Stochastic Simulation Library

Patrick Frostholm Østergaard
Student No.: 20195087
pfas19@student.aau.dk

June 3, 2023

This report documents my implementation of the Stochastic Simulation Library for the Selected Topics in Programming course exam at AAU 2023.

The source code for the project can also be found in my GitHub repository for the course.

1 Project Files

The following two sections show the CMakeLists.txt files used to build the project as well as the source code itself. I used CLion as my IDE on Windows 11 with the bundled toolchains. Please see Figure 1 for the toolchains setup.

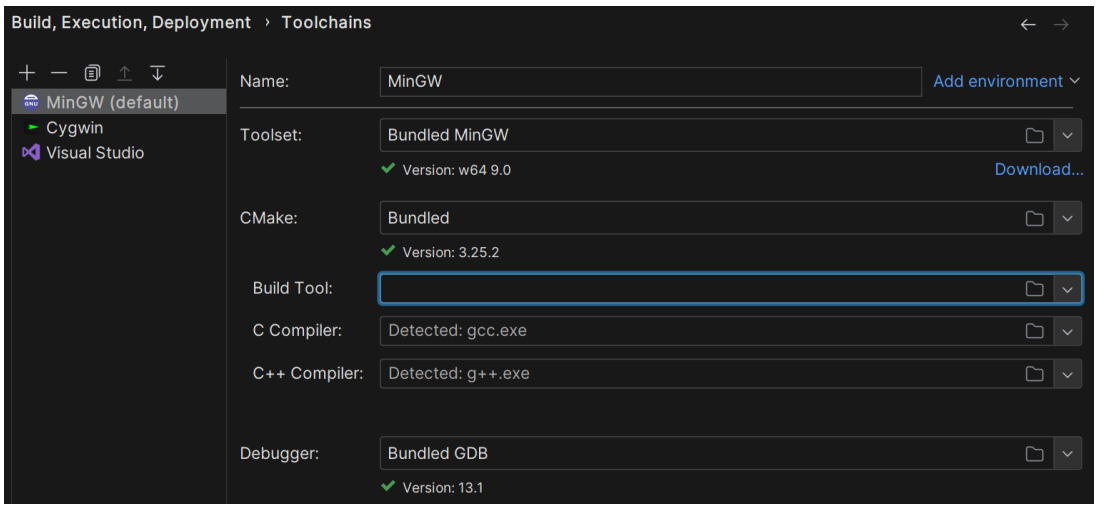

Figure 1: Toolchains setup in CLion.

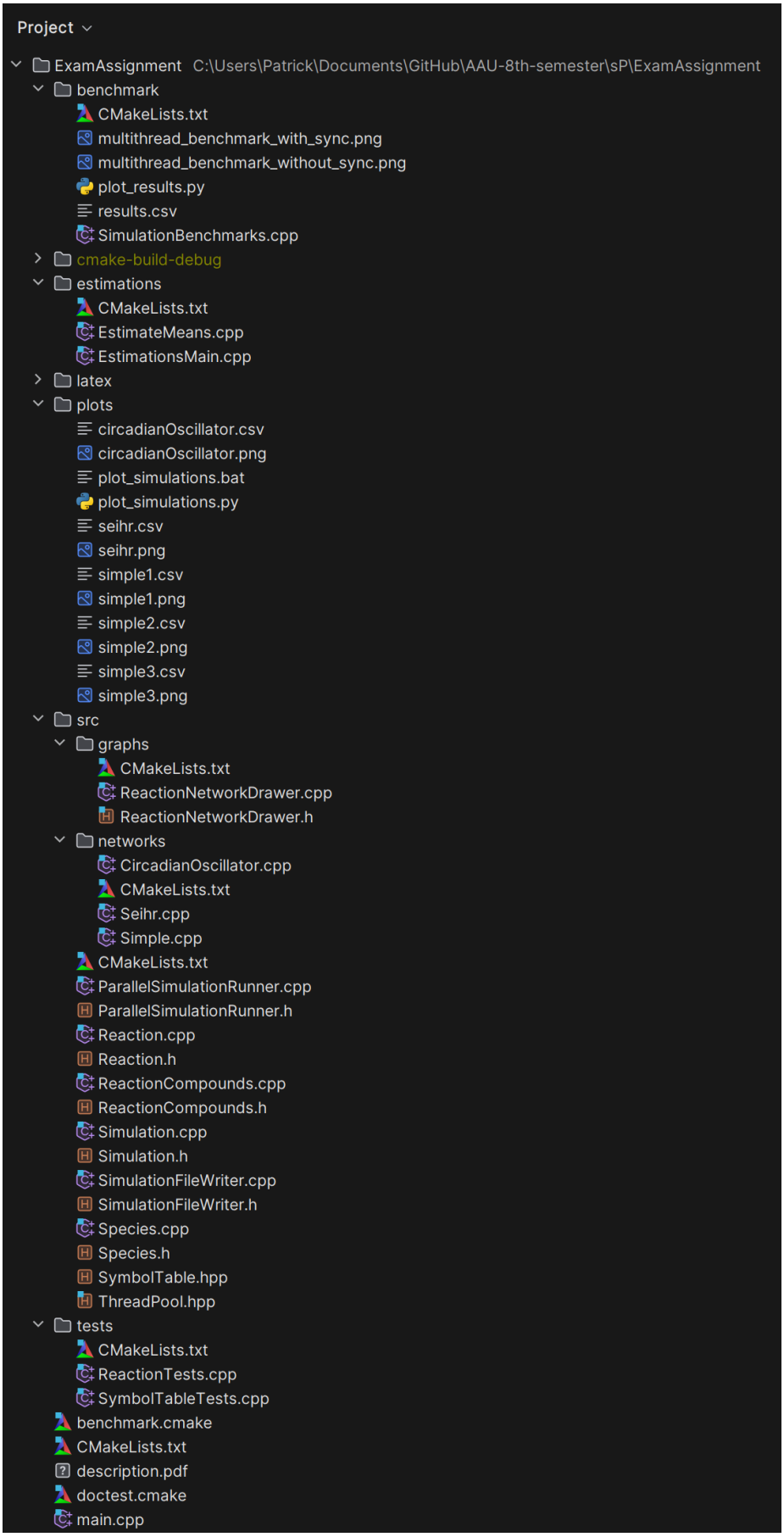

Figure 2: Project structure in CLion.

In addition, Figure 2 shows the project structure in CLion.

1.1 CMakeLists

The following listings show the CMakeLists.txt files used to build the project.

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.24)
project(StochSimLib)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
set(BENCHMARK_DOWNLOAD_DEPENDENCIES ON)

include(doctest.cmake)
include(benchmark.cmake)

enable_testing()
include_directories(src)

# Graphviz paths
set(GRAPHVIZ_INCLUDE_DIR "C:/Program Files/Graphviz/include")
set(GRAPHVIZ_LIBRARY_DIR "C:/Program Files/Graphviz/lib")

# Add Graphviz directories
include_directories(${GRAPHVIZ_INCLUDE_DIR})
link_directories(${GRAPHVIZ_LIBRARY_DIR})

# Add src subdirectory
add_subdirectory(src)

# Link StochSimLib to main executable
add_executable(StochSimLibMain main.cpp)

# Link Graphviz and StochSimLib and StochSimLibGraphing to main executable
target_link_libraries(StochSimLibMain PRIVATE StochSimLib StochSimLibGraphing cgraph gvc)

add_subdirectory(tests)
add_subdirectory(benchmark)
add_subdirectory(estimations)
```

Listing 2: src/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.24)

set(CMAKE_CXX_STANDARD 20)

# source files without the 'src/' prefix
set(SOURCE_FILES
        Simulation.cpp
        Species.cpp
        Reaction.cpp
        ReactionCompounds.cpp
        ParallelSimulationRunner.cpp
        SimulationFileWriter.cpp
        ThreadPool.hpp)

add_library(StochSimLib ${SOURCE_FILES})

add_subdirectory(networks)
add_subdirectory(graphs)
```

Listing 3: src/networks/CMakeLists.txt

```
set(EXAMPLE_FILES
        CircadianOscillator.cpp
        Seihr.cpp
        Simple.cpp
)

add_library(StochSimLibExamples ${EXAMPLE_FILES})
```

Listing 4: src/graphs/CMakeLists.txt

```
set(GRAPH_FILES
        ReactionNetworkDrawer.h ReactionNetworkDrawer.cpp
)

add_library(StochSimLibGraphing ${GRAPH_FILES})
```

Listing 5: estimations/CMakeLists.txt

```
set(ESTIMATION_FILES
        EstimationsMain.cpp
        EstimateMeans.cpp
)

foreach(ESTIMATION_FILE ${ESTIMATION_FILES})
    get_filename_component(FILE_NAME ${ESTIMATION_FILE} NAME_WE)
    add_executable(${FILE_NAME} ${ESTIMATION_FILE})
    target_link_libraries(${FILE_NAME} PRIVATE StochSimLib)
endforeach()
```

Listing 6: tests/CMakeLists.txt

```
set(TEST_FILES
        SymbolTableTests.cpp
        ReactionTests.cpp
)

foreach(TEST_FILE ${TEST_FILES})
    get_filename_component(TEST_NAME ${TEST_FILE} NAME_WE)
    add_executable(${TEST_NAME} ${TEST_FILE})
    target_compile_definitions(${TEST_NAME} PRIVATE DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN)
    target_link_libraries(${TEST_NAME} PRIVATE doctest::doctest StochSimLib)  # Linking to
↪StochSimLib
    add_test(NAME ${TEST_NAME} COMMAND ${TEST_NAME})
endforeach()
```

Listing 7: benchmark/CMakeLists.txt

```
set(BENCHMARK_FILES
        SimulationBenchmarks.cpp
)

foreach(BENCHMARK_FILE ${BENCHMARK_FILES})
    get_filename_component(BENCHMARK_NAME ${BENCHMARK_FILE} NAME_WE)
    add_executable(${BENCHMARK_NAME} ${BENCHMARK_FILE})
    target_link_libraries(${BENCHMARK_NAME} PRIVATE StochSimLib benchmark::benchmark)
endforeach()
```

1.2 Source code

The following listings show the source code for the project. While I used the `Graphviz` C API for the graphs, I used `matplotlib` in Python for the plots, as external tools were allowed.

