Assignment 1
particles go brr: Parallel Particle Collision Simulator

CS3210 – 2024/25 Semester 1

Learning Outcomes
This assignment aims to familiarize you with parallel programming and performance evaluation on a single node, multi-core system using OpenMP.

Please read this entire document before you start! [and yes, we know it's long, but it's necessary :)]
We will provide a lot of starter code and utilities, so don't worry about the long description.

Contents

1 Problem Description

Note that any amendments to the initial PDF will be in this color.

1.1 Introduction

In this assignment, you will write a program that **simulates the movement of particles in a 2D space**, which is governed by particle velocities, and collisions with their surroundings and other particles. As simulating tens of thousands or even millions of particles over a long time is computationally intensive, you will *parallelize* this simulation to make it fast. Such fast simulations are crucial for large-scale problems such as climate modelling, fluid simulation, and even understanding how aircraft behave in practice.

1.2 Simulation Overview

The goal of your program is to simulate the movement of particles in a 2D space over a number of discrete *timesteps*. The particles move according to their velocities and have *elastic collisions* with other particles and the boundaries of the simulation.

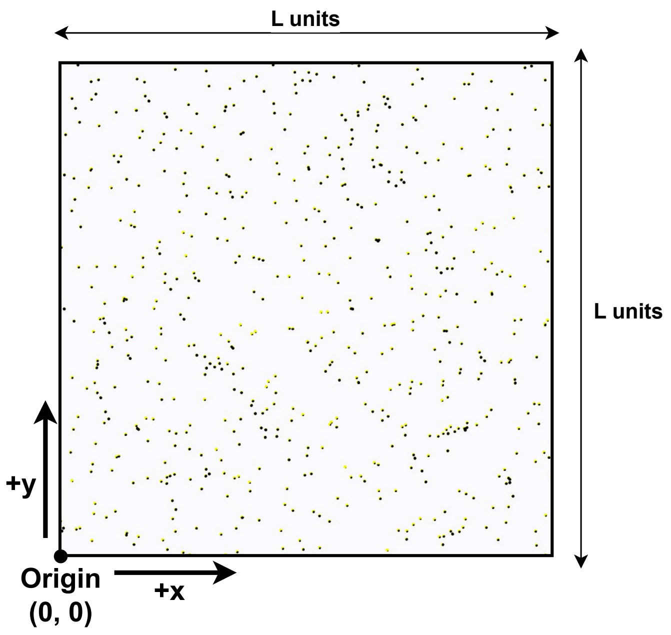


Figure 1: Illustration of particles within the square simulation area at a specific point in time

Figure 1 shows a representation of the 2D area (*simulation area*) that contains a number of *particles*. We now further define the terms and behavior of the simulation. **Note that we have already provided reference implementations of many of the definitions and behaviors we describe below.**

1.3 Simulation Objects

The *simulation area* and list of *particles* define the state of the simulation at some specific time.

Simulation Area

- A square 2D region. All particles in the simulation start with their centers within this region. The boundaries of the 2D region ("walls") are infinitely thick outside the bounds of the simulation area, and have infinite mass.
- The simulation area has a length of L units (an integer (`int`)) on each side.
- Each point in the area is defined by (x, y) coordinates, where $0 \leq x, y \leq L$. The coordinate system starts at the bottom-left of the simulation area, where x increases to the right and y increases upwards.

The simulation area contains a number of individual *particles*.

Particles

- Each particle is defined by 5 properties: (i, x, y, v_x, v_y), where i is the particle's unique index, (x, y) are the particle's center coordinates, and (v_x, v_y) are the particle's velocity components. The index is an integer (`int`), and the coordinates and velocities are all double-precision floating-point (`double`) values.
- Every particle is circular and has a radius r units (an integer (`int`)) that is the same for all particles. Every particle also has the same (unspecified) mass.

1.4 Simulation Behavior: Timesteps

The simulation progresses in discrete *timesteps*, where the particles move and interact with each other and the boundaries of the simulation area.

