N-Body Gravity Simulation Documentation

Overview: 2D N-body gravitational simulation built in Python. This models celestial bodies and Newtonian gravity, visualizing their movements via Pygame.

Components: Body, Engine, Renderer, and Loader.

- Body: represents the physical object
- Engine: computes physics, updates motion
- Renderer: visualization
- Loader: loads initial conditions for the object's data via JSON

Rohtak Patwardhan

Body Class: represents a single celestial object with physical properties

- Name, mass, pos (position), vel (velocity), and acc (acceleration)
- Vectorized calculations were used
- Position, velocity, and acceleration are stored as NumPy arrays for vector operations

```python
import numpy as np



class Body:
   def __init__(self, name, mass, pos, vel):
       self.name = name
       self.mass = mass
       self.pos = np.array(pos, dtype=float)  # Position vector [x, y]
       self.vel = np.array(vel, dtype=float)  # Velocity vector [vx, vy]
       self.acc = np.zeros(2, dtype=float)    # Acceleration vector
```

Engine Class: handles all the physical calculations and updates the system state

- G: Gravitational constant
- Methods
    - Compute_forces()
        - Resets acceleration for all bodies
        - Computes pairwise gravitational interactions
        - Uses Newton's law of gravitation
        - Updates acceleration in real-time
    - Euler_step(dt)
        - Updates the velocity and position in time increments
        - **Limitations:** Euler integration is not very accurate for long-term simulations, does not assume conservation of energy

Renderer Class: handles graphics using PyGame

- Opens a window and manages frame rate, draws bodies as circles, radius scales with mass
- Methods
    - Draw(bodies)
        - Clear screen, draw each body at the given location, update the display, cap frame rate at 60 FPS (configurable dependent on display refresh rate)

```
class Renderer:
   def __init__(self, width=800, height=600):
       pygame.init()
       self.screen = pygame.display.set_mode((width, height))
       self.clock = pygame.time.Clock()
       self.width = width
       self.height = height

   def draw(self, bodies):
       self.screen.fill((0, 0, 0))  # Black background
       for body in bodies:
           x, y = int(body.pos[0]), int(body.pos[1])
           pygame.draw.circle(self.screen, (255, 255, 255), (x, y), max(2,
int(body.mass**0.33)))
       pygame.display.flip()
       self.clock.tick(60)  # 60 FPS
```

Loader: loads initial simulation data from a JSON

- load_bodies_from_json(file_path)
    - Reads JSON config
    - Instantiate body objects
    - Return a list of bodies

```
[
   {"name": "Sun", "mass": 1000, "pos": [400, 300], "vel": [0, 0]},
   {"name": "Planet1", "mass": 10, "pos": [500, 300], "vel": [0, 50]},
   {"name": "Planet2", "mass": 20, "pos": [300, 300], "vel": [0, -50]}
]
```

Design Notes

Strengths: easy to extend and iterate, modular design, and uses NumPy
Potential Improvements:

- Use Verlet or Runge-Kutta
- Barnes-Hut algorithm implementation for improved complexity
    - Partition space into a 2D quadtree
    - Nodes would be a region of space
    - Build a quadtree from all bodies, compute CoM for each node, and for each body: traverse the tree; if a region is far, approximate; else, recurse into children
    -
- Use CUDA
- Improve UI/UX with more camera controls and smoothness
- Add softening factor, **ε**

Future Extension (Summer 2026)

1. Adding GPU Acceleration
2. Improving UI and user capacity to add/remove bodies and create custom bodies
3. Ideally, 3D simulation support?