A Virtual Solar System

PHAS0100 2022/23: Research Computing with C++ Assignment 2

Errata

None.

If you believe you have found a mistake in the assignment, please contact the module leader.

Late Submission Policy

Please be aware of the UCL Late Submission Policy (Section 3.12).

Extenuating Circumstances

The course tutors are not allowed to grant Extenuating Circumstances for late submission. You must apply by submitting an Extenuating Circumstances form which can be found on the Extenuating Circumstances information website. The Department EC Panel will then decide whether or not to grant extenuating circumstances and email the module lead 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 `tar.gz` format archive to Moodle. This is different to the first assignment but will allow you to check your submission before the deadline. Inside your archive, you must have a single top-level folder, whose folder name is your student number, so that this folder appears when the archive is opened. This top-level folder must contain all the parts of your solution. Inside the top-level folder should be a git repository named exactly `PHAS0100Assignment2`. All code, images, results and documentation should be inside this git repository. You must include the `.git` folder otherwise we cannot mark your git commit log. Your folder structure should look similar to (but not exactly the same as):

```
- 314159   // YOUR STUDENT NUMBER
  - PHAS0100Assignment2
    - .git
    - README.md
    - src
    - app
    - test
    - ...
```

You can create a correctly formatted `.tar.gz` file containing the folder `314159` from inside the docker container by running the following command from the folder than includes `314159`:

```
tar czvf submission.tar.gz 314159
```

You should test that your archive produces this structure **from inside the docker container** by **moving the archive to a different location** and running:

```
tar xzvf submission.tar.gz
```

If you see the same folder as the one you extracted, it has been correctly archived. This will guarantee that your archive will extract correctly during the submission process (which uses the docker container).

You can use GitHub for your work if you wish, although we will only mark the code you submit on Moodle. Due to the need to avoid plagiarism, do not use a public GitHub repository for your work. We have provided a private repo unique to you accessible through Github Classroom. **To access the starting project, using the following link**:

https://classroom.github.com/a/Mi_A2qj4

After you accept the permissions, you will have access to a repository named `nbody-<gh_username>`. This will contain the starting project for your own work.

Important: You may have already cloned a similar starting repository from last year. **It is vital you use the above link to access the starting project.**

It is important that you follow the file/folder/executable structure of the starting project as well as the form and name of any functions/classes/executables described in any sections below. This is because we will be automating basic checks of the submitted work so if you have not followed the pre-defined structure those checks will fail and you will lose marks. Where it is not specified, you can and should create new files, functions, classes and edit the `CMakeLists.txt` as you see fit.

Your code should be split between the `src`, `lib`, `include` and `test` directories. This is to simulate a real-life project where you are writing both a **library** (containing code another developer might want to use if they were developing their own simulation) and an **application** (containing code related to an application a user might use to run the simulation). All app-related code should go in `app` while all code that could be part of the library should go in `src`. Only code inside `src` can be tested by the unit tests in `test`. You should take care to chose an appropriate location for each piece of code you write.

CTest will be used to run all unit tests. You can use CTest by running `ctest` from inside the `build` directory as described in `PHAS0100Assignment2/README.md`. **Since the first assignment, CTest has been set up so that it automatically detects all tests**. We have already provided sample tests in the `test` directory. You can add new tests inside the existing test file and these will be run without any other changes, however if you add an entirely new test file (i.e. `test/particle_test.cpp`) you will have to edit `PHAS0100Assignment2/test/CMakeLists.txt` to ensure the new file is built into the test executable. Instructions are provided in the starting project `README.md`.

We recommend you use Visual Studio Code IDE with the supplied devcontainer running Ubuntu 22.04 docker image for development (the correct Dockerfile is included for you in the above repository). This is not a requirement but if you do use a different setup then you must ensure that your code compiles and runs on Ubuntu 22.04 and with g++ 11.3.0 (enforcing C++17) and CMake 3.22.1 as this is the environment the markers will use to test the code. By default Docker should permit unlimited access to your CPU but if you notice Docker cannot access as many cores as you expect, you may need to change your settings. See the Docker Desktop Windows Resources documentation, Docker Desktop MacOS Resources documentation, or the Docker Desktop Linux Resources documentation. Setting can also be set via the command-line docker interface, see the relevant documentation.

Marks will be deducted for code that does not compile or run. Marks will also be deducted if code is poorly formatted. You can choose to follow your own formatting style but it must be applied consistently. See the Google C++ style guide for a good set of conventions regarding code formatting.

