Ideal Gas Particle based Simulation

In this project, we aim to simulate an ideal gas with a particle based approach. The "Ideal Gas Particle Based Simulation" is a Python program that simulates the behavior of a collection of ideal gas particles in a 3D space. The simulation utilizes a particle-based approach, where each particle represents a gas molecule. The particles move within a defined space, experiencing collisions with each other and the walls, following the principles of ideal gas behavior.

Key Components

Ball Class

The simulation revolves around the Ball class, representing a gas particle. Each instance of this class maintains the position, velocity, and acceleration of the particle. Key methods include:

- `__init__(self, position, velocity, acceleration)`: Initializes a particle with given initial conditions. check_wall_collision(self, position, velocity): Checks and handles collisions with the walls of the simulation space.
- `update(self)`: Updates the position, velocity, and handles collisions for each time step.

Simulation Logic

The simulation involves the following crucial components:

- Collision Logic: The collide function handles collisions between pairs of particles, applying the principles of conservation of linear momentum and kinetic energy. The simulation assumes an elastic collision model.
- Particle Update: The updateParameters function updates the position, velocity, and acceleration of each particle based on the defined time step (delta_t).
- Drawing Frames: The simulation generates frames for visualization, with options for both text and PNG formats. Visualization includes the 3D positions of particles and histograms of their velocities.

Visualization

The simulation produces a series of frames, each representing a snapshot of the particle system at a specific time. Visualization options include text-based animations and 3D scatter plots with velocity histograms.

Project Overview

The simulation first begins with creating a set of particles that are stored in a list. Then, the simulation loop begins.

The following steps make up the simulation loop:

1. Check for collisiosn with other particles, and update velocities accordingly.
2. Update the positions of the particles based on their velocities.
3. Draw the positions of the particles into a scatterplot. This visualization makes up one frame of the simulation.

A pre-specified number of frames are generated following the above simulation loop, after which all the frames are stitched together into a single mp4 video.

The following variables are available to adjust in the beginning of the simulation:

- `number_of_frames`: The number of frames of the simulation.
- `dimensions`: The dimensions of the box in which the particles are simulated.
- `delta_t`: The change in time between the frames of the simulation
- `number_of_balls`: The number of particles being simulated.

The remainder of this report will describe how the above steps are implemented in the code.

Video Generation

The backbone of the project structure is the video generation engine. For this, a function called `generate_animation` was written. A folder with all the frames of the simulation is kept, where each frame is stored with it's numerical index. The function looks at all the files in this folder, and uses the OpenCV library to generate an MP4 Video with the images in order.

> ***Future Work:*** An alternative pipeline is in development, which involves saving the frames as a text file with the data on all the particle positions,

with a function to plot and generate the video together.

With this function ready, the code can be simplified into the following outline:

```python
#Importing the Nessecary Libraries

#Defining the simulation constants
number_of_balls = 200
dimensions = np.asarray([500,500, 500])
number_of_frames = 500
delta_t = 10 # Update with the actual time step

#Particle Class Defintion
class Ball:

    def __init__(self, position, velocity, acceleration):
        '''Taking Position, velocity, acceleration as inputs


    def check_wall_collision(self, position, velocity):
        '''Check if Collisions occurs with the walls and impl



    def update(self):
        """
        Update the position, velocity, and handle collisions.
        """


################################Generating the Ball Objects #

balls = generate_list_of_balls(number_of_balls)

################################ The Actual Simulation ######


#Simulating and generating each individual plot

zfilll = len(str(number_of_balls)) + 2


for frame_count in range(number_of_frames):

    draw_frame(frame_count)

    print(f"Frame number {frame_count + 1} has been generatec


print("Generating Video from the frames...")
generate_animation.generate_animation(folder_path, fps = 25)
```

The Particle Class

For a simpler coding experience, we are using Python's object oriented capabilities, and defining a custom class which we call as Ball. This class will have the following methods:

1. Initialize the object.
2. Check if a Ball collides with a wall
3. Update the position of the ball.

Initialization Function:

The ball object is initialized with the `__init__` method, which takes in the following arguments:

1. Initial Position of the Ball.
2. Initial velocity of the ball.
3. Initial Acceleration on the Ball.

These arguments are stored as class variables, and are updated each frame.

Collision with Walls:

We check if any of the particles are exceeding the bounds of the simulation, which was defined with the variable `dimensions`. If so, we implement elastic collisions and change the velocity accordingly.

If the particle exceeds the bounds of the simulation in the x direction, then we change the x component of the velocity to be negative of itself. Similar checks are implemented for the y and z directions as well.