Simulation Timestep Behavior
This section describes the behavior of the particles in this simulation during each **timestep**. The simulation progresses in discrete timesteps, starting at step 0, and ending at the end of step S (i.e., step S is also simulated). During each timestep, these behaviors occur in sequence:

1. **Phase I: Position Update:** Every particle moves according to its velocity vector. For example, a particle with velocity (v_x, v_y) moves to $(x + v_x, y + v_y)$.
2. **Phase II: Velocity Update – Collision Detection and Resolution:** After all particles have moved in the previous phase, each particle's position is fixed for the remainder of the step. The simulation must now update the *velocity* of each particle, if necessary. The rules of of this phase are as follows:

   (a) **If a particle collides with a wall, it follows the Particle-Wall collision rule.**

   (b) **If two particles collide, they follow the Particle-Particle collision rule.**

   (c) **The simulation should follow the law of conservation of energy – assuming particle p^i has velocity (v_x^i, v_y^i) and energy $E = \sum_{i=0}^{N} (v_x^i)^2 + (v_y^i)^2$, the energy E at the end of this phase should equal the energy at the start of the phase.**

   (d) **At the end of this phase, no particle should be colliding with any other particle or wall**. Note that you may have to *repeatedly* check and apply the rules above multiple times to resolve all collisions. You may assume that the inputs given converge in such a way that this step will eventually complete. If you find a test case that does *not* satisfy this (e.g., number of collisions does not decrease over time within a step), please let us know. We also *do not specify the order* in which you should check and apply collision rules (e.g., we don't specify whether you should check collisions between Particle 1 and 2 before Particle 2 and 3).

1.5 Collision Behavior

The simulation area contains both particles and walls. Collisions can occur between particles and walls, and between any two particles.

Collision
We define a collision as an event between a particle and either another particle or a wall. These are two conditions that are *necessary and sufficient* for a collision to occur:

- **Overlap Rule:** The two objects must be *overlapping*. Two particles are overlapping if the distance between their centers is $\leq 2r$. A particle and wall are overlapping if the particle's center is $\leq r$ units of the simulation boundary.
- **Velocity Rule:** The two objects must be *moving closer together*, i.e., their relative velocity vector points towards the other object.

If both conditions are met, the two objects are considered to be colliding. If only one or the other is met (e.g., particles overlapping but not moving closer together), they are not considered to be colliding.

When a collision is detected, the simulation must *resolve* the collision by updating the velocities of the particles involved. The rules for resolving collisions are as follows:

Particle-Wall Collision Rule
If a particle collides with a wall, the particle's velocity is "reflected" off that particular wall. Enumerating all possibilities:

- If a particle collides with the top/bottom wall, the y component of its velocity is negated.
- If a particle collides with the left/right wall, the x component of its velocity is negated.
- A particle may collide with two walls simultaneously (e.g., a corner). In this case, both components of its velocity are negated.

Particle-Particle Collision Rule
If any two particles collide, their velocities are exchanged via the laws of **elastic collisions**, where the *total energy and momentum* are preserved before and after the collision.

Conservation of Energy: Using the notation in the section above on timestep behavior, if two particles p_i and p_j collide, $E = (v_x^i)^2 + (v_y^i)^2 + (v_x^j)^2 + (v_y^j)^2$ should be same at the start and end of the collision.
Conservation of Momentum: For two particles p_i and p_j with the same mass, the momentum in the x and y directions is conserved. This gives the conditions:

$$v_x^i + v_x^j = v_x'^i + v_x'^j \quad \text{and} \quad v_y^i + v_y^j = v_y'^i + v_y'^j$$

where v_x^i, v_y^i are the initial velocity components of particle p_i, and $v_x'^i, v_y'^i$ are the velocity components after the collision, similarly for p_j.

Reference Implementation
You don't need to write the collision rules and math in code yourself – we include reference implementations of them in `collisions.h`, which you are free to change as necessary as long as the final behavior is correct.

