

Women In Data Datathon: Conjunction Risk Analysis

Ashley Loong

September 15, 2025

1 Background

1.1 Introduction

From 2020 to 2024, satellites operating in Earth orbit grew from 3,371 to 11,539. In this year alone, more than 1,200 satellites were launched into orbit from January to April. SpaceX led with 573 Starlink satellites during the Q1 of 2025.

Our space environment is becoming increasingly crowded as the number of satellites and large constellations like Starlink continues to grow. In addition to these new launches, inactive satellites in Low Earth Orbit (LEO) can remain in orbit for years to centuries.

Each additional satellite increases conjunction frequency and thus creates more chances for collision. When two satellites collide, they can produce thousands of pieces of debris and trigger cascading collision events.

Sources:

- [Satellite Industry Association Releases the 28th Annual State of the Satellite Industry Report](#)
- [Orbital debris and the market for satellites](#)
- [Modeling Orbital Decay of Low-Earth Orbit Satellites due to Atmospheric Drag](#)
- [NASA Spacecraft Conjunction Assessment and Collision Avoidance Best Practices Handbook](#)
- [Satellite orbital conjunction reports assessing threatening encounters in space \(SOCRATES\)](#)

1.2 Key Terms and Concepts

1.2.1 What is a shell?

A shell is band of altitudes where satellites are placed. It is not as single orbit but a “layer” above Earth where satellites can exist with different inclinations and longitudes.

Risks of different shells

- Shells below 500 km are less crowded, but satellites decay faster due to atmospheric drag.
- Shells between 500–600 km very popular because they balance longer lifetime with lower launch cost. However, conjunctions risks are higher since the space is more crowded. Lifetime could be years to decades, e.g. a dead satellite at 550 km might remain in orbit for 10-25 years before atmospheric reentry.

- There is less drag in shells above 800 km, so satellites can stay for decades to centuries. This is bad for long-term sustainability since debris also lingers forever.
- Objects in the geostationary orbit shell (~36,000 km) remain essentially forever as there is meaningful drag at all. Satellites must be moved to a “graveyard orbit” when retired.

Lifetimes

Consider a typical satellite-sized object that is about 100–1,000 kg with moderate drag area.

Below are its orbital lifetime estimates by altitude:

- 300-400 km: ~0-2 years before atmospheric reentry
- 500-600 km: ~5–30 years
- 700-800 km: ~80–400 years
- 900-1,000 km: ~500–1,500 years
- 1,200 km and above: 2,000+ years

1.2.2 Space Object Types

<i>Object Type</i>	<i>Description</i>	<i>Importance in Conjunction Risk Analysis</i>
Payload	Operational or defunct satellites	Valuable, often maneuverable, critical to protect
Rocket body	Spent propulsion units to deploy satellites into orbit, i.e. launch vehicle stages	Large, non-maneuverable, collision threat
Debris	Fragments from explosions, collisions, breakups	Numerous and unpredictable
Unknown	Identified but unclassified objects	Complicates modeling with uncertainty, could be a hazard or payload

Why are rocket bodies catalogued differently than standard debris?

- From [Space Track Documentation](#):

They can have mechanisms or fuel on board that can affect the orbital behavior of the rocket body even after long periods of time. Rocket bodies are also constructed to endure high temperatures and stresses associated with launch, so they have a greater probability of surviving reentry and require closer attention than most debris.

1.2.3 Conjunction vs. Collision

Conjunction: A close approach between two objects in space, defined by a threshold distance.

Collision: An event where two objects hit each other.

1.3 Historical Context

2009 - The First Satellite Collision

The [collision of Iridium 33 and Cosmos 2251](#) produced more than 1800 pieces of debris that were larger than 10 cm. Some of which will remain in orbit through 2100.

1.4 Goal: Forecast Conjunction Risk

Phase 1: Setup

- Define scope
- Import libraries and dataset
- Data wrangling
- Configure conjunction analysis
- Build propagators

Phase 2: Execution

- Run coarse propagation
- Deduplicate coarse candidates
- Refine locally

Phase 3: Analysis and Reporting

- Risk report
- Summary and recommendations

2 Setup

2.1 Scope

Objective

As satellites are being launched at an accelerating rate each year, we want to know: Is it getting too crowded in popular shells? Can we track or predict the risk of conjunctions?

Our reported findings are intended to support policymakers in making evidence-based decisions regarding space safety and environmental regulations.

Dataset

We will work with a combined dataset from:

1. United States Space Force (USSF)’s 18th Space Defense Squadron Element Sets
2. NASA Goddard Space Flight Center’s CDDIS (Crustal Dynamics Data Information System) Ephemeris Archive

Deliverable

Risk report and recommendations.

Model Choice

The **SGP4** is a standard orbital propagation model used to predict to position and velocity of satellites over time.

- Strengths: fast, efficient, standardized
- Limitations: not the most precise, best for short-term predictions

Methodology

1) What time horizon are we forecasting?

Screen for close approaches over the next 24 hours. This is long enough to see interesting traffic but short enough to run fast on a laptop with thousands of objects.

2) How fine is our sampling of time?

For the first “coarse” pass, propagate every 5 minutes. For any potential close approach we find, we’ll re-check just those two satellites around that time at 30-second steps to refine the minimum distance. This will keep runtime manageable and still finds the real minimum distance accurately.

3) What counts as a “conjunction”?

Flag pairs that ever get within 10 km (and we’ll also count the stricter 5 km subset). We’ll use an initial search radius of 20 km during the coarse pass to make sure we don’t miss events that dip below 10 km between 5-minute samples.

4) Which objects are we analyzing?

The densest shell by object count. Exclude anything that’s not currently orbiting (`isOrbiting == False`) so we don’t propagate dead/decayed entries.

5) What coordinate system/units are we using?

SGP4 returns positions in the TEME/ECI frame, in kilometers. We’ll compute distances directly in that frame with plain Euclidean distance.

6) What should we expect from TLE/SGP4?

TLE+SGP4 is screening-level only (good for finding candidates, not for computing formal probability of collision). Accuracy drops as you move far from the TLE’s epochDate, so we’ll start the forecast at the latest epoch among your selected objects to reduce bias.

7) What do we need from the dataframe?

The dataframe contains classical mean elements we can feed into SGP4: inclination, raan, argOfPerigee, meanAnomaly, eccentricity, meanMotion, bStar, plus epochDate. Check these fields exist and have minimal missing data for the chosen shell.