Update Function:

Recall that the motion of a particle can be described by the following diffrential equations:

$$\frac{dv}{dt} = a$$

$$\frac{ds}{dt} = v$$

In this function, we update the position and velocity of the ball to the new values. We first check for collisions with the walls and update the velocity accordingly. Following this, we use Euler's Method of Solving Diffrential Equations to update the position and velocity of the balls.

> ***Future Work:*** For greater accuracy when describing the motion, we need to update to use the Runge Kutta methods instead.

Generating the Balls

We use a loop to generate the set of balls that interact. The number of balls is pre-defined in the function `number_of_balls`. The parameters of Position are generated with `random.randint()` with values between 0 and the max value allowed by the dimension of the box. The Velocity parameter is randomly generated with `random.random() - 0.5`, which allows for values between -0.5 and 0.5 as the initial value of the velocities. The acceleration parameter is set to zero for all it's components in this versoin of the code.

Particle-Particle Collision Logic:

The collide function is designed to handle the collision between two objects. It calculates the distance between the two objects based on their positions and checks if a collision occurs. If a collision occurs (determined by the distance being less than or equal to a predefined constant `COLLISION_DISTANCE`), it updates the velocities of the two objects using an elasticity formula. It takes in the following parameters:

1. Position of the First Object
2. Position of the Second Object
3. Velocity of the First Object
4. Velocity of the Second Object

If a collision occurs, update the velocities of the two objects using the elasticity formula:

$$vel1_{new} = \frac{(e+1)\cdot vel1 + vel2\cdot(1-e)}{2}$$

$$vel2_{new} = \frac{vel1\cdot(1-e) + (1+e)\cdot vel2}{2}$$

The coefficient of restitution e (`ELASTICITY_COEFFICIENT`) determines the elasticity of the collision, affecting how much kinetic energy is retained after the collision.

Within the primary simulation loop, the collision detection process involves examining each particle to determine if it is colliding with any other particle. During this process, the velocities of the involved particles are updated to simulate the effects of the collision. Given the commutative nature of collisions (i.e., the collision between particle 1 and particle 2 is equivalent to particle 2 colliding with particle 1), we optimize the collision-checking procedure.

To achieve this optimization, the inner loop responsible for collision checks iterates only over a subset of particles. Specifically, it iterates from the current particle to the last particle. This approach capitalizes on the fact that collisions between the current particle and the preceding particles have already been assessed. By doing so, we effectively halve the number of collision checks performed, maintaining the accuracy of the simulation while optimizing computational efficiency. This optimization becomes increasingly impactful as the number of particles in the simulation grows.

Code Optimization

In order to optimize and reduce the time required for generating each frame, a number of optimizations have been implemented.

In the previous section, we described how the we're reducing the number of collision checks we perform.

Beyond this, we also isolate all the mathematical calculations to seperate functions, which we implement Just In Time Compilation wth Numba, in order to speed up the computation.

Future Work: The mathematical calcualtions are currently implemented in python and optimized with Just in Time compilation. We can improve the performace by implementing an external library written in C or Fortran.

Features yet to be Implemented

Several enhancements can be considered to augment the functionality and performance of the current simulation:

1. **Particle-Particle Interactions:**

    - The introduction of interactions between particles, such as gravitational or electrostatic forces, stands as a potential feature. Implementing such interactions would impact the acceleration parameter, which is currently set to zero.
2. **Runge-Kutta Integration Method:**
    - A refinement to the simulation accuracy can be achieved by updating the numerical integration method. Transitioning from the currently employed Euler Method to more sophisticated techniques like Runge-Kutta methods offers the potential for increased precision. Moreover, this adjustment would allow the use of larger time intervals for the simulation.
3. **Optimization with External Libraries:**
    - The mathematical calculations, currently implemented in Python and optimized with Just-In-Time Compilation, could benefit from further performance improvements. Exploring the integration of external libraries written in languages like C or Fortran may offer notable speed enhancements.
4. **Alternative Plotting Methods:**
    - Addressing a significant performance bottleneck involves reevaluating the plotting strategy. Instead of using Matplotlib, an alternative approach is to store position values at each time step in a text file and perform plotting at a later time. This method can lead to more efficient and scalable plotting.
5. **Parallelized Collision Detection:**
    - The collision detection algorithm is a potential area for improvement through parallel processing. Although not currently implemented due to skill constraints, exploring parallelization could significantly enhance performance.

These potential enhancements collectively contribute to the adaptability, accuracy, and efficiency of the simulation, providing opportunities for further exploration and development.

Author Information and Contact

- **Name:** Behara Sasi Mitra
- **Email:** beharasasimitra211141@students.iisertirupati.ac.in
- **GitHub:** SasiMitraB