2 Running Your Simulation, Inputs, and Automated Validation

2.1 Running Your Simulation

We assume that calling `make` at the top level of your submission will compile your program into *two* executables named `sim` and `sim.perf` (the difference is explained in Section 2.4.3). Taking `sim` as an example (the arguments will be the same for `sim.perf`), your simulation will then be called as such:

```
./sim <input_file_path> <num_threads>
```

2.2 Input

Your program will be given a path to an input file, which strictly follows the structure shown below:

1. N – Number of particles in the simulation
2. L – Size of the square (in *units*)
3. r – Radius of the particle (in *units*)
4. S – Number of timesteps to run the simulation for
5. For each particle, the following space-delimited information will be present on one line:
   - i – the index of the particle from 0 to $N-1$
   - x – initial position of particle index i on x axis (in *units*)
   - y – initial position of particle index i on y axis (in *units*)
   - v_x – initial velocity on the x axis of particle i (in *units per timestep*)
   - v_y – initial velocity on the y axis of particle i (in *units per timestep*)

Sample Input

```
4
13
1
25
0 3 5 -1 0
1 5.05 5 0 0
2 7.1 5 0 0
3 9.15 5 0 0
```

Note that this input file describes a simulation with 4 particles in a simulation area with side length 13, a radius of 1, and the simulation will run for 25 timesteps. The particles are initially located at $(3, 5)$, $(5.05, 5)$, $(7.1, 5)$, and $(9.15, 5)$, respectively. The first particle has an initial velocity of $(-1, 0)$, while the other particles have an initial velocity of $(0, 0)$.

2.3 Number of Threads

The final argument to your program is the number of OpenMP threads to use when running your program. You *must* call `omp_set_num_threads` to set the number of threads to this value (and not bypass this by changing environment variables or setting a different number of threads later, etc.).

2.4 Validation

Printing an output file containing the positions and velocities of each particle at each timestep would result in (a) far too much time performing I/O, and (b) far too much disk space used. Instead, we provide a **simulation validator** (`sim_validator.h` and `sim_validator.o`) that contains functions that you are **required to call in your code.**

2.4.1 Calling the Validator

Here, we explictly describe how you must call our validator. The sample code (in `sim.cc`) includes some but not all of these requirements already – specifically, step (3) is not done for you yet.

(1) You should first include the header file in your code (we assume the object file `sim_validator.a` will be compiled together with your executable, which is already done for you in the provided Makefile)

```
#include "sim_validator.h"
```

(2) After reading the input file, you should initialize the validator:

```
#if CHECK == 1
    // Initialize collision checker with required arguments
    SimulationValidator validator(params.param_particles,
                                  params.square_size,
                                  params.param_radius);
    // Initialize with starting initial positions
    validator.initialize(particles);
#endif
```

(3) From this point onwards, you must validate each step (from 1 to S) by calling the code below:

```
#if CHECK == 1
    // Called for each step 1 to S inclusive
    validator.validate_step(particles);
#endif
```

If your simulation fails our validation, the validator will print an error message and forcibly exit your program with a failure code (`EXIT_FAILURE`). If your code passes validation, the validator will print a success message, and will not exit your program forcibly at any point.

2.4.2 Why Step-by-Step Validation?

The goal of our validator is to check that the simulation rules and behavior we have specified so far are followed during each step (e.g., positions and velocities are updated correctly, energy and momentum are conserved at the right times, etc). We need to validate the physics of each step individually as the final positions/velocities of your program are not guaranteed to be the same as other students or even our own code, for a variety of reasons (floating point inaccuracies, collision resolution order...)

2.4.3 Validation Executable vs Performance Executable

Your program will be automatically compiled into two executables with our given `Makefile`:

1. `sim` – This is the executable that you will use to validate your program ("validation" mode). It will be compiled with the `-DCHECK=1` flag, which will trigger the `#if CHECK == 1` section in the code above, and therefore include the validator code in your program.
2. `sim.perf` – This is the executable that you will use to test the performance of your program ("performance" mode). It will be compiled with the `-DCHECK=0` flag, which will exclude the validator code entirely from your program. You can therefore be sure that our validator does not affect the performance of your code.

3 Implementation Guidelines

We don't provide you working code from the start as that might limit the approaches you might take, taking into account that there isn't a specific "correct answer". However, this section provides some potentially useful guidelines and advice to help you implement your simulation both correctly and efficiently.