2.2 Import Libraries and Dataset

```
[ ]: import pandas as pd
import numpy as np

from pathlib import Path
from datetime import datetime, timedelta, timezone
from dataclasses import dataclass
from sgp4.api import Satrec, SGP4_ERRORS, jday, WGS72
from scipy.spatial import cKDTree

FINAL_DIR = Path.cwd().parents[1] / "data" / "02_final"
```

```
print(FINAL_DIR)
```

```
c:\Users\ash\Desktop\wid-datathon\data\02_final
```

```
[42]: df = pd.read_csv(FINAL_DIR / "satellite_data_clean.csv")
```

```
# global setting to show all columns
```

```
pd.set_option('display.max_columns', None)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 29140 entries, 0 to 29139
```

```
Data columns (total 57 columns):
```

#	Column	Non-Null Count	Dtype
0	argOfPerigee	29140 non-null	float64
1	bStar	29132 non-null	float64
2	createdAt	29140 non-null	object
3	eccentricity	29140 non-null	float64
4	semiMajorAxis	29109 non-null	float64
5	satNo	29140 non-null	int64
6	revNo	29109 non-null	float64
7	raan	29140 non-null	float64
8	period_els	29109 non-null	float64
9	meanMotionDot	29118 non-null	float64
10	meanMotionDDot	29118 non-null	float64
11	meanMotion	29140 non-null	float64
12	meanAnomaly	29140 non-null	float64
13	inclination_els	29140 non-null	float64
14	idOnOrbit	29140 non-null	int64
15	epoch	29140 non-null	object
16	epochDate	29140 non-null	object
17	intldes	29140 non-null	object
18	noradCatId	29140 non-null	int64
19	objectType	29140 non-null	object
20	satName	29140 non-null	object
21	country	29140 non-null	object
22	launch	29140 non-null	object
23	site	29140 non-null	object
24	decay	1198 non-null	object
25	inclination_sat	29134 non-null	float64
26	rscsValue	29140 non-null	int64
27	rscsSize	28728 non-null	object
28	file	29140 non-null	int64
29	launchYear	29140 non-null	int64
30	launchNum	29140 non-null	int64
31	launchPiece	29133 non-null	object

```

32  objectName          29140 non-null  object
33  objectId            29140 non-null  object
34  objectNumber        29140 non-null  int64
35  perigee_alt_km      29109 non-null  float64
36  apogee_alt_km       29109 non-null  float64
37  apogee_mismatch     29140 non-null  bool
38  perigee_mismatch    29140 non-null  bool
39  orbitClass          29140 non-null  object
40  launchDecade        29140 non-null  object
41  inclinationBand     29140 non-null  object
42  eccClass            29140 non-null  object
43  ageInYears          29140 non-null  float64
44  shell_idx_100km     29109 non-null  float64
45  shell_100km         29140 non-null  object
46  shell_center_km     29109 non-null  float64
47  isDecayed           29140 non-null  bool
48  isOrbiting          29140 non-null  bool
49  isStarlink          29140 non-null  bool
50  isOneweb            29140 non-null  bool
51  isIridium           29140 non-null  bool
52  isConstellation     29140 non-null  bool
53  dwelling_alt_km     29109 non-null  float64
54  dwelling_alt_km_weighted 29109 non-null  float64
55  dwelling_shell_idx  29109 non-null  float64
56  dwelling_shell_100km 29140 non-null  object
dtypes: bool(8), float64(21), int64(8), object(20)
memory usage: 11.1+ MB

```

2.3 Data Wrangling

```

[43]: # null counts

df.isna().sum().reset_index() \
    .rename(columns={0: 'n_missing'}) \
    .query("n_missing > 0") \
    .sort_values(by='n_missing', ascending=False)

```

```

[43]:
      index  n_missing
24      decay    27942
27    rcsSize     412
35  perigee_alt_km     31
54  dwelling_alt_km_weighted 31
53    dwelling_alt_km     31
46    shell_center_km     31
44    shell_idx_100km     31
36    apogee_alt_km     31
55  dwelling_shell_idx     31

```

4	semiMajorAxis	31
8	period_els	31
6	revNo	31
10	meanMotionDDot	22
9	meanMotionDot	22
1	bStar	8
31	launchPiece	7
25	inclination_sat	6

```
[44]: # parse epochDate to timezone-aware UTC timestamps
df = df.copy()
df["epochDate"] = pd.to_datetime(df["epochDate"], utc=True, errors="coerce")

# replace missing bStar with 0.0 (common practice for SGP4 init if unknown)
if "bStar" in df.columns:
    df["bStar"] = df["bStar"].fillna(0.0)

# create a reliable inclination in degrees
df["inclination_deg"] = df["inclination_els"].where(
    ~df["inclination_els"].isna(),
    df["inclination_sat"]
)

# coerce all numeric inputs we'll send to SGP4 to numeric dtype
for c in [
    "eccentricity", "meanAnomaly", "raan", "argOfPerigee", "meanMotion", "inclination_deg", "bStar"
]:
    df[c] = pd.to_numeric(df[c], errors="coerce")
```

```
[45]: shell_counts = (
    df.loc[df["isOrbiting"] == True]
    .groupby("shell_100km", dropna=False)
    .size()
    .sort_values(ascending=False)
)

print("Objects by 100-km shell (orbiting only):")
print(shell_counts.head(10))
```

```
Objects by 100-km shell (orbiting only):
shell_100km
500- 599 km      6253
400- 499 km      4624
700- 799 km      3286
800- 899 km      2285
600- 699 km      2201
300- 399 km      1364
900- 999 km      1167
```

```

1400-1499 km      1048
35700-35799 km    726
1000-1099 km      661
dtype: int64

```

```

[46]: SHELL_TO_USE = shell_counts.idxmax() # densest shell
      SHELL_COUNT = int(shell_counts.max())

      print(f"\nChosen shell: {SHELL_TO_USE}, {SHELL_COUNT} objects")

      # filter dataframe to shell and still-orbiting objects
      df_shell = df[
          (df["shell_100km"] == SHELL_TO_USE) &
          (df["isOrbiting"] == True) &
          (df["orbitClass"] == "LEO")
      ].copy()

      # valid eccentricity range for SGP4: [0,1)
      ecc_mask = df_shell["eccentricity"].between(0.0, 1.0, inclusive="left")
      df_shell_clean = df_shell[ecc_mask].copy()
      print(f"  After setting eccentricity range: {len(df_shell_clean)} rows remain.")

      # mean motion must be positive (revs/day)
      mm_mask = df_shell_clean["meanMotion"].astype(float) > 0.0
      df_shell_clean = df_shell_clean[mm_mask].copy()
      print(f"  After meanMotion > 0: {len(df_shell_clean)} rows remain.")

      # epochDate must be valid (not NaT)
      df_shell_clean = df_shell_clean[df_shell_clean["epochDate"].notna()].copy()
      print(f"  After valid epoch: {len(df_shell_clean)} rows remain.")
      df_shell = df_shell_clean.copy()

```

```

Chosen shell: 500- 599 km, 6253 objects
  After setting eccentricity range: 6093 rows remain.
  After meanMotion > 0: 6093 rows remain.
  After valid epoch: 6093 rows remain.

