PHAS0100: Research Computing with C++

Assignment 2: Solving pedestrian flow dynamics using the Social Force Model

Late Submission Policy: Please be aware of the UCL Late Submission Policy, which is here: https://www.ucl.ac.uk/academic-manual/chapters/chapter-4-assessment-framework-taught-programmes/section-3-module-assessment#3.12

Extenuating Circumstances
The course tutors are not allowed to grant Extenuating Circumstances for late submission. You must apply to your programme (Masters course, MRes, UG etc) organiser. If granted, your programme organiser or programme administrator must email j.dobson@ucl.ac.uk confirming that Extenuating Circumstances have been granted and providing a revised submission date.

Assignment Submission
For this coursework assignment, you must submit by uploading a single Zip format archive to Moodle. You must use only the zip tool, not any other archiver, such as .tgz or .rar. Inside your zip archive, you must have a single top-level folder, whose folder name is your student number, so that on running unzip this folder appears. This top-level folder must contain all the parts of your solution. Inside the top-level folder should be a git repository named PHAS0100Assignment2. All code, images, results and documentation should be inside this git repository. You should git commit your work regularly as the exercise progresses.

Due to the need to avoid plagiarism, do not use a public GitHub repository for your work - instead, use git on your local disk (with git commit but not git push), and **ensure the hidden .git folder is part of your zipped archive** as otherwise the git history is lost. You can push to a private GitHub repository but it is your responsibility to ensure that it is private and that others cannot see or copy your work as pieces of identical (or suspiciously similar) work will both be treated as plagiarised. Whilst it is fine to look up definitions and general documentation for C++ libraries online it is not acceptable to use code taken from the internet or other sources for the problem solutions. The exception to this is the base PHAS0100Assignment2 project that can be used as a template CMake project.

You are encouraged to use https://github.com/jeyjeyj0bson/PHAS0100Assignment2 as the starting point for your solution to this assignment as it comes with the required `pos2d`, `vec2d` and `dir2d` types needed for periodic boundary conditions in the `x` and `y` directions, but you can also choose to implement your own project from scratch if you want. Do be aware that the code is expected to compile and run, and marks will be deducted if the markers cannot run the code. Your documentation should give very clear build and run instructions in `README.md`.

In general, you are free to develop in your own environment, but you need to ensure that your code compiles and runs on Ubuntu 18.04.3 and with g++ 7.4.0 (enforcing C++14) as this is the environment the markers will use to test the code.

Preliminaries

The aim is to write a piece of software that solves the **Social Force Model (SFM) for pedestrian dynamics** from [Helbing & Molnar Phys. Rev. E 51, 4282 (1995)]. While implementations of this are fairly widespread in many libraries, especially in other languages such as Python, the aim here is to demonstrate that you can produce a C++ project that can be run by other people, on their machines. Based on the covered topics in class and in Assignment 1, this project will make use of:

- Build setup using CMake
- Unit testing
- Error handling using exceptions
- Avoiding raw pointers – either use STL containers, or smart pointers
- Program to Interfaces
- Dependency Injection
- Factory patterns

- Shared memory parallelism
- Code profiling and optimisation

and the aim here is to put this all together into a coherent project.

Reporting Errors
You can post questions to the Moodle Discussion/SLACK channel, if you believe that there is a problem with the example CMake code, or the build process in general. To make error reporting useful, you should include in your description of the error: error messages, operating system version, compiler version, and an exact sequence of steps to reproduce the error. Questions regarding the clarity of the instructions are allowed, but obvious "Please tell me the answer" type questions will be ignored.

Important: Read all these instructions **and the paper** before starting coding. In particular, some marks are awarded for various quality aspects including:

1. Version control with a reasonable git history that should reflect your thinking as you developed
2. Layout/naming of files, classes, methods and variables
3. Error handling
4. Information hiding

Part A: Social Force Model for pedestrian dynamics (35 marks)

The first part of this coursework is to get you to setup with the basic data types and functionalities needed for the completion of the project and to reproduce a standard result. These instructions will guide you through.

1. Please read **Social Force Model (SFM) for pedestrian dynamics** from [Helbing & Molnar Phys. Rev. E 51, 4282 (1995)] (a PDF is available on the moodle coursework page) [0 marks]

2. To help you getting started with simulations in 2D, we provide a number of custom defined 2D vector types to facilitate basic vector arithmetic and logic (periodic boundary conditions). At the core is `vec2d` which implements vector addition, subtraction, scalar product and scaling). Based on this are the derived `pos2d` which implements periodic boundary conditions in the *x* and *y* directions together with methods calculating the *distance* between two positions, the *direction* vector from a given vector from the current one and the position in the given direction from the initial position (*displace*). `dir2d` implements the cosine between the current vector and a given one. Based upon these vector types, you are expected to:
    a. Create a header file for `Pedestrian` class that contains the following data members to be of the most suited type:
        i. Origin – 2D starting position $\overrightarrow{r_\alpha}(t=0)$ [m]
        ii. Destination – 2D position $\overrightarrow{r_\alpha^0}$ [m]
        iii. Velocity – 2D velocity $\overrightarrow{v_\alpha}(t)$ [m/s]
        iv. Position – current 2D position $\overrightarrow{r_\alpha}(t)$ [m]
        v. Desired speed - v_α^0 [m/s]
        vi. Relaxation time - τ [s]

    b. Create an implementation file for the `Pedestrian` class and ensure both files are added to `CMakeLists.txt`, so that your build environment will know it exists.

    c. Create a unit test file, that will instantiate an instance of your new concrete class.

    d. Check that you can compile and run the test and then add some unit tests to convince yourself that you understand the periodic boundary conditions.

