sP Exam Project

Caspar Emil Jensen, Lucas Lybek Højlund Pedersen, Pætur Magnussen
Thomas Dam Nykjær & Thomas Ilum Andersen

June 14, 2025

sp-exam-project

Authors

Caspar Emil Jensen, Lucas Lybek Højlund Pedersen, Pætur Magnussen, Thomas Dam Nykjær & Thomas Ilum Andersen

Requirements

Install **graphviz** - https://graphviz.org/download/
- Setup path in environment variables to make the “dot” file - command available

Install **qt6-charts** - https://www.qt.io/download/

- CMAKE 3.30 or later

Additional Requirements

This library requires:
gcc-14 compiler
cpp 23+ standard
Which is included in bleeding edge package managers and will be added to others later
CLion does not yet come packaged with gcc-14+/cpp 23+, so for Windows systems: Install **MSYS2** - ensure CLion configuration uses ucrt64

FAQ:

Generator errors are related to missing or wrong gcc version. See Additional Requirements

Network graph Generation:

To generate the network graph, run the following command in a terminal:

```
cd /graphs
dot -Tpng "name-of-file".dot -o network.png
```

Listing 1: ./bin/src/utils.hpp

```cpp
#ifndef UTILS_HPP
#define UTILS_HPP

#include "utils.hpp"
#include <filesystem>
#include <fstream>

#include "vessels.hpp"

namespace fs = std::filesystem;

inline fs::path find_project_root() {
    fs::path current = fs::current_path();
    fs::path last_with_cmake;

    while (!current.empty()) {
        if (fs::exists(current / ".git")) {
            return current;
        }
        if (fs::exists(current / "CMakeLists.txt")) {
            last_with_cmake = current;
        }
        current = current.parent_path();
    }

    if (!last_with_cmake.empty()) {
        return last_with_cmake;
    }
    throw std::runtime_error("Project root not found (no .git or CMakeLists.txt).");
}


inline void generate_dot_file(StochasticSimulation::Vessel &ves, const std::string &name) {
    fs::path out_path = find_project_root() / "graphs" / ("network_" + name + ".dot");
    std::cout << out_path.string() << std::endl;

    std::ofstream out_file{out_path};
    if (!out_file.is_open()) {
        throw std::runtime_error("Failed to open file: " + out_path.string());
    }
    out_file << to_dot_network(ves.get_reactions(), ves.get_species()) << std::endl;
    out_file.close();
}

#endif // UTILS_HPP
```

Listing 2: ./lib/examples/circadian_rhythm.hpp

```cpp
#ifndef CIRCADIAN_RHYTH_HPP
#define CIRCADIAN_RHYTH_HPP

#include "vessels.hpp"

namespace StochasticSimulation::Examples {
    Vessel circadian_rhythm ();
    std::vector<SimulationState> run_circadian_sim();

}

#endif //CIRCADIAN_RHYTH_HPP
```

Listing 3: ./lib/examples/covid-19.hpp

```
#ifndef COVID_19_HPP
#define COVID_19_HPP

#include "vessels.hpp"
#include <cstdint> // For uint32_t

namespace StochasticSimulation::Examples {
    Vessel seihr(uint32_t N);
    std::vector<SimulationState> run_covid_sim();
    void estimate_peak_hospitalized();
} // namespace StochasticSimulation::Examples

#endif // COVID_19_HPP
```

Listing 4: ./lib/examples/exponential_decay.hpp

```
#ifndef EXPONENTIAL_DECAY_HPP
#define EXPONENTIAL_DECAY_HPP

#include "vessels.hpp"

namespace StochasticSimulation::Examples {
    Vessel exponential_decay(uint32_t q_a, uint32_t q_b, uint32_t q_c);
    std::vector<SimulationState> run_exponential_decay_a();
    std::vector<SimulationState> run_exponential_decay_b();
    std::vector<SimulationState> run_exponential_decay_c();
    Vessel exponential_decay_a();
    Vessel exponential_decay_b();
    Vessel exponential_decay_c();
} // namespace StochasticSimulation::Examples

#endif // EXPONENTIAL_DECAY_HPP
```

Listing 5: ./lib/examples/multi_threading.hpp

```
#ifndef MULTI_THREADING_H
#define MULTI_THREADING_H
#include "multithreading.hpp"


namespace StochasticSimulation::Examples {
    void get_peak_average(float endtime, Vessel& baseVessel, uint32_t numberOfRuns, const
↪std::string&);
    void get_peak_average_serial(float endtime, Vessel& baseVessel, uint32_t numberOfRuns, const
↪std::string&);
    void get_peak_average_serial_optimized(float endtime, Vessel& baseVessel, uint32_t
↪numberOfRuns, const std::string&);

}
#endif
```

Listing 6: ./lib/include/charter.hpp

```
#ifndef CHARTER_HPP
#define CHARTER_HPP

#include <vector>
#include <QWidget>

#include "state.hpp"

```

```
namespace StochasticSimulation {

    // Requirement 6 – Display simulation trajectories of how the amounts change. We use qtCharts
    class Charter {
        // QString since implicitly shared (copy-on-write, thread-aware)
        public:
            static void showChart(std::vector<SimulationState>, uint32_t, uint32_t, const QString& title);
        static void showChart(std::vector<std::vector<double> > &data, std::vector<std::string> names,
            uint32_t width, uint32_t height, const QString &title);
    };

} // StochasticSimulation

#endif //CHARTER_HPP
```

Listing 7: ./lib/include/debug_print.hpp

```
#ifndef DEBUG_PRINT_HPP
#define DEBUG_PRINT_HPP
#include <string>

inline void debug_print(const std::string& msg) {
#ifndef NDEBUG
    std::cout << msg << std::endl;
#endif
}
#endif //DEBUG_PRINT_HPP
```

Listing 8: ./lib/include/multithreading.hpp

```
#ifndef MULTITHREADING_HPP
#define MULTITHREADING_HPP
#include <functional>
#include <future>
#include <vector>

#include "simulator.hpp"
#include "state.hpp"
#include "vessels.hpp"

namespace StochasticSimulation {
    class Multithreading {
    public:
        template<class observerReturnType> // generic
        static std::vector<observerReturnType> runObserve(
            float endtime,
            Vessel& baseVessel,
            const std::function<observerReturnType(SimulationState)>& observer,
            const uint32_t numberOfRuns)
        {
            std::vector<std::future<observerReturnType>> futures; // futures = promises

            for (int i = 0; i < numberOfRuns; i++) {
                Vessel vesselCopy = baseVessel;
                SimulationState state = vesselCopy.createSimulationState();

                futures.emplace_back(std::async(std::launch::async,
                    [endtime, state = std::move(state), vesselCopy = std::move(vesselCopy), observer]() mutable { // TODO: Why mutable?
```

```
                        return Simulator::simulate_observer<observerReturnType>(endtime, state, vesselCopy, observer); // Pass reserver into simulation, and return observer
                    }));
            }

            std::vector<observerReturnType> observerResults; // Accumulate oberverResults in a vector
            for (auto& f : futures) {
                observerResults.emplace_back(f.get());
            }

            return observerResults;
        }
    };
}
#endif //MULTITHREADING_HPP
```

