 0$$

Solve for t:

$$t = -\frac{r_1 \cdot d}{d \cdot d}$$

In the code: `rd = np.sum(r1 * d, axis=0)`, `lambda_f = -rd / dd`, where `lambda_f` is t.

This t (called `lambda_f` in the code) gives the parameter for the point on the line closest to the origin:

$$p = r_1 + td$$

Step 3: Distance to the Origin

The distance from the origin to the line is the distance to point p. However, a more efficient approach (used in the code) is to compute the distance using the cross product. The distance from the origin to the line is given by:

$$\text{distance} = \frac{|r_1 \times r_2|}{|d|}$$

- The cross product $r_1 \times r_2$ gives the area of the parallelogram formed by r_1 and r_2.
- Dividing by $|d|$ (distance between satellites) gives the perpendicular distance from the origin to the line.

In the code:

```python
cross = np.cross(r1, r2)
cross_mag = np.linalg.norm(cross, axis=0) if cross.ndim
dist = cross_mag / distance
```

Step 4: Occultation Condition

The line of sight is blocked if:

1. **Distance Condition**: The distance from the origin to the line is less than the Earth’s radius:

$$|r_1 \times r_2| < R$$

$$\frac{}{|d|} < R_E$$

(In the code: `dist < EARTH_RADIUS_KM` .)

2. **Segment Condition**: The closest point must lie on the line segment between r_1 and r_2, i.e., $t \in [0, 1]$:

$$0 \leq -\frac{r_1 \cdot d}{d \cdot d} \leq 1$$

(In the code: `lambda_f >= 0) & (lambda_f <= 1` .)

If both conditions are true, the Earth intersects the line segment, meaning the satellites cannot see each other (occultation).

Step 5: Additional Check

The code also checks if the satellites are too far apart:

$$|d| > d_{\max}$$

where $d_{\max}$ is the maximum visible distance (e.g., 10,000 km, from `cfg.max_visible_distance_meters`). If true, the satellites are considered not visible, even without occultation.

In the code:

```python
too_far = distance > max_dist_km
occulted = (dist < EARTH_RADIUS_KM) & (lambda_f >= 0) &
vis = np.logical_not(occulted | too_far)
```

Full Visibility Condition

The satellites are visible if:

$$\text{vis} = \neg \left(\left(\frac{|r_1 \times r_2|}{|d|} < R_E \land 0 \leq -\frac{r_1 \cdot d}{d \cdot d} \leq 1 \right) \lor |d| > d_{\max} \right)$$

- Returns 1.0 if visible (no occultation and within max distance).
- Returns 0.0 if not visible (occulted or too far).

Edge Case

If $d = 0$ (satellites at the same position), the code checks only the distance condition:

python

```python
if np.any(dd == 0):
    vis = (distance <= max_dist_km)
    return vis.astype(float)
```

This avoids division by zero in the distance and t calculations.

Why This Works

- **Geometric Intuition**: The cross product $|r_1 \times r_2|$ relates to the area of the parallelogram, and dividing by $|d|$ gives the perpendicular distance to the line. If this is less than R_E, the line could pass through the Earth.
- **Segment Check**: Ensuring $t \in [0, 1]$ confirms the intersection is between the satellites, not beyond them.
- **Efficiency**: Uses vector operations (dot and cross products), which are fast, especially with NumPy's array handling for multiple times.

Example

- **Positions**: $r_1 = (7000, 0, 0)\,\text{km}$, $r_2 = (0, 7000, 0)\,\text{km}$.
- **Vector d**: $r_2 - r_1 = (-7000, 7000, 0)$.
- **Distance**: $|d| = \quad 7000^2 + 7000^2 \approx 9899\,\text{km}$.
- **Cross Product**: $r_1 \times r_2 = (0, 0, 7000 \cdot 7000) = (0, 0, 49000000)$, $|r_1 \times r_2| = 49000000$.
- **Distance to Line**: $\frac{49000000}{9899} \approx 4950\,\text{km} < 6371\,\text{km}$.
- **Parameter t**: $r_1 \cdot d = 7000 \cdot (-7000) = -49000000$, $d \cdot d = 98000000$, $t = -\frac{-49000000}{98000000} = 0.5$.

- **Result**: Since $4950 < 6371$ and $t = 0.5 \in [0, 1]$, the Earth blocks the line of sight. If $d_{\max} = 10000\,\mathrm{km}$, and $9899 < 10000$, the occultation makes them not visible.

This math is directly implemented in your `visibility_func`, making it both accurate and computationally efficient for determining satellite-to-satellite visibility. If you want a visual diagram or a specific example worked out, let me know!

how to turn this chat into pdf?

To convert this chat conversation into a PDF, you can follow methods similar to those used for exporting conversations from platforms like ChatGPT or WhatsApp, as described in the web results. Since this is a conversation with Grok, there's no built-in export feature mentioned, but you can use browser-based or manual methods to achieve this. Below, I'll outline a simple approach using the browser's print-to-PDF feature, which is widely applicable and doesn't require third-party tools. I'll also provide an artifact with a step-by-step guide in markdown format.

Simple Method: Print to PDF from Browser

1. **Open the Conversation**: Ensure the entire chat is visible in your browser or app interface. If it's long, scroll to load all messages.
2. **Select All**: Highlight the conversation text using your mouse or keyboard (`Ctrl + A` on Windows, `Cmd + A` on Mac) to select everything.
3. **Print**: Right-click the selected text or use the browser's menu (click the three-dot icon or press `Ctrl + P` / `Cmd + P`) and choose "Print."
4. **Save as PDF**: In the print dialog, set the destination to "Save as PDF" (available in most browsers like Chrome, Edge, or Firefox). Adjust settings like layout or margins if needed.
5. **Save**: Click "Save," choose a file name and location, and

download the PDF.

Notes

- **Formatting**: The PDF will capture the conversation as displayed, including text and any visible formatting. If the interface includes unwanted elements (e.g., sidebars), hide them via browser settings or zoom in before printing.
- **Long Conversations**: If the conversation spans multiple pages, you may need to scroll and repeat the process for each section, or use a browser extension like GoFullPage (for Chrome) to capture a scrolling screenshot and convert it to PDF.
- **Alternative**: Copy the text into a text editor like Google Docs or Microsoft Word, format as needed, and export as a PDF (`File > Download > PDF` in Google Docs or `File > Save As > PDF` in Word).

Artifact: Step-by-Step Guide

How_to_Convert_Chat_to_PDF.md
markdown

Edit in files • Show inline

This approach is straightforward and works across platforms without needing additional software. If you prefer a specific tool or have a long conversation requiring automation, let me know, and I can suggest alternatives like screenshot-to-PDF conversion or third-party extensions. pdfgear.com chromewebstore.google.com