

Dust Clearance via Resonances in a Rotating Binary (Sun–Jupiter) System

Department of Physics, University of California, Santa Barbara

Group 3: Hongyu, Alex, Kuntong, Shibiao, Zhe

Instructor: David Berenstein

June 2025

1 Introduction

Dust and small bodies around stars experience complex dynamical interactions. Understanding how dust particles evolve under the gravitational influence of two stars (or a star and a giant planet) has applications to:

- **Debris disks and planetesimal formation:** The clearing or accumulation of dust in binary systems informs models of how terrestrial planets form in binary or multi-star environments.
- **Circumstellar dynamics:** Over 50% of Sun-like stars are in binaries. Simulating dust in a simplified two-body configuration helps elucidate how binaries clear or trap dust, affecting observable debris disks.
- **Asteroid-belt analogues:** The classical Kirkwood gaps in our own Solar System’s asteroid belt arise from Jupiter-driven resonances. By modeling a mass-ratio 1:3 binary at separation unity, we can reproduce analogous resonance gaps and study clearing efficiency.

In this lab project, we simulate dust-particle motion in both non-dimensionalized, planar, inertial and rotating frames of two massive bodies (“Sun” mass $M_1 = 1$, “Jupiter” mass $M_2 = 3$). Our goal is to show how such a binary clears dust via mean-motion resonances, and to compare surviving fractions across initial semi-major axes.

2 Theory

In this project we simulate a planar, non-dimensionalized “Sun–Jupiter–dust” system in the *inertial frame*. Two massive bodies of masses $M_1 = 1$ and $M_2 = 3$ are placed on circular orbits around their common center of mass (COM), separated by $D = 1$. Dust particles move under Newtonian gravity from both primaries. Below we summarize the key equations and numerical implementation.

2.1 Equations of Motion

Denote

$$G = 1, \quad M_1 = 1, \quad M_2 = 3, \quad D = 1,$$

so that the two primaries orbit each other with angular frequency

$$\omega = \sqrt{\frac{G(M_1 + M_2)}{D^3}} = \sqrt{\frac{4}{1^3}} = 2.$$

We place the COM at the origin,

$$\mathbf{r}_{\text{COM}} = (0, 0),$$

and at $t = 0$ choose the two stars on opposite sides of the COM along the x -axis:

$$\mathbf{r}_1(0) = \left(-D \frac{M_2}{M_1 + M_2}, 0\right) = \left(-\frac{3}{4}, 0\right), \quad \mathbf{r}_2(0) = \left(+D \frac{M_1}{M_1 + M_2}, 0\right) = \left(+\frac{1}{4}, 0\right).$$

At each time step n , the primaries execute a rigid rotation about the COM by angle $\omega \Delta t$. In matrix form, if

$$\theta = \omega \Delta t = 2 \times 0.02 = 0.04, \quad R(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix},$$

then

$$\mathbf{r}_1^{n+1} = \mathbf{r}_{\text{COM}} + R(\theta) (\mathbf{r}_1^n - \mathbf{r}_{\text{COM}}), \quad \mathbf{r}_2^{n+1} = \mathbf{r}_{\text{COM}} + R(\theta) (\mathbf{r}_2^n - \mathbf{r}_{\text{COM}}).$$

Thus each primary follows a perfect circle of radius $\frac{3}{4}$ (for M_1) or $\frac{1}{4}$ (for M_2) around the COM.

Each dust particle at position $\mathbf{r} = (x, y)$ feels combined acceleration

$$\ddot{\mathbf{r}} = \mathbf{a}(\mathbf{r}) = G \left[M_1 \frac{\mathbf{r}_1 - \mathbf{r}}{\|\mathbf{r}_1 - \mathbf{r}\|^3} + M_2 \frac{\mathbf{r}_2 - \mathbf{r}}{\|\mathbf{r}_2 - \mathbf{r}\|^3} \right].$$

In practice, to avoid singularities at very small separation, we enforce a minimum distance of 0.05 when computing $\|\mathbf{r}_i - \mathbf{r}\|$. Hence, for each dust particle the gravitational acceleration is

$$\mathbf{a}(\mathbf{r}) = G \left[M_1 \frac{\mathbf{r}_1 - \mathbf{r}}{\max(\|\mathbf{r}_1 - \mathbf{r}\|, 0.05)^3} + M_2 \frac{\mathbf{r}_2 - \mathbf{r}}{\max(\|\mathbf{r}_2 - \mathbf{r}\|, 0.05)^3} \right].$$

2.2 Choice of Numerical Integrator: Why RK4?

In principle, one could integrate the dust-particle equations of motion using a variety of explicit methods—e.g. Forward Euler, a second-order Runge–Kutta (RK2), or the classical fourth-order Runge–Kutta (RK4). However, compared to Forward Euler (which is only first-order accurate and accumulates large energy-drift unless Δt is extremely small) and RK2 (which, although simple and only requires two force evaluations per step, still exhibits significant phase-space drift over thousands of steps), RK4 offers a crucial advantage: by sampling

the acceleration at four points within each time step, it achieves $\mathcal{O}(\Delta t^4)$ local truncation error. In practice—when integrating thousands of dust particles for 4000 steps—RK4 remains essentially indistinguishable from the true solution for simple test problems (e.g. solving $\dot{y} = y$), whereas RK2 begins to lag and Euler diverges catastrophically. Thus, RK4 allows us to choose a moderately large step size $\Delta t = 0.02$ without incurring unacceptable energy or orbital-phase errors. For these reasons (accuracy per computational cost, particularly over long-duration runs), we employ RK4 for dust integration rather than Euler or RK2.

2.3 Runge–Kutta 4 Integration for Dust

We advance each dust particle $\{\mathbf{r}, \mathbf{v}\}$ using a classical 4th-order Runge–Kutta (RK4) step of size $\Delta t = 0.02$. Denoting

$$\mathbf{k}_1^v = \mathbf{a}(\mathbf{r}), \quad \mathbf{k}_1^r = \mathbf{v},$$

we compute

$$\begin{aligned} \mathbf{k}_2^v &= \mathbf{a}\left(\mathbf{r} + \frac{\Delta t}{2} \mathbf{k}_1^r\right), & \mathbf{k}_2^r &= \mathbf{v} + \frac{\Delta t}{2} \mathbf{k}_1^v, \\ \mathbf{k}_3^v &= \mathbf{a}\left(\mathbf{r} + \frac{\Delta t}{2} \mathbf{k}_2^r\right), & \mathbf{k}_3^r &= \mathbf{v} + \frac{\Delta t}{2} \mathbf{k}_2^v, \\ \mathbf{k}_4^v &= \mathbf{a}\left(\mathbf{r} + \Delta t \mathbf{k}_3^r\right), & \mathbf{k}_4^r &= \mathbf{v} + \Delta t \mathbf{k}_3^v. \end{aligned}$$