    Hints:

A simple 2D rectangular world can be realised using `pos2d` to represent positions; the periodic boundary conditions at 0 and `POS2D_XWRAP` in `x` and 0 and `POS2D_YWRAP` in `y` mean that a pedestrian that steps beyond a boundary will appear to be transported back to the start of the opposite boundary to complete the step.

To understand the periodic boundary conditions try creating a Pedestrian with an initial position beyond a boundary; or creating a Pedestrian within the world but then displacing (stepping) their position outside using `dir2d`.

[1 + 1 + 1 + 2, 5 marks total]

3. Now we implement the three necessary forces for a rudimentary example of pedestrian dynamics, namely i) the attractive force to destination, ii) the pedestrian-pedestrian repulsive force and iii) the obstacle (wall) -pedestrian repulsive force:
    a. Using the Pedestrian class, implement the attractive force to destination for a single pedestrian instance.

    (eq 2 in Helbing & Molnar)

    b. Implement the pedestrian-pedestrian repulsive force *(eq 3 in Helbing & Molnar)* and the overall resultant force that one pedestrian experiences from all other pedestrians.
        i. This should make use of helper methods to compute, for instance, the value of the pedestrian private "ellipse" ellipse semiminor axis `b` variable.

        (eq 4 in Helbing & Molnar)

        ii. The force on pedestrian α should be repulsive and directed along the line connecting the two pedestrians
        iii. Implement the field-of-vision form factor to be applied to this force *(eq 7 in Helbing & Molnar)*
        iv. Sum the contributing forces from all other pedestrians β to calculate the overall force on pedestrian α

    c. Implement a border-pedestrian repulsive force from the horizontal borders at `y` = `0` and `y` = `POS2D_YWRAP`:
        i. This should make use of the distance between the pedestrian and the closest border
        ii. The force should be directed along the line connecting closest border and the pedestrian such that the pedestrian is repelled from the border

        (eq 5 in Helbing & Molnar)

    d. Implement a method to calculate the resultant of the three forces above given an input of the current pedestrian and a list
    (for example `std::vector<shared_ptr<Pedestrian> >` of other pedestrians).

    e. Create unit tests that will check all the previous classes.

Hints:

The above can be implemented as methods in the `sfm` namespace in a new `Forces` library declared and defined in `Forces.h` and `Forces.cpp`.

When implementing parts *b* and *c* you should use the potentials as described in *eq 5 in Helbing & Molnar* with constants as defined in the text; as these are exponentials it is straightforward to differentiate $V_{\alpha\beta}(b)$ and $U_{\alpha B}(|\vec{r}_{\alpha B}|)$ with respect to the semi-minor axis `b` and the distance between the pedestrian and the nearest wall $|\vec{r}_{\alpha B}|$, respectively, when calculating the forces.

[3 + 4 + 3 + 2 + 3, 15 marks total]

4. The idea is now to solve the Newtonian equations using two nested loops: an outer loop in time and then an inner loop over pedestrians to calculate forces and update the velocity and position values.
    a. First, create an application that initialises a `std::vector` containing pedestrians.

    b. Then, implement a loop in time from t = `0` to t = `finish_time_s` with a parametrised time spacing `dt`.

    c. Then implement a loop over all the pedestrians and for each pedestrian calculate the resultant of all forces using the `Forces` library from section *3.d*

    d. Then based on the calculated resultant force on each pedestrian update the pedestrian's velocity and position variables:
        i. Calculate the new velocity $\overrightarrow{v_\alpha}(t+dt) = \overrightarrow{v_\alpha}(t) + \overrightarrow{F_\alpha}(t) \times dt$
        ii. If the magnitude of $\overrightarrow{v_\alpha}(t+dt)$ is greater than $v_\alpha^{max} = 1.3 \times v_\alpha^0$ then renormalise it to v_α^{max} *(see eqs 11 and 12 in Helbing & Molnar)*
        iii. Then update the position $\overrightarrow{r_\alpha}(t+dt) = \overrightarrow{r_\alpha}(t) + \overrightarrow{v_\alpha}(t+dt) \times dt$

Hints:

Whilst the Helbing & Molnar paper uses a time spacing of 2 s more recent implementations of the social force model yield better results with shorter time spacing and you are advised to use a step size of between 0.1 and 0.5 s.

When developing the above try some initial configurations of 1, 2 and 3 pedestrians with different Origin and Destination positions and print key calculated quantities to screen to check them.