Listing 9: ./lib/include/reaction.hpp

```
#ifndef REACTION_HPP
#define REACTION_HPP

#include <iomanip>
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <sstream>

#include "species.hpp"

namespace StochasticSimulation {
    struct SimulationState;

    struct Reaction {
        std::vector<Species> reactants;
        std::vector<Species> products;
        const double rate;
        double delay = 0.0;

        virtual ~Reaction() = default;

        explicit Reaction(std::vector<Species> reactants = {}, std::vector<Species> products = {}, double rate = 0.0)
            : reactants(reactants), products(products), rate(rate) { }

        std::string createFingerprint() const {
            std::ostringstream oss;

            auto names = [](const std::vector<Species> vec) {
                std::vector<std::string> n;
                for (const auto &s: vec) n.push_back(s.name);
                std::sort(n.begin(), n.end());
                return n;
            };

            for (const auto &n: names(reactants)) oss << "R:" << n << ";";
            for (const auto &n: names(products)) oss << "P:" << n << ";";
            oss << "Rate:" << std::fixed << std::setprecision(6) << rate;

            return oss.str();
        }
```

```
43
44          void print() const;
45
46          [[nodiscard]] virtual std::string to_string(int padAmount = 7) const;
47
48          static std::string center(const std::string& s, int width);
49
50          void calculateDelay(SimulationState &);
51      };
52
53      // Requirement 1: The library should overload operators to support the reaction rule
    ↪typesetting directly in C++ code.
54      std::ostream &operator<<(std::ostream &os, const Species &s);
55
56      Reaction operator+(const Species &a, const Species &b);
57
58      bool operator==(const Species &a, const Species &b);
59
60      Reaction operator+(const Reaction &reaction, const Species &species);
61
62      Reaction operator>>(const Reaction &reaction, const double rate);
63
64      Reaction operator>>(const Species &species, const double rate);
65
66      Reaction operator>>(const Species &species, const int rate);
67
68      Reaction operator>>=(const Reaction &reaction, const Species &product);
69
70      Reaction operator>>=(const Reaction &reactionA, const Reaction &reactionB);
71
72      // Requirement 2: Provide pretty-printing of the reaction network in: b) network graph.
73      std::string to_dot(const Reaction &reaction, const int index);
74
75      std::string to_dot(const Species &species);
76
77      std::string to_dot_network(const std::vector<Reaction> &reactions, const
    ↪std::vector<Species> &species);
78  }
79
80  #endif //REACTION_HPP
```

Listing 10: ./lib/include/simulator.hpp

```
1   #ifndef SIMULATOR_HPP
2   #define SIMULATOR_HPP
3
4   #include <functional>
5   #include <generator>
6   #include <limits>
7
8   #include "state.hpp"
9   #include "trajectory_logger.hpp"
10  #include "vessels.hpp"
11
12  #pragma once
13
14  namespace StochasticSimulation
15  {
16      class Simulator
17      {
18      public:
19          // Requirement 4: Implement the stochastic simulation (Alg. 1) of the system using the
```

```
↪reaction rules.

        // Requirement 7: ...provide a lazy trajectory generation interface (coroutine)...
        static std::generator<SimulationState> simulate_lazy(float endtime, SimulationState&
↪state, Vessel vessel, int resolutionAmount = 1);

        // Implements observer


        // Requirement 7: Implement a generic support for (any) user-supplied state observer
↪function object
        // The observer itself should be part by the user/test program and not part of the library.

        // Template can only reside in header files
        template <class observerReturnType>
        static observerReturnType simulate_observer(
            float endtime, SimulationState& state, Vessel vessel,
            const std::function<observerReturnType(SimulationState)>& observer =
↪[](SimulationState state) {
                return 0;
            })
        {
            observerReturnType result{};
            observer(state);

            while (state.time < endtime) {
                for (auto& reaction : vessel.get_reactions()) {
                    reaction.calculateDelay(state);
                }
                auto r = getSmallestDelay(vessel);
                if (r.delay == std::numeric_limits<double>::infinity()) {
                    break;
                }
                state.time += r.delay;

                if (!allReactantsQuantitiesLargerThanZero(r, state))
                    continue;

                for (auto& species : r.reactants) {
                    state.species.get(species.name).decrease_quantity();
                }
                for (auto& product : r.products) {
                    state.species.get(product.name).increase_quantity();
                }

                // Record the current time and snapshot of all species quantities into the
↪trajectory log
                result = observer(state);
            }
            return result;
        }

        // Implements observer

        // Requirement 7: Implement a generic support for (any) user-supplied state observer
↪function object
        // The observer itself should be part by the user/test program and not part of the library.
        template <class observerReturnType>
        static observerReturnType simulate_observer_optimized(
            float endtime, SimulationState& state, Vessel vessel,
            const std::function<observerReturnType(SimulationState)>& observer)
```

```
        {
            observerReturnType result{};
            observer(state);

            while (state.time < endtime) {
                for (auto& reaction : vessel.get_reactions()) {
                    reaction.calculateDelay(state);
                }
                auto r = getSmallestDelay(vessel);
                if (r.delay == std::numeric_limits<double>::infinity()) {
                    break;
                }
                state.time += r.delay;

                if (!allReactantsQuantitiesLargerThanZero(r, state))
                    continue;

                for (auto& species : r.reactants) {
                    state.species.get(species.name).decrease_quantity();
                }
                for (auto& product : r.products) {
                    state.species.get(product.name).increase_quantity();
                }

                result = observer(state);
            }
            return result;
        }

    private:
        static bool allReactantsQuantitiesLargerThanZero(const Reaction& reaction, const
↪SimulationState& state);

        static const Reaction& getSmallestDelay(Vessel& vessel);
    };
} // namespace StochasticSimulation

#endif // SIMULATOR_HPP
```

Listing 11: ./lib/include/species.hpp

```
#ifndef SPECIES_HPP
#define SPECIES_HPP
#include <functional>
#include <ranges>
#include <string>

namespace StochasticSimulation
{
    struct Reaction;

    struct Species
    {
        virtual ~Species() = default;
        std::string name;
        mutable int _quantity;
        std::unordered_map<std::string, std::function<void()>> mark_for_recalculation;

        explicit Species(std::string name, int quantity = 0);
        Species();

        void increase_quantity(int amount = 1)
```

```
        {
            for (const auto& func : mark_for_recalculation | std::views::values) {
                func();
            }
            _quantity += amount;
        }

        void decrease_quantity(int amount = 1)
        {
            for (const auto& func : mark_for_recalculation | std::views::values) {
                func();
            }
            _quantity -= amount;
        }

        void create_delay_marker_reference(const std::string& reactionName,
↪std::function<void()> delay_marker_func)
        {
            if (mark_for_recalculation.contains(reactionName))
                return;

            mark_for_recalculation[reactionName] = delay_marker_func;
        }

        virtual std::string to_string() const { return name; }
    };
} // namespace StochasticSimulation

#endif // SPECIES_HPP
```