3.1 Implementation Recommendation 1: Make a working version first

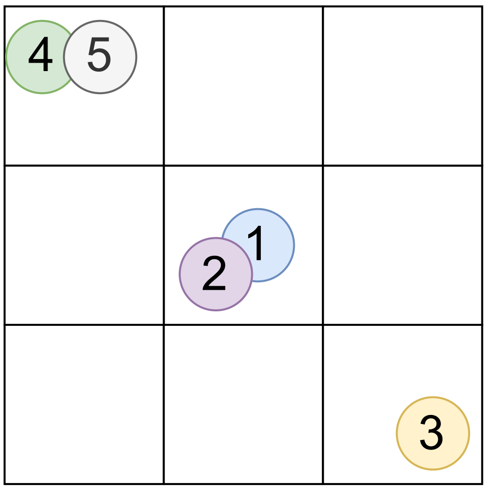
It might be tempting to start by implementing complex algorithms first. **Do not do this.** Start with the simplest possible algorithm that you can verify the correctness of, to make sure that you have interpreted our rules correctly. As a guideline, a simple collision resolution procedure per timestep may involve checking every (particle – wall) collision first, resolving the collisions, then checking every unique (particle – particle) collision pair, resolving them, and then repeating this process until no more collisions are detected at that timestep.

3.2 Implementation Recommendation 2: Spatial Partitioning and Caching

You are likely to find naively checking for *all-pairs collisions* too slow. A common option that **we recommend for this assignment** is to use **spatial-based algorithms**, where you divide the simulation area into smaller regions, and only check for collisions between particles in the same region and/or adjacent regions. This is often also known as a "broadphase" collision detection algorithm. We include a very high-level description of such an algorithm below.

Algorithm 1 Pseudocode: Broadphase Collisions

```
1: Initialize simulation parameters and particles
2: Assign particles to grid cells
3: for each step in simulation do
4:     Move each particle based on velocity
5:     Reassign particles to new grid cells
6:     while unresolved collisions exist do
7:         for each particle $p_1$ in grid do
8:             Check for wall and particle collisions
9:             within own and adjacent grid cells
10:            for each collision that's found do
11:                Resolve collision
12:            end for
13:        end for
14:    end while
15: end for
```



The figure above shows particles assigned to grid cells, and how particles within a grid ((1 – 2) and (4 – 5)) are far more likely to be colliding with each other than particles in other grids. Such a technique may significantly reduce the number of collision checks you need to perform.

Caching Collisions: In addition to implementing spatial techniques, consider that checking for overlaps between particles is a common and expensive operation. Consider caching the results of such operations.

3.3 Implementation Recommendation 3: Profile, then optimize

While this should go without saying, do not optimize your programs blindly. Use any tools at your disposal to *profile* the behavior of your program – what parts take most time, what metrics are higher or lower than expected, and make and test hypotheses as to the performance of your program.

4 Requirements, Grading, and Provided Template Code

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You are not required to have the same teammate for the following assignments. You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalised. If you use external references, cite them or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.). *This also includes use of AI tools such as ChatGPT, Copilot, etc*. Please refer to our policy in the introductory lecture on AI tool usage.

The grades are divided as follows:

- 7 marks – your parallel implementations with OpenMP, split into:
  - 3 marks – the *correctness* of your simulation and implementation, described in Section 4.1.
  - 4 marks – the *performance* of your simulation, described in Section 4.2.
- 5 marks – your report, described in Section 4.3.
- Up to 2 bonus marks, described in Section 4.4

4.1 Correctness Requirements

In this section, we define the correctness requirements for your program. These requirements are in *addition* to any other information presented earlier in this document.