Reporting Errors

Contact the lecturer directly if you believe that there is a problem with the starting 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. Any corrections or clarifications made after the initial release of this coursework will be written in the errata section and announced.

Assignment: A Virtual Solar System

Important: Read all these instructions before starting coding. In particular, some marks are awarded for a reasonable git history (as described in Part 3), that should show what you were thinking as you developed.

You should first read through the Wikipedia article on Numerical models of the Solar System to understand the high-level physics behind the simulation we will be writing. You may also find Wikipedia's article on N-Body simulations useful but it goes into more complex methods of simulation than you will implement in this assignment.

The instructions will be more specific at the start of the assignment and become less specific towards the end. This is to allow you the opportunity to show your understanding of good software engineering practices.

1 Solar System Simulator (30 marks)

In this first part of the assignment, you will write and test a simple N-body simulator that models the dynamics of the Solar System. We can consider the Solar System as a system of particles which interact via gravity. You will build up the simulation piece by piece, adding numerical approximations to the governing equations as you go. The high-level overview of the simulation's algorithm is:

1. For every particle, calculate the gravitational force applied by every other particle.
2. For every particle, update the particle's acceleration using the previously calculated force.
3. Move the simulation forward in time by a small amount dt, specifically:

- Update the velocity according to the particle's current acceleration.
- Update the particle's position according to the particle's updated velocity.

4. Repeat until the final time is reached.

1.1 Particles

1.1.1 Creating the `Particle` class

First you will create a `Particle` class that holds a particle's position, velocity, acceleration and mass, and can update its position and velocity when it is stepped through time. The class should provide at least the following four public methods:

- `Eigen::Vector3d getPosition()` that returns the current position,
- `Eigen::Vector3d getVelocity()` that returns the current velocity,
- `double getMass()` that returns the particle's mass,
- `void update(double dt)` which updates the position and velocity according to the equations below.

Given an acceleration and a timestep `dt`, `update` should update the position and velocity according to the following equations:

$\vec{x}_{n+1} = \vec{x}_n + dt\ \vec{v}_n$

$\vec{v}_{n+1} = \vec{v}_n + dt\ \vec{a}$

That is, in the time dt the velocity changes by an amount $dt\ \vec{a}$ and the position by an amount $dt\ \vec{v}$. Here $\vec{v}$ is a vector with components (v_x, v_y, v_z). This method of integrating the equations is known as the forward Euler method of numerical integration. There are alternative methods that are more accurate but forward Euler is particularly simple. As an example of reading this equation, here is the snippet for updating a particle's position:

```
position += dt * velocity;
```

Hints:

- Consider whether any of `Particle`'s methods, arguments or return values should be declared `const`, and modify them if so.

- Decide which arguments and return values should be references and/or pointers.

1.1.2 Testing the `Particle` class

Write unit tests to confirm that a particle moves as it should when:

1. There is no acceleration $\vec{a} = \vec{0}$.
2. There is a constant acceleration.
3. When an artificial acceleration $\vec{a} = -\vec{x}$ is applied. You will have to update the acceleration as $\vec{x}$ changes during the simulation.

In the 3rd test case, this form of acceleration should result in a particle orbiting in a circle around the origin. Setting $\vec{x} = (1, 0, 0)$ and $\vec{v} = (0, 1, 0)$ and integrating the system using a small $dt = 0.001$ until a final time of 2π should result in the particle's final position being approximately equal to its initial position.

Hints:

- The `Eigen::Vector3d` type supports vector addition and subtraction, as well as scalar multiplication. You should include `<Eigen/Core>` in your code to make use of it.
- For testing, Eigen provides the method `Eigen::Vector3d::isApprox` to allow the comparison of two vectors with `v1.isApprox(v2)`. Extra arguments can be supplied to change the precision.

[4 marks implementation, 4 marks unit test: 8 total]

1.2 The gravitational force

1.2.1 Calculating acceleration

Now create a function that calculates the acceleration felt by one particle due to another. It should have a signature similar to:

```
Eigen::Vector3d calcAcceleration(<particle one>, <particle two>, epsilon=0.0)
```

You must decide the exact nature of the arguments, e.g. whether to pass by value or reference, the type of the arguments, and whether the arguments should be declared `const`. `epsilon`, the softening factor found in the below relation for acceleration should take a default value of 0 but will be set in a later section.