Listing 12: ./lib/include/state.hpp

```
#ifndef STATE_HPP
#define STATE_HPP
#include <unordered_map>
#include <string>

#include "reaction.hpp"
#include "species.hpp"
#include "symbol_table.hpp"

namespace StochasticSimulation {
    struct SimulationState {
        double time = 0.0;
        SymbolTable<std::string, Species> species;

        explicit SimulationState(SymbolTable<std::string, Species> species);

        // Requirement 2: Provide pretty-printing of the reaction network in a) human readable.
        std::string to_string();
    };
}
#endif //STATE_HPP
```

Listing 13: ./lib/include/symbol_table.hpp

```
#ifndef SYMBOL_TABLE_HPP
#define SYMBOL_TABLE_HPP

#include <format>
#include <stdexcept>
#include <unordered_map>
```

```cpp
#include <vector>

#include "symbol_table.hpp"

// Requirement 3: Implement a generic symbol table to store and lookup objects of user-defined key and value types.
// Support failure cases when a) the table does not contain a looked up symbol,
// b) the table already contains a symbol that is being added.
// Demonstrate the usage of the symbol table with the reactants (names and initial counts).


namespace StochasticSimulation {
    // Requirement 3: Implement a generic symbol table to store and lookup objects of user-defined key and value types
    template<typename Key, typename Value>
    class SymbolTable {
        std::unordered_map<Key, Value> table;

    public:
        // Requirement 3: Support failure cases when b) Table already contains a symbol that is being added
        const Value &add(const Key &key, const Value &value) {
            if (table.contains(key)) throw std::runtime_error("Key already exists");
            table[key] = value;
            return value;
        }

        // Requirement 3: Support failure cases when a) the table does not contain a looked up symbol
        Value &get(const Key &key) {
            if (!table.contains(key)) throw std::out_of_range(
                std::format("(SymbolTable) - Key {} not found, passed wrong state to simulator?", key));
            return table.at(key);
        }

        // Requirement 3: Support failure cases when a) the table does not contain a looked up symbol
        const Value &get(const Key &key) const {
            // Overload for const symboltables
            if (!table.contains(key)) throw std::out_of_range(
                std::format("(SymbolTable:const) - Key {} not found, passed wrong state to simulator?", key));
            return table.at(key);
        }

        bool contains(const Key &key) const {
            return table.contains(key);
        }

        std::vector<Value> getValues() const {
            std::vector<Value> values;
            for (const auto &[_, value]: table) {
                values.push_back(value);
            }
            return values;
        }

        auto begin() { return table.begin(); }
        auto end() { return table.end(); }
        auto begin() const { return table.begin(); }
```

```
        auto end() const { return table.end(); }
    };
}
#endif //SYMBOL_TABLE_HPP
```

Listing 14: ./lib/include/trajectory_chart_widget.hpp

```
#ifndef TRAJECTORYCHARTWIDGET_HPP
#define TRAJECTORYCHARTWIDGET_HPP

#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
#include <vector>
#include <string>

#include "trajectory_logger.hpp"
#include "state.hpp"

QT_USE_NAMESPACE

namespace StochasticSimulation {


    class TrajectoryChartWidget : public QWidget
    {
        Q_OBJECT

    public:
        explicit TrajectoryChartWidget(QWidget* parent = nullptr);

        // Pass trajectory data to display
        void setTrajectory(const std::vector<SimulationState>& trajectory);
        void setTrajectory(std::vector<std::vector<double>>& data, std::vector<std::string>& names);

    private:
        QChart* chart_;
        QChartView* chartView_;

        // Map species name -> QLineSeries for that species
        std::unordered_map<std::string, QLineSeries*> seriesMap_;

        void setupChart();
    };

}

#endif // TRAJECTORYCHARTWIDGET_HPP
```

Listing 15: ./lib/include/trajectory_logger.hpp

```
#ifndef TRAJECTORY_LOGGER_HPP
#define TRAJECTORY_LOGGER_HPP

#include <vector>
#include <unordered_map>
#include <string>

#include "symbol_table.hpp"
#include "species.hpp"

namespace StochasticSimulation {
    struct TimeStep {
```

```
        double time = 0.0;
        std::unordered_map<std::string, int> speciesQuantities;
    };

    class TrajectoryLogger {
    public:
        // Requirement 3: Demonstrate the usage of the symbol table with the reactants (names and initial counts).
        void log(const double time, const SymbolTable<std::string, Species>& speciesTable) {
            TimeStep step;
            step.time = time;

            for (const auto& species : speciesTable.getValues()) {
                step.speciesQuantities[species.name] = species._quantity;
            }

            trajectory_.emplace_back(std::move(step));
        }

        [[nodiscard]]
        const std::vector<TimeStep>& getTrajectory() const {
            return trajectory_;
        }

    private:
        std::vector<TimeStep> trajectory_;
    };

}

#endif // TRAJECTORY_LOGGER_HPP
```

Listing 16: ./lib/include/vessels.hpp

```
#ifndef VESSELS_HPP
#define VESSELS_HPP

#include <string>
#include <utility>
#include <vector>

#include "symbol_table.hpp"
#include "reaction.hpp"
#include "state.hpp"

namespace StochasticSimulation {
    class Vessel {
        // Requirement 3: Demonstrate the usage of the symbol table with the reactants (names and initial counts).
        SymbolTable<std::string, Species> species;
        std::vector<Reaction> reactions;
        std::string name = "Vessel";
        Species _env = Species("env");

    public:
        explicit Vessel(std::string name) : name(std::move(name)) {
        }

        Species add(const std::string &name, double amount) {
            return species.add(name, Species(name, amount));
        }

```

```
        void add(const Reaction reaction) {
            reactions.push_back(reaction);
        }

        Species environment() {
            species.add(_env.name, _env);
            return _env;
        }

        std::vector<Reaction>& get_reactions() {
            return reactions;
        }

        std::vector<Species> get_species() {
            return species.getValues();
        }

        SimulationState createSimulationState() {
            auto a = species;
            return SimulationState(species);
        }

        // Requirement 2: Provide pretty-printing of the reaction network in a) human readable format and b) network graph
        void prettyPrintReactions(bool printHeader = true) const {
            if (printHeader) {
                std::cout << name << ":\n";
            }
            for (const auto& reaction: reactions) {
                std::cout << reaction.to_string() << std::endl;
            }
        }
    };
}
#endif // VESSELS_HPP
```