```

```

[47]: df_shell["objectType"].value_counts(dropna=False)

```

```

[47]: objectType
Payload      4967
Debris        845
Unknown       150
Rocket body   131
Name: count, dtype: int64

```


2.4 Configure Conjunction Analysis

```
[48]: from dataclasses import dataclass
from typing import Optional

@dataclass
class Config:
    # what we're analyzing
    shell_name: str = "mixed"          # e.g. "500-600 km"
    n_objects: int = 0                 # filled after df_shell is built

    # constant
    earth_radius_km = 6378.137         # Earth's mean equatorial radius (WGS-84)

    # time window
    forecast_hours: int = 24
        # how far into the future to forecast
        # start with 24h for speed; you can extend later
    explicit_start_utc: Optional[datetime] = None # set to fix start; else
    ↪ latest TLE epoch

    # sampling step sizes
    coarse_step_minutes: int = 5       # coarse propagation step for full set
    refine_step_seconds: int = 30      # refinement step for candidate pairs

    # thresholds
    search_radius_km: float = 20.0     # coarse neighbor query radius
    report_thresh_km: float = 10.0     # main reporting threshold
    report_strict_km: float = 5.0      # stricter subset for "very close"
    ↪ approaches

    # derived (computed after df_shell is known)
    t_start: Optional[datetime] = None
    t_end: Optional[datetime] = None

cfg = Config()

def initialize_config(cfg: Config, df_shell: pd.DataFrame) -> Config:
    if df_shell is None or df_shell.empty:
        raise ValueError("df_shell cannot be empty")

    # shell name from column
    if "shell_100km" not in df_shell.columns:
        raise ValueError("expected 'shell_100km' in df_shell for labeling the
    ↪ working shell")

    shell_values = df_shell["shell_100km"].dropna().unique()
```

```

if len(shell_values) == 1:
    cfg.shell_name = str(shell_values[0])
    print(f"Single shell detected: {cfg.shell_name}")
else: # edge case: mixed labels
    cfg.shell_name = "mixed"
    print("WARNING: Multiple shell labels found in df_shell:")
    for val in shell_values:
        print(f"    - {val}")
    print("Proceeding with shell_name='mixed'")

cfg.n_objects = int(len(df_shell))

if cfg.explicit_start_utc is not None:
    t_start = cfg.explicit_start_utc
else:
    t_start = df_shell["epochDate"].max() if "epochDate" in df_shell.
↳columns else None
    # if epoch parsing failed earlier and this is NaT, fall back to 'now'
↳in UTC.
    if t_start is None or pd.isna(t_start):
        t_start = datetime.now(timezone.utc)

cfg.t_start = t_start
cfg.t_end = t_start + timedelta(hours=cfg.forecast_hours)
return cfg

def print_config(cfg: Config) -> None:
    print("\nConjunction summary:")

    rows = {
        "Shell label": cfg.shell_name,
        "Object count": cfg.n_objects,
        "Time window": f"{cfg.t_start} to {cfg.t_end}",
        "Coarse step": f"{cfg.coarse_step_minutes} min",
        "Refine step": f"{cfg.refine_step_seconds} sec",
        "Coarse search radius": f"{cfg.search_radius_km} km",
        "Risk thresholds": f"<{cfg.report_thresh_km} km, <{cfg.
↳report_strict_km} km",
        "Frame & units": "SGP4 TEME/ECI; distances in km (Euclidean).
↳"
    }

    for field, val in rows.items():
        print(f"{field}: {val}")

```

```
[49]: cfg = initialize_config(cfg, df_shell)
      print_config(cfg)
```

Single shell detected: 500- 599 km

Conjunction summary:

Shell label: 500- 599 km

Object count: 6093

Time window: 2025-08-03 00:00:00+00:00 to 2025-08-04 00:00:00+00:00

Coarse step: 5 min

Refine step: 30 sec

Coarse search radius: 20.0 km

Risk thresholds: <10.0 km, <5.0 km

Frame & units: SGP4 TEME/ECI; distances in km (Euclidean).

```
[50]: # EPOCH FRESHNESS FILTER

# goal: keep satellites whose TLE epoch is "close" to the common start time
#       ↳ t_start.
# avoids SGP4 numerical/pathology issues when propagating far from an object's
#       ↳ own epoch

# compute age of each epoch relative to t_start (days; positive means epoch
#       ↳ BEFORE t_start)
df_shell = df_shell.copy()
df_shell["epoch_age_days"] = (cfg.t_start - df_shell["epochDate"]).dt.
#       ↳ total_seconds() / 86400.0

# pick a freshness window, like ±3 days for LEO screening
max_age_days = 1.0

fresh_mask = df_shell["epochDate"].between(
    cfg.t_start - pd.Timedelta(days=max_age_days),
    cfg.t_start + pd.Timedelta(days=max_age_days)
)

df_shell_fresh = df_shell[fresh_mask].copy()

print("Epoch freshness filter:")
print(f" Window: |epoch - t_start| <= {max_age_days:.1f} days")
print(f" Kept:      {len(df_shell_fresh)}")
print(f" Dropped: {len(df_shell) - len(df_shell_fresh)}")

# summary of how far the kept epochs are from t_start
print("\nAge (days) among kept rows:")
print(df_shell_fresh['epoch_age_days'].describe().to_string())
```

```
# if you dropped too many rows and want to relax the window - bump max_age_days
↳ to 5-7.
# if you still drop a lot - consider redefining t_start (e.g., median/quantile
↳ of epochs).
```

Epoch freshness filter:

```
Window: |epoch - t_start| <= 1.0 days
Kept:    5874
Dropped: 219
```

Age (days) among kept rows:

```
count    5874.000000
mean      0.148621
std       0.355745
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       1.000000
```

2.5 Build Propagators

sgp4 is a standard model for predicting satellite positions from TLE data (two-line elements).