Then the position and velocity update are

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \frac{\Delta t}{6} \left(\mathbf{k}_1^r + 2 \mathbf{k}_2^r + 2 \mathbf{k}_3^r + \mathbf{k}_4^r \right), \quad \mathbf{v}^{n+1} = \mathbf{v}^n + \frac{\Delta t}{6} \left(\mathbf{k}_1^v + 2 \mathbf{k}_2^v + 2 \mathbf{k}_3^v + \mathbf{k}_4^v \right).$$

2.4 Initialization and Removal Criteria

- **Dust Initialization.** We sample $N = 5000$ dust particles uniformly in a ring of radii $r_0 \in [0.5, 2.0]$ with random polar angle $\theta \in [0, 2\pi)$. In Cartesian form:

$$x = r_0 \cos \theta, \quad y = r_0 \sin \theta.$$

- **Initial Velocity.** Each dust particle is assigned a Keplerian circular speed about the two-body mass $M_1 + M_2$,

$$v_{\text{kep}} = \sqrt{\frac{G(M_1 + M_2)}{\sqrt{x^2 + y^2}}},$$

directed tangentially. Concretely,

$$v_x = -v_{\text{kep}} \sin \theta + \mathcal{N}(0, 0.05), \quad v_y = +v_{\text{kep}} \cos \theta + \mathcal{N}(0, 0.05),$$

where $\mathcal{N}(0, 0.05)$ is Gaussian noise with standard deviation 0.05.

- **Removal Conditions.** After each RK4 update, any dust particle whose updated position \mathbf{r} satisfies

$$\min\{\|\mathbf{r} - \mathbf{r}_1\|, \|\mathbf{r} - \mathbf{r}_2\|\} < R_{\text{ACC}} = 0.1 \quad \text{or} \quad \|\mathbf{r}\| > R_{\text{EJECT}} = 50$$

is declared “accreted” and its coordinates are set to NaN. Such particles are excluded from further integration and plotting.

- **Storage of History.** We record at each time step $n = 0, \dots, 3999$:
 - The instantaneous positions of the two stars in arrays $\star 1_n, \star 2_n$.
 - The dust positions $\{\mathbf{r}_i^n\}_{i=1}^{5000}$ in a 3D array of shape (4000, 5000, 2).
 - We also accumulate two density profiles via histograms (next section).

2.5 Density Profiles

At each time step we compute two one-dimensional histograms (“radial densities”) of surviving dust:

1. **COM-centric histogram:** Compute each surviving particle’s distance

$$r_{\text{COM}} = \|\mathbf{r}_{\text{dust}} - \mathbf{r}_{\text{COM}}\|,$$

then bin into $B = 300$ equally spaced intervals over $[0, 3.0]$. The resulting counts per bin are stored in `com_density[n, :]`.

2. **M2-centric histogram:** Compute each surviving particle’s distance

$$r_2 = \|\mathbf{r}_{\text{dust}} - \mathbf{r}_2\|,$$

then bin into $B = 300$ intervals over $[0, 1.5]$ (half the COM range). We store these counts in `m2_density[n, :]`.

These two time-dependent density arrays allow us to visualize how the dust distribution evolves both relative to the COM and relative to the secondary of mass M_2 .

2.6 Rotating-Frame Formulation

We also use the same settings to run the simulation in a co-rotating (synodic) frame where the two primary masses $M_1 = 1$ and $M_2 = 3$ remain *fixed* in space. This simplifies visualization (both stars appear stationary) and makes it straightforward to identify resonance gaps in the dust distribution. Below we summarize the rotating-frame equations of motion are implemented in the code:

Fixed primary positions. The total mass is

$$M_{\text{tot}} = M_1 + M_2 = 4, \quad \mu = \frac{M_2}{M_{\text{tot}}} = \frac{3}{4}.$$

In a non-dimensionalized system with separation $D = 1$, the two stars lie at the constant coordinates

$$\mathbf{r}_1 = (-\mu, 0) = (-0.75, 0), \quad \mathbf{r}_2 = (1 - \mu, 0) = (0.25, 0).$$

Their angular velocity about the barycenter is

$$\omega = \sqrt{\frac{G M_{\text{tot}}}{D^3}} = \sqrt{\frac{1 \times 4}{1^3}} = 2.$$

Because we remain in the rotating frame, these two points never move in our plots or in the integration.

Dust initialization. We again generate $N = 5000$ dust particles uniformly in polar coordinates:

$$r_0 \sim \mathcal{U}(0.5, 2.0), \quad \theta_0 \sim \mathcal{U}(0, 2\pi),$$

so that

$$x_0 = r_0 \cos \theta_0, \quad y_0 = r_0 \sin \theta_0.$$

Each particle's initial speed is set to the *Keplerian circular velocity* about the total mass M_{tot} at radius r_0 :

$$v_{\text{kep}} = \sqrt{\frac{G M_{\text{tot}}}{r_0}} = \sqrt{\frac{4}{r_0}},$$

oriented tangentially. In Cartesian components:

$$v_x = -v_{\text{kep}} \sin \theta_0, \quad v_y = +v_{\text{kep}} \cos \theta_0.$$

Since we work in the rotating frame, we subtract the rigid-rotation velocity $\omega \hat{\mathbf{z}} \times \mathbf{r}$. Concretely,

$$v_{\text{rot}}^x = -\omega y_0, \quad v_{\text{rot}}^y = +\omega x_0,$$

so that each particle's initial rotating-frame velocity becomes

$$v'_x = v_x - v_{\text{rot}}^x, \quad v'_y = v_y - v_{\text{rot}}^y.$$

Finally, we add a small Gaussian perturbation $\mathcal{N}(0, 0.05)$ independently to each component ($\sigma = 0.05$) to model random noise. This completes the rotating-frame initialization:

$$(\mathbf{r}_i, \mathbf{v}_i) = (x_0, y_0), \quad (v'_x + \delta v_x, v'_y + \delta v_y).$$

Equations of motion in rotating frame. A dust particle at $\mathbf{r} = (x, y)$ with rotating-frame velocity $\mathbf{v} = (v_x, v_y)$ experiences three contributions to its acceleration:

$$\ddot{\mathbf{r}} = \mathbf{a}(\mathbf{r}, \mathbf{v}) = \underbrace{-G \left[M_1 \frac{(\mathbf{r} - \mathbf{r}_1)}{\|\mathbf{r} - \mathbf{r}_1\|^3} + M_2 \frac{(\mathbf{r} - \mathbf{r}_2)}{\|\mathbf{r} - \mathbf{r}_2\|^3} \right]}_{\text{gravitational pull}} + \underbrace{\omega^2 \mathbf{r}}_{\text{centrifugal}} + \underbrace{2\omega (v_y, -v_x)}_{\text{Coriolis}}.$$