`calcAcceleration` should return the acceleration felt by particle i due to the gravitation force produced by particle j, calculated via the following equation:

$\vec{a}_{ji} = m_j(\vec{x}_j - \vec{x}_i)/(d_{ji}^2 + \epsilon^2)^{3/2}$,

where m_j is the mass of particle j, $d_{ji} = |\vec{x}_i - \vec{x}_j|$ is the distance between particles i and j, and ϵ is a small value called a softening factor that helps with stability when two particles are very close and d_{ij} is nearly 0. Due to the choice of scaling seen later, the Gravitational constant normally found in this equation is 1.

You should then add a way to calculate the acceleration on a particle due to all other particles. This may be a method inside `Particle`, a function, or some other structure. This is for you to decide. It should calculate the sum of all accelerations from each particle:

$\vec{a}_i = \sum_{j \neq i} \vec{a}_{ji}$

You must decide how to store a list of particles and how to ensure a particle does not interact with itself. The result of this calculation should be stored in the `Particle`'s `acceleration` variable. The value will then be used in the `update` method during a later stage. Do not calculate the acceleration inside `update` because these two operations should remain decoupled as we later parallelise the code.

1.2.2 Testing the acceleration code

You should test that:

1. The gravitational force between two particles is as expected. You should calculate a few test cases by hand then encode them in a unit test.
2. Providing a list of particles that includes itself to a particle will not add any acceleration.
3. If a particle has two equally sized particles an equal distance either side of it (e.g. at $\vec{x}_1 = (1, 0, 0)$ and $\vec{x}_2 = (-1, 0, 0)$) your code correctly calculates zero acceleration.

[5 marks implementation, 3 marks unit test: 8 total]

1.3 Building the Solar System

Using the `Particle` class you developed you will now build an application to model the motion of the Sun and planets in the Solar System:

a. Create a command line app that can be run as `build/solarSystemSimulator` from the project root directory. It should have arguments to control the timestep `dt` and either the total length of time or total number of timesteps to simulate. Since this is intended to be a HPC application, is it not acceptable to use `std::cin` anywhere. **[2 marks]**

b. The command line app should print a useful help message to tell the user what arguments to use if the `-h` or `--help` switches are used. With zero command line arguments, the program should not crash, and should respond with the help message. **[1 marks]**

c. First the application should create a list of `Particle`s corresponding to the major bodies (the Sun plus 8 planets) within the Solar System with appropriate initial conditions. We will refer to the function or class that creates this list as an *initial condition generator*. You should use the following list of masses and distances from the Sun, all scaled so that the Sun's mass is 1 and the distance between the Sun and the Earth is also 1. Using this scaling, the Earth will travel around the Sun exactly once every 2π, i.e. one year corresponds to a time of 2π. The initial data is:

```
// Masses in order Sun, Mercury, Venus, etc
{1.,1./6023600,1./408524,1./332946.038,1./3098710,1./1047.55,1./3499,1./22962,1./19352}
// Distances from Sun
{0.0, 0.4, 0.7, 1, 1.5, 5.2, 9.5, 19.2, 30.1}
```

You should set the Sun's position and velocity to zero and randomly choose an angle θ for each of the planets. You should set their positions and velocities to result in a stable, circular orbit using the following equations:

$x_x = r\sin(\theta)$,

$x_y = r\cos(\theta)$,

$v_x = \frac{-1}{\sqrt{r}}\cos(\theta)$,

$v_y = \frac{1}{\sqrt{r}}\sin(\theta)$,

where r is the given planet's distance from the Sun. You may set the z components of the position and velocity to 0.

You should write a unit test testing this generator. **[4 marks]**

d. Next you need to implement the evolution of the Solar System with time. To do this you should create an outer loop to loop over all timesteps. This can either be a `while` loop that tests if the final time has been reached or a `for` loop that iterates until the total number of timesteps has been reached. Within this loop there should be two separate `for` loops, each looping over all of the Solar System bodies:

   i. In the first for loop the code should update the gravitational acceleration for all bodies given their current positions.
   ii. In the second for loop the position and velocity of each body should be updated. **[4 marks]**

e. Print appropriate messages to screen summarising the position of the Solar System bodies at the start and end of the simulation. Choose a suitably small timestep and simulate the system for 1 full year (i.e. until a time of 2π). You should copy the output into a section in the `README.md` as proof before submitting. **[3 marks]**

Hints:

- To test the main loop you should manufacture simple test cases like a single body orbiting around the Sun and test the expected positions after just one timestep.
- You should expect the Earth to return to close to its original starting position after a time of 2π
- You should use built-in values for constants like π rather than defining your own
- The simulation's accuracy and stability is dependent on the size of the timestep `dt`. You should test with an unnecessarily small value of 0.0001 to ensure correct behaviour over 1 year.
- You may reuse the `Timer` class from the OpenMP examples.