Our dataframe doesn't store raw tle strings, but it does have the equivalent parameters. We'll feed those into **sgp4.api.Satrec** objects using the function **sgp4init**.

Units and conversions: - **sgp4init** expects angles in radians, not degrees - **meanMotion** must be converted from revolutions/day to radians/minute - **epochDate** must be expressed as a Julian date split into (jd, fraction)

Variable names (inclo, nodeo, argpo, mo, no_kozai, etc.) below were chosen to match the sgp4 C/fortran heritage. The python wrapper **sgp4.api** preserves those names for consistency.

sgp4init() will return a satellite record object, called a **satrec**, that knows how to compute that satellite's position at any time. **satellites** is a list where each element is dictionary containing a satellite's metadata and how to propagate it.

```
[51]: from sgp4.api import Satrec, SGP4_ERRORS, jday, WGS72

satellites = []    # list of dictionaries with satNo, name, and Satrec object
errors = []        # tracks any rows we fail to convert

EPOCH0 = datetime(1949, 12, 31, 0, 0, 0, tzinfo=timezone.utc)

for idx, row in df_shell.iterrows(): # loop through each row in df_shell
    try:
        epoch_dt = row["epochDate"].to_pydatetime() # extract epoch as datetime

        # days since 1949-12-31 (as float)
```

```

epoch_days = (epoch_dt - EPOCH0).total_seconds() / 86400.0

# convert to radians
inclo = np.deg2rad(row["inclination_deg"])
nodeo = np.deg2rad(row["raan"])
argpo = np.deg2rad(row["argOfPerigee"])
mo     = np.deg2rad(row["meanAnomaly"])

# convert revs/day to rad/min
no_kozai = float(row["meanMotion"]) * 2.0 * np.pi / (24.0 * 60.0)

# orbital scalars
ecco = float(row["eccentricity"])      # must be in [0,1)
bstar = float(row.get("bStar", 0.0))   # ok if 0.0

# initialize satellite record
satrec = Satrec()
satrec.sgp4init(
    WGS72,                # Earth gravity model (standard for SGP4)
    'i',                  # 'i' = initialize
    int(row["satNo"]),     # satellite ID
    epoch_days,           # Julian date
    bstar,                # drag term
    0.0, 0.0,             # ndot, nddot (not used here; 0.0 okay)
    ecco,                 # eccentricity
    argpo,                # argument of perigee [rad]
    inclo,                # inclination [rad]
    mo,                   # mean anomaly [rad]
    no_kozai,             # mean motion [rad/min]
    nodeo                 # RAAN [rad]
)

satellites.append({ # add satellite to list
    "satNo": row["satNo"],
    "satName": row.get("satName", ""),
    "objectType": row.get("objectType", ""),
    "satrec": satrec
})

except Exception as e:
    errors.append((idx, str(e)))

print(f"Built {len(satellites)} propagators successfully.")
if errors:
    print(f"Failed on {len(errors)} rows. Example error:")
    print(errors[0])

```

Built 6093 propagators successfully.

```

[52]: # TEST PROPAGATION

def norm3(vec):
    return float(np.sqrt(vec[0]**2 + vec[1]**2 + vec[2]**2))
    # returns Euclidean norm of a 3-vector

# pick the first satellite we built
if not satellites:
    raise RuntimeError("No satellites in `satellites`, build propagators first")
test_sat = satellites[0] # change index if desired

# convert t_start to Julian date parts for SGP4
jd, fr = jday(
    cfg.t_start.year,
    cfg.t_start.month,
    cfg.t_start.day,
    cfg.t_start.hour,
    cfg.t_start.minute,
    cfg.t_start.second + cfg.t_start.microsecond * 1e-6
)

# propagate and inspect results
err, r, v = test_sat["satrec"].sgp4(jd, fr)

print(f"Testing '{test_sat.get('satName','')}' (satNo. {test_sat['satNo']}) at_
↳ t_start={cfg.t_start}\n")
if err != 0:
    print(f"  SGP4 ERROR: {SGP4_ERRORS[err]}")
else:
    # r, v are TEME/ECI position (km) and velocity (km/s)
    r_mag = norm3(r) # distance from Earth's center (km)
    v_mag = norm3(v) # speed (km/s)
    alt_km = r_mag - cfg.earth_radius_km

    print("  r (km):", [round(x, 3) for x in r])
    print("  v (km/s):", [round(x, 5) for x in v])
    print(f"  |r| = {r_mag:,.2f} km (altitude {alt_km:,.2f} km above mean_
↳ equator)")
    print(f"  |v| = {v_mag:,.3f} km/s")

    # LEO range check
    if 6500 <= r_mag <= 7500 and 6.5 <= v_mag <= 8.5:
        print("\nCheck: Values look reasonable for LEO ")
    else:
        print("\nCheck: Values are unusual for LEO, recheck inputs ")

```

Testing 'FENGYUN 1C DEB' (satNo. 35230) at t_start=2025-08-03 00:00:00+00:00

```

r (km): [-4223.135, -905.095, -5454.942]
v (km/s): [5.25127, 2.94944, -4.5768]
|r| = 6,957.76 km (altitude 579.63 km above mean equator)
|v| = 7.565 km/s

```

Check: Values look reasonable for LEO

3 Execution Phase

3.1 Coarse Propagation

Build a time grid from `t_start` → `t_end` every `coarse_step_minutes`.

For each time: - Propagate each satellite via `satrec.sgp4()` to get position `r = (x,y,z)` in TEME/ECI (km) - Build a KD-tree on the 3D positions - Query for all pairs within a search radius - Store those pairs as candidates with their coarse distance and timestamp

```

[53]: from scipy.spatial import cKDTree    # fast KD-tree (compiled)

# returns Euclidean norm ||a-b|| if b is provided
def norm3(a, b=None):
    if b is None:
        return float(np.sqrt((a*a).sum()))
    d = a - b
    return float(np.sqrt((d*d).sum()))

times_coarse = pd.date_range( # build shared time grid
    start = cfg.t_start,
    end = cfg.t_end,
    freq = f"{cfg.coarse_step_minutes}min",
    inclusive = "both" # include t_end
).to_pydatetime().tolist()

print(f"Coarse grid has {len(times_coarse)} timestamps "
      f"\n({cfg.t_start} to {cfg.t_end}, step={cfg.coarse_step_minutes} min)")

```

Coarse grid has 289 timestamps
(2025-08-03 00:00:00+00:00 to 2025-08-04 00:00:00+00:00, step=5 min)

```

[54]: # convenience arrays for satellite metadata
n = len(satellites)
satNos = np.array([int(s["satNo"]) for s in satellites], dtype=np.int64)
satNames = np.array([s.get("satName","") for s in satellites], dtype=object)

candidates = [] # store in a list of dicts

# simple progress printing every k steps
print_every = max(1, len(times_coarse)//12) # ~12 status lines over the run

```

3.2 Neighbor Search (KD-tree)

Why use a KD-tree?