Numerically, to avoid singularities at extremely small star–dust separations, we enforce $\|\mathbf{r} - \mathbf{r}_i\| \geq 0.05$ when computing the gravitational term. In code form, for a batch of particle positions `pos[: , :]` and velocities `vel[: , :]`, we do:

```

dist1 = np.linalg.norm(pos - star1_pos, axis=1).clip(min=0.05)
dist2 = np.linalg.norm(pos - star2_pos, axis=1).clip(min=0.05)
a_grav = -G * ( M1 * (pos - star1_pos)/dist1**3
               + M2 * (pos - star2_pos)/dist2**3 )
a_cent = omega**2 * pos
a_corl = np.empty_like(pos)
a_corl[:,0] = +2*omega * vel[:,1]
a_corl[:,1] = -2*omega * vel[:,0]
acc = a_grav + a_cent + a_corl

```

Fourth-order Runge–Kutta (RK4) step in rotating frame. We advance the dust system using the classical RK4 integrator on the coupled system

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \quad \frac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{r}, \mathbf{v}).$$

At each step n , we compute

$$\begin{aligned}
k_1^r &= \mathbf{v}^n, & k_1^v &= \mathbf{a}(\mathbf{r}^n, \mathbf{v}^n), \\
k_2^r &= \mathbf{v}^n + \frac{\Delta t}{2} k_1^v, & k_2^v &= \mathbf{a}\left(\mathbf{r}^n + \frac{\Delta t}{2} k_1^r, k_2^r\right), \\
k_3^r &= \mathbf{v}^n + \frac{\Delta t}{2} k_2^v, & k_3^v &= \mathbf{a}\left(\mathbf{r}^n + \frac{\Delta t}{2} k_2^r, k_3^r\right), \\
k_4^r &= \mathbf{v}^n + \Delta t k_3^v, & k_4^v &= \mathbf{a}\left(\mathbf{r}^n + \Delta t k_3^r, k_4^r\right).
\end{aligned}$$

Then the updates are

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \frac{\Delta t}{6} (k_1^r + 2k_2^r + 2k_3^r + k_4^r), \quad \mathbf{v}^{n+1} = \mathbf{v}^n + \frac{\Delta t}{6} (k_1^v + 2k_2^v + 2k_3^v + k_4^v).$$

This step is implemented vectorized for all surviving dust particles at once.

Removal criterion. After each RK4 update, any dust with

$$\min\{\|\mathbf{r} - \mathbf{r}_1\|, \|\mathbf{r} - \mathbf{r}_2\|\} < 0.1$$

is set to NaN (accreted) and excluded from further steps. We do *not* explicitly check for ejection ($\|\mathbf{r}\| > 50$) in the rotating-frame code, because most resonant orbits either collide or become highly unstable and rapidly leave the plotted region; any remnant beyond plotting range appears as NaN and is ignored in the histograms.

Density histograms. At each of $n = 0, 1, \dots, 3999$ we bin surviving dust radii both relative to the COM and relative to the secondary M_2 :

$$r_{\text{COM}} = \|\mathbf{r} - \mathbf{r}_{\text{COM}}\|, \quad r_2 = \|\mathbf{r} - \mathbf{r}_2\|.$$

We discretize $r_{\text{COM}} \in [0, 3.0]$ into $B = 300$ bins and $r_2 \in [0, 3]$ into 300 bins, storing the counts in arrays `com_density[n, :]` and `m2_density[n, :]`. These time-dependent histograms allow us to generate side-by-side “density vs. radius” plots at each frame of the animation.

Comparison with Inertial-Frame RK4. We ran both the simulations and generated two identical results. The code for rotating-frame runs much faster compared to the code for inertial frame. So we mainly use the rotating-frame to analyze the data. Here we will also present only rotating-frame results (so that both primaries remain fixed in the animation) to simplify presentation. we show only rotating-frame screenshots and histograms, with the understanding that inertial-frame and rotating-frame results coincide to within numerical error. The code for both frames will be provided at the end of this report.

3 Results

3.1 Evolution and Animation

We ran the RK4 integration for 4000 steps ($\Delta t = 0.02$), which corresponds to a nondimensional time span of 80 units. Figure 1 shows three representative snapshots from the triple-panel animation (star positions + dust scatter, M_2 -centric density, COM-centric density).

Qualitative Evolution:

- *Initial (Step 0):* Dust is uniformly distributed in a ring $0.5 \leq r \leq 2.0$. Stars lie at ± 0.25 on the x -axis.
- *Mid (Step 2000):* Many particles have either collided with one of the primaries (distance < 0.1) or moved beyond $r > 50$ and been removed. Survivors form transient spiral structures, particularly around M_2 . The M_2 -centric density histogram begins to show a trough near $r_2 \approx 0.7$ – 0.8 .
- *Final (Step 3999):* Only particles on stable, nonresonant orbits remain. In the COM-centric histogram, deep dips appear at $r_{\text{COM}} \approx 0.63$ and $r_{\text{COM}} \approx 0.87$, corresponding to 2:1 and 3:2 mean-motion resonances with the secondary. A pronounced outer ring of survivors exists around $r_{\text{COM}} \approx 1.8$ – 2.0 .

3.2 M_2 -Centric and COM-Centric Density Profiles

At the final time step $n = 3999$, we extract and plot the two radial density profiles as bar charts (Figure 2).

Key Features:

- **2:1 Resonance Gap** ($r_2 \approx 0.63$). In both histograms, the bin centered at $r_2 \approx 0.63$ shows near-zero surviving dust. Particles initially placed near this radius experience coherent gravitational kicks from M_2 (period ratio 2:1) and are quickly scattered or accreted.
- **Outer Survivor Ring** ($r_2 \approx 1.8$ – 2.5). A great population of particles remains at this larger radii. These dust grains never cross the critical radii of either star and thus survive until the end of the simulation. Their distribution peaks around $r_{\text{COM}} \approx 2.0$.