2 Improving the simulation (30 marks)

In this part of the assignment you will quantify the accuracy of the simulation and benchmark its performance using timers. You will then improve the performance of the simulation using OpenMP to parallelise the relevant parts.

2.1 Calculating numerical energy loss

As there is no energy being added to or removed from the system of bodies the total energy should remain constant as the simulation evolves. However given the simplistic numerical integration total energy will not be conserved and its loss will depend on the step size. You will use the loss in the total energy to measure the accuracy of the simulation. Total energy in this system is made up of kinetic and potential energy. The kinetic energy of particle i is:

$E_i^{kinetic} = \frac{1}{2} m_i |\vec{v}_i|^2$

and its potential energy is:

$E_i^{potential} = -\frac{1}{2} \sum_{i \neq j} \frac{m_i m_j}{d_{ij}}$

The total energy of a particle i is $E_i^{total} = E_i^{kinetic} + E_i^{potential}$ and the total energy of the entire system is the sum of all individual particle energies:

$E^{total} = \sum_i E_i^{total}$

Again, you must take care in the calculation of the potential energy term to avoid calculating the potential energy due to an interaction of a particle with itself.

You should add code to calculate the kinetic, potential and total energy for the system and print the values out out before and after the simulation has run. By how much does the total energy drop over a single simulation run? You should simulate 100 years of time ($100 \times 2\pi$ simulation time) using 8 different timesteps and summarise the results in your `README.md`. You should find that more energy is lost when `dt` is large, i.e. the simulation is less actuate.

You should decide whether the new code should be part of the `Particle` class, standalone functions, part of a new class, or in another place. There are many valid designs for this code and you will be marked on your design choices and how well it is implemented.

[5 marks total]

2.2 Benchmarking the simulation

Next you should implement a timer and benchmark the runtime of your code using `std::chrono::high_resolution_clock`. You should choose an appropriate location to start and end your timer to capture the performance of the

most relevant parts of your code.

Measure the time it takes your simulation to run for 100 year's ($100 \times 2\pi$ simulation time) and for 8 different timestep sizes. You should print out the total time and the average time per timestep (the total time divided by the total number of timesteps). While your total time should change with the size of the timestep, your time per timestep should remain relatively constant. You should ensure compiler optimisations are enabled and set to the second level (i.e. passing `-O2` to the compiler through CMake). Run a single simulation without compiler optimisations (i.e. with `-O0`).

Add a summary of the performance results to the `README.md` file. Comment on the difference in performance with and without compiler optimisations. Based on the performance measurements and the accuracy measurements from the previous sections, you choose a value of timestep that is a good balance between simulation run time (a few minutes is reasonable) and accuracy and document this in the `README.md`.

Hints:

- You do not want your performance measurements to be affected by printing to the screen (e.g. with `std::cout`). If you print any output in the section being timed, ensure you have some ability to disable output. You should wrap any output lines with the preprocessor directives `#ifdef DEBUG` and `#endif` like: `#ifdef DEBUG  std::cout << "Hello world!\n";  #endif`
- You should be able to enable or disable any debug during the CMake configure process with `cmake -S . -B build -DCMAKE_BUILD_TYPE=Debug ...` or `-DCMAKE_BUILD_TYPE=Release`.

[4 marks total]

2.3 Increasing the scale of the system

The small scale of the Solar System is not suitable for parallelisation. In order to show your program parallelises appropriately, we must increase the size of the system (i.e. the number of bodies) by many orders of magnitude. You will add a new initial condition generator to generate a system made up of many random particles, still orbiting a central, large star. Recall, for a body at some angle θ and distance from the central star `r` in a stable, circular orbit, its position and velocity is given by:

$x_x = r\sin(\theta)$,

$x_y = r\cos(\theta)$,

$v_x = \frac{-1}{\sqrt{r}}\cos(\theta)$,

$v_y = \frac{1}{\sqrt{r}}\sin(\theta)$,

Since you already developed an initial condition generator in section 1.3c, it is good practice at this point to consider if the code should be refactored to support two initial condition generators with a common interface. In the interest of testing your understanding of inheritance and polymorphism, you will refactor this existing generator.