Listing 17: ./benchmark/benchmarks.cpp

```
#include <benchmark/benchmark.h>

#include "circadian_rhythm.hpp"
#include "debug_print.hpp"
#include "../lib/examples/covid-19.hpp"
#include "../lib/examples/exponential_decay.hpp"
#include "../lib/examples/multi_threading.hpp"


using namespace StochasticSimulation;

// Requirement 10: Benchmark and compare the stochastic simulation performance (e.g. the time it takes to compute 100 simulations
// a single core, multiple cores, or improved implementation). Record the timings and make your conclusions.

static void seihr_single_core(benchmark::State &state) {
    auto vessel = Examples::seihr(20000);

    debug_print("\nStarting single core seihr");
    for (const auto _: state) {
        Examples::get_peak_average_serial(100, vessel, 100, "H");
    }
    debug_print("End single core seihr");
```

```
}


BENCHMARK(seihr_single_core)->Unit(benchmark::kMillisecond)->Iterations(50);;


static void seihr_multi_core(benchmark::State &state) {
    auto vessel = Examples::seihr(20000);

    debug_print("\nStarting multi-core seihr");
    for (const auto _: state) {
        Examples::get_peak_average(100, vessel, 100, "H");
    }
    debug_print("End multi-core seihr");
}

BENCHMARK(seihr_multi_core)->Unit(benchmark::kMillisecond)->Iterations(50);


static void circadian_rhythm_single_core_100_runs(benchmark::State &state) {
    auto vessel = Examples::circadian_rhythm();

    debug_print("\nStarting circadian_rhythm_single_core_100_runs");
    for (const auto _: state) {
        Examples::get_peak_average_serial(48, vessel, 100, "DA");
        // This finds peak average of H - H does not exist in all examples
    }
    debug_print("End circadian_rhythm_single_core_100_runs");
}

BENCHMARK(circadian_rhythm_single_core_100_runs)->Unit(benchmark::kMillisecond)->Iterations(50);

static void circadian_rhythm_single_core_100_runs_optimized(benchmark::State &state) {
    auto vessel = Examples::circadian_rhythm();

    debug_print("\nStarting circadian_rhythm_single_core_100_runs_optimized");
    for (const auto _: state) {
        Examples::get_peak_average_serial_optimized(48, vessel, 100, "DA");
        // This finds peak average of H - H does not exist in all examples
    }
    debug_print("End circadian_rhythm_single_core_100_runs_optimized");
}

BENCHMARK(circadian_rhythm_single_core_100_runs_optimized)->Unit(benchmark::kMillisecond)->Iterations(50);


static void circadian_rhythm_multi_core_100_runs(benchmark::State &state) {
    auto vessel = Examples::circadian_rhythm();

    debug_print("\nStarting circadian_rhythm_multi_core_100_runs");
    for (const auto _: state) {
        Examples::get_peak_average(48, vessel, 100, "DA");
    }
    debug_print("End circadian_rhythm_multi_core_100_runs");
}

BENCHMARK(circadian_rhythm_multi_core_100_runs)->Unit(benchmark::kMillisecond)->Iterations(50);


static void exponential_decay_single_core(benchmark::State &state) {
    auto vessel = Examples::exponential_decay(50, 0, 50);
```

```
84
85      debug_print("\nStarting exponential_decay_single_core");
86      for (const auto _: state) {
87          Examples::get_peak_average_serial(48, vessel, 100, "C");
88      }
89      debug_print("End exponential_decay_single_core");
90  }
91
92  BENCHMARK(exponential_decay_single_core)->Unit(benchmark::kMillisecond)->Iterations(50);
93
94
95  static void exponential_decay_multi_core(benchmark::State &state) {
96      auto vessel = Examples::exponential_decay(50, 0, 50);
97
98      debug_print("\nStarting exponential_decay_multi_core");
99      for (const auto _: state) {
100         Examples::get_peak_average(1500, vessel, 100, "C");
101     }
102     debug_print("End exponential_decay_multi_core");
103 }
104
105 BENCHMARK(exponential_decay_multi_core)->Unit(benchmark::kMillisecond)->Iterations(50);
106
107 BENCHMARK_MAIN();
```

Listing 18: ./bin/src/main.cpp

```
1   #include <QApplication>
2
3   #include "charter.hpp"
4   #include "utils.hpp"
5   #include "trajectory_chart_widget.hpp"
6
7   #include "covid-19.hpp"
8   #include "exponential_decay.hpp"
9   #include "circadian_rhythm.hpp"
10
11  using namespace StochasticSimulation;
12  namespace fs = std::filesystem;
13
14  void runSimulations(float endtime, Vessel &baseVessel);
15
16  int main(int argc, char *argv[]) {
17      QApplication app(argc, argv);
18
19      // Requirement 7 ...To demonstrate the generic support, estimate the peak of hospitalized
    ↪agents in Covid-19 example
20      // without storing an entire trajectory. Record the peak hospitalization values for
    ↪population sizes of NNJ and NDK.
21      Examples::estimate_peak_hospitalized();
22
23      // Requirement 5: Demonstrate the application of the library on the three examples
24      // Example 1: Exponential Decay
25      auto exponential_decay_a = Examples::run_exponential_decay_a();
26      auto exponential_decay_b = Examples::run_exponential_decay_b();
27      auto exponential_decay_c = Examples::run_exponential_decay_c();
28      Charter::showChart(exponential_decay_a, 800, 600, "exponential_decay_a");
29      Charter::showChart(exponential_decay_b, 800, 600, "exponential_decay_b");
30      Charter::showChart(exponential_decay_c, 800, 600, "exponential_decay_c");
31
32      // Requirement 2 (b): Provide pretty-printing of the reaction network in a) human readable
    ↪format and b) network graph
```

```
    // - note, "dot -Tpng <*.dot> -o <*.png>still needs to be executed in the "graphs" folder
    auto exponential_decay_vessel_a = Examples::exponential_decay_a();
    generate_dot_file(exponential_decay_vessel_a, "exponential_decay_a");


    // Example 2: Circadian Rhythm
    auto circadian_rhythm = Examples::run_circadian_sim();
    Charter::showChart(circadian_rhythm, 800, 600, "circadian_rhythm");

    // Example 3: Covid 19
    // Create chart widget and set data
    auto covid_sim = Examples::run_covid_sim();
    Charter::showChart(covid_sim, 800, 600, "covid_sim");

    // Requirement 2 (a): Provide pretty-printing of the reaction network in a) human readable
↪format and b) network graph
    auto circadian_rhythm_vessel = Examples::circadian_rhythm();
    circadian_rhythm_vessel.prettyPrintReactions();

    return app.exec();
}
```