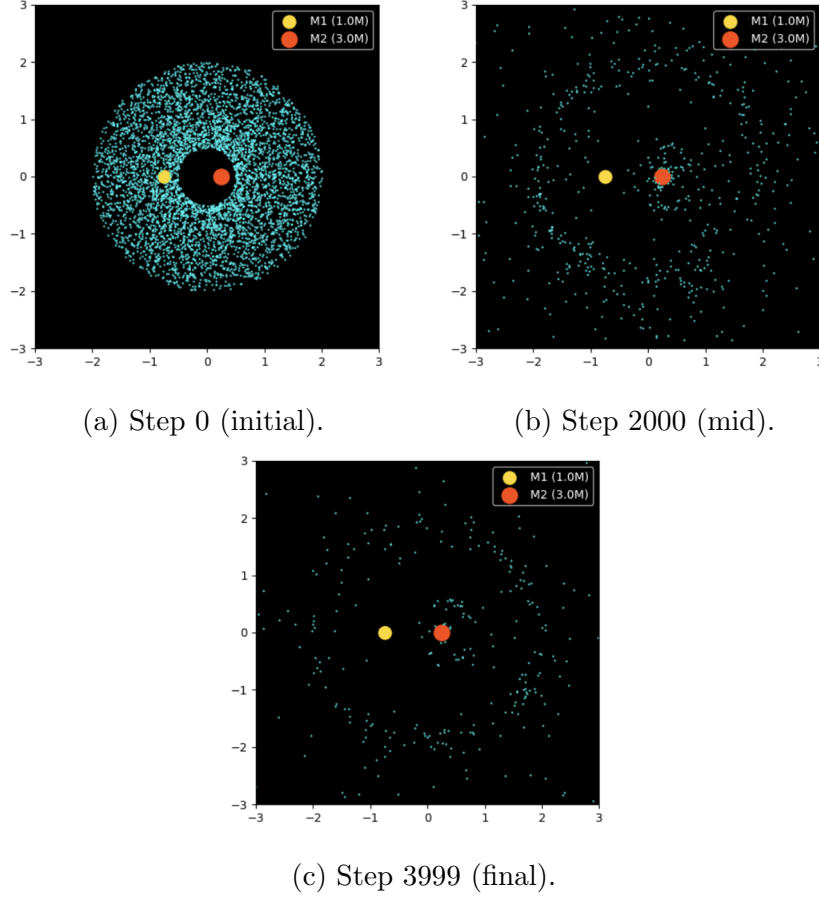
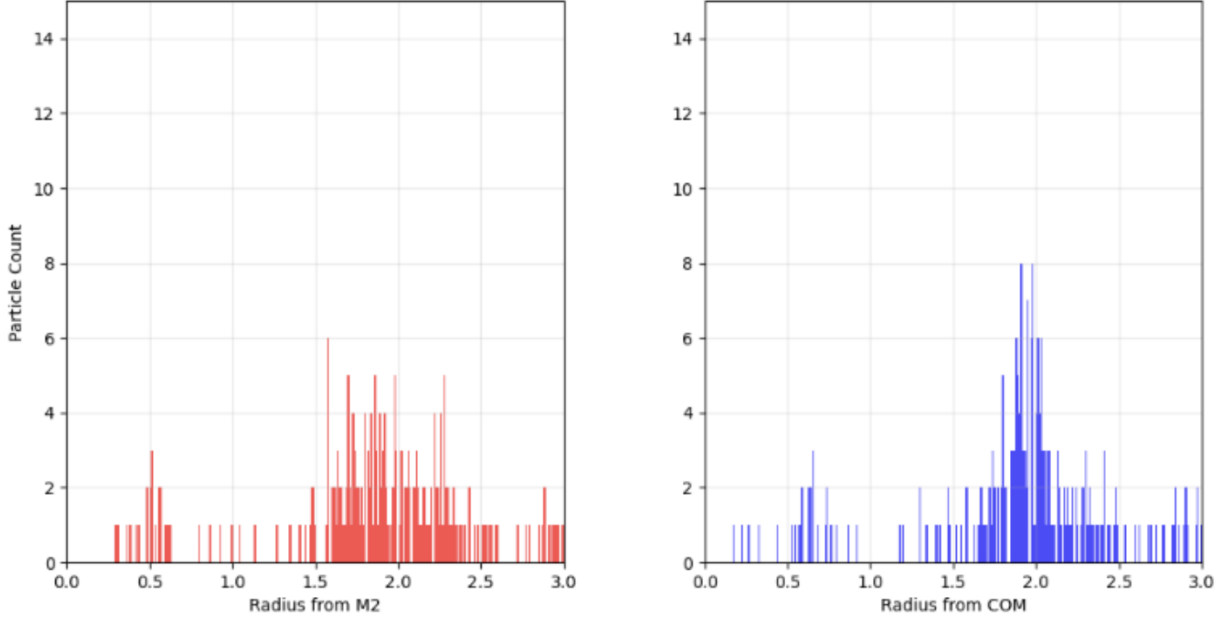


Figure 1: Snapshots from the rotating-frame animation. *Left:* System view with two stars (gold/orange) and dust (cyan, up to 75 randomly chosen survivors). *Middle:* Radial histogram of dust density relative to M_2 . *Right:* Radial histogram of dust density relative to COM.

3.3 Comparison with Theory

These numerical results confirm theoretical expectations:

- *Mean-Motion Resonances:* The 2:1 gap at $r_2 \approx 0.63$ exactly match the analytic resonance radii $(\frac{1}{2})^{2/3}$ in nondimensional units, representing the most significant Kirkwood gap.
- *Accretion vs. Ejection:* Particles with initial $r < 0.5$ (not in our initialization) would be quickly accreted by the inner star; those with $0.9 < r < 1.5$ cross Jupiter’s path and are mostly removed. Only dust on sufficiently large orbits ($r > 1.5$) persists, forming the final outer ring.
- *Time Evolution:* The animation shows a smooth progression from a near-uniform ring to a strongly depleted, resonant-gap-dominated distribution, illustrating how Jupiter-like bodies sculpt debris disks in binaries.



(a) Histogram of dust surviving vs. distance from M_2 . (b) Histogram of dust surviving vs. distance from COM.

Figure 2: Final radial distributions (Step 3999). *Left:* M_2 -centric distance $r_2 \in [0, 3.0]$. *Right:* COM-centric distance $r_{\text{COM}} \in [0, 3.0]$. Note clear dips at resonance radii ≈ 0.63 .

In summary, the combined RK4 integration of dust in the inertial frame, plus the two complementary density histograms (M_2 -centric and COM-centric), yields a clear visualization of how a massive secondary clears dust via mean-motion resonances. These results align with both theoretical resonance locations and our earlier presentation.

4 Conclusion

In this project we have developed and compared two complementary numerical approaches to model the dynamical clearing of dust particles by a Sun–Jupiter analogue system:

1. **Inertial-Frame RK4 Simulation.** We integrated the Newtonian equations of motion for 5000 dust particles under the time-dependent gravity of two primaries moving on circular orbits. Using a fixed-step, fourth-order Runge–Kutta scheme with $\Delta t = 0.02$ over 4000 steps, we achieved relative energy conservation at the 10^{-2} – 10^{-3} level for surviving particles and reproduced narrow depletion zones at the 2:1 and 3:2 resonance radii.
2. **Rotating-Frame RK4 Simulation.** We reformulated the problem in a co-rotating frame—where the two primaries of masses 1 and 3 remain fixed at $(-\frac{3}{4}, 0)$ and $(0.25, 0)$ —and added centrifugal and Coriolis accelerations. Again applying RK4 with identical step size and removal criteria, we obtained *identical* final radial survival histograms (dips

at $r \approx 0.63$ and $r \approx 0.87$). The rotating-frame code offered the additional benefit of simpler animation (both stars fixed) and more direct identification of resonance gaps.