You should create a new *abstract* class called `InitialConditionGenerator` with at least a pure virtual method called `generateInitialConditions` which returns a container of `Particles`. You may choose to add other methods and data methods to this class but it should remain an abstract class. This class should be the base class for two subclasses: a refactored version of the Solar System generator developed in part 1.3c and a random generator described below. You must show your understanding of polymorphism by using these through the base class.

The random initial condition generator should generate a list of particles. The first particle should represent a central star with mass 1, zero velocity and positioned at the origin. The remaining particles should have random masses and random positions in stable orbits (as given by the relations above). To simulate a real but randomised star system, the masses should be distributed between `1./6000000` and `1./1000` using a uniform distribution, and the distances from the star should be distributed between `0.4` and `30`, again using a uniform distribution. The initial angle for each body should be randomly distributed between 0 and 2π, again using a uniform distribution.

Although you are generating random values, you should manually set a seed inside the generator to ensure reproducibility while testing. Fixing the seed should result in the exact same initial conditions being generated, even in different runs of the simulation. The user does not need to be able to set the seed directly, but you should make it clear in the code how a developer can fix the seed to a particular value when appropriate.

You should also set the softening factor involved in the acceleration calculation to $\epsilon = 0.001$ at an appropriate place in the code and pass this to the acceleration calculation. If you do not intend to change the value during runtime, and can calculate its value during compilation, it is acceptable to set this as a global constant outside `main`. It is better practice, however, to ensure the user can set its value at runtime using, for example, a command-line flag. You will not lose marks for opting for a compile-time constant but you may gain marks for a well-designed way of allowing the user to set ϵ at runtime.

You should then use your generator to initialise simulations with 8, 64, 256, 1024 and 2048 particles. Summarise the performance of your simulation in each case in your `README.md`.

Hints:

- Your git log must show evidence of the changes you make as you refactor the existing Solar System initial conditions generator into an inheritance-based generator.

[6 marks total]

2.4 Parallelising with OpenMP

Next you should use OpenMP to parallelise your simulation to improve its performance. Use the 2048 particle initial conditions as the use-case and choose an appropriate step size and simulation time that results in a run time of between 1 and 5 minutes before parallelisation.

a. Parallelise your simulation using OpenMP. Think carefully about which parts of the simulation can be parallelised. It is mathematically important that within each timestep the loop over bodies to calculate forces needs to be completed for all bodies before the subsequent loop over bodies to update their position and velocities starts. You should make good use of unit tests and manual checks to ensure your code remains correct as you parallelise it.

   Where appropriate you should experiment with the `collapse` and `schedule` clauses and comment on the performance differences (if any) in the `README.md`. **[6 marks]**

b. You should now run a strong scaling experiment, like that found in *example 4* of the OpenMP notes, and a weak scaling experiment, like that found in *example 6* of the same notes. You should use enough particles in both cases so that the runtime using a single thread is at least 30 seconds. If you are using a laptop you should ensure it is plugged in, i.e. not using its internal battery since this can strongly affect performance. You should also ensure you have compiled using compiler optimisations, but you should choose the optimisation level and document this in your summary.

   For the strong scaling experiment you should increase the number of OpenMP threads until you reach the number of cores on your machine. For example, if you have 4 *physical* cores, you should increase the threads as 1, 2, 3, and 4. You may need to check if your particular CPU supports hyperthreading so you know the number of physical cores. You should then run one additional simulation with the number of threads equal to double the number of cores. Include the data as a table in your `README.md` using the following template and comment on the scaling:

```
| `OMP_NUM_THREADS | Time (<units>) | Speedup |
|---|---|---|
| 1 | x | x |
| 2 | x | x |
| ... | ... | ... |
```

   You should specify the units you chose to measure your runtimes in and calculate the speedup achieved in each case over the single-threaded runtime.

For the weak scaling experiment, you should run the same experiment as in the strong scaling experiment but you should increase the number of particles by the same factor, i.e. if you simulate 1024 particles with 1 thread, you should simulate 2048 with 2 threads, 3072 with 3 threads, and so on. Again, present the data in your `README.md` and comment on the weak scaling. The weak scaling template is:

```
| `OMP_NUM_THREADS | Num Particles | Time (<units>) | Speedup |
|---|---|---|---|
| 1 | x | x | x |
| 2 | x | x | x |
| ... | ... | ... | ... |
```