It avoids $O(N^2)$ all-pairs checks. For ~6,000 satellites, all-pairs would be ~18M distance checks per timestep. KD-tree gives you only the nearby ones.

```
[55]: for t_idx, t in enumerate(times_coarse):
    # convert this timestamp to Julian date parts
    jd, fr = jday(t.year, t.month, t.day,
                  t.hour, t.minute, t.second + t.microsecond * 1e-6)

    # pre-allocate arrays for positions; mark invalid slots with NaN
    R = np.full((n, 3), np.nan, dtype=float)
    valid_mask = np.zeros(n, dtype=bool)

    for i, s in enumerate(satellites): # propagate all satellites to time t
        err, r, v = s["satrec"].sgp4(jd, fr)
        if err == 0:
            R[i, :] = r # km
            valid_mask[i] = True
        else:
            if t_idx == 0 and i < 3:
                print("sgp4 error:", SGP4_ERRORS.get(err, err), "for satNo", s["satNo"])

    # keep only valid positions for the KD-tree
    if not valid_mask.any():
        continue # unlikely if epochs were filtered

    Rv = R[valid_mask] # positions of valid satellites
    idx_valid = np.where(valid_mask)[0] # lookup table, tells you which
    # original satellite each row of Rv came from

    tree = cKDTree(Rv) # build KD-tree and query all pairs within coarse search
    # radius
    pair_set = tree.query_pairs(r=cfg.search_radius_km) # returns indices (i,
    # j) in the *compressed* array Rv

    if not pair_set: # skip if zero candidates at this time
        if t_idx % print_every == 0:
            print(f"t={t.isoformat()} candidates=0")
        continue

    # for each candidate pair, compute the coarse distance and collect metadata
    for (ia, ib) in pair_set:
        gi = idx_valid[ia] # global index into satellites list
        gj = idx_valid[ib]
```



```

ra = Rv[ia]
rb = Rv[ib]
d_km = norm3(ra, rb)

# altitudes (quick context, not used as a filter here)
altA = norm3(ra) - cfg.earth_radius_km
altB = norm3(rb) - cfg.earth_radius_km

candidates.append({
    "t_coarse": t,                # coarse timestamp
    "idxA": gi, "idxB": gj,      # indices into `satellites` list
    "satNoA": int(satNos[gi]),
    "satNoB": int(satNos[gj]),
    "d_coarse_km": d_km,
    "altA_km": altA,
    "altB_km": altB,
})

# PROGRESS LOG OPTIONS

# --- VERBOSE: print every timestep
# print(f"[t={t:%Y-%m-%d %H:%M:%S} cand={len(pair_set)}] ␣
↳total={len(candidates)}")

# --- QUIET: only print when candidates exist
# if pair_set:
#     print(f"HIT t={t:%Y-%m-%d %H:%M} +{len(pair_set)}] ␣
↳total={len(candidates)}")

# --- LIGHT: periodic heartbeat (default)
if t_idx % print_every == 0:
    print(f"t={t:%Y-%m-%d %H:%M} cand={len(pair_set)}] ␣
↳total={len(candidates)}")

```

```

t=2025-08-03 00:00 cand=18896 total=18896
t=2025-08-03 02:00 cand=21424 total=462047
t=2025-08-03 04:00 cand=15567 total=886074
t=2025-08-03 06:00 cand=20181 total=1310431
t=2025-08-03 08:00 cand=15430 total=1725958
t=2025-08-03 10:00 cand=19796 total=2145682
t=2025-08-03 12:00 cand=15349 total=2556526
t=2025-08-03 14:00 cand=19428 total=2977171
t=2025-08-03 16:00 cand=15333 total=3384110
t=2025-08-03 18:00 cand=19067 total=3797569
t=2025-08-03 20:00 cand=15288 total=4200928
t=2025-08-03 22:00 cand=18671 total=4612588

```

```
t=2025-08-04 00:00 cand=15259 total=5011577
```

```
[56]: candidates_df = pd.DataFrame(candidates)

if candidates_df.empty:
    print("\nNo candidates found on the coarse grid, try increasing the search_
    ↪radius or extending the time window")
else:
    # sort by time, then distance
    candidates_df.sort_values(["t_coarse", "d_coarse_km"], inplace=True)

    # preview
    preview_cols = ["t_coarse", "satNoA", "satNoB", "d_coarse_km", "altA_km",
    ↪"altB_km"]
    print("\nCandidate preview:")
    print(candidates_df[preview_cols].head(10).to_string(index=False))

    print(f"\nTotal coarse candidates collected: {len(candidates_df)}")
```

Candidate preview:

	t_coarse	satNoA	satNoB	d_coarse_km	altA_km	altB_km
2025-08-03	00:00:00+00:00	53215	53213	0.012744	566.036699	566.040957
2025-08-03	00:00:00+00:00	48367	48374	0.014597	547.983245	547.995372
2025-08-03	00:00:00+00:00	58190	56119	0.019235	558.789126	558.804504
2025-08-03	00:00:00+00:00	55420	55271	0.026606	574.568089	574.590435
2025-08-03	00:00:00+00:00	52858	52856	0.028376	540.663827	540.640100
2025-08-03	00:00:00+00:00	56918	57997	0.032693	558.750160	558.735401
2025-08-03	00:00:00+00:00	55435	55432	0.034817	574.578019	574.572876
2025-08-03	00:00:00+00:00	58195	56134	0.040379	558.828136	558.830876
2025-08-03	00:00:00+00:00	56699	57106	0.043731	558.911380	558.916606
2025-08-03	00:00:00+00:00	64532	64531	0.055167	520.602541	520.578733