Key findings:

- *Resonance-Driven Gaps.* Both methods exhibit deep, narrow depletion zones of width $\Delta r \sim 0.01$ at the analytically predicted mean-motion resonance radii

$$r_{2:1} = \left(\frac{1}{2}\right)^{2/3} \approx 0.63$$

These are the numerical analogues of the Solar System’s Kirkwood gaps (e.g. 3.28 AU for Jupiter at 5.2 AU).

- *Frame-Invariance.* The agreement between inertial-frame and rotating-frame results confirms that the macroscopic clearing pattern is a true dynamical phenomenon, independent of coordinate choice or the presence of non-inertial force terms.
- *Integrator Performance.* The classical RK4 integrator strikes an effective balance between accuracy and computational cost: $\Delta t = 0.02$ resolves close encounters well enough to keep energy/Jacobi constant drifts below a few percent over 4000 steps, while allowing the simulation of thousands of particles in a few minutes.
- *Dust Distribution Features.* Besides the resonance gaps, we observe (i) a broad depletion zone for $0.9r_{1.5}$ caused by repeated close encounters leading to accretion or ejection, and (ii) an outer ring of survivors at $r \approx 1.8\text{--}2.0$ corresponding to orbits that never cross either primary’s sphere of influence.

Limitations and Future Work:

- *Two-Dimensional Approximation.* Extending to 3D would capture inclination-driven resonances and vertical structure in debris disks.
- *Non-Gravitational Forces.* Including radiation pressure and gas drag could model protoplanetary disk evolution more realistically.
- *Adaptive or Symplectic Integrators.* To improve long-term invariants, future studies could employ adaptive-step Runge–Kutta or higher-order symplectic schemes, achieving energy/Jacobi-constant errors below 10^{-6} .
- *Parameter Survey.* Varying the mass ratio M_2/M_1 , separation D , or initial dust distribution would map out the full range of resonance-clearing efficiency in binary and multi-star systems.

In summary, our dual-frame RK4 simulations robustly reproduce the expected resonance-driven clearing of dust by a Jupiter-mass companion, validate the frame-invariance of the effect, and provide a solid foundation for more sophisticated future explorations of debris disk dynamics in binary systems.

A Appendix: Code Listings and Usage

Below we present the two Python modules used in this project (`BinarySystem` for the inertial frame, and `RotatingBinarySystem` for the rotating frame), together with brief instructions on how to run them in Jupyter Notebook.

A.1 Inertial-Frame RK4 Code

Listing 1: `inertial_system.py`: Inertial-Frame RK4 Simulation

```
#Animation and Plot for Inertial frame simulation
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib.gridspec import GridSpec
from IPython.display import HTML
from tqdm import tqdm
import matplotlib as mpl
mpl.rcParams['animation.embed_limit'] = 500 # Increase to 500MB
# Constants
G = 1.0
M1 = 1.0
M2 = 3.0
DISTANCE = 1.0
DT = 0.02
STEPS = 4000
PARTICLES = 5000
BINS = 300
MAX_RADIUS = 3.0
MAX_PARTICLES = 75

class BinarySystem:
    def __init__(self):
        # Initialize stars
        self.com_pos = np.array([0.0, 0.0])
        self.star1_pos = np.array([-DISTANCE*M2/(M1+M2), 0.0])
        self.star2_pos = np.array([DISTANCE*M1/(M1+M2), 0.0])

        # Initialize velocities
        omega = np.sqrt(G*(M1+M2)/DISTANCE**3)
        self.star1_vel = np.array([0, -omega*self.star1_pos[0]])
        self.star2_vel = np.array([0, -omega*self.star2_pos[0]])

        # Initialize dust
        self.dust_pos, self.dust_vel = self._init_dust()

        # History storage
        self.star1_history = np.zeros((STEPS, 2))
```

```

self.star2_history = np.zeros((STEPS, 2))
self.dust_history = np.zeros((STEPS, PARTICLES, 2))

# Density plot bins
self.com_bins = np.linspace(0, MAX_RADIUS, BINS)
self.m2_bins = np.linspace(0, MAX_RADIUS/2, BINS) # Smaller
range for M2 view
self.com_density = np.zeros((STEPS, BINS-1))
self.m2_density = np.zeros((STEPS, BINS-1))

def _init_dust(self):
    # Initialize dust particles in a cloud around the system
    # Random positions in a 2D cloud
    r = np.random.uniform(0.5, 2.0, PARTICLES)
    theta = np.random.uniform(0, 2*np.pi, PARTICLES)
    x = r * np.cos(theta)
    y = r * np.sin(theta)

    # Give particles initial velocity
    v_kep = np.sqrt(G*(M1+M2)/np.sqrt(x**2 + y**2))
    vx = -v_kep * np.sin(theta) + np.random.normal(0, 0.05,
        PARTICLES)
    vy = v_kep * np.cos(theta) + np.random.normal(0, 0.05,
        PARTICLES)

    return np.column_stack((x, y)), np.column_stack((vx, vy))

def gravity(self, pos):
    r1 = self.star1_pos - pos
    r2 = self.star2_pos - pos
    dist1 = max(np.linalg.norm(r1), 0.05)
    dist2 = max(np.linalg.norm(r2), 0.05)
    return G*(M1*r1/dist1**3 + M2*r2/dist2**3)

def rk4_integrate(self, pos, vel, dt):
    k1v = self.gravity(pos)
    k1r = vel

    k2v = self.gravity(pos + 0.5*dt*k1r)
    k2r = vel + 0.5*dt*k1v

    k3v = self.gravity(pos + 0.5*dt*k2r)
    k3r = vel + 0.5*dt*k2v

    k4v = self.gravity(pos + dt*k3r)
    k4r = vel + dt*k3v

```

```

new_pos = pos + dt/6*(k1r + 2*k2r + 2*k3r + k4r)
new_vel = vel + dt/6*(k1v + 2*k2v + 2*k3v + k4v)

return new_pos, new_vel

def evolve(self):
    for step in tqdm(range(STEPS), desc="Simulating"):
        # Update stars (circular orbits)
        theta = np.sqrt(G*(M1+M2)/DISTANCE**3) * DT
        rot = np.array([[np.cos(theta), -np.sin(theta)],
                        [np.sin(theta), np.cos(theta)]])

        self.star1_pos = self.com_pos + rot @ (self.star1_pos -
            self.com_pos)
        self.star2_pos = self.com_pos + rot @ (self.star2_pos -
            self.com_pos)

        # Update dust
        for i in range(PARTICLES):
            self.dust_pos[i], self.dust_vel[i] = self.
                rk4_integrate(
                    self.dust_pos[i], self.dust_vel[i], DT)

            # Remove accreted particles
            if (np.linalg.norm(self.dust_pos[i] - self.star1_pos
                ) < 0.1 or
                np.linalg.norm(self.dust_pos[i] - self.star2_pos
                ) < 0.1):
                self.dust_pos[i] = np.array([np.nan, np.nan])

        # Store history
        self.star1_history[step] = self.star1_pos
        self.star2_history[step] = self.star2_pos
        self.dust_history[step] = self.dust_pos

        # Calculate both density profiles
        valid_dust = ~np.isnan(self.dust_pos[:,0])

        # COM-centric
        com_distances = np.linalg.norm(
            self.dust_pos[valid_dust] - self.com_pos,
            axis=1
        )
        self.com_density[step], _ = np.histogram(com_distances,
            bins=self.com_bins)

        # M2-centric