Listing 8: src/Species.h

```h
#ifndef STOCHSIMLIB_SPECIES_H
#define STOCHSIMLIB_SPECIES_H

#include <string>

namespace StochSimLib {

    class Species {
        std::string m_name;
        size_t m_quantity;

    public:
        Species(std::string name, size_t quantity);

        [[nodiscard]] std::string name() const;

        [[nodiscard]] size_t quantity() const;

        void increaseQuantity(size_t amount);

        void decreaseQuantity(size_t amount);
    };

}

#endif
```

Listing 9: src/Species.cpp

```cpp
#include <stdexcept>
#include "Species.h"

namespace StochSimLib {

    Species::Species(std::string name, size_t quantity) : m_name(std::move(name)), m_quantity(quantity) {}

    std::string Species::name() const {
        return m_name;
    }

    size_t Species::quantity() const {
        return m_quantity;
    }

    void Species::increaseQuantity(size_t amount) {
        m_quantity += amount;
    }

    void Species::decreaseQuantity(size_t amount) {
        if (amount > m_quantity) {
            throw std::runtime_error("Cannot decrease quantity of " + m_name + " by " + std::to_string(amount) + " as it only has " + std::to_string(m_quantity) + " left.");
        }

        m_quantity -= amount;
    }

}
```

Listing 10: src/Reaction.h

```h
#ifndef STOCHSIMLIB_REACTION_H
#define STOCHSIMLIB_REACTION_H

#include <sstream>
#include <string>
#include <random>
#include "ReactionCompounds.h"

namespace StochSimLib {

    class Reaction {
        ReactionCompounds m_reactants;
        ReactionCompounds m_products;

    public:
        double rate = 0;
        double delay = 0;

        [[nodiscard]] ReactionCompounds &reactants();

        [[nodiscard]] ReactionCompounds &products();

        /** Returns the reaction as a human readable string.
         * This fulfills part of requirement 2: "Pretty-print the reaction network in a human readable format". */
        [[nodiscard]] std::string name() const;

        void addReactant(const std::shared_ptr<Species> &reactant);

        void addProduct(const std::shared_ptr<Species> &product);

        /** Computes the delay of the reaction using equation 1 from the PDF.
         * This fulfills part of requirement 4:
         * "Implement the stochastic simulation of the system using the reaction rules." */
        void computeDelay(std::mt19937 &generator);

        friend std::ostream &operator<<(std::ostream &os, const Reaction &reaction);
    };

    /* Models a reaction where multiple reactants (compounds) form a single product (species). E.g., A + B -> C.
     * This fulfills part of requirement 1:
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
    Reaction operator>>=(ReactionCompounds &&compounds, const std::shared_ptr<Species> &species);

    /* Models a reaction where a single reactant (species) forms multiple products (compounds). E.g., A -> B + C.
     * This fulfills part of requirement 1:
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
    Reaction operator>>=(const std::shared_ptr<Species> &species, ReactionCompounds &&compounds);

    /* Models a reaction where multiple reactants (compounds) form multiple products (compounds). E.g., A + B -> C + D.
     * This fulfills part of requirement 1:
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
    Reaction operator>>=(const ReactionCompounds &compoundsL, const ReactionCompounds &compoundsR);

    /* Models a unary reaction where 's1' transforms into 's2'. E.g., A -> B.
     * This fulfills part of requirement 1:
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
```

```
    Reaction operator>>=(const std::shared_ptr<Species> &speciesL, const
↪std::shared_ptr<Species> &speciesR);

}

#endif
```

Listing 11: src/Reaction.cpp

```
#include "Reaction.h"
#include <random>

namespace StochSimLib {

    ReactionCompounds &Reaction::reactants() {
        return m_reactants;
    }

    ReactionCompounds &Reaction::products() {
        return m_products;
    }

    std::string Reaction::name() const {
        std::stringstream ss;
        ss << m_reactants << " -> " << m_products << " (rate = " << rate << ")";

        return ss.str();
    }

    void Reaction::addReactant(const std::shared_ptr<Species> &reactant) {
        m_reactants.addSpecies(reactant);
    }

    void Reaction::addProduct(const std::shared_ptr<Species> &product) {
        m_products.addSpecies(product);
    }

    void Reaction::computeDelay(std::mt19937 &generator) {
        double lambdaK = rate;

        for (const auto &species : reactants()) {
            lambdaK *= static_cast<double>(species->quantity());
        }

        std::exponential_distribution<double> distribution(lambdaK);
        delay = distribution(generator);
    }

    std::ostream &operator<<(std::ostream &os, const Reaction &reaction) {
        return os << reaction.name();
    }

    Reaction operator>>=(ReactionCompounds &&compounds, const std::shared_ptr<Species> &species) {
        Reaction reaction;

        for (const auto &s: compounds.species()) {
            reaction.addReactant(s);
        }

        reaction.addProduct(species);

        return std::move(reaction);
```

```
    }

    Reaction operator>>=(const std::shared_ptr<Species> &species, ReactionCompounds &&compounds) {
        Reaction reaction;
        reaction.addReactant(species);

        for (const auto &s: compounds.species()) {
            reaction.addProduct(s);
        }

        return std::move(reaction);
    }

    Reaction operator>>=(const ReactionCompounds &compoundsL, const ReactionCompounds &compoundsR) {
        Reaction reaction;

        for (const auto &s: compoundsL.species()) {
            reaction.addReactant(s);
        }

        for (const auto &s: compoundsR.species()) {
            reaction.addProduct(s);
        }

        return reaction;
    }

    Reaction operator>>=(const std::shared_ptr<Species> &speciesL, const std::shared_ptr<Species> &speciesR) {
        Reaction reaction;
        reaction.addReactant(speciesL);
        reaction.addProduct(speciesR);

        return reaction;
    }

}
```

Listing 12: src/ReactionCompounds.h

```
#ifndef STOCHSIMLIB_REACTIONCOMPOUNDS_H
#define STOCHSIMLIB_REACTIONCOMPOUNDS_H

#include <iostream>
#include <vector>
#include <memory>
#include "Species.h"

namespace StochSimLib {

    class ReactionCompounds {
        std::vector<std::shared_ptr<Species>> m_species;

    public:
        [[nodiscard]] const std::vector<std::shared_ptr<Species>> &species() const;

        void addSpecies(const std::shared_ptr<Species> &species);

        [[nodiscard]] std::vector<std::shared_ptr<Species>>::const_iterator begin() const;

        [[nodiscard]] std::vector<std::shared_ptr<Species>>::const_iterator end() const;
```

```

        std::vector<std::shared_ptr<Species>>::iterator begin();

        std::vector<std::shared_ptr<Species>>::iterator end();

        /** Operator overload for pretty-printing a ReactionCompounds object to an output stream.
         * Fulfills part of requirement 2: "Pretty-print the reaction network in human readable format". */
        friend std::ostream &operator<<(std::ostream &os, const ReactionCompounds &compounds);
    };

    /** Operator overload for creating a reaction rule between two species (resulting in a ReactionCompounds object).
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
    ReactionCompounds operator+(const std::shared_ptr<Species> &speciesL, const std::shared_ptr<Species> &speciesR);

    /** Operator overload for creating a reaction rule between a ReactionCompounds object and a species
     * (adding the species to the ReactionCompounds object). Fulfills part of requirement 1:
     * "Use operator overloading to support the reaction rule typesetting directly in C++ code." */
    ReactionCompounds operator+(ReactionCompounds &&compounds, const std::shared_ptr<Species> &species);
}

#endif
```

Listing 13: src/ReactionCompounds.cpp

```
#include "ReactionCompounds.h"

namespace StochSimLib {

    const std::vector<std::shared_ptr<Species>> &ReactionCompounds::species() const {
        return m_species;
    }

    void ReactionCompounds::addSpecies(const std::shared_ptr<Species> &species) {
        m_species.emplace_back(species);
    }

    std::vector<std::shared_ptr<Species>>::const_iterator ReactionCompounds::begin() const { return m_species.begin(); }

    std::vector<std::shared_ptr<Species>>::const_iterator ReactionCompounds::end() const { return m_species.end(); }

    std::vector<std::shared_ptr<Species>>::iterator ReactionCompounds::begin() { return m_species.begin(); }

    std::vector<std::shared_ptr<Species>>::iterator ReactionCompounds::end() { return m_species.end(); }

    std::ostream &operator<<(std::ostream &os, const ReactionCompounds &compounds) {
        for (size_t i = 0; i < compounds.m_species.size(); ++i) {
            os << compounds.m_species[i]->name();

            if (i < compounds.m_species.size() - 1) {
                os << " + ";
            }
        }

```

```
30          return os;
31      }
32
33      ReactionCompounds operator+(const std::shared_ptr<Species> &speciesL, const
    ↪std::shared_ptr<Species> &speciesR) {
34          ReactionCompounds compounds;
35
36          compounds.addSpecies(speciesL);
37          compounds.addSpecies(speciesR);
38
39          return compounds;
40      }
41
42      ReactionCompounds operator+(ReactionCompounds &&compounds, const std::shared_ptr<Species>
    ↪&species) {
43          compounds.addSpecies(species);
44
45          return compounds;
46      }
47
48  }
```