Total coarse candidates collected: 5011577

3.3 Deduplication

De-duplicate coarse candidates so we don't refine the same pair many times. We'll keep the shortest coarse distance per unique pair.

```
[57]: # check: ensure we have coarse candidates
if 'candidates_df' not in locals() or candidates_df.empty:
    raise RuntimeError("candidates_df is missing or empty.")

coarse = candidates_df.copy()

# create an order-independent pair key
pair_key = np.where(coarse["satNoA"] < coarse["satNoB"],
```

```

        coarse["satNoA"].astype(str) + "-" + coarse["satNoB"].
        ↪astype(str),
        coarse["satNoB"].astype(str) + "-" + coarse["satNoA"].
        ↪astype(str))
coarse["pair_id"] = pair_key

# keep the closest coarse occurrence per pair
# (to get multiple per pair across time, group by day/hour buckets later)
coarse_best = (
    coarse.sort_values(["pair_id", "d_coarse_km"])
        .groupby("pair_id", as_index=False)
        .first()
)

print(f"Unique pairs to refine: {len(coarse_best)} (from {len(coarse)} coarse_
    ↪rows)")
print(coarse_best[["t_coarse", "satNoA", "satNoB", "d_coarse_km"]].head(10).
    ↪to_string(index=False))

```

Unique pairs to refine: 41670 (from 5011577 coarse rows)

	t_coarse	satNoA	satNoB	d_coarse_km
2025-08-03	02:20:00+00:00	10095	43678	17.895829
2025-08-03	07:40:00+00:00	42831	10096	18.119522
2025-08-03	00:40:00+00:00	65055	10188	14.506981
2025-08-03	10:15:00+00:00	10761	40290	12.681518
2025-08-03	04:00:00+00:00	30879	10974	16.870528
2025-08-03	01:30:00+00:00	34316	10974	7.241115
2025-08-03	00:00:00+00:00	52377	11114	16.188481
2025-08-03	00:25:00+00:00	11267	39416	17.834428
2025-08-03	10:00:00+00:00	11267	43490	11.359098
2025-08-03	00:45:00+00:00	11267	44886	5.931519

3.4 Local Refinement

Find true TCA and minimum distance! Steps:

- For each unique pair, build a refinement time window centered on its coarse timestamp (e.g., ± 10 minutes, step `cfg.refine_step_seconds`).
- Propagate only those two satellites across the window.
- Compute distance at each substep; pick the minimum \rightarrow that's the TCA and `d_min`.
- Grab relative speed at TCA (from SGP4 velocities).
- Save a tidy row for each refined event.

```

[58]: def refine_pair(satA, satB, t_center, half_window_minutes=10, step_seconds=30):
        # build the fine time grid centered at t_center

```

```

t_start = t_center - timedelta(minutes=half_window_minutes)
t_end    = t_center + timedelta(minutes=half_window_minutes)

# se seconds-based frequency (capital 'S')
times = pd.date_range(start=t_start, end=t_end,
                      freq=f"{step_seconds}S", inclusive="both").
↳to_pydatetime()

# pre-allocate arrays for distances and relative speed
nT = len(times)
dists = np.full(nT, np.nan, dtype=float)
vrels = np.full(nT, np.nan, dtype=float)

# loop over sub-steps; propagate both sats and compute separation and rel_
↳speed
for k, tk in enumerate(times):
    jd, fr = jday(tk.year, tk.month, tk.day,
                  tk.hour, tk.minute, tk.second + tk.microsecond * 1e-6)

    errA, rA, vA = satA["satrec"].sgp4(jd, fr)
    errB, rB, vB = satB["satrec"].sgp4(jd, fr)

    if errA != 0 or errB != 0:
        # skip this sub-step if either failed (can happen near stale epochs)
        continue

    # distance between positions (km)
    dists[k] = norm3(np.array(rA) - np.array(rB))

    # relative speed magnitude (km/s)
    rel_v = np.array(vA) - np.array(vB)
    vrels[k] = norm3(rel_v)

# choose minimum valid distance
if np.all(np.isnan(dists)):
    return {
        "satNoA": satA["satNo"], "satNoB": satB["satNo"],
        "satNameA": satA.get("satName", ""), "satNameB": satB.
↳get("satName", ""),
        "t_TCA": None, "d_min_km": np.nan, "v_rel_km_s": np.nan,
        "t_center": t_center, "n_steps": nT,
        "status": "refine_failed_all_nan"
    }

kmin = np.nanargmin(dists)
return {
    "satNoA": satA["satNo"], "satNoB": satB["satNo"],

```

```

        "satNameA": satA.get("satName",""), "satNameB": satB.get("satName",""),
        "t_TCA": times[kmin],
        "d_min_km": float(dists[kmin]),
        "v_rel_km_s": float(vrels[kmin]) if not np.isnan(vrels[kmin]) else np.
nan,
        "t_center": t_center, # coarse time around which we refine

        "n_steps": nT,
        "status": "ok"
    }

```

```

[59]: refined_rows = []
half_window_minutes = 10 # ±10 minutes around the coarse time
step_seconds = cfg.refine_step_seconds

for _, row in coarse_best.iterrows():
    iA = int(row["idxA"]) # indices into satellites
    iB = int(row["idxB"])
    t_center = pd.to_datetime(row["t_coarse"], utc=True).to_pydatetime()

    satA = satellites[iA]
    satB = satellites[iB]

    result = refine_pair(
        satA, satB, t_center,
        half_window_minutes=half_window_minutes,
        step_seconds=step_seconds
    )
    refined_rows.append(result)

refined_df = pd.DataFrame(refined_rows)

print(f"Refinement complete. Rows: {len(refined_df)}")
print(refined_df.head(10).to_string(index=False))

```

C:\Users\ash\AppData\Local\Temp\ipykernel_22416\2767047286.py:8: FutureWarning: 'S' is deprecated and will be removed in a future version, please use 's' instead.

```
times = pd.date_range(start=t_start, end=t_end,
```