```

```

        m2_distances = np.linalg.norm(
            self.dust_pos[valid_dust] - self.star2_pos,
            axis=1
        )
        self.m2_density[step], _ = np.histogram(m2_distances,
            bins=self.m2_bins)

def create_triple_animation(self):
    fig = plt.figure(figsize=(18, 6))
    gs = GridSpec(1, 3, width_ratios=[1, 1, 1])

    # Animation panel
    ax1 = fig.add_subplot(gs[0])
    ax1.set_xlim(-MAX_RADIUS, MAX_RADIUS)
    ax1.set_ylim(-MAX_RADIUS, MAX_RADIUS)
    ax1.set_aspect('equal')
    ax1.set_facecolor('black')
    ax1.set_title("System View", color='white')

    # M2-centric density
    ax2 = fig.add_subplot(gs[1])
    ax2.set_xlim(0, MAX_RADIUS)
    ax2.set_ylim(0, MAX_PARTICLES)
    ax2.set_xlabel('Radius from M2')
    ax2.set_ylabel('Particle Count')
    ax2.set_title("M2-Centric View", color='white')
    ax2.grid(True, alpha=0.3)

    # COM-centric density
    ax3 = fig.add_subplot(gs[2])
    ax3.set_xlim(0, MAX_RADIUS)
    ax3.set_ylim(0, MAX_PARTICLES)
    ax3.set_xlabel('Radius from COM')
    ax3.set_title("COM-Centric View", color='white')
    ax3.grid(True, alpha=0.3)

    # Initialize plots
    star1 = ax1.scatter([], [], s=120, c='gold', label=f'M1 ({M1}M)')
    star2 = ax1.scatter([], [], s=180, c='orangered', label=f'M2 ({M2}M)')
    dust = ax1.scatter([], [], s=1, c='cyan', alpha=0.6)
    ax1.legend(loc='upper right', facecolor='black', labelcolor='white')

    # Density plot bars
    m2_bin_centers = (self.m2_bins[1:] + self.m2_bins[:-1])/2

```

```

m2Bars = ax2.bar(m2_bin_centers, np.zeros(BINS-1),
                 width=self.m2_bins[1]-self.m2_bins[0],
                 color='red', alpha=0.7)

com_bin_centers = (self.com_bins[1:] + self.com_bins[:-1])/2
comBars = ax3.bar(com_bin_centers, np.zeros(BINS-1),
                 width=self.com_bins[1]-self.com_bins[0],
                 color='blue', alpha=0.7)

title = fig.suptitle('Frame_0/{0}'.format(STEPS), color='
white')

def update(frame):
    # Update animation
    star1.set_offsets(self.star1_history[frame])
    star2.set_offsets(self.star2_history[frame])
    valid_dust = ~np.isnan(self.dust_history[frame,:,0])
    dust.set_offsets(self.dust_history[frame][valid_dust])

    # Update M2-centric plot
    for rect, h in zip(m2Bars, self.m2_density[frame]):
        rect.set_height(h)

    # Update COM-centric plot
    for rect, h in zip(comBars, self.com_density[frame]):
        rect.set_height(h)

    title.set_text('Frame_{0}/{0}'.format(frame, STEPS))
    return star1, star2, dust, *m2Bars, *comBars

ani = FuncAnimation(fig, update, frames=STEPS, interval=50,
                    blit=False)
plt.close()
return HTML(ani.to_jshtml())

```

Run simulation and display

How to use:

1. Copy above code to a jupyter notebook cell.
2. In a new cell:

```

system = BinarySystem()
system.evolve()
animation = system.create_triple_animation()
animation

```

A.2 Rotating-Frame RK4 Code

Listing 2: rotating_system.py: Rotating-Frame RK4 Simulation

```
#Animation and Plot for Rotational Frame Simulation (Much Faster!)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib.gridspec import GridSpec
from IPython.display import HTML
from tqdm import tqdm
import matplotlib as mpl
import pandas as pd
mpl.rcParams['animation.embed_limit'] = 500 # Increase to 500MB

# Constants
G = 1.0
M1 = 1.0
M2 = 3.0
DISTANCE = 1.0
DT = 0.02
STEPS = 4000
PARTICLES = 5000
BINS = 300
MAX_RADIUS = 3.0
MAX_PARTICLES = 75

class RotatingBinarySystem:
    def __init__(self):
        # Compute reduced mass fraction and angular velocity
        self.M_total = M1 + M2
        self.mu = M2 / self.M_total
        self.omega = np.sqrt(G * self.M_total / DISTANCE**3)

        # In rotating frame, stars are fixed at:
        self.com_pos = np.array([0.0, 0.0])
        self.star1_pos = np.array([-self.mu * DISTANCE, 0.0])
        self.star2_pos = np.array([(1 - self.mu) * DISTANCE, 0.0])

        # Initialize dust positions and velocities in rotating frame
        self.dust_pos, self.dust_vel = self._init_dust()

        # History storage
        self.star1_history = np.zeros((STEPS, 2))
        self.star2_history = np.zeros((STEPS, 2))
        self.dust_history = np.zeros((STEPS, PARTICLES, 2))

        # Density bins and storage
```

```

self.com_bins = np.linspace(0, MAX_RADIUS, BINS)
self.m2_bins = np.linspace(0, MAX_RADIUS/2, BINS)
self.com_density = np.zeros((STEPS, BINS-1))
self.m2_density = np.zeros((STEPS, BINS-1))

def _init_dust(self):
    # Random positions in a 2D annulus [0.5, 2.0] around COM
    r = np.random.uniform(0.5, 2.0, PARTICLES)
    theta = np.random.uniform(0, 2*np.pi, PARTICLES)
    x = r * np.cos(theta)
    y = r * np.sin(theta)

    # Initial Keplerian velocity around COM, then transform to rotating frame
    v_kep = np.sqrt(G * self.M_total / np.sqrt(x**2 + y**2))
    vx = -v_kep * np.sin(theta)
    vy = v_kep * np.cos(theta)

    # Subtract rigid rotation component: v_rot = omega * r
    v_rot_x = -self.omega * y
    v_rot_y = self.omega * x

    vx_rot = vx - v_rot_x
    vy_rot = vy - v_rot_y

    # Add small random perturbation
    vx_rot += np.random.normal(0, 0.05, PARTICLES)
    vy_rot += np.random.normal(0, 0.05, PARTICLES)

    return np.column_stack((x, y)), np.column_stack((vx_rot, vy_rot))

def accelerations_rot(self, pos, vel):
    """
    Compute acceleration in rotating frame:
    - Gravitational from fixed stars at star1_pos, star2_pos
    - Centrifugal: +omega^2 * (pos)
    - Coriolis: 2*omega * (vel rotated by +90 )
    """
    r1_vec = pos - self.star1_pos
    r2_vec = pos - self.star2_pos
    dist1 = np.linalg.norm(r1_vec, axis=1).clip(min=0.05)
    dist2 = np.linalg.norm(r2_vec, axis=1).clip(min=0.05)

    # Gravitational acceleration
    a1 = -G * M1 * (r1_vec / dist1[:,None]**3)
    a2 = -G * M2 * (r2_vec / dist2[:,None]**3)