Since the performance should scale with the square of the number of particles, the speedup in the weak case will not be close to 1, as in example 6 in the OpenMP notes. [**4 marks**]

c. Although it is not part of the main loop, the calculation of the energies could become a bottleneck for large numbers of particles. Parallelise the calculation using an OpenMP reduction and test (using unit tests) that it provides the same result as the serial version. [**4 marks**]

Hints:

- Marks will be awarded based on the correctness of the parallelisation and the ability to benchmark, not on actual performance achieved so you will not be penalised for having a machine with a low number of cores.
- When running scaling tests, it can save a little time to use Bash `for` loops to automate the running of each simulation. Here is a useful tutorial on Bash for loops An example of running 4 simulations each with an increasing number of cores is: `for n in 1 2 3 4; do OMP_NUM_THREADS=$n ./a.out; done` The above can be run diretcly in the VSCode terminal.

[**14 marks total**]

3 Additional Considerations (10 marks)

As in the first assignment, you are expected to engage in good software engineering practices. Specifically, you will be marked on:

- Appropriate naming for functions and classes.
- Good code organisation (i.e. appropriate choice of files and folders)
- Informative and useful git commit log.
- Effective in-code commenting.
- Code formatting.

You are expected to implement unit tests as you see fit, not only where indicated in the assignment instructions.

Hints:

- See section 5
- Familiarise yourself with writing good git commit messages.
- Familiarise yourself with Best practices for writing code comments.
- You can use formatting tools like `clang-format` from inside VS Code to help format your code.

[**10 marks**]

4 Additional Hints

This section contains only extra, optional hints; there are no specific instructions or associated marks. This section is identical to that found in the first assignment.

4.1 Code Layout

- Make sure all header (`.h`) files have an include guard to prevent accidental double declaration of variables/functions/classes. Alternatively you can use `#pragma once`; it's less typing but it is not part of standard so not guaranteed to be available for every compiler.
- Header files should only contain function/class/variable declarations (the interface) and not their definitions (the implementation). Definitions should go in a corresponding source (`.cpp`) file; this keeps interface and implementation separate, speeds up compilation and avoids double definitions of functions/classes/variables. Templates are a caveat to this.
- Choose a consistent formatting style (for example the google C++ style guide) and stick to it. The focus should be on readability.
- Use descriptive variable, function and class names that describe the purpose/intent of an object and would be understandable to someone reading the code for the first time.
- Be consistent with indentation. Use 2, 4, etc spaces and stick to it. Avoid tabs as these can show up differently on different editors. You can use automatic tooling to help you with that.
- Comments in the header files should describe what a class/function is for and how it should be used (unless it is obvious from the function name). Avoid stating the obvious or describing literally what the code does when commenting in the implementation (`.cpp`) source files. Use comments for *tricky, non-obvious, interesting, or important parts of your code* (from cppguide#Comments).

4.2 Unit Tests

- Each unit test `TEST_CASE` should at least one meaningful assertion to check a condition has been met. Avoid meaningless checks such as `REQUIRE(true)` that will always pass. To check that a class can be instantiated you can use `REQUIRE_NOTHROW` or `REQUIRE_THROWS` to check that an exception is not/is thrown as expected, for example:

```
TEST_CASE( "FileReader file reader instantiation" , "[FileReader]")
{
  REQUIRE_NOTHROW(FileReader("file_that_exists.txt"));
  // should throw an exception for non-existent file
  REQUIRE_THROWS(FileReader("file_that_does_not_exist.txt"));
}
```

- To improve coverage of your unit tests consider whether you have fully tested the behaviour of the object or function that you are unit testing:
  - Have you tested for known solutions that you can compare to?
  - Have you tested a range of representative use cases, not just one specific case?
  - Have you tested for failure (like invalid inputs where you expect the constructor to throw an exception) and edge cases (special cases like the square root of 0)?
- Use the `Approx` catch2 method when comparing floating point numbers, for example:

```
TEST_CASE( "Average", "[MathsUtils]" ) {
  // to compare within floating point precision
  REQUIRE(GetAverage(10.0,20.0) == Approx(15.0));
  // or to compare a result to within your own defined precision
  REQUIRE(GetAverage(10.1,10.2) == Approx(10.0).epsilon(0.02));
}
```

4.3 Commit Log

- Commit messages should describe the changes being made and have enough information that someone could come back and find out where a specific change was introduced
- Avoid bundling many changes into a single commit: this makes it harder to understand what has been changed in each commit and also harder to use tools such as `git bisect` to locate a change that may have introduced a bug

- Keep the first line of commit message concise (i.e. brief but with lots of info) and add more details as part of the message body if needed