Your implementations should:

- Be written in C or C++, and compile and run successfully with `gcc/g++` **and** `clang/clang++` with flags `-std=c++20 -fopenmp -O3` on all PDC lab machines.
- Create a simulation executable named `sim` that includes the correct calls to our validator, and another executable `sim.perf` that does not call our validator, when you run `make` in the top level of your submission.
- Be parallelized using OpenMP *only* – you are not allowed to use any other parallelization libraries, frameworks, or even C++ threading libraries. You are also not allowed to submit a fully-sequential simulation – your program must show speedup over a single-threaded execution of your program.
- Be tested and run via Slurm – we will use the default CPU mapping and bindings for `srun`.
- Have no memory leaks or undefined behavior. You may check this via the `valgrind` tool, for example, although it may have false positives.
- Not use any external utilities or libraries (e.g., those not provided with the C/C++ environment) that are not otherwise approved by the teaching team. Such approvals will be listed publicly in the FAQ document mentioned later. If what you want is not listed, you may request permission via our contact details in Section 5.2. Note that we are unlikely to allow things that directly impact performance.
- Use the provided `io.h` and `io.cc` files *without any modifications*, as we don't want you to focus on this. You may change other parts of our code skeleton (e.g., `collisions.h`) assuming that all other requirements are fulfilled. Our utility scripts can all be modified freely.
- **Correctly call and pass checks in our sim_validate library**
  - We will check your code on test cases provided to you, and a number of private test cases.
  - We may update the `sim_validate` library if we discover any issues with it w.r.t to our requirements.
- Not have any race conditions, data races, deadlocks, or any other synchronization issues, regardless of what our validator says.

4.2 Performance Requirements

We provide a number of **benchmark executables** to compare your solution against. Each are named in the format `bench_i`, where i is an integer. The i value does not necessarily represent any specific ordering of the benchmarks. You will be assigned performance marks based on your performance against these reference benchmarks. We do not disclose the specific mark allocation per benchmark.

Performance Metric: We will compare the wall-clock runtime of your program against the wall-clock time of each of the benchmarks. For each run, we will run each program with the same input file and the same number of threads. We will run each program a few times (unspecified) and take the average time as the final time.
Requirements: To receive marks for a specific benchmark, you must be faster than it under these conditions:

- The input file will have $N \geq 10,000$ particles.
- Define the *density* of the simulation as $\frac{N \times \pi R^2}{L^2}$ (intuition: maximum % of the simulation area that could be covered by particles at the start). The density of the initial simulation state will be ≥ 0.7.
- The input file will have $S \geq 100$ steps.
- Your program will run with at least 8 threads (on a machine with at least 8 hardware threads). We may choose any machine or machines within the PDC lab for our tests.
- Your program will be run via Slurm.

We provide some reference test cases in the `tests/` directory. You can run the benchmarks with the provided `run_bench.sh` **Note that the provided tests are not exhaustive, and therefore you should test your program with a variety of inputs to ensure correctness and performance.**

We have tested our benchmark implementations but as always they are possibly incorrect. If you notice any issues, please do let us know. Our contact details are available in Section 5.2.

Slurm Usage
We reiterate that you should test and run your programs via Slurm. We will run your programs via Slurm, and if your program does not run correctly via Slurm, you will not receive marks. Furthermore, the CS3210 cluster will be heavily utilized during the assignment period, so Slurm allows a fair allocation of testing resources. **Do not test your program on login nodes!**

"Spirit of the Assignment" Note
We included a number of rules and constraints in this assignment for clarity. However, this is not a legal document – we cannot and do not want to cover every single eventuality in writing, and we also have certain learning objectives for you, so bypassing the spirit of the rules is not what we're aiming for.

At the same time, we want to encourage exploration and interesting solutions. If you believe you have an interpretation of any these rules that is different from the one we might have intended, **please ask us for clarification** (even privately if necessary). For avoidance of doubt, if you have any concern that you are not interpreting the requirements as intended, you must ask us.

4.3 Report Requirements

4.3.1 Format

Your report should follow these specifications:

- Five pages maximum for main content (excluding appendix).
- All text in your report should be minimum 11-point Arial (any typeface and size is ok so long as it's readable and not trying to bypass the page limit).
- All page margins (top, bottom, left, right) should be at least 1 inch (2.54cm).
- Have visually distinct headers for each content item in Section 4.3.2.
- It should be self-contained. If you write part of your report somewhere else and reference that in your submitted "report", we reserve the right to ignore any content outside the submitted document. An exception is referencing a document containing measurement data that you created as part of the assignment - we encourage you to do this.
- If headers, spacing or diagrams cause your report to *slightly* exceed the page limit, that's ok - we prefer well-organised, easily readable reports.