```

```

a_grav = a1 + a2

# Centrifugal acceleration
a_centrifugal = self.omega**2 * pos

# Coriolis acceleration: (2*omega*vy, -2*omega*vx)
a_coriolis = np.empty_like(pos)
a_coriolis[:,0] = 2.0 * self.omega * vel[:,1]
a_coriolis[:,1] = -2.0 * self.omega * vel[:,0]

return a_grav + a_centrifugal + a_coriolis

def rk4_integrate_rot(self, pos, vel, dt):
    """
    RK4 step for rotating-frame equations of motion:
        dr/dt = v
        dv/dt = a_rot(pos, v)
    """
    k1v = self.accelerations_rot(pos, vel)
    k1r = vel

    k2r = vel + 0.5 * dt * k1v
    k2v = self.accelerations_rot(pos + 0.5*dt * k1r, k2r)

    k3r = vel + 0.5 * dt * k2v
    k3v = self.accelerations_rot(pos + 0.5*dt * k2r, k3r)

    k4r = vel + dt * k3v
    k4v = self.accelerations_rot(pos + dt * k3r, k4r)

    new_pos = pos + (dt/6.0) * (k1r + 2*k2r + 2*k3r + k4r)
    new_vel = vel + (dt/6.0) * (k1v + 2*k2v + 2*k3v + k4v)
    return new_pos, new_vel

def evolve(self):
    for step in tqdm(range(STEPS), desc="Simulating (Rotating Frame)"):
        # Stars remain fixed in rotating frame
        self.star1_history[step] = self.star1_pos
        self.star2_history[step] = self.star2_pos

        # Update dust with RK4 in rotating frame
        self.dust_pos, self.dust_vel = self.rk4_integrate_rot(
            self.dust_pos, self.dust_vel, DT
        )

```

```

        # Remove accreted particles (close enough to either star
        )
        d1 = np.linalg.norm(self.dust_pos - self.star1_pos, axis
            =1)
        d2 = np.linalg.norm(self.dust_pos - self.star2_pos, axis
            =1)
        accreted = (d1 < 0.1) | (d2 < 0.1)
        self.dust_pos[accreted] = np.array([np.nan, np.nan])

        # Store dust positions
        self.dust_history[step] = self.dust_pos

        # Compute densities
        valid = ~np.isnan(self.dust_pos[:,0])
        # COM-centric
        com_dist = np.linalg.norm(self.dust_pos[valid] - self.
            com_pos, axis=1)
        self.com_density[step], _ = np.histogram(com_dist, bins=
            self.com_bins)
        # M2-centric
        m2_dist = np.linalg.norm(self.dust_pos[valid] - self.
            star2_pos, axis=1)
        self.m2_density[step], _ = np.histogram(m2_dist, bins=
            self.m2_bins)

def create_triple_animation(self):
    fig = plt.figure(figsize=(18, 6))
    gs = GridSpec(1, 3, width_ratios=[1, 1, 1])

    # System view (rotating frame)
    ax1 = fig.add_subplot(gs[0])
    ax1.set_xlim(-MAX_RADIUS, MAX_RADIUS)
    ax1.set_ylim(-MAX_RADIUS, MAX_RADIUS)
    ax1.set_aspect('equal')
    ax1.set_facecolor('black')
    ax1.set_title("Rotating□Frame□View", color='white')

    # M2-centric density
    ax2 = fig.add_subplot(gs[1])
    ax2.set_xlim(0, MAX_RADIUS)
    ax2.set_ylim(0, MAX_PARTICLES)
    ax2.set_xlabel('Radius□from□M2')
    ax2.set_ylabel('Particle□Count')
    ax2.set_title("M2-Centric□Density", color='white')
    ax2.grid(True, alpha=0.3)

    # COM-centric density

```

```

ax3 = fig.add_subplot(gs[2])
ax3.set_xlim(0, MAX_RADIUS)
ax3.set_ylim(0, MAX_PARTICLES)
ax3.set_xlabel('Radius from COM')
ax3.set_title("COM-Centric Density", color='white')
ax3.grid(True, alpha=0.3)

# Initialize scatter/hist plots
star1_scatter = ax1.scatter([], [], s=120, c='gold', label=f'
M1_{M1}M')
star2_scatter = ax1.scatter([], [], s=180, c='orangered',
label=f'M2_{M2}M')
dust_scatter = ax1.scatter([], [], s=1, c='cyan', alpha=0.6)
ax1.legend(loc='upper right', facecolor='black', labelcolor=
'white')

# Prepare histogram bars
m2_centers = (self.m2_bins[:-1] + self.m2_bins[1:]) / 2
m2Bars = ax2.bar(m2_centers, np.zeros(BINS-1),
width=self.m2_bins[1]-self.m2_bins[0],
color='red', alpha=0.7)

com_centers = (self.com_bins[:-1] + self.com_bins[1:]) / 2
comBars = ax3.bar(com_centers, np.zeros(BINS-1),
width=self.com_bins[1]-self.com_bins[0],
color='blue', alpha=0.7)

title = fig.suptitle('Frame 0_{}/{}'.format(STEPS), color='
white')

def update(frame):
    # Update star positions (constant)
    star1_scatter.set_offsets(self.star1_history[frame])
    star2_scatter.set_offsets(self.star2_history[frame])

    # Update dust scatter
    valid = ~np.isnan(self.dust_history[frame,:,0])
    dust_scatter.set_offsets(self.dust_history[frame][valid
])

    # Update M2-centric histogram
    for rect, h in zip(m2Bars, self.m2_density[frame]):
        rect.set_height(h)

    # Update COM-centric histogram
    for rect, h in zip(comBars, self.com_density[frame]):
        rect.set_height(h)