Listing 14: src/Simulation.h

```
1   #ifndef STOCHSIMLIB_SIMULATION_H
2   #define STOCHSIMLIB_SIMULATION_H
3
4   #include <iostream>
5   #include <fstream>
6   #include <random>
7   #include <functional>
8   #include "Species.h"
9   #include "SymbolTable.hpp"
10  #include "Reaction.h"
11
12  namespace StochSimLib {
13
14      class Simulation {
15          std::shared_ptr<Species> env = std::make_shared<Species>("env", 0);
16          SymbolTable <Species> m_species;
17          SymbolTable <Reaction> m_reactions;
18
19          double m_time;
20
21          /** Returns true if the reaction can proceed (all reactants have a sufficient quantity). */
22          static bool canReact(Reaction& reaction);
23
24          /** Performs the reaction by decreasing the quantity of reactants and increasing the
    ↪quantity of products. */
25          static void react(Reaction& reaction);
26
27          /** Returns the reaction with the minimum delay. */
28          Reaction& getReactionWithMinDelay();
29
30      public:
31          std::shared_ptr<Species> environment() const;
32
33          SymbolTable <Species> species() const;
34
35          SymbolTable <Reaction> reactions() const;
36
37          /** The current time of the simulation. */
```

```
        double time() const;

        /** Creates a new species and adds it to the simulation. */
        std::shared_ptr<Species> addSpecies(const std::string &name, size_t quantity);

        /** Creates a new reaction and adds it to the simulation. */
        void addReaction(Reaction &&reaction, double rate);

        /** Simulates the reactions until the given end time, invoking the callback function after each iteration with
         * the current simulation state.
         * This fulfills part of requirement 4:
         * "Implement the stochastic simulation of the system using the reaction rules."
         * The callback function is used to fulfill part of requirement 7:
         * "Implement a generic support for the state monitor in the stochastic simulation algorithm." */
        void simulate(double endTime, std::optional<std::function<void(const Simulation&)>> callback = std::nullopt);

        /** Pretty-prints the reactions in the simulation. This fulfills part of requirement 2:
         * "Pretty-print the reaction network in a human readable format". */
        friend std::ostream &operator<<(std::ostream &os, const Simulation &simulation);
    };

}

#endif
```

Listing 15: src/Simulation.cpp

```
#include "Simulation.h"

namespace StochSimLib {

    bool Simulation::canReact(Reaction &reaction) {
        return std::all_of(
                reaction.reactants().begin(),
                reaction.reactants().end(),
                [](const auto &reactant) {
                    return reactant->quantity() > 0;
                }
        );
    }

    void Simulation::react(Reaction &reaction) {
        for (auto &reactant : reaction.reactants()) {
            reactant->decreaseQuantity(1);
        }

        for (auto &product : reaction.products()) {
            product->increaseQuantity(1);
        }
    }

    Reaction &Simulation::getReactionWithMinDelay() {
        auto reactionMinDelayIter = std::min_element(m_reactions.begin(), m_reactions.end(),
              [](const auto &a, const auto &b) {
                return a.second->delay < b.second->delay;
            }
        );

        return *reactionMinDelayIter->second;
```

```
33      }
34
35      std::shared_ptr<Species> Simulation::environment() const {
36          return env;
37      }
38
39      SymbolTable<Species> Simulation::species() const {
40          return m_species;
41      }
42
43      SymbolTable<Reaction> Simulation::reactions() const {
44          return m_reactions;
45      }
46
47      double Simulation::time() const {
48          return m_time;
49      }
50
51      std::shared_ptr<Species> Simulation::addSpecies(const std::string &name, size_t quantity) {
52          std::shared_ptr<Species> species = std::make_shared<Species>(name, quantity);
53
54          m_species.add(species->name(), species);
55
56          return species;
57      }
58
59      void Simulation::addReaction(Reaction &&reaction, double rate) {
60          reaction.rate = rate;
61          m_reactions.add(reaction.name(), std::make_shared<Reaction>(reaction));
62      }
63
64      void Simulation::simulate(double endTime, std::optional<std::function<void(const Simulation
    ↪&)>> callback) {
65          std::random_device rd;
66          std::mt19937 generator(rd());
67
68          while (m_time <= endTime) {
69              for (auto &[_, reaction] : m_reactions) {
70                  reaction->computeDelay(generator);
71              }
72
73              Reaction& reactionMinDelay = getReactionWithMinDelay();
74              m_time += reactionMinDelay.delay;
75
76              if (canReact(reactionMinDelay)) {
77                  react(reactionMinDelay);
78              }
79
80              if (callback) {
81                  (*callback)(*this);
82              }
83          }
84      }
85
86      std::ostream &operator<<(std::ostream &os, const Simulation &simulation) {
87          for (const auto &[_, reaction]: simulation.m_reactions) {
88              os << reaction;
89              os << " (rate = " << reaction->rate << ")";
90              os << std::endl;
91          }
92
```

```
93          return os;
94      }
95
96  }
```

Listing 16: src/ParallelSimulationRunner.h

```
1   #ifndef STOCHSIMLIB_PARALLELSIMULATIONRUNNER_H
2   #define STOCHSIMLIB_PARALLELSIMULATIONRUNNER_H
3
4   #include <thread>
5   #include <functional>
6   #include "Simulation.h"
7
8   namespace StochSimLib {
9
10      class ParallelSimulationRunner {
11          std::vector<Simulation> m_simulations;
12
13          /** Callback invoked after each simulation iteration with the index of the simulation
    ↪and the simulation state. */
14          std::optional<std::function<void(size_t, const Simulation&)>> m_simulationCallback;
15
16      public:
17          /** Sets the callback invoked after each simulation iteration with the index of the
    ↪simulation and the simulation state. */
18          void setSimulationCallback(const std::function<void(size_t, const Simulation&)>
    ↪&simulationCallback);
19
20          /** Runs the given number of simulations in parallel on the given number of threads
    ↪where each simulation runs
21           * until the given end time. The given simulation factory is used to create each simulation.
22           * This fulfills part of requirement 8: "Implement support for multiple computer cores
    ↪by parallelizing the
23           * computation of several simulations at the same time. */
24          void runSimulationsInParallel(size_t count, double endTime, const
    ↪std::function<Simulation()> &simulationFactory, size_t numThreads = 0);
25
26          /** Returns the simulations. */
27          [[nodiscard]] const std::vector<Simulation>& getSimulations() const;
28      };
29
30  }
31
32  #endif
```

Listing 17: src/ParallelSimulationRunner.cpp

```
1   #include <queue>
2   #include "ParallelSimulationRunner.h"
3   #include "ThreadPool.hpp"
4
5   namespace StochSimLib {
6
7       void ParallelSimulationRunner::setSimulationCallback(const std::function<void(size_t, const
    ↪Simulation &)> &simulationCallback) {
8           m_simulationCallback = simulationCallback;
9       }
10
11      void ParallelSimulationRunner::runSimulationsInParallel(size_t count,
12                                                              double endTime,
```

```
                                                        const std::function<Simulation()>   ↙
↪&simulationFactory,
                                                        size_t numThreads) {
        m_simulations.resize(count);

        // Helper lambda to run a simulation
        auto runSimulation = [this, endTime, &simulationFactory](size_t index) {
            m_simulations[index] = simulationFactory();

            if (m_simulationCallback) {
                m_simulations[index].simulate(endTime, [this, index](const Simulation   ↙
↪&simulation) {
                    (*m_simulationCallback)(index, simulation);
                });
            } else {
                m_simulations[index].simulate(endTime);
            }
        };

        if (numThreads <= 0) {
            // Run on all available threads
            std::vector threads = std::vector<std::thread>(count);

            for (size_t i = 0; i < count; ++i) {
                threads[i] = std::thread(runSimulation, i);
            }

            for (std::thread &thread : threads) {
                thread.join();
            }
        } else {
            // Run on specified number of threads
            ThreadPool threadPool{numThreads};

            for (size_t i = 0; i < count; ++i) {
                threadPool.addTask([runSimulation, i]() {
                    runSimulation(i);
                });
            }
        }

        std::cout << "Done." << std::endl;

    }

    const std::vector<Simulation> &ParallelSimulationRunner::getSimulations() const {
        return m_simulations;
    }

}
```

Listing 18: src/SimulationFileWriter.h

```
#ifndef STOCHSIMLIB_SIMULATIONFILEWRITER_H
#define STOCHSIMLIB_SIMULATIONFILEWRITER_H

#include <memory>
#include "Simulation.h"

namespace StochSimLib {

    class SimulationFileWriter {
```

```
        std::unique_ptr<std::ofstream> m_outputFile;

    public:
        SimulationFileWriter(const Simulation& simulation, const std::string& fileName);

        ~SimulationFileWriter();

        void logState(const Simulation& simulation);

    };
}

#endif
```

Listing 19: src/SimulationFileWriter.cpp

```
#include "SimulationFileWriter.h"

namespace StochSimLib {

    SimulationFileWriter::SimulationFileWriter(const Simulation &simulation, const std::string &fileName) {
        m_outputFile = std::make_unique<std::ofstream>(fileName);

        for (const auto &species : simulation.species()) {
            *m_outputFile << species.second->name() << ",";
        }

        *m_outputFile << "Time\n";
    }

    SimulationFileWriter::~SimulationFileWriter() {
        m_outputFile->close();
    }

    void SimulationFileWriter::logState(const Simulation &simulation) {
        for (const auto &species : simulation.species()) {
            *m_outputFile << species.second->quantity() << ",";
        }

        *m_outputFile << simulation.time() << "\n";
    }

}
```

Listing 20: src/SymbolTable.hpp

```
#include <optional>
#include <unordered_map>
#include <string>
#include <memory>

namespace StochSimLib {

    /** Symbol table for storing objects of type T.
     *  This fulfills requirement 3:
     *  "Implement a generic symbol table to store and lookup objects of user-defined types (e.g. information about
     *  agents and reactions). Support failure cases when the table does not contain a looked up value." */
    template<typename T>
    class SymbolTable {
```

```
        std::unordered_map<std::string, std::shared_ptr<T>> m_table;

        void checkDuplicate(const std::string &symbol) {
            if (m_table.find(symbol) != m_table.end()) {
                throw std::runtime_error("Symbol \"" + symbol + "\" already exists in symbol table!");
            }
        }

    public:
        void add(const std::string &symbol, std::shared_ptr<T> object) {
            checkDuplicate(symbol);
            m_table.emplace(symbol, object);
        }

        void add(const std::string &symbol, T &&object) {
            checkDuplicate(symbol);
            m_table.emplace(symbol, std::make_shared<T>(object));
        }

        void remove(const std::string &symbol) {
            m_table.erase(symbol);
        }

        std::optional<std::shared_ptr<T>> tryGet(const std::string &symbol) {
            auto it = m_table.find(symbol);

            if (it != m_table.end()) {
                return it->second;
            }

            return std::nullopt;
        }

        // Convenience operator for subscripting the symbol table
        std::shared_ptr<T> operator[](const std::string &symbol) {
            if (tryGet(symbol)) {
                return *tryGet(symbol);
            } else {
                throw std::runtime_error("Symbol\"" + symbol + "\"not found in symbol table!");
            }
        }

        [[nodiscard]] size_t size() const {
            return m_table.size();
        }

        // Iterator support
        auto begin() { return m_table.begin(); }
        auto end() { return m_table.end(); }
        auto begin() const { return m_table.begin(); }
        auto end() const { return m_table.end(); }
    };

}
```