Refinement complete. Rows: 41670

satNoA	satNoB	satNameA	satNameB	t_TCA	d_min_km
v_rel_km_s		t_center	n_steps	status	
10095	43678	COSMOS 921	DIWATA 2B	2025-08-03 02:20:00+00:00	17.895829
3.029118	2025-08-03 02:20:00+00:00		41	ok	
42831	10096	FLYING LAPTOP	SL-14 R/B	2025-08-03 07:40:00+00:00	18.119522
11.653229	2025-08-03 07:40:00+00:00		41	ok	
65055	10188	PRSC-S1 DELTA 1	DEB	2025-08-03 00:40:00+00:00	14.506981

```

1.997945 2025-08-03 00:40:00+00:00      41      ok
   10761   40290      DELTA 1 DEB    CZ-2C DEB 2025-08-03 10:15:00+00:00 12.681518
14.566158 2025-08-03 10:15:00+00:00      41      ok
   30879   10974  FENGYUN 1C DEB    SL-14 R/B 2025-08-03 04:00:00+00:00 16.870528
15.185883 2025-08-03 04:00:00+00:00      41      ok
   34316   10974  COSMOS 2251 DEB    SL-14 R/B 2025-08-03 01:30:00+00:00  7.241115
1.227373 2025-08-03 01:30:00+00:00      41      ok
   52377   11114  STARLINK-3805    SL-8 DEB 2025-08-03 00:00:00+00:00 16.188481
2.736324 2025-08-03 00:00:00+00:00      41      ok
   11267   39416      SL-14 R/B APRIZESAT 7 2025-08-03 00:25:00+00:00 17.834428
13.970884 2025-08-03 00:25:00+00:00      41      ok
   11267   43490      SL-14 R/B    CZ-2D DEB 2025-08-03 10:00:00+00:00 11.359098
7.671557 2025-08-03 10:00:00+00:00      41      ok
   11267   44886      SL-14 R/B    OBJECT H 2025-08-03 00:45:00+00:00  5.931519
2.051345 2025-08-03 00:45:00+00:00      41      ok

```

3.4.1 status column tags: ok or fail

Not every coarse candidate can be successfully refined. Common reasons:

- bad/missing TLE parameters for one of the objects
- numerical failure in the propagator
- epochs too far out of date

```

[60]: # drop rows where refinement failed completely
events_df = refined_df[refined_df["status"] == "ok"].copy()
events_df.info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41670 entries, 0 to 41669
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  -
 0   satNoA          41670 non-null  int64
 1   satNoB          41670 non-null  int64
 2   satNameA        41670 non-null  object
 3   satNameB        41670 non-null  object
 4   t_TCA           41670 non-null  datetime64[ns, UTC]
 5   d_min_km        41670 non-null  float64
 6   v_rel_km_s      41670 non-null  float64
 7   t_center        41670 non-null  datetime64[ns, UTC]
 8   n_steps         41670 non-null  int64
 9   status          41670 non-null  object
dtypes: datetime64[ns, UTC] (2), float64(2), int64(3), object(3)
memory usage: 3.2+ MB

```

```

[61]: total = len(refined_df) # total coarse candidates refined

status_counts = refined_df["status"].value_counts()

```

```

success = status_counts.get("ok", 0)
fail = total - success
success_rate = 100.0 * success / total if total > 0 else 0.0

print(f"Refinement results:")
print(f"  Total coarse candidates: {total}")
print(f"  Successful refinements:  {success} ({success_rate:.1f}%)")
print(f"  Failed refinements:      {fail}")

```

```

Refinement results:
  Total coarse candidates: 41670
  Successful refinements:  41670 (100.0%)
  Failed refinements:      0

```

4 Analysis and Reporting

4.1 Totals

```

[62]: # apply thresholds
events_df["below_10km"] = events_df["d_min_km"] < cfg.report_thresh_km
events_df["below_5km"]  = events_df["d_min_km"] < cfg.report_strict_km

events_10 = events_df[events_df["below_10km"]].copy()
events_5  = events_df[events_df["below_5km"]].copy()

# sort by TCA then by distance
events_10.sort_values(["t_TCA", "d_min_km"], inplace=True)
events_5.sort_values(["t_TCA", "d_min_km"], inplace=True)

```

```

[125]: print(f"Events under {cfg.report_thresh_km} km: {len(events_10)}")
print(f"Events under {cfg.report_strict_km} km:  {len(events_5)}\n")

print(f"Conjunctions by shortest distance (Top 10):")
preview_cols = ["satNoA", "satNoB", "d_min_km", "v_rel_km_s"]
print(events_5[preview_cols].head(10).to_string(index=False))

```

```

Events under 10.0 km: 20307
Events under 5.0 km:  12959

```

Conjunctions by shortest distance (Top 10):

satNoA	satNoB	d_min_km	v_rel_km_s
52092	52310	0.045807	0.000165
63096	63080	0.143227	0.000305
64544	64543	0.191631	0.000303
62712	62706	0.350875	0.000761
64092	64090	0.383235	0.000776
56349	56352	0.391792	0.000352

51758	51756	0.482989	0.000531
58211	58218	0.519249	0.000640
64155	64166	0.600434	0.001913
51152	51754	0.621888	0.000807

```
[82]: # count by day/hour (for multi-day runs)
events_10["date"] = events_10["t_TCA"].dt.floor("D")
daily_10 = events_10.groupby("date").size()

events_5["date"] = events_5["t_TCA"].dt.floor("D")
daily_5 = events_5.groupby("date").size()

daily_table = pd.DataFrame({
    "<5 km": daily_5,
    "<10 km": daily_10
})

daily_table.index = daily_table.index.strftime("%Y-%m-%d")
daily_table
```

```
[82]:
```

	<5 km	<10 km
date		
2025-08-02	115	173
2025-08-03	12842	20132
2025-08-04	2	2

4.2 Which object types dominate conjunction risk?

- by pair count
- by weighted risk

4.2.1 By pair count

```
[65]: df_events = events_df.copy()

# normalize pair ID so A-B and B-A collapse to the same key
pair_id_refined = np.where(df_events["satNoA"] < df_events["satNoB"],
                           df_events["satNoA"].astype(str) + "-" +
                           df_events["satNoB"].astype(str),
                           df_events["satNoB"].astype(str) + "-" +
                           df_events["satNoA"].astype(str))
df_events["pair_id"] = pair_id_refined

df_events.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41670 entries, 0 to 41669
Data columns (total 13 columns):
```