```

```

        title.set_text('Frame_{}/_{}'.format(frame, STEPS))
        return (star1_scatter, star2_scatter, dust_scatter, *
                m2_bars, *com_bars)

ani = FuncAnimation(fig, update, frames=STEPS, interval=50,
                    blit=False)
plt.close()
return HTML(ani.to_jshtml())
def save_final_distributions(self):

    """Save final particle positions in both COM and M2 frames
    as CSV"""
    # Get valid particles (non-accreted)
    valid_mask = ~np.isnan(self.dust_pos[:, 0])
    final_pos = self.dust_pos[valid_mask]

    # COM frame coordinates
    com_coords = final_pos - self.com_pos
    com_distances = np.linalg.norm(com_coords, axis=1)

    # M2 frame coordinates
    m2_coords = final_pos - self.star2_pos
    m2_distances = np.linalg.norm(m2_coords, axis=1)

    # Create DataFrames
    com_df = pd.DataFrame({
        'x': com_coords[:, 0],
        'y': com_coords[:, 1],
        'distance': com_distances,
        'frame': 'COM'
    })

    m2_df = pd.DataFrame({
        'x': m2_coords[:, 0],
        'y': m2_coords[:, 1],
        'distance': m2_distances,
        'frame': 'M2'
    })

    # Save to CSV
    com_df.to_csv('final_particles_com_frame.csv', index=False)
    m2_df.to_csv('final_particles_m2_frame.csv', index=False)
    print("Saved final particle distributions to CSV files")

    # Return the DataFrames for inspection
    return com_df, m2_df

```

```

def plot_final_distributions(self):
    """Plot final radial distributions for both frames"""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

    # Get valid particles
    valid_mask = ~np.isnan(self.dust_pos[:, 0])
    final_pos = self.dust_pos[valid_mask]

    # COM frame distribution
    com_distances = np.linalg.norm(final_pos - self.com_pos,
                                    axis=1)
    ax1.hist(com_distances, bins=self.com_bins, color='blue',
             alpha=0.7)
    ax1.set_xlabel('Distance from COM')
    ax1.set_ylabel('Particle Count')
    ax1.set_title('COM Frame Distribution')
    ax1.grid(True, alpha=0.3)

    # M2 frame distribution
    m2_distances = np.linalg.norm(final_pos - self.star2_pos,
                                   axis=1)
    ax2.hist(m2_distances, bins=self.m2_bins, color='red', alpha
            =0.7)
    ax2.set_xlabel('Distance from M2')
    ax2.set_title('M2 Frame Distribution')
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    return fig

```

How to use:

1. Copy above code to a jupyter notebook cell.
2. In a new cell:

```

# Run the rotating-frame simulation and display animation
system_rot = RotatingBinarySystem()
system_rot.evolve()

# Generate and save outputs
final_com_df, final_m2_df = system_rot.save_final_distributions()
final_plot = system_rot.plot_final_distributions()
plt.show()

animation_rot = system_rot.create_triple_animation()

```

animation_rot

A.3 Changing Initial Conditions

The dust-particle initial distribution is set in the `_init_dust()` method of each class. To explore different scenarios, you can adjust the following parameters:

- **Number of particles:** At the top of your script, change the constant

```
PARTICLES = 5000
```

to any desired integer.

- **Radial range:** In `_init_dust()`, modify the call

```
r = np.random.uniform(0.5, 2.0, PARTICLES)
```

by replacing 0.5 and 2.0 with your chosen minimum and maximum radii.

- **Angular distribution:** Also in `_init_dust()`, change

```
theta = np.random.uniform(0, 2*np.pi, PARTICLES)
```

to limit the dust to any angular sector (e.g. $\text{np.pi}/4$ to $3\text{np.pi}/4$).

- **Velocity perturbation:** The random noise level is set by

```
vx += np.random.normal(0, 0.05, PARTICLES)
vy += np.random.normal(0, 0.05, PARTICLES)
```

Change the 0.05 standard deviation to a larger or smaller value to adjust the initial velocity spread.

After making these edits, simply rerun the `evolve()` method (and the subsequent animation or histogram routines) to see how the new initial conditions affect the dust-clearing behavior.

A.4 Post-Processing: Binning and Plotting Script

We also have the following python script to generate the final radial distribution histogram from a CSV of particle positions, the following Python script can be used. It reads in a file (with either a 'distance' column or separate 'x','y' columns), bins the distances into 300 intervals over $[0, 3.0]$, writes out the binned counts, and produces a bar-plot.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

```

# Read the original CSV file (must contain 'distance' or 'x','y'
  columns)
data = pd.read_csv('particles_com_frame.csv')

# Determine which column holds the radial distance
if 'distance' in data.columns:
    distances = data['distance']
elif 'x' in data.columns and 'y' in data.columns:
    distances = np.sqrt(data['x']**2 + data['y']**2)
else:
    raise ValueError("CSV must contain either a 'distance' column or
      'x','y' columns")

# Create 300 bins from 0.0 to 3.0
bins = np.linspace(0, 3.0, 301)
centers = (bins[:-1] + bins[1:]) / 2

# Histogram the data
counts, _ = np.histogram(distances, bins=bins)

# Save binned data to new CSV
binned_df = pd.DataFrame({
    'Distance': centers,
    'Particle_Count': counts
})
binned_df.to_csv('binned_particle_com_distribution.csv', index=False
)

# Plot the distribution
plt.figure(figsize=(8,5))
plt.bar(centers, counts, width=bins[1]-bins[0], alpha=0.7, edgecolor
  ='k')
plt.xlabel('Distance from COM')
plt.ylabel('Number of Particles')
plt.title(f'Radial Distribution (N={len(distances)})')
plt.grid(alpha=0.3, linestyle='--')
plt.tight_layout()
plt.savefig('particle_distribution_plot.png', dpi=300)
plt.show()

```

After saving the Python script as `bin_and_plot.py` in the same directory as your data file `particles_com_frame.csv`, run the following in your terminal:

```
python bin_and_plot.py
```

The script will output the following files:

- `binned_particle_com_distribution.csv`

- `particle_distribution_plot.png`

References

- [1] Astrophysics Formulas. (n.d.). *Keplerian orbital velocity*. Retrieved June 6, 2025, from <https://astrophysicsformulas.com/astrometry-formulas-astrophysics-formulas/keplerian-orbital-velocity/>
- [2] LibreTexts. (2021). “The Runge–Kutta method.” *Mathematics LibreTexts*. Retrieved June 6, 2025, from https://math.libretexts.org/Courses/Monroe_Community_College/MTH_225_Differential_Equations/03%3A_Numerical_Methods/3.03%3A_The_Runge-Kutta_Method
- [3] Kluyver, T., Ragan-Kelley, B., Pérez, F., ... & Jupyter Development Team. (2016). *Jupyter Notebooks—A publishing format for reproducible computational workflows*. In F. Loizides & B. Schmidt (Eds.), *Positioning and Power in Academic Publishing* (pp. 87–90). IOS Press. <https://doi.org/10.3233/978-1-61499-649-1-87>