Listing 21: src/ThreadPool.hpp

```
#include <thread>
#include <condition_variable>
#include <future>
#include <functional>
```

```cpp
#include <queue>

namespace StochSimLib {

    /** A class for managing a pool of threads executing tasks. */
    class ThreadPool {
        std::queue<std::function<void()>> m_tasks;  // Queue of tasks to be executed by the threads.
        std::vector<std::jthread> m_threads;        // Vector of worker threads.
        std::mutex m_mutex;                         // Mutex for synchronizing access to tasks.
        std::condition_variable m_cv;               // Condition variable to wake up threads when tasks are available.

        bool m_stop = false;                        // Boolean flag indicating whether to stop the thread pool.

    public:
        /**
         * @brief Constructs a ThreadPool with a specified number of worker threads.
         * @param num_threads Number of worker threads to create.
         */
        explicit ThreadPool(size_t num_threads) {
            // Create and start worker threads.
            for (int i = 0; i < num_threads; ++i) {
                m_threads.emplace_back([this] {
                    while (true) {
                        std::function<void()> task;

                        {
                            std::unique_lock lock(m_mutex);

                            // Wait until a task is available or stop is signaled.
                            m_cv.wait(lock, [this] {
                                return m_stop || !m_tasks.empty();
                            });

                            // If stop is signaled and no tasks are available, exit the thread.
                            if (m_stop && m_tasks.empty()) {
                                return;
                            }

                            // Get the next task from the queue.
                            task = std::move(m_tasks.front());
                            m_tasks.pop();
                        }

                        // Execute the task.
                        task();
                    }
                });
            }
        }

        /** Destructor. Stops all worker threads and waits for them to finish. */
        ~ThreadPool() {
            {
                std::scoped_lock lock(m_mutex);
                m_stop = true;  // Signal all threads to stop.
            }

            m_cv.notify_all();  // Wake up all threads.

```

```
            // Wait for all threads to finish.
            for (auto &thread: m_threads) {
                thread.join();
            }
        }

        /** Adds a new task to be executed by the worker threads.
         * @param task Task to be added. This should be a callable object. */
        void addTask(std::function<void()> task) {
            {
                std::scoped_lock lock(m_mutex);
                m_tasks.push(std::move(task));  // Add the task to the queue.
            }

            m_cv.notify_one();  // Wake up one thread to handle the new task.
        }
    };

}
```

Listing 22: src/networks/Simple.cpp

```
#ifndef STOCHSIMLIB_SIMPLE_CPP
#define STOCHSIMLIB_SIMPLE_CPP

#include "Simulation.h"
#include "Reaction.h"

namespace StochSimLib::Networks {

    /** Creates a simulation of a simple reaction network based on the model from the PDF. */
    Simulation simple(size_t countA = 100, size_t countB = 0, size_t countC = 1) {
        const double lambda = 0.001;

        Simulation simulation{};

        auto A = simulation.addSpecies("A", countA);
        auto B = simulation.addSpecies("B", countB);
        auto C = simulation.addSpecies("C", countC);

        simulation.addReaction(A + C >>= B + C, lambda);

        return simulation;
    }

}

#endif
```

Listing 23: src/networks/Seihr.cpp

```
#ifndef STOCHSIMLIB_SEIHR
#define STOCHSIMLIB_SEIHR

#include "Simulation.h"
#include "Reaction.h"

namespace StochSimLib::Networks {

    /** Creates a simulation of the SEIHR model based on the model from the PDF. */
    Simulation seihr(uint32_t N = 10000) {
        Simulation simulation{};
```

```
12
13          const auto eps = 0.0009; // initial fraction of infectious
14          const auto I0 = size_t(std::round(eps * N)); // initial infectious
15          const auto E0 = size_t(std::round(eps * N * 15)); // initial exposed
16          const auto S0 = N - I0 - E0; // initial susceptible
17          const auto R0 = 2.4; // basic reproductive number (initial, without lockdown etc)
18          const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
19          const auto gamma = 1.0 / 3.1; // recovery rate (I -> R) ~3.1 days
20          const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
21          const auto P_H = 0.9e-3; // probability of hospitalization
22          const auto kappa = gamma * P_H * (1.0 - P_H); // hospitalization rate (I -> H)
23          const auto tau = 1.0 / 10.12; // recovery/death rate in hospital (H -> R) ~10.12 days
24
25          auto S = simulation.addSpecies("S", S0); // susceptible
26          auto E = simulation.addSpecies("E", E0); // exposed
27          auto I = simulation.addSpecies("I", I0); // infectious
28          auto H = simulation.addSpecies("H", 0); // hospitalized
29          auto R = simulation.addSpecies("R", 0); // removed/immune (recovered + dead)
30
31          simulation.addReaction(S + I >>= E + I, beta / N); // susceptible becomes exposed
    ↪through infectious
32          simulation.addReaction(E >>= I, alpha); // exposed becomes infectious
33          simulation.addReaction(I >>= R, gamma); // infectious becomes removed
34          simulation.addReaction(I >>= H, kappa); // infectious becomes hospitalized
35          simulation.addReaction(H >>= R, tau); // hospitalized becomes removed
36
37          return simulation;
38      }
39
40  }
41
42  #endif
```

Listing 24: src/networks/CircadianOscillator.cpp

```
1   #ifndef STOCHSIMLIB_CIRCADIANOSCILLATOR_CPP
2   #define STOCHSIMLIB_CIRCADIANOSCILLATOR_CPP
3
4   #include "Simulation.h"
5   #include "Reaction.h"
6
7   namespace StochSimLib::Networks {
8
9       /** Creates a simulation of the circadian oscillator based on the model from the PDF. */
10      Simulation circadianOscillator() {
11          auto alphaA = 50.0;
12          auto alpha_A = 500.0;
13          auto alphaR = 0.01;
14          auto alpha_R = 50.0;
15          auto betaA = 50.0;
16          auto betaR = 5.0;
17          auto gammaA = 1.0;
18          auto gammaR = 1.0;
19          auto gammaC = 2.0;
20          auto deltaA = 1.0;
21          auto deltaR = 0.2;
22          auto deltaMA = 10.0;
23          auto deltaMR = 0.5;
24          auto thetaA = 50.0;
25          auto thetaR = 100.0;
26
27          Simulation simulation{};
```

```
28
29          auto env = simulation.environment();
30
31          auto DA = simulation.addSpecies("DA", 1);
32          auto D_A = simulation.addSpecies("D_A", 0);
33          auto DR = simulation.addSpecies("DR", 1);
34          auto D_R = simulation.addSpecies("D_R", 0);
35          auto MA = simulation.addSpecies("MA", 0);
36          auto MR = simulation.addSpecies("MR", 0);
37          auto A = simulation.addSpecies("A", 0);
38          auto R = simulation.addSpecies("R", 0);
39          auto C = simulation.addSpecies("C", 0);
40
41          simulation.addReaction(A + DA >>= D_A, gammaA);
42          simulation.addReaction(D_A >>= DA + A, thetaA);
43          simulation.addReaction(A + DR >>= D_R, gammaR);
44          simulation.addReaction(D_R >>= DR + A, thetaR);
45          simulation.addReaction(D_A >>= MA + D_A, alpha_A);
46          simulation.addReaction(DA >>= MA + DA, alphaA);
47          simulation.addReaction(D_R >>= MR + D_R, alpha_R);
48          simulation.addReaction(DR >>= MR + DR, alphaR);
49          simulation.addReaction(MA >>= MA + A, betaA);
50          simulation.addReaction(MR >>= MR + R, betaR);
51          simulation.addReaction(A + R >>= C, gammaC);
52          simulation.addReaction(C >>= R, deltaA);
53          simulation.addReaction(A >>= env, deltaA);
54          simulation.addReaction(R >>= env, deltaR);
55          simulation.addReaction(MA >>= env, deltaMA);
56          simulation.addReaction(MR >>= env, deltaMR);
57
58          return simulation;
59      }
60
61  }
62
63  #endif
```

Listing 25: src/graphs/ReactionNetworkDrawer.h

```
1   #ifndef STOCHSIMLIB_REACTIONNETWORKDRAWER_H
2   #define STOCHSIMLIB_REACTIONNETWORKDRAWER_H
3
4   #include <sstream>
5   #include <iomanip>
6   #include <string>
7
8   #include <graphviz/gvc.h>
9   #include <graphviz/cgraph.h>
10
11  #include "Simulation.h"
12
13  namespace StochSimLib::Graphs {
14
15      class ReactionNetworkDrawer {
16          /** Truncates a double to 9 decimal places and removes trailing zeros.
17           * Also removes the decimal point if it's the last character. */
18          static std::string truncateDecimal(double num);
19
20          /** Creates a node in the graph with the given name, fill color, and shape. */
21          static void createNode(Agraph_t *g, const std::string &name, const std::string &fillColor,
22                                 const std::string &shape = "ellipse");
23
```

```
24          /** Creates an edge in the graph with the given source and destination.
25           * Ignores the edge if either the source or destination is the environment. */
26          static void createEdge(Agraph_t *g, const std::string &source, const std::string
↪&destination);
27
28      public:
29          /** Fulfills part of requirement 2:
30          * "Pretty print the reaction network in a network graph". */
31          static void draw(const StochSimLib::Simulation &simulation, const std::string &filename
↪= "graph", const std::string &format = "png");
32      };
33
34  }
35
36  #endif
```