#	Column	Non-Null Count	Dtype
0	satNoA	41670 non-null	int64
1	satNoB	41670 non-null	int64
2	satNameA	41670 non-null	object
3	satNameB	41670 non-null	object
4	t_TCA	41670 non-null	datetime64[ns, UTC]
5	d_min_km	41670 non-null	float64
6	v_rel_km_s	41670 non-null	float64
7	t_center	41670 non-null	datetime64[ns, UTC]
8	n_steps	41670 non-null	int64
9	status	41670 non-null	object
10	below_10km	41670 non-null	bool
11	below_5km	41670 non-null	bool
12	pair_id	41670 non-null	object

dtypes: bool(2), datetime64[ns, UTC](2), float64(2), int64(3), object(4)
memory usage: 3.6+ MB

```
[66]: # map satNo -> objectType
id_to_type = df_shell.set_index("satNo")["objectType"]

# annotate conjunction pairs with object types
df_conj = df_events.assign(
    type1 = df_events["satNoA"].map(id_to_type),
    type2 = df_events["satNoB"].map(id_to_type)
).copy()

# categorize the pair (order independent)
def pair_category(row):
    t1, t2 = sorted([row["type1"], row["type2"]])
    return f"{t1}-{t2}"

df_conj["pair_type"] = df_conj.apply(pair_category, axis=1)
df_conj.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41670 entries, 0 to 41669
Data columns (total 16 columns):
#   Column      Non-Null Count  Dtype
---  -
0   satNoA      41670 non-null  int64
1   satNoB      41670 non-null  int64
2   satNameA    41670 non-null  object
3   satNameB    41670 non-null  object
4   t_TCA       41670 non-null  datetime64[ns, UTC]
5   d_min_km    41670 non-null  float64
6   v_rel_km_s  41670 non-null  float64
7   t_center    41670 non-null  datetime64[ns, UTC]
```

```

8   n_steps      41670 non-null  int64
9   status       41670 non-null  object
10  below_10km   41670 non-null  bool
11  below_5km    41670 non-null  bool
12  pair_id      41670 non-null  object
13  type1        41670 non-null  object
14  type2        41670 non-null  object
15  pair_type    41670 non-null  object
dtypes: bool(2), datetime64[ns, UTC](2), float64(2), int64(3), object(7)
memory usage: 4.5+ MB

```

```

[115]: # aggregate
pair_summary = (
    df_conj.groupby("pair_type")
    .agg(
        n_events=("d_min_km", "size"),
        median_miss_km=("d_min_km", "median"),
        min_miss_km=("d_min_km", "min"),
        n_events_below_1km=("d_min_km", lambda x: (x < 1).sum())
    )
    .sort_values("n_events", ascending=False)
    .rename_axis("pair_type").reset_index()
)
pair_summary

```

```

[115]:

```

	pair_type	n_events	median_miss_km	min_miss_km	\
0	Payload-Payload	38713	10.145515	0.006495	
1	Payload-Unknown	958	10.452237	0.110107	
2	Debris-Payload	910	14.500108	0.411050	
3	Unknown-Unknown	494	7.407047	0.030465	
4	Payload-Rocket body	293	14.086566	0.265819	
5	Debris-Debris	193	12.881185	1.183780	
6	Debris-Rocket body	43	14.850467	4.545728	
7	Debris-Unknown	37	12.814541	1.015493	
8	Rocket body-Unknown	19	13.422841	1.979328	
9	Rocket body-Rocket body	10	18.328455	2.595006	

	n_events_below_1km
0	5249
1	32
2	1
3	42
4	2
5	0
6	0
7	0
8	0

4.2.2 By weighted risk

We want close approaches to count more heavily than distant ones.

- 0.1 km miss \rightarrow weight = 10
- 1.0 km miss \rightarrow weight = 1
- 10 km miss \rightarrow weight = 0.1

```
[117]: # map to buckets
bucket_map = {"Payload": "Payload", "Rocket body": "Debris", "Debris": "Debris", "Unknown": "Unknown"}

dfw2 = df_conj.copy()
dfw2["bucket1"] = dfw2["type1"].map(bucket_map).fillna("Other/Unknown")
dfw2["bucket2"] = dfw2["type2"].map(bucket_map).fillna("Other/Unknown")

# stack both sides so each event counts for both participants
tall = pd.concat([
    dfw2[["d_min_km", "bucket1"]].rename(columns={"bucket1": "bucket"}),
    dfw2[["d_min_km", "bucket2"]].rename(columns={"bucket2": "bucket"})
], ignore_index=True)

# recompute weights on the stacked view
tall["weight"] = 1.0 / (tall["d_min_km"] + 1e-6)

bucket_summary = (
    tall.groupby("bucket")
    .agg(
        weighted_risk=("weight", "sum"),
        count=("bucket", "size"),
        q1_miss_km=("d_min_km", lambda x: x.quantile(0.25)),
        median_miss_km=("d_min_km", "median"),
        q3_miss_km=("d_min_km", lambda x: x.quantile(0.75))
    )
    .sort_values("weighted_risk", ascending=False)
    .rename_axis("object_type").reset_index()
    .assign(
        weighted_risk=lambda df: df["weighted_risk"].round(0).astype(int),
        q1_miss_km=lambda df: df["q1_miss_km"].round(2),
        median_miss_km=lambda df: df["median_miss_km"].round(2),
        q3_miss_km=lambda df: df["q3_miss_km"].round(2),
        risk_share=lambda df: (
            (100 * df["weighted_risk"] / df["weighted_risk"].sum())
            .round(1).astype(str) + "%"
        )
    )
)
```

```
bucket_summary
```

```
[117]:
```

	object_type	weighted_risk	count	q1_miss_km	median_miss_km	q3_miss_km	\
0	Payload	60168	79587	3.07	10.22	14.96	
1	Unknown	678	2002	4.51	8.85	13.58	
2	Debris	187	1751	9.65	14.11	16.98	

	risk_share
0	98.6%
1	1.1%
2	0.3%

4.3 Summary and Recommendations

Our analysis confirms that close approaches in the thousands can occur daily in low-Earth orbit.

Median miss distances are typically in the 8-14 km range. For Payload-Payload conjunctions, 5249 events (about 14%) have miss distances below 1 km. Following with 74 events with miss distances below 1 km, conjunctions involving Unknown objects should not be overlooked. Although debris and unknown objects contribute marginally, their presence still complicates risk management.

Weighted risk is dominated by payloads at 98.6%. With Q1 miss distances lower than those of other objects, conjunctions between operational spacecraft are both more frequent and riskier.

Recommendations

Space Traffic Management - Establish binding international standards for conjunction assessment and collision avoidance - Mandate maneuver protocols and thresholds, e.g. automated avoidance systems - Classify “Unknown” objects to better assess risk - Promote international agreements on information sharing, debris mitigation, and maneuver coordination

```
[122]: # export
FINAL_DIR = Path.cwd().parents[1] / "data" / "02_final" / "conjunction_risk"

events_10.to_csv(FINAL_DIR / "conjunctions_df.csv", index=False)
pair_summary.to_csv(FINAL_DIR / "pair_summary.csv", index=False)
bucket_summary.to_csv(FINAL_DIR / "bucket_summary.csv", index=False)
```