4.3.2 Content

Your report should contain:

- (1 mark) A brief description of your implementation, including:
  - An overview of your chosen algorithm, data structures, and parallelization strategy, why these were chosen.
  - What OpenMP constructs did you use, and why did you use those specific constructs.
  - How work is divided among threads.
  - How you handled synchronization in your program.
  - How and why your program's performance scales with the number of threads. Vary the number of threads and present the data and trends clearly.

  Diagrams are not required but may help you explain something clearly without taking much space. Include any relevant details you think will help us understand your implementation.
- (2 marks) Description, visualization, and data on your execution, including:
  - How and why your program's performance changes based on parameters in the input file.
  - How and why your program's performance is affected by the type of machine you run it on. Include a comparison of at least *three* different hardware types.
- (2 marks) Describe at least TWO performance optimizations you tried, including supporting measurements and hypotheses on why they worked or did not work. Do not include optimizations which have already been recommended in the writeup, instead focus on optimizations that you have come up with.

Please make sure to support your statements with clear data – tables, graphs, etc. We are looking for good quality, scientifically sound reports.

Additionally, your report should have an appendix (does not count towards page limit) containing:

- Details on exactly how to reproduce your results, e.g. nodes, inputs, execution time measurement, etc.
- Relevant performance measurements, if you don't want to link to an external document.

Tips:

- There could be many variables that contribute to performance, and studying every combination could be highly impractical and time-consuming. You will be graded more on the quality of your investigations, not so much on the quantity of things tried or even whether your hypothesis turned out to be correct.
- Performance analysis may take longer than expected and/or run into unexpected obstacles (like your program failing halfway). **Start early** and test selectively.

4.4 Bonus - Speed Contest

You may obtain up to 2 bonus marks for achieving some of the fastest implementations in the class. More details can be found in the Assignment 1 Bonus section of the Student Guide.

4.5 Template Code

We provide skeleton code that can be used as a starting point for your implementation. The code is written using C++, but feel free to change this code, including changing to C (though other languages are not acceptable), as long as all other requirements are met. Furthermore, you might consider writing your program using a mix of C and C++.

The template code provided includes the files listed in Table 1:

File name	**Description**
`sim.cc`	Example initial code for the assignment that calls our collision and I/O libraries.
`collision.h`	Our reference implementation of the collision checking and resolution logic.
`io.h` and `io.cc`	Our reference input/output code for reading the input file and writing to the output file, including reference implementations of the Particle type. **Do not change either of these files!**
`sim_validator.h` and `sim_validator.a`	The header file and library file for our simulation validation checker.
`bench-{i}.perf`	Benchmark executables in performance mode to compare against the speed of your code.
`Makefile`	Default Makefile for compiling all the provided code. Modify this as you add more source and header files.
`tests/`	Subfolder with example test cases.
`gen_testcase.py`	Script to help you generate test cases.
`viz.py`	Script to visualize small runs.
`run_bench.sh`	Example script to run your program on Slurm against the existing benchmarks, on a specific machine type.
`slurm_job.sh`	Example Slurm job script to run non-interactively.

Table 1: List of provided files in the skeleton

4.5.1 Quickstart

1. Run `make` in the folder with the `Makefile`
2. This will produce two executables from your code: `sim` and `sim.perf`. Recall that `sim` should call our validator, and `sim.perf` will have those calls removed automatically.
3. Check the correctness of your code by running `./sim tests/correctness/cradle.in 8` for example. You should observe that the validator fails and exits your program as currently, no particle positions are being updated.
4. Run a performance benchmark (assuming you are executing it within a `soctf` node) by running: `./run_bench.sh i7-7700 tests/small/random10.in 8`. You'll notice that your code "beats" all the benchmarks! Great! Actually, this is because your code is not doing anything yet (and this is not detected as the performance benchmark does not run correctness checks).
5. Go forth and make your code correct, and then fast!