Listing 19: ./lib/examples/circadian_rhythm.cpp

```
#ifndef CIRCADIAN_RHYTHM_HPP
#define CIRCADIAN_RHYTHM_HPP

#include "simulator.hpp"
#include "vessels.hpp"
#include "../../bin/src/utils.hpp"

namespace StochasticSimulation::Examples {
    Vessel circadian_rhythm()
    {
        const auto alphaA = 50;
        const auto alpha_A = 500;
        const auto alphaR = 0.01;
        const auto alpha_R = 50;
        const auto betaA = 50;
        const auto betaR = 5;
        const auto gammaA = 1;
        const auto gammaR = 1;
        const auto gammaC = 2;
        const auto deltaA = 1;
        const auto deltaR = 0.2;
        const auto deltaMA = 10;
        const auto deltaMR = 0.5;
        const auto thetaA = 50;
        const auto thetaR = 100;
        auto v = StochasticSimulation::Vessel{"Circadian Rhythm"};
        const auto env = v.environment();
        const auto DA = v.add("DA", 1);
        const auto D_A = v.add("D_A", 0);
        const auto DR = v.add("DR", 1);
        const auto D_R = v.add("D_R", 0);
        const auto MA = v.add("MA", 0);
        const auto MR = v.add("MR", 0);
        const auto A = v.add("A", 0);
        const auto R = v.add("R", 0);
        const auto C = v.add("C", 0);
        v.add((A + DA) >> gammaA >>= D_A);
        v.add(D_A >> thetaA >>= DA + A);
```

```
        v.add((A + DR) >> gammaR >>= D_R);
        v.add(D_R >> thetaR >>= DR + A);
        v.add(D_A >> alpha_A >>= MA + D_A);
        v.add(DA >> alphaA >>= MA + DA);
        v.add(D_R >> alpha_R >>= MR + D_R);
        v.add(DR >> alphaR >>= MR + DR);
        v.add(MA >> betaA >>= MA + A);
        v.add(MR >> betaR >>= MR + R);
        v.add((A + R) >> gammaC >>= C);
        v.add(C >> deltaA >>= R);
        v.add(A >> deltaA >>= env);
        v.add(R >> deltaR >>= env);
        v.add(MA >> deltaMA >>= env);
        v.add(MR >> deltaMR >>= env);
        return v;
    }

    std::vector<SimulationState> run_circadian_sim() {
        auto vessel = circadian_rhythm();
        auto state = vessel.createSimulationState();

        std::vector<SimulationState> trajectory;
        //Observer version of simulate
        // auto test = [&trajectory](const SimulationState& state) -> int {
↪trajectory.emplace_back(state); return 0; };
        // Simulator::simulate_observer<int>(1500, state, vessel, test);

        //Lazy evaluation version of simulate

        int i = 0;
        for (auto&& simState : Simulator::simulate_lazy(48, state, vessel)) { // Consume
            trajectory.emplace_back(simState);
            i++;
        }
        generate_dot_file(vessel,"Circadian-Rhythm-Dot-Graph");
        std::cout << trajectory.size() << '\n';
        return trajectory;
    }
}
#endif //CIRCADIAN_RHYTHM_HPP
```

Listing 20: ./lib/examples/covid-19.cpp

```
#ifndef COVID_19_HPP
#define COVID_19_HPP

#include <cmath>

#include "simulator.hpp"
#include "vessels.hpp"
#include "../../bin/src/utils.hpp"

namespace StochasticSimulation::Examples {
    Vessel seihr(uint32_t N) {
        auto v = Vessel{"COVID19 SEIHR: " + std::to_string(N)};
        const auto eps = 0.0009; // initial fraction of infectious
        const auto I0 = size_t(std::round(eps * N)); // initial infectious
        const auto E0 = size_t(std::round(eps * N * 15)); // initial exposed
        const auto S0 = N - I0 - E0; // initial susceptible
        const auto R0 = 2.4; // initial basic reproductive number
        const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
        const auto gamma = 1.0 / 3.1; // recovery rate (I -> R) ~3.1 days
```

```
        const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
        const auto P_H = 0.9e-3; // probability of hospitalization
        const auto kappa = gamma * P_H * (1.0 - P_H); // hospitalization rate (I -> H)
        const auto tau = 1.0 / 10.12; // removal rate in hospital (H -> R) ~10.12 days
        const auto S = v.add("S", S0); // susceptible
        const auto E = v.add("E", E0); // exposed
        const auto I = v.add("I", I0); // infectious
        const auto H = v.add("H", 0); // hospitalized
        const auto R = v.add("R", 0); // removed/immune (recovered + dead)
        v.add((S + I) >> beta / N >>= E + I); // susceptible becomes exposed by infectious
        v.add(E >> alpha >>= I); // exposed becomes infectious
        v.add(I >> gamma >>= R); // infectious becomes removed
        v.add(I >> kappa >>= H); // infectious becomes hospitalized
        v.add(H >> tau >>= R); // hospitalized becomes removed
        return v;
    }

    std::vector<SimulationState> run_covid_sim() {
        auto vessel = seihr(100);
        auto state = vessel.createSimulationState();
        std::vector<SimulationState> trajectory;

        // Observer version of simulate
        // auto test = [&trajectory](const SimulationState& state) {   ↙
↪trajectory.emplace_back(state); };
        // Simulator::simulate(1500, c, covid, test);

        //Lazy evaluation version of simulate
        for (auto &&simState: Simulator::simulate_lazy(1500, state, vessel)) {
            // Consume
            trajectory.emplace_back(simState);
        }

        generate_dot_file(vessel,"Covid-Dot-Graph");

        return trajectory;
    }

    // Big covid sim, Req 7B
    void estimate_peak_hospitalized() {
        // Uses lazy evaluation with limited population sizes
        std::vector<std::pair<std::string, uint32_t> > regions = {
            {"NNJ", 10000},
            {"NDK", 20000}
        };

        std::vector<std::string> peaks;
        for (const auto &[region, population]: regions) {
            auto vessel = seihr(population);
            auto state = vessel.createSimulationState();
            int peak = 0;
            for (auto &&simState: Simulator::simulate_lazy(500.0, state, vessel)) {
                int currentH = simState.species.get("H")._quantity;
                if (currentH > peak) {
                    peak = currentH;
                }
            }
            peaks.emplace_back(
                "Peak hospitalized in " + region + " (population " + std::to_string(population)   ↙
↪+ "): " +
                std::to_string(peak));
```

```
        }
        std::cout << "Peak hospitalized in regions:\n";
        for (const auto &entry: peaks) {
            std::cout << "  - " << entry << "\n";
        }
    }
}

#endif //COVID_19_HPP
```