[2 + 2 + 3 + 3, 10 marks total]

5. In order to check the validity of the previous model we now implement and visualise a more realistic corridor model.
    a. First, create an initial set of pedestrians of two groups starting at either end of the world (at `x = 0` and `x = POS2D_XWRAP)` and set their `Destination` to the opposite wall (at `x = POS2D_XWRAP` and `x = 0`).
    b. Incorporate the visualisation tool that will be provided and pass it the vector of pedestrians after each update
    c. Run and visualise the example with the tool provided and convince yourself it is working. To demonstrate take screen grabs of visualisation at t = 0 and after a period of time, for example after 10 seconds of simulated timesteps, and commit them.

Hints:

A reasonable initial configuration is 20 Pedestrians at each end of world with y positions distributed uniformly along the vertical wall with $v_\alpha^0 = 1.3\,m/s$ and $\tau = 0.5\,\mathrm{m/s}$.

When setting target positions for pedestrians starting at x = 0 do not set the target x position to exactly `POS2D_XWRAP` as due to the periodic boundary conditions this will be set to x = 0 and the pedestrian will try to get to the wrong boundary. Instead set it to a value just below, for example `POS2D_XWRAP - 0.01`

[2 + 2 + 1, 5 marks total]

Part B: Improvements (25 marks)

6. Next you will work on developing a design pattern suited for this class of problems called the Factory Pattern. The core idea is to develop an interface (factory) which is responsible for the creation of products (pedestrians). Thus, when the caller requests a product, it instantiates a factory which holds all the details about the creation of potentially different products (in this case different types of pedestrians). In detail, this means:
    a. Create classes that implement the following two types of pedestrian:
        i. "targeted" pedestrians which always try to travel to a target point `pos2d`
        ii. "directional" pedestrians which always try to travel along a given `dir2d`

    b. Design and implement a factory for the creation of different types of pedestrians and initial distributions.
        i. The idea is to develop a `PedestrianSpawner` class (the factory) to create different initial setups for the pedestrian crowds and which can be used to create pedestrians of either "targeted" or "directional" type

    c. Develop two methods in `PedestrianSpawner` for the creation of a "uniform" initial distribution of pedestrians and a "distributed" initial distribution
        i. "uniform" means pedestrians with starting positions randomly generated within the whole 2D world
        ii. "distributed" means pedestrians generated randomly within a box defined by `x_start` to `x_end` and `y_start` to `y_end`

    d. Setup, run and visualise a group of targeted pedestrians starting within a box at one end of the world and with a target at the opposite end

    e. Setup, run and visualise a group of targeted pedestrians as above but now together with a group of directional pedestrians that start at the opposite end of the world and travel in the opposite `-x` direction

    To demonstrate d. and e. you should take screen grabs of the visualisation at t = 0 and after a period of time, for example after 10 seconds of simulated timesteps and commit them.

    Hints:

    To implement the "targeted" and "directional" pedestrian types consider making the Pedestrian class an abstract base class with a pure virtual method `virtual pos2d GetTarget() = 0` which is then overridden by the concrete derived TargetedPedestrian and DirectionalPedestrian classes. The "targetted" derived class would then simply return the

value of the data member storing the Target `pos2d` variable and the "directional" derived class would instead calculate a `pos2d` to return based on the desired direction.

An example of a base class with a static factory method to produce different derived classes can be found here sourcemaking.com/design_patterns/factory_method/cpp/1 although note that for this course you should avoid the use of raw pointers.

[3 + 3 + 3 + 2 + 2, 13 marks total]

7. The final point of investigation is performance.
   a. Benchmark your code using `std::chrono::high_resolution_clock` and `std::clock,` see example here: en.cppreference.com/w/cpp/chrono/c/clock.

   b. Try to improve performance using OpenMP to implement parallelisation.

   c. Benchmark again with respect to incrementing the number of threads through the `OMP_NUM_THREAD` and provide a summary in the README.md of benchmarking results before/after parallelisation.

N.B. marks will be based on the implementation of parallelisation and the ability to benchmark and not on actual performance achieved.

Hints:

You may need to update the settings on your VM to make sure it can use multiple cores if they are available on the host machine: power off the virtual machine and then go to VirtualBox Settings -> System -> Processor and increase to the maximum available.

When implementing parallelisation see Lecture 8 and Homework 36 05OpenMP/cpp/forloop/openmpforloop.cc for an example of how to parallelise a loop with `#pragma omp for.` Consider splitting the loop over pedestrians into two separate loops each of which is parallelised separately: an initial loop that calculates the forces and then a subsequent loop that updates the positions and velocities.

[3 + 7 + 2, 12 marks total]

Part B: Additional Considerations (10 marks)

8. Overall code quality: variable/class/method naming, code/class layout, test coverage, error handling and information hiding

   [6 marks total, at markers discretion]

9. Nice git commit log. Effective commenting in code

   [2 marks]

10. Update README.md to give clear build instructions, and instructions for use

    [2 marks]