4.5.2 Using the `run_bench.sh` benchmark runner

The `run_bench.sh` script is used to run your program against the provided benchmarks. The script has the following usage: `./run_bench.sh <machine> <input_file> <num_threads>`. The script will run your program with the given input file and number of threads on the specified machine type via `srun`. It will compare the runtime of your program against the provided benchmarks by running each three times. We may not use this exact script in our grading, and it's more for your convenience.

Caching results (important!): Our script caches the results of benchmark runs in a subfolder called `checker_cache/`. This is to avoid running the benchmark multiple times for the same input and machine type (since the results should not substantially change, as our benchmarks don't change). If you would like to clear this cache for any reason, *delete the checker_cache folder*.

4.5.3 Using and Generating Test Cases

We provide some starter test cases in the `tests/` subfolder. These include: `correctness/` (very short cases can be visually verified), `small/`, `standard/` (test cases that meet the requirements for benchmarking), and `large/` (very long execution time).

We provide a Python script to help you generate more test cases: `gen_testcase.py`. You can run it as such:
`./gen_testcase.py num_particles grid_length radius steps min_velocity max_velocity`

A testcase with these parameters will be printed to `stdout`, with particle velocities randomized between `min_velocity` and `max_velocity`. The generator will also print to `stderr` the *particle density* and *whether the test case is valid for performance checks* (see Section 4.2). We let you generate testcases failing this requirement as smaller/different test cases may be useful for testing.

4.5.4 Visualization for Debugging

Debugging the correctness of your programs might be difficult without visualizing what is going on. To do so, *only for relatively small testcases*, ensure this line is below `validator.initialize(particles)`:

```
validator.enable_viz_output("test.out");
```

Now, when you run `sim`, an output file containing particle information for each step will be created called `test.out` (specify any filename you want). To visualize the simulation as an `.mp4` file, run:
`python3 viz.py <path to test case> <path to out file>`

This will create a `simulation.mp4` file (filename can be changed with `--output` argument). Have fun!

5 Admin

5.1 Accessing the Template Code

We will use GitHub Classroom for this assignment. **Name your team** `a1-e0123456` (if you work by yourself) or `a1-e0123456-e0654321` (if you work with another student) – substitute your NUSNET number accordingly. You can access the Classroom via https://classroom.github.com/a/RDMgcxBt. We will provide the template code for the assignment through this repository.

You **must** use only the GitHub Classroom repository during this assignment. That is, you should not create other repositories for your codebase.

5.2 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered in this document here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions**.

If there are any questions regarding the assignment, please use the Discussion Section on Canvas (preferred, as we need to disseminate information to students regardless) or email Sriram (srirams@comp.nus.edu.sg).

5.3 Deadline and Submission

Assignment submission is due on **Friday, 27 September, 2pm** (note: **not midnight!**).

You are required to do **two important steps** for submission.

- **GitHub Classroom**: The implementation and report must be submitted through your GitHub Classroom repository.
  - Push your **code and report** to your team's GitHub Classroom repository.
  - Your report must be a PDF named `<teamname>.pdf`. For example, `a1-e0123456-e0654321.pdf`. `<teamname>` should **exactly match** your team's name; if you are working in a pair, please DO NOT flip the order of your NUSNET IDs.
  - **Tag the commit** that you want us to grade with `a1-submission`; if you forget to add such a tag, we will be forced to use the most recent commit.
- **Canvas Quiz for Assignment 1**: Take the Assignment 1 quiz on Canvas and provide *both* the **name of your GitHub Classroom repository** and the **commit hash** corresponding to the `a1-submission` tag; if you are working in a team, only one team member needs to submit the quiz. If both of you submit, we will take the latest submission.

A penalty of 5% per day (out of your grade for this assignment) will be applied for late submissions.

Final check: Ensure that you have submitted to both **Canvas** and **GitHub Classroom**. Canvas should contain your commit hash and repository name, and GitHub Classroom should contain your code and report.