Listing 21: ./lib/examples/exponential_decay.cpp

```
#ifndef EXPONENTIAL_DECAY_HPP
#define EXPONENTIAL_DECAY_HPP

#include "simulator.hpp"
#include "vessels.hpp"
#include "../../bin/src/utils.hpp"

namespace StochasticSimulation::Examples {
    Vessel exponential_decay(uint32_t q_a, uint32_t q_b, uint32_t q_c) {
        auto v = Vessel{"Exponential Decay"};

        constexpr auto rate = 0.001;
        const auto A = v.add("A", q_a);
        const auto B = v.add("B", q_b);
        const auto C = v.add("C", q_c);

        v.add((A + C) >> rate >>= B + C);
        return v;
    }
    std::vector<SimulationState> run_exponential_decay(Vessel vessel) {
        auto state = vessel.createSimulationState();

        //Observer version of simulate
        //auto test = [&trajectory2](const SimulationState& state) {   ↙
↪trajectory2.emplace_back(state); };
        //Simulator::simulate(1500, c, covid, test);

        //Lazy evaluation version of simulate
        std::vector<SimulationState> trajectory;
        for (auto&& simState : Simulator::simulate_lazy(1500, state, vessel)) { // Consume
            trajectory.emplace_back(simState);
        }

        return trajectory;
    }

    Vessel exponential_decay_a() {
        return exponential_decay(100, 0, 1);
    }
    Vessel  exponential_decay_b() {
        return exponential_decay(100, 0, 2);
    }
    Vessel  exponential_decay_c() {
        return exponential_decay(50, 50, 1);
    }

    // Requirement 5: Demonstrate the application of the library on the three examples
    std::vector<SimulationState> run_exponential_decay_a() {
        return run_exponential_decay(exponential_decay_a());
    }
```

```
    std::vector<SimulationState> run_exponential_decay_b() {
        return run_exponential_decay(exponential_decay_b());
    }
    std::vector<SimulationState> run_exponential_decay_c() {
        return run_exponential_decay(exponential_decay_c());
    }

    void generate_dot_graph_exponential_decay() {
        auto ves = Vessel{"Exponential Decay"};
        generate_dot_file(ves,"Exponential-Decay-Dot-Graph");
    }
}

#endif //EXPONENTIAL_DECAY_HPP
```

Listing 22: ./lib/examples/multi_threading.cpp

```
#include "multi_threading.hpp"

#include "covid-19.hpp"

namespace StochasticSimulation::Examples {
    // Requirement 8: Implement support for multiple CPU cores by parallelizing the computation of several simulations at the same time.
    // Estimate the likely (average) value of the hospitalized peak over 100 simulations.

    void get_peak_average(float endtime, Vessel &baseVessel, const uint32_t numberOfRuns,
                          const std::string &peakProperty) {
        auto observer = [&peakProperty](const SimulationState &state) -> int {
            thread_local int peak = 0;
            if (int currentProperty = state.species.get(peakProperty)._quantity; currentProperty > peak)
                peak = currentProperty;
            return peak;
        };

        auto peaks = Multithreading::runObserve<int>( //<ObserverReturnType>
            endtime, baseVessel, observer, numberOfRuns
        );

        int sum = 0;
        for (const auto &peak: peaks) {
            sum += peak;
        }

        float average = static_cast<float>(sum) / peaks.size();
        std::cout << "Average sum(" << sum << ") Peak: " << average << std::endl;
    }

    void get_peak_average_serial(float endtime, Vessel &baseVessel, const uint32_t numberOfRuns,
                                 const std::string &peakProperty) {
        int peak = 0;
        auto peak_serial_observer = [&peak, &peakProperty](const SimulationState &state) {
            const int currentProperty = state.species.get(peakProperty)._quantity;
            if (currentProperty > peak)
                peak = currentProperty;
            return peak;
        };

        auto peak_vessel_serial = baseVessel;
        std::vector<int> peaks_serial;
        for (int i = 0; i < numberOfRuns; i++) {
```

```
            peak = 0;
            auto vessel_serial = peak_vessel_serial;
            auto state_serial = peak_vessel_serial.createSimulationState();
            peaks_serial.emplace_back(
                Simulator::simulate_observer<int>(endtime, state_serial, vessel_serial,
↪peak_serial_observer));
        }

        int sum = 0;
        for (int a: peaks_serial)
            sum += a;
        float average = static_cast<float>(sum) / peaks_serial.size();
        std::cout << "Average sum(" << sum << ") serial Peak: " << average << std::endl;
    }

    void get_peak_average_serial_optimized(float endtime, Vessel &baseVessel, const uint32_t
↪numberOfRuns,
                                           const std::string &peakProperty) {
        int peak = 0;
        auto peak_serial_observer = [&peak, &peakProperty](const SimulationState &state) {
            const int currentProperty = state.species.get(peakProperty)._quantity;
            if (currentProperty > peak)
                peak = currentProperty;
            return peak;
        };

        auto peak_vessel_serial = baseVessel;
        std::vector<int> peaks_serial;
        for (int i = 0; i < numberOfRuns; i++) {
            peak = 0;
            auto vessel_serial = peak_vessel_serial;
            auto state_serial = peak_vessel_serial.createSimulationState();
            peaks_serial.emplace_back(
                Simulator::simulate_observer_optimized<int>(endtime, state_serial, vessel_serial,
                                                            peak_serial_observer));
        }

        int sum = 0;
        for (int a: peaks_serial)
            sum += a;
        float average = static_cast<float>(sum) / peaks_serial.size();
        std::cout << "Average sum(" << sum << ") serial Peak: " << average << std::endl;
    }
}
```

Listing 23: ./lib/src/charter.cpp

```
#include "../include/charter.hpp"
#include "trajectory_chart_widget.hpp"

namespace StochasticSimulation {
    void Charter::showChart(std::vector<SimulationState> trajectory, uint32_t width, uint32_t
↪height, const QString& title) {
        auto* chartWidget = new TrajectoryChartWidget();
        chartWidget->setAttribute(Qt::WA_DeleteOnClose);

        chartWidget->setTrajectory(trajectory);
        chartWidget->resize(width, height);
        chartWidget->setWindowTitle(title);
        chartWidget->show();
    }
} // StochasticSimulation
```