Listing 26: src/graphs/ReactionNetworkDrawer.cpp

```
1   #include "ReactionNetworkDrawer.h"
2
3   namespace StochSimLib::Graphs {
4
5       std::string ReactionNetworkDrawer::truncateDecimal(double num) {
6           std::ostringstream out;
7           out << std::setprecision(9) << std::fixed << num;
8           std::string truncated = out.str();
9           truncated.erase(truncated.find_last_not_of('0') + 1, std::string::npos);
10
11          if (truncated.back() == '.') {
12              truncated.pop_back();
13          }
14
15          return truncated;
16      }
17
18      void ReactionNetworkDrawer::createNode(Agraph_t *g, const std::string &name, const
↪std::string &fillColor,
19                                             const std::string &shape) {
20          auto node = agnode(g, (char *) name.c_str(), 1);
21          agsafeset(node, (char *) "style", (char *) "filled", (char *) "");
22          agsafeset(node, (char *) "fillcolor", (char *) fillColor.c_str(), (char *) "");
23          agsafeset(node, (char *) "shape", (char *) shape.c_str(), (char *) "");
24      }
25
26      void ReactionNetworkDrawer::createEdge(Agraph_t *g, const std::string &source, const
↪std::string &destination) {
27          if (source == "env" || destination == "env") {
28              return;
29          }
30
31          agedge(g, agnode(g, (char *) source.c_str(), 1), agnode(g, (char *) destination.c_str(),
↪1), nullptr, 1);
32      }
33
34      void ReactionNetworkDrawer::draw(const Simulation &simulation, const std::string &filename,
↪const std::string &format) {
35          GVC_t *gvc = gvContext();
36          Agraph_t *g = agopen((char*)"g", Agdirected, nullptr);
37
38          // Create species nodes
39          for (const auto &[_, species]: simulation.species()) {
40              if (species->name() != "env") {
```

```
                createNode(g, species->name(), "lightgreen");
            }
        }

        // Draw edges between species based on reactions
        int rateNodeCount = 0;

        for (const auto &[reactionName, reaction]: simulation.reactions()) {
            // Create rate node
            double rate = reaction->rate;

            std::string rateStr = truncateDecimal(rate);
            std::string rateNodeName = "rate" + std::to_string(rateNodeCount++);

            createNode(g, rateNodeName, "lightblue", "box");
            agset(agnode(g, (char *) rateNodeName.c_str(), 1), (char*)"label",
↪(char*)rateStr.c_str());

            // Draw edges between every species in the reaction as well as the rate node
            for (const auto &x: reaction->reactants()) {
                createEdge(g, x->name(), rateNodeName);
            }

            for (const auto &x: reaction->products()) {
                createEdge(g, rateNodeName, x->name());
            }
        }

        // Render the graph and free resources
        gvLayout(gvc, g, "dot");
        gvRenderFilename(gvc, g, (char*)format.c_str(), (char*)(filename + "." + format).c_str());
        gvFreeLayout(gvc, g);
        agclose(g);
        gvFreeContext(gvc);
    }

}
```

Listing 27: estimations/EstimateMeans.cpp

```
#include <vector>
#include "ParallelSimulationRunner.h"
#include "networks/Seihr.cpp"

namespace StochSimLib::Estimations {

    /** Fulfills part of requirement 7:
     * "Use it to estimate the peak of hospitalized agents in Covid-19 example without storing
↪trajectory data". */
    std::pair<size_t, double> estimatePeakAndMean(size_t N = 10000, size_t simulationCount =
↪100, double endTime = 100.0) {
        std::vector<size_t> peakValues(simulationCount, 0);
        size_t globalPeak = 0;
        std::mutex mtx;

        ParallelSimulationRunner runner;
        runner.setSimulationCallback([&](size_t index, const Simulation &simulation) {
            std::shared_ptr<Species> hospitalized = simulation.species()["H"];
            size_t quantity = hospitalized->quantity();

            std::lock_guard<std::mutex> guard(mtx);

```

```
            if (quantity > peakValues[index]) {
                peakValues[index] = quantity;
            }

            if (quantity > globalPeak) {
                globalPeak = quantity;
            }
        });
        runner.runSimulationsInParallel(simulationCount, endTime, [N]() { return
↪Networks::seihr(N); });

        // Calculate mean peak value
        double mean = 0.0;

        for (size_t value: peakValues) {
            mean += static_cast<double>(value);
        }

        mean /= static_cast<double>(simulationCount);

        return {mean, globalPeak};
    }

}
```

Listing 28: EstimationsMain.cpp

```
#include "EstimateMeans.cpp"
#include <iostream>

using namespace StochSimLib::Estimations;

int main() {
    size_t N_DK = 5822763;
    size_t N_NJ = 589755;
    size_t simulationCount = 100;

    auto result_DK = estimatePeakAndMean(N_DK, simulationCount);
    auto result_NJ = estimatePeakAndMean(N_NJ, simulationCount);

    std::cout << "Mean DK hospitalized: " << result_DK.first << std::endl;
    std::cout << "Mean NJ hospitalized: " << result_NJ.first << std::endl;
    std::cout << "Peak DK hospitalized: " << result_DK.second << std::endl;
    std::cout << "Peak NJ hospitalized: " << result_NJ.second << std::endl;

    return 0;
}
```

Listing 29: tests/ReactionTests.cpp

```
#include <doctest/doctest.h>
#include "Reaction.h"
#include "Species.h"

using namespace StochSimLib;

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wparentheses"

/** Tests the Reaction class pretty-printing.
 * Fulfills part of requirement 9:
```

```
 * "Implement unit tests (e.g. test symbol table methods and pretty-printing of reaction
↪rules)". */

TEST_CASE("Test pretty printing") {
    Reaction reaction;
    reaction.addReactant(std::make_shared<Species>("A", 1));
    reaction.addReactant(std::make_shared<Species>("C", 1));
    reaction.addProduct(std::make_shared<Species>("B", 0));
    reaction.addProduct(std::make_shared<Species>("C", 0));

    reaction.rate = 50.0;

    CHECK(reaction.name() == "A + C -> B + C (rate = 50)");
}

#pragma GCC diagnostic pop
```

Listing 30: tests/SymbolTableTests.cpp

```
#include <doctest/doctest.h>
#include "SymbolTable.hpp"

using namespace StochSimLib;

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wparentheses"

/** Tests the SymbolTable class.
 * Fulfills part of requirement 9:
 * "Implement unit tests (e.g. test symbol table methods and pretty-printing of reaction
↪rules)". */

TEST_SUITE("SymbolTable Test Suite") {
    TEST_CASE("SymbolTable - Adding and Retrieving") {
        SymbolTable<int> table;

        table.add("a", 1);
        table.add("b", 2);
        table.add("c", 3);

        CHECK(*table["a"] == 1);
        CHECK(*table["b"] == 2);
        CHECK(*table["c"] == 3);
    }

    TEST_CASE("SymbolTable - Modifying") {
        SymbolTable<int> table;

        table.add("a", 1);
        table.add("b", 2);
        table.add("c", 3);

        *table["a"] = 4;
        *table["b"] = 5;
        *table["c"] = 6;

        CHECK(*table["a"] == 4);
        CHECK(*table["b"] == 5);
        CHECK(*table["c"] == 6);
    }

    TEST_CASE("SymbolTable - Removing") {
```

```
43          SymbolTable<int> table;
44
45          table.add("a", 1);
46          table.add("b", 2);
47          table.add("c", 3);
48
49          table.remove("a");
50          table.remove("b");
51          table.remove("c");
52
53          CHECK(table.tryGet("a") == std::nullopt);
54          CHECK(table.tryGet("b") == std::nullopt);
55          CHECK(table.tryGet("c") == std::nullopt);
56      }
57
58      TEST_CASE("SymbolTable - Size Check") {
59          SymbolTable<int> table;
60
61          table.add("a", 1);
62          table.add("b", 2);
63          table.add("c", 3);
64
65          CHECK(table.size() == 3);
66
67          table.remove("a");
68          table.remove("b");
69          table.remove("c");
70
71          CHECK(table.size() == 0);
72      }
73
74      TEST_CASE("SymbolTable - Duplicate Add") {
75          SymbolTable<int> table;
76
77          table.add("a", 1);
78
79          CHECK_THROWS_AS(table.add("a", 2), std::runtime_error);
80      }
81
82      TEST_CASE("SymbolTable - Nonexistent Remove") {
83          SymbolTable<int> table;
84
85          CHECK_NOTHROW(table.remove("a"));
86      }
87
88      TEST_CASE("SymbolTable - Nonexistent Subscript") {
89          SymbolTable<int> table;
90
91          CHECK_THROWS_AS(table["a"], std::runtime_error);
92      }
93
94      TEST_CASE("SymbolTable - Add and Retrieve Shared Ptr") {
95          SymbolTable<int> table;
96          std::shared_ptr<int> ptr = std::make_shared<int>(1);
97
98          table.add("a", ptr);
99
100         CHECK(*table["a"] == 1);
101         CHECK(table["a"] == ptr);
102     }
103
```

```
    TEST_CASE("SymbolTable - Iteration") {
        SymbolTable<int> table;

        table.add("a", 1);
        table.add("b", 2);
        table.add("c", 3);

        int sum = 0;

        for(auto& [key, value] : table) {
            sum += *value;
        }

        CHECK(sum == 6);
    }
}

#pragma GCC diagnostic pop
```