Listing 24: ./lib/src/reaction.cpp

```cpp
#include <string>
#include <utility>
#include <vector>
#include <cmath>
#include <sstream>
#include <random>

#include "reaction.hpp"
#include "state.hpp"

namespace StochasticSimulation {

    Species::Species()
        : name(), _quantity(0) {
    }

    Species::Species(std::string name, int quantity)
        : name(std::move(name)), _quantity(quantity) {
    }

    std::string Reaction::to_string(int padAmount) const {
        std::ostringstream lhs;
        for (size_t i = 0; i < reactants.size(); ++i) {
            lhs << reactants[i].to_string();
            if (i < reactants.size() - 1) lhs << " + ";
        }

        std::ostringstream out;
        out << std::left << std::setw(padAmount) << lhs.str();

        std::ostringstream rateStr;
        rateStr << "(" << std::fixed << std::setprecision(2) << rate << ")";
        out << " >>" << center(rateStr.str(), 10) << ">>= ";

        for (size_t i = 0; i < products.size(); ++i) {
            out << products[i].to_string();
            if (i < products.size() - 1) out << " + ";
        }

        return out.str();
    }

    std::string Reaction::center(const std::string& s, int width) {
        int pad = width - static_cast<int>(s.length());
        if (pad <= 0) return s;
        int pad_left = pad / 2;
        int pad_right = pad - pad_left;
        return std::string(pad_left, ' ') + s + std::string(pad_right, ' ');
    }

    void Reaction::calculateDelay(SimulationState &state) {

        int product = 1;
        for (const Species &sp: reactants) {
            product *= state.species.get(sp.name)._quantity;
        }

        double lambda = rate * product;
```

```
        if (lambda <= 0.0 || !std::isfinite(lambda)) {
            delay = std::numeric_limits<double>::infinity();
            return;
        }

        static std::mt19937 rng(std::random_device{}());
        std::exponential_distribution<> dist(lambda);
        delay = dist(rng);
    }

    // Requirement 1 - operator overloading
    std::ostream &operator<<(std::ostream &os, const Species &s) {
        os << "Species(name=" << s.name << ")";
        return os;
    }

    Reaction operator+(const Species &a, const Species &b) {
        return Reaction({a, b});
    }

    bool operator==(const Species &a, const Species &b) {
        return a.name == b.name && a._quantity == b._quantity;
    }

    // (A + B) + C --> adds another Species to the list of reactants
    Reaction operator+(const Reaction &reaction, const Species &species) {
        std::vector<Species> new_reactants = reaction.reactants;
        new_reactants.push_back(species);
        return Reaction(new_reactants, reaction.products, reaction.rate);
    }

    // (A + B) >> 0.01 --> sets the reaction rate     - intrinsic
    Reaction operator>>(const Reaction &reaction, const double rate) {
        return Reaction(reaction.reactants, reaction.products, rate);
    }

    Reaction operator>>(const Species &species, const double rate) {
        return Reaction({species}, {}, rate);
    }

    Reaction operator>>(const Species &species, const int rate) {
        return Reaction({species}, {}, rate);
    }

    ///((A + B)) >> 0.01 >>= C --> completes the reaction and creates a Reaction object
    Reaction operator>>=(const Reaction &reaction, const Species &product) {
        return Reaction{reaction.reactants, {product}, reaction.rate};
    }

    Reaction operator>>=(const Reaction &reactionA, const Reaction &reactionB) {
        return Reaction{reactionA.reactants, {reactionB.reactants}, reactionA.rate};
    }


    std::string to_dot(const Reaction &reaction, const int index) {
        std::ostringstream out;
        std::string rname = "r" + std::to_string(index);
        out << "  " << rname << " [label=\"λ=" << reaction.rate <<
                "\",shape=\"oval\",fillcolor=\"yellow\",style=\"filled\"];\n";
        for (const auto &reactant: reaction.reactants) {
            out << "  " << reactant.name << " -> " << rname << ";\n";
```

```
        }
        for (const auto &product: reaction.products) {
            out << "  " << rname << " -> " << product.name << ";\n";
        }
        return out.str();
    }

    std::string to_dot(const Species &species) {
        std::ostringstream out;

        out << "  " << species.name << " [shape=\"rect\",fillcolor=\"cyan\",style=\"filled\"];\n";
        return out.str();
    }


    std::string to_dot_network(const std::vector<Reaction>& reactions,const std::vector<Species>& species) {
        std::ostringstream out;
        out << "digraph {\n";
        for (size_t i = 0; i < reactions.size(); ++i) {
            out << to_dot(reactions[i], static_cast<int>(i));
        }
        for (size_t i = 0; i < species.size(); ++i) {
            out << to_dot(species[i]);
        }
        out << "}\n";
        return out.str();
    }
}
```

Listing 25: ./lib/src/simulator.cpp

```
#include <chrono>
#include <functional>
#include <vector>
#include <generator>
#include "vessels.hpp"
#include "state.hpp"

#include "simulator.hpp"


namespace StochasticSimulation {
    // Implements Lazy evaluation through coroutine. Only works in version > c++ 23
    // Generator = C++ lazy evaluation (generates a sequence of elements by repeatedly resuming the coroutine from which it was returned.)
    std::generator<SimulationState> Simulator::simulate_lazy(float endtime, SimulationState &state, Vessel vessel, int resolution_amount)
    {
        // Yield initial state
        co_yield state;

        while (state.time < endtime) {
            for (auto& reaction : vessel.get_reactions()) {
                reaction.calculateDelay(state);
            }
            auto r = getSmallestDelay(vessel);
            if (r.delay == std::numeric_limits<double>::infinity()) {
                std::cout << "No valid reactions left - simulation stopping." << std::endl;
                break;
            }
            state.time += r.delay;
```

```
29
30
31              if (!allReactantsQuantitiesLargerThanZero(r, state))
32                  continue;
33
34              for (auto& species : r.reactants) {
35                  state.species.get(species.name).decrease_quantity(resolution_amount);
36              }
37              for (auto& product : r.products) {
38                  state.species.get(product.name).increase_quantity(resolution_amount);
39              }
40
41              // Record the current time and snapshot of all species quantities into the trajectory log
42              // Yield state and only run next iteration when next state is required from calling entity (lazy evaluation)
43              co_yield state;
44          }
45      }
46
47      // For smallest reaction (reaction with smallest delay) all reactants (species) must have a quantity of x>0 (otherwise they can't create a reaction
48      bool Simulator::allReactantsQuantitiesLargerThanZero(const Reaction& reaction, const SimulationState &state) {
49          for (const auto& species: reaction.reactants) {
50              if (species._quantity > 0 && state.species.get(species.name)._quantity <= 0)
51                  return false;
52          }
53          return true;
54      }
55
56
57      const Reaction &Simulator::getSmallestDelay(Vessel &vessel) {
58          auto &reactions = vessel.get_reactions();  // Must be a reference!
59          if (reactions.empty()) throw std::runtime_error("No reactions");
60
61          Reaction* smallest = &reactions[0];
62          for (auto& r : reactions) {
63              if (r.delay < smallest->delay)
64                  smallest = &r;
65          }
66          return *smallest;
67      }
68  };
```

Listing 26: ./lib/src/state.cpp

```
1  #include <iostream>
2  #include <utility>
3
4  #include "state.hpp"
5
6  namespace StochasticSimulation {
7      // Requirement 3: Demonstrate the usage of the symbol table with the reactants (names and initial counts).
8      SimulationState::SimulationState(SymbolTable<std::string, Species> species) : species(std::move(species)) {
9      }
10
11     std::string SimulationState::to_string() {
12         std::stringstream ss;
13         ss << "CurrentTime: " << time << " Species:" << std::endl;
```

```
        for (const auto &species: species | std::views::values) {
            ss << "  " << species.to_string() << ": " << species._quantity << std::endl;
        }
        return ss.str();
    }
}
```

Listing 27: ./lib/src/trajectory_chart_widget.cpp

```
#include <QtCharts/QValueAxis>
#include <QVBoxLayout>

#include "trajectory_chart_widget.hpp"
#include "state.hpp"

namespace StochasticSimulation {
    TrajectoryChartWidget::TrajectoryChartWidget(QWidget* parent)
        : QWidget(parent),
          chart_(new QChart()),
          chartView_(new QChartView(chart_, this))
    {
        auto layout = new QVBoxLayout(this);
        layout->addWidget(chartView_);
        setLayout(layout);

        setupChart();
    }

    void TrajectoryChartWidget::setupChart()
    {
        chart_->setTitle("Species Quantities Over Time");
        chart_->legend()->setVisible(true);
        chart_->legend()->setAlignment(Qt::AlignBottom);

        // Configure axes
        auto axisX = new QValueAxis;
        axisX->setTitleText("Time");
        axisX->setLabelFormat("%.1f");
        chart_->addAxis(axisX, Qt::AlignBottom);

        auto axisY = new QValueAxis;
        axisY->setTitleText("Quantity");
        axisY->setLabelFormat("%d");
        chart_->addAxis(axisY, Qt::AlignLeft);
    }

    void TrajectoryChartWidget::setTrajectory(const std::vector<SimulationState>& trajectory)
    {
        if (trajectory.empty())
            return;

        // Clear previous series
        for (auto& [name, series] : seriesMap_) {
            chart_->removeSeries(series);
            delete series;
        }
        seriesMap_.clear();

        // Get all species names from first time step
        const auto& firstStep = trajectory.front();
        std::map<std::string, QVector<QPointF>> pointsMap;
        for (const auto& [speciesName, _] : firstStep.species) {
```

```
            if (speciesName == "env") continue; // Guard to avoid printing the "env"

            auto series = new QLineSeries();
            series->setName(QString::fromStdString(speciesName));
            chart_->addSeries(series);
            seriesMap_[speciesName] = series;
            series->attachAxis(chart_->axisX());
            series->attachAxis(chart_->axisY());
            pointsMap[speciesName].reserve(trajectory.size());
        }

        // Fill the QVector<QPointF> for each species
        for (const auto& step : trajectory) {
            for (const auto& [speciesName, quantity] : step.species) {
                pointsMap[speciesName].append(QPointF(step.time, quantity._quantity));
            }
        }

        // Bulk replace data in each series
        for (const auto& [speciesName, series] : seriesMap_) {
            series->replace(pointsMap[speciesName]);
        }

        // Set axes range
        auto axisX = static_cast<QValueAxis*>(chart_->axisX());
        auto axisY = static_cast<QValueAxis*>(chart_->axisY());

        axisX->setRange(0, trajectory.back().time);

        int maxQuantity = 0;
        for (const auto& step : trajectory) {
            for (const auto& [speciesName, quantity] : step.species) {
                if (speciesName == "env") continue; // Guard to avoid printing the "env"
                if (quantity._quantity > maxQuantity)
                    maxQuantity = quantity._quantity;
            }
        }
        axisY->setRange(0, maxQuantity + 1);

        // Disable animations and antialiasing for performance
        chart_->setAnimationOptions(QChart::NoAnimation);
        chartView_->setRenderHint(QPainter::Antialiasing, false);

        chartView_->repaint();
    }
}
```

Listing 28: ./test/reaction_test.cpp

```
#include <doctest/doctest.h>

#include "species.hpp"
#include "reaction.hpp"
#include "vessels.hpp"
#include <cmath>

using namespace StochasticSimulation;

// Requirement 9: Implement unit tests (e.g. test symbol table methods, their failure cases, pretty-printing reaction rules, etc).
TEST_CASE("Species_Test") {
    SUBCASE("Species Default Constructor") {
```

```
        Species s;
        CHECK(s.name.empty());
        CHECK(s._quantity == 0);
    }

    SUBCASE("Constructor with empty name") {
        Species s("", 10);
        CHECK(s.name.empty());
        CHECK(s._quantity == 10);
    }
}

TEST_CASE("Reaction_Test") {
    SUBCASE("Reaction get reactants test") {
        const std::vector<Species> reactants{};
        const std::vector<Species> products{Species{"agent_a", 0}, Species{"agent_b", 0}};
        constexpr double rate = 5.0;


        Reaction r(reactants, products, rate);

        for (std::size_t i = 0; i < r.reactants.size(); i++) {
            CHECK(r.reactants[i].name == reactants[i].name);
        }
    }

    SUBCASE("Reaction get products test") {
        const std::vector<Species> reactants{};
        const std::vector<Species> products{Species{"agent_a", 0}, Species{"agent_b", 0}};
        constexpr double rate = 5.0;

        Reaction r(reactants, products, rate);

        for (std::size_t i = 0; i < r.products.size(); i++) {
            CHECK(r.products[i].name == products[i].name);
        }
    }
}


TEST_CASE("Delay calculation") {
    // lambda = rate x product
    SUBCASE("Delay with positive lambda test") {
        auto vessel = Vessel("Name");
        SimulationState state = vessel.createSimulationState();
        state.species.add("A", Species{"A", 5});
        Species A{"A"};

        Reaction r({A}, {}, 2.0);

        r.calculateDelay(state);

        CHECK(r.delay >= 0.0);
        CHECK(std::isfinite(r.delay));
    }

    SUBCASE("Delay with zero quantity test") {
        auto vessel = Vessel("Name");
        SimulationState state = vessel.createSimulationState();
        state.species.add("A", Species{"A", 0});
        Species A{"A"};
```

```
        Reaction r({A}, {}, 2.0);
        r.calculateDelay(state);
        CHECK(std::isinf(r.delay));
    }
}

// Identical reactions should yield identical fingerprints
TEST_CASE("Fingerprint is consistent test") {
    Reaction r1({Species{"X"}, Species{"Y"}}, {Species{"Z"}}, 1.0);
    Reaction r2({Species{"X"}, Species{"Y"}}, {Species{"Z"}}, 1.0);
    CHECK(r1.createFingerprint() == r2.createFingerprint());
}


TEST_CASE("Fingerprint is unique test") {
    Reaction r1({Species{"X"}, Species{"Y"}}, {Species{"Z"}}, 1.0);
    Reaction r2({Species{"X"}, Species{"Y"}}, {Species{"Z"}}, 1.5);
    CHECK(r1.createFingerprint() != r2.createFingerprint());
}


TEST_CASE("Reaction operator overloads (DSL) test") {
    SUBCASE("Species + Species gives correct reactants") {
        Species A("A");
        Species B("B");
        Species C("C");

        Reaction r = A + B;

        CHECK(r.reactants.size() == 2);
        CHECK(r.reactants[0].name == "A");
        CHECK(r.reactants[1].name == "B");
    }

    SUBCASE("Adding species to reaction appends to reactants") {
        Species A("A");
        Species B("B");
        Species C("C");

        Reaction r1 = A + B;
        Reaction r2 = r1 + C;

        CHECK(r2.reactants.size() == 3);
        CHECK(r2.reactants[2].name == "C");
    }

    SUBCASE("Reaction >> product finalizes with correct product") {
        Species A("A");
        Species B("B");

        Reaction r = A >> 0.01 >>= B;

        CHECK(r.reactants.size() == 1);
        CHECK(r.reactants[0].name == "A");

        CHECK(r.products.size() == 1);
        CHECK(r.products[0].name == "B");

        CHECK(r.rate == doctest::Approx(0.01));
    }

```

```
    SUBCASE("to string returns expected format") {
        Species A("A");
        Species B("B");

        Reaction r = A + B >> 0.02 >>= B;

        std::string expected = "A + B   >>  (0.02)  >>= B";
        CHECK(r.to_string() == expected);
    }

    SUBCASE("to string returns