Listing 31: plots/plot_simulations.py

```
import pandas as pd
import matplotlib.pyplot as plt
import glob

"""
This Python script plots the simulation results from the CSV files in the current directory.
Fulfills requirement 6:
"Display simulation trajectories of how the amounts change. External tools/libraries can be used to visualize".
"""

# Define data transformations in a dictionary
transformations = {
    'seihr.csv': {'column': 'H', 'transform': lambda x: x * 1000, 'note': '* 1000'}
}

# Get all CSV files in the current directory
csv_files = glob.glob('*.csv')

for file in csv_files:
    data = pd.read_csv(file)

    # Strip leading and trailing spaces from column names
    data.columns = data.columns.str.strip()
    data['Time'] = data['Time'].astype(float)

    # Check if there are transformations defined for this file
    if file in transformations:
        # Get the transformation parameters
        column_name = transformations[file]['column']
        transform_func = transformations[file]['transform']
        transform_note = transformations[file]['note']

        # Apply the transformation
        if column_name in data.columns:
            data[column_name] = data[column_name].apply(transform_func)
            # Rename the column to include the transformation note
            data.rename(columns={column_name: f'{column_name} {transform_note}'}, inplace=True)

    # Plot the data
    plt.figure(figsize=(10,6))
```

```
    for column in data.columns:
        if column != 'Time':
            plt.plot(data['Time'], data[column], label=column)

    plt.xlabel('Time')
    plt.ylabel('Quantity')
    plt.legend()

    # Save the figure to a PNG file
    plt.savefig(file.replace('.csv', '.png'), dpi=300)
    plt.close()
```

Listing 32: benchmark/plot_results.py

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
import argparse
import numpy as np

# Python script used to plot the benchmark results.

def plot_data(df, show_sync):
    colors = ['#E91E63', '#4CAF50', '#2196F3', '#FFC107', '#9C27B0', '#FF5722']

    plt.figure(figsize=(10,6))

    # Replace 'OS' in 'Thread count' column with a unique numeric identifier
    df['Thread count'] = df['Thread count'].replace('OS', -1)

    # Convert 'Thread count' column to numeric
    df['Thread count'] = pd.to_numeric(df['Thread count'])

    # Filter data
    single_threaded = df[(df['Thread count'] == 0) & (df['Benchmark'].str.contains('Synchronous'))]
    os_managed = df[(df['Thread count'] == -1) & (df['Benchmark'].str.contains('AndThreads'))]
    multithreaded = df[(df['Thread count'] > 0) & (df['Benchmark'].str.contains('AndThreads'))]

    if show_sync:
        plt.plot(single_threaded['Simulation count'], single_threaded['Time (ms)'], marker='o', linestyle='-', color=colors[0], label='Single-threaded')

    plt.plot(os_managed['Simulation count'], os_managed['Time (ms)'], marker='o', linestyle='-', color=colors[1], label='OS Managed')

    # Loop through the unique thread counts and create a plot for each
    for idx, thread_count in enumerate(multithreaded['Thread count'].unique(), start=2):
        subset = multithreaded[multithreaded['Thread count'] == thread_count]
        plt.plot(subset['Simulation count'], subset['Time (ms)'], marker='o', linestyle='-', color=colors[idx % len(colors)], label=f'Multithreaded ({thread_count} threads)')

    # Configure plot
    plt.xlabel('Simulation Count')
    plt.ylabel('Time (ms)')
    plt.title('Benchmark Results')
    plt.legend()
    plt.grid(True)
    plt.xticks(df['Simulation count'].unique())

```

```
    # Save the plot
    file_name = 'multithread_benchmark_with_sync.png' if show_sync else '↩
        ↪multithread_benchmark_without_sync.png'
    plt.savefig(file_name, dpi=300)
    plt.show()



def main(show_sync):
    df = pd.read_csv('results.csv')
    plot_data(df, show_sync)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Plot benchmark data.')
    parser.add_argument('--show_sync', action='store_true', help='Include the single-threaded data↩
        ↪ in the plot')
    args = parser.parse_args()
    main(args.show_sync)
```

Listing 33: benchmark/SimulationBenchmarks.cpp

```
#include <benchmark/benchmark.h>
#include "ParallelSimulationRunner.h"
#include "networks/CircadianOscillator.cpp"

/** Benchmarks fulfilling requirement 10:
 * "Benchmark and compare the stochastic simulation performance (e.g. the time it takes to   ↙
↪compute 20 simulations
 * a single core, multiple cores, or improved implementation)". */

constexpr double endTime = 24;
constexpr size_t simulationCount = 128;
constexpr size_t numIterations = 100;

StochSimLib::Simulation simulationFactory() {
    return StochSimLib::Networks::circadianOscillator();
}

// Benchmark for varying number of simulations with a single thread
static void BM_Simulations_Synchronous(benchmark::State& state) {
    size_t numSimulations = state.range(0);

    std::vector<StochSimLib::Simulation> simulations(numSimulations);

    for (auto _ : state) {
        for (auto& simulation : simulations) {
            simulation = simulationFactory();
            simulation.simulate(endTime);
        }
    }
}

BENCHMARK(BM_Simulations_Synchronous)->Iterations(numIterations)->RangeMultiplier(2)->Range(1,   ↙
↪simulationCount);

// Benchmark for varying number of threads and simulations together
static void BM_SimulationsAndThreads(benchmark::State& state) {
    StochSimLib::ParallelSimulationRunner runner;
    size_t numSimulations = state.range(0);
    size_t numThreads = state.range(1);

    for (auto _ : state) {
```

```
        runner.runSimulationsInParallel(numSimulations, endTime, simulationFactory, numThreads);
    }
}

// Custom Arguments for the above benchmark
static void customThreadAndSimulationCountArguments(benchmark::internal::Benchmark* b) {
    std::vector<int> threadCounts;

    for (int i = 2; i <= std::thread::hardware_concurrency(); i *= 2) {
        threadCounts.push_back(i);
    }

    // Add the case with 0 threads for the simulation runner (becomes OS choice)
    for (int simulations = 1; simulations <= simulationCount; simulations *= 2) {
        b->Args({simulations, 0});
    }

    for (const auto& threads : threadCounts) {
        for (int simulations = 1; simulations <= simulationCount; simulations *= 2) {
            b->Args({simulations, threads});
        }
    }
}

BENCHMARK(BM_SimulationsAndThreads)->Iterations(numIterations)->Apply(customThreadAndSimulationCountArguments


int main(int argc, char** argv) {
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();
}
```

Listing 34: main.cpp

```
#include "networks/Simple.cpp"
#include "networks/CircadianOscillator.cpp"
#include "networks/Seihr.cpp"
#include "SimulationFileWriter.h"
#include "graphs/ReactionNetworkDrawer.h"

using namespace StochSimLib;
using namespace StochSimLib::Graphs;

/** Demonstrates the application of the library on the three examples from the PDF. Fulfills
↪requirement 5. */
int main() {
    auto example1 = Networks::simple();
    auto example2 = Networks::circadianOscillator();
    auto example3 = Networks::seihr();

    SimulationFileWriter fileWriter1{example1, "simple.csv"};
    SimulationFileWriter fileWriter2{example2, "circadianOscillator.csv"};
    SimulationFileWriter fileWriter3{example3, "seihr.csv"};

    ReactionNetworkDrawer::draw(example1, "simple", "png");
    ReactionNetworkDrawer::draw(example2, "circadianOscillator", "png");
    ReactionNetworkDrawer::draw(example3, "seihr", "png");

    example1.simulate(2000, [&fileWriter1](const Simulation &simulation) {
        fileWriter1.logState(simulation);
    });

```

```
    example2.simulate(72, [&fileWriter2](const Simulation &simulation) {
        fileWriter2.logState(simulation);
    });

    example3.simulate(100, [&fileWriter3](const Simulation &simulation) {
        fileWriter3.logState(simulation);
    });

    return 0;
}
```

2 Figures

2.1 Reaction Networks

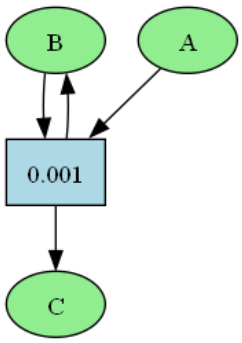

Figure 3: Reaction network for the simple model.

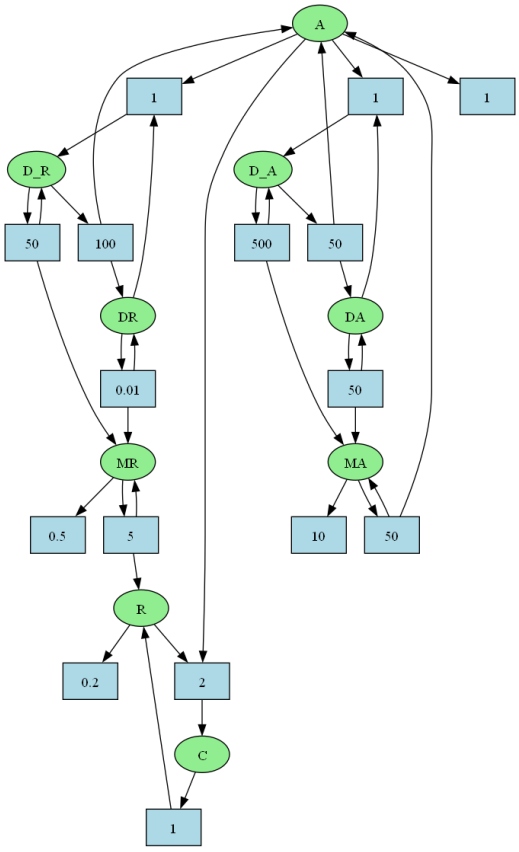

Figure 4: Reaction network for the circadian oscillator model.

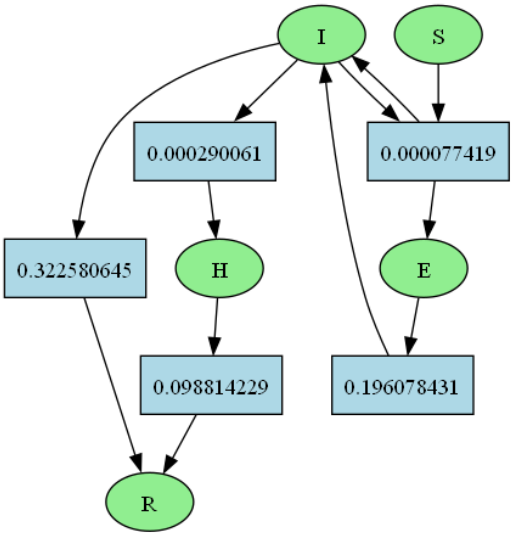

Figure 5: Reaction network for the SEIHR model.

2.2 Plots

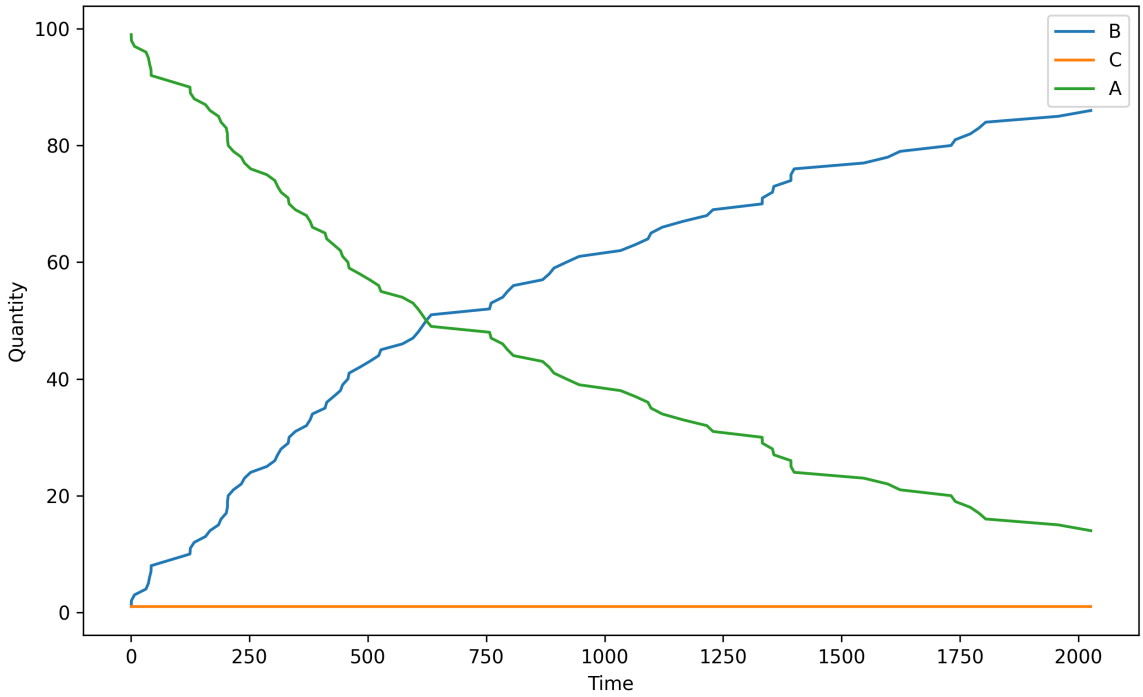

Figure 6: Plot of running the simple model simulation with A(0)=100, B(0)=0, C(0)=1.

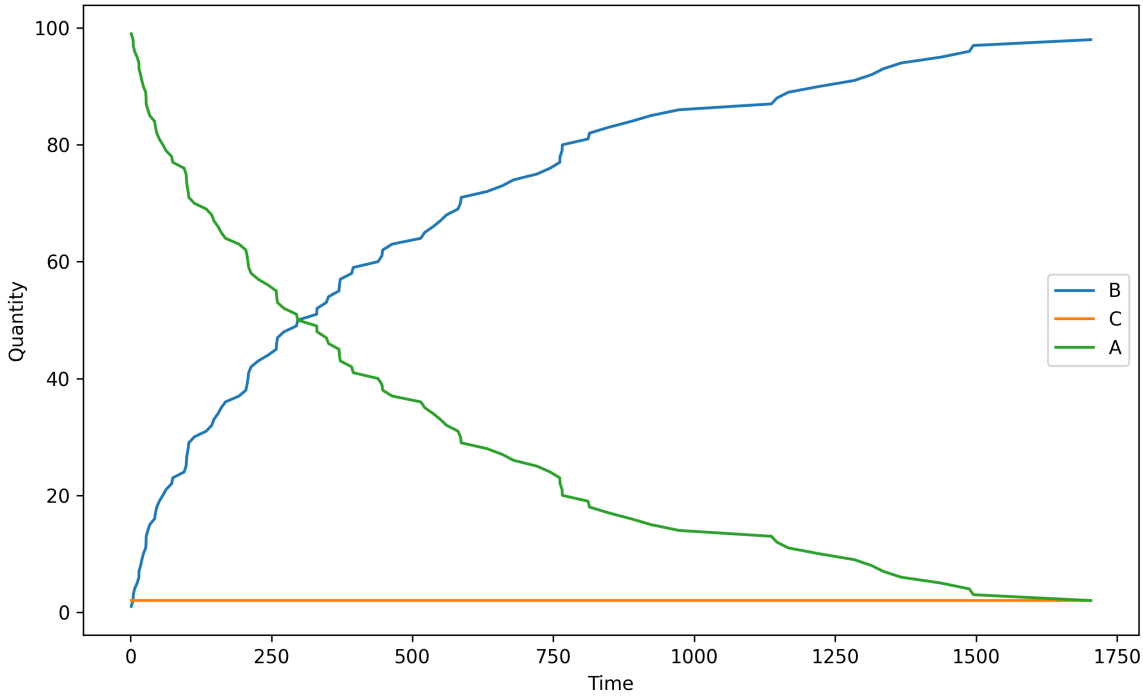

Figure 7: Plot of running the simple model simulation with A(0)=100, B(0)=0, C(0)=2.

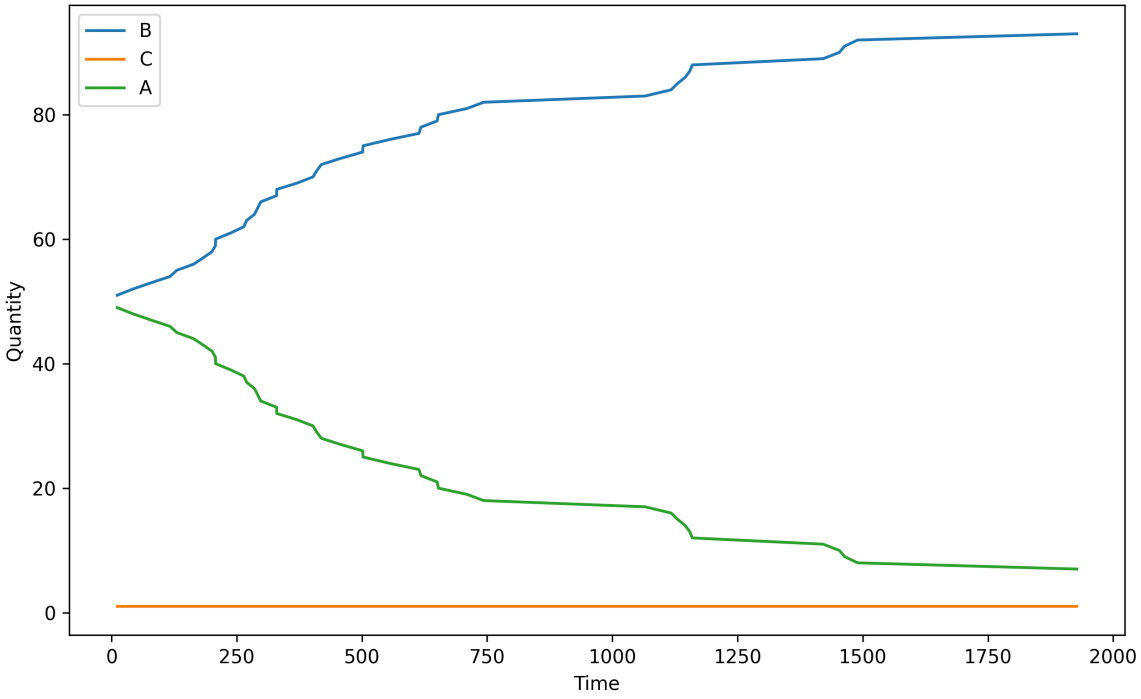

Figure 8: Plot of running the simple model simulation with A(0)=50, B(0)=50, C(0)=1.

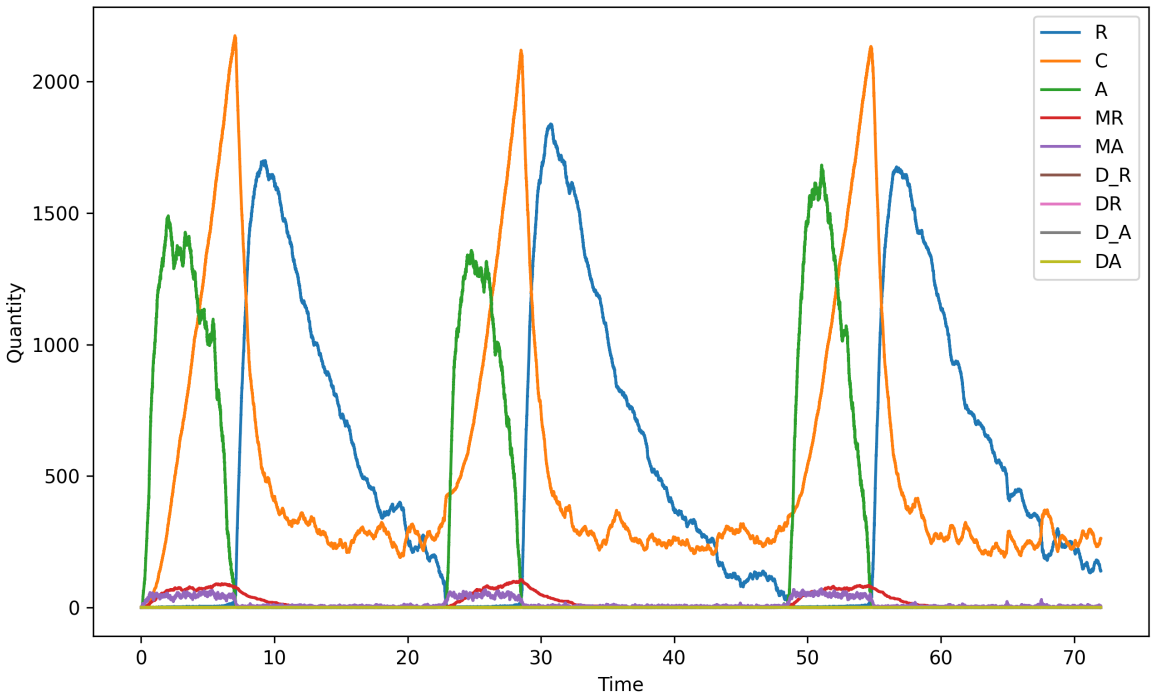

Figure 9: Plot of running the circadian oscillator model simulation.

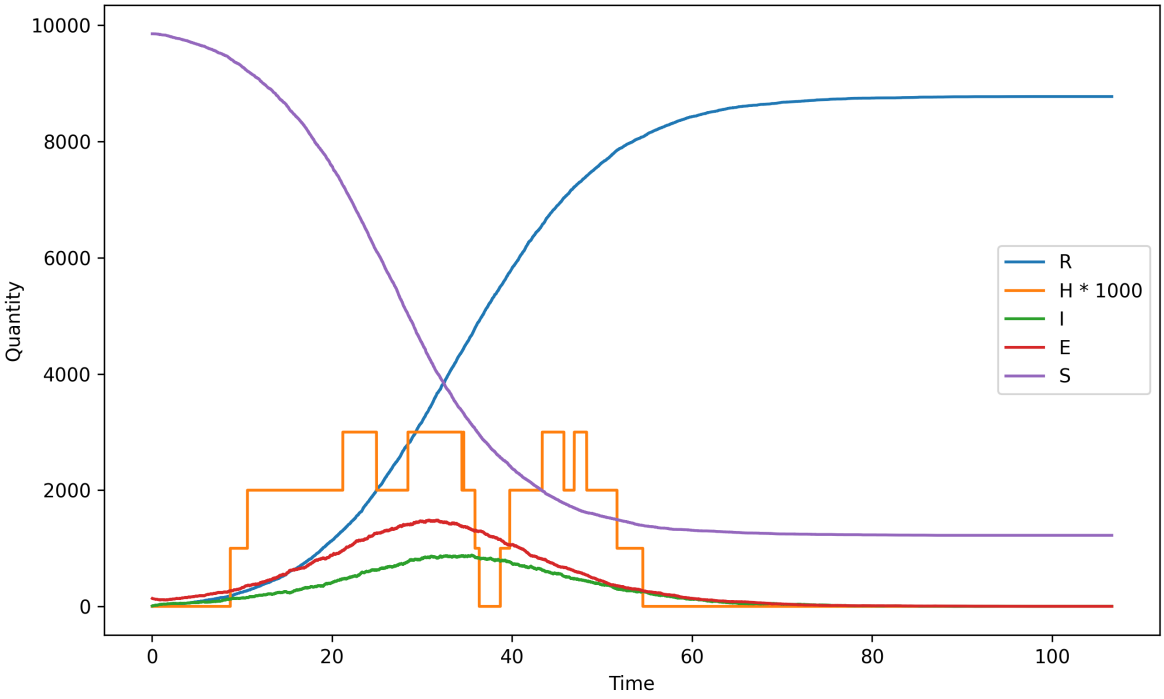

Figure 10: Plot of running the SEIHR model simulation.

2.3 Benchmarks

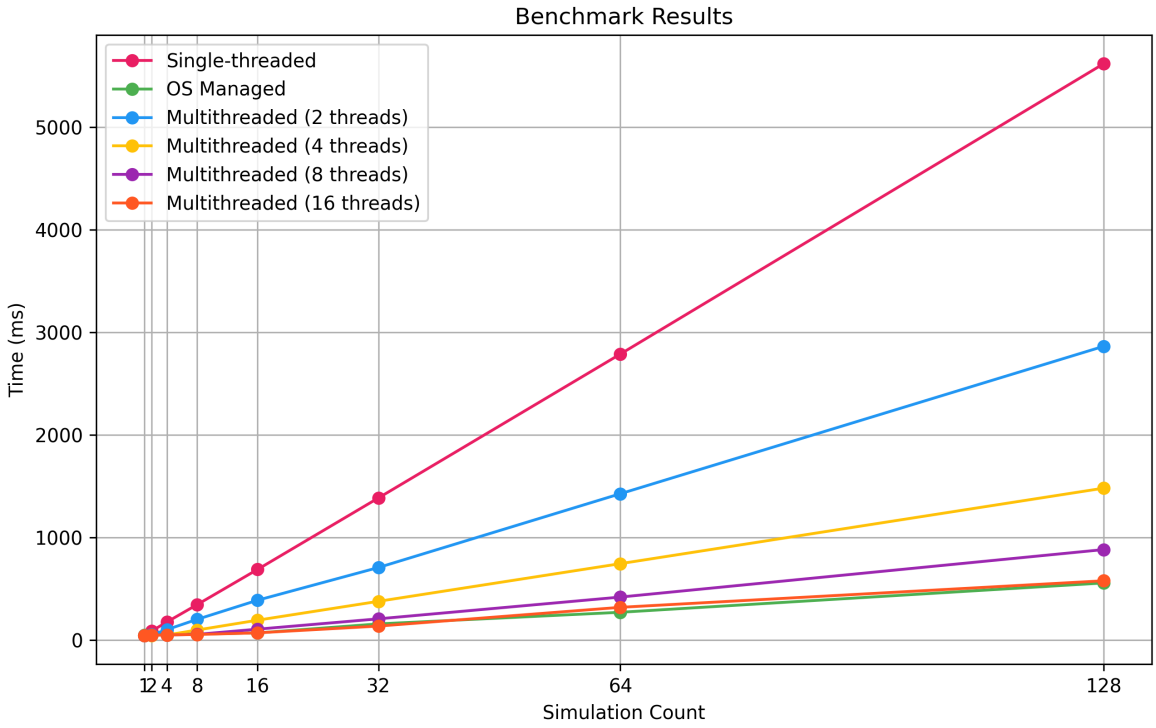

Figure 11: Benchmark of model simulation.

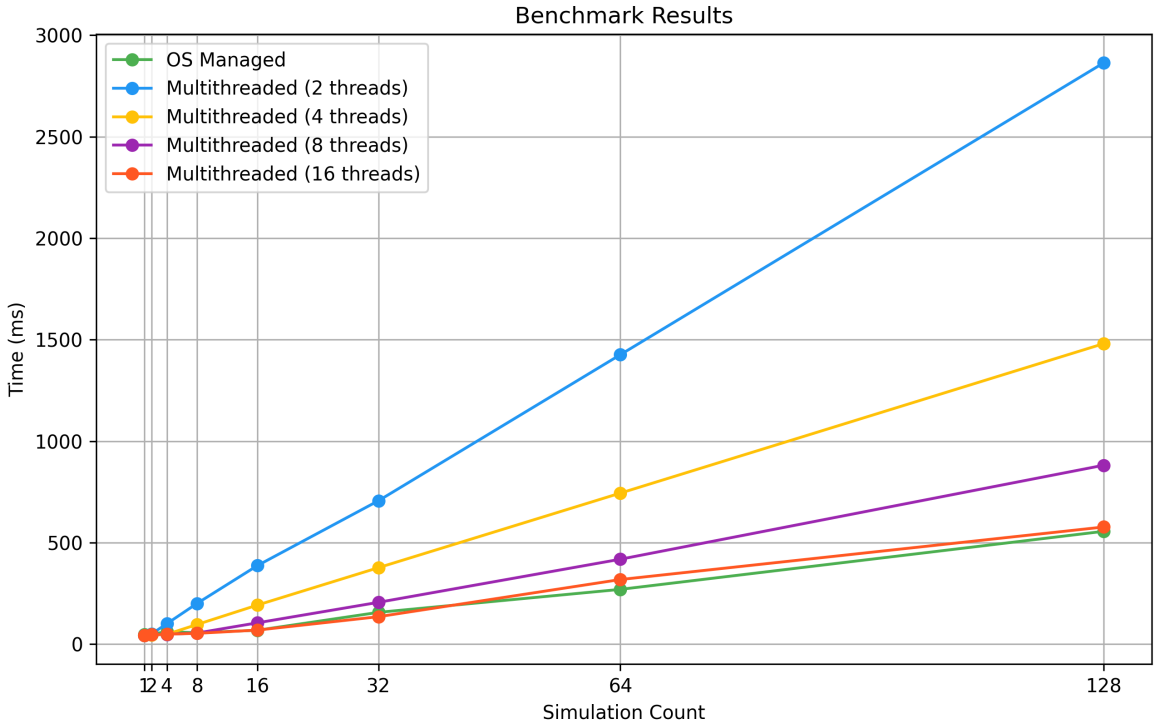

Figure 12: Benchmark of model simulation not including benchmarks for synchronized model simulation.

3 Conclusions

This section contains the results and conclusions of estimating the number of infected people in Denmark and North Jutland, as well as the benchmarks for running the simulations in parallel and sequentially.

3.1 Estimations

The following table shows the estimations for the number of infected people in Denmark (DK) as well as North Jutland (NJ) specifically. These estimations were calculated by running the `main` function in Listing 28.

	Mean Hospitalized	**Peak Hospitalized**
DK	1193	1268
NJ	128	161

Table 1: Hospitalization estimations.

3.2 Benchmarks

The benchmarks clearly indicate that running multiple simulations in parallel is faster than running them sequentially. This is to be expected, as the simulations are independent of each other, and can therefore be run in parallel without any issues. In other words, there are no data dependencies between the simulations, and therefore no need to wait for one simulation to finish before starting the next one.

As is also expected, running the simulations on a 16 core machine and letting the OS schedule the threads is about the same as running the simulations in a thread pool with 16 threads. According to my results, at least, the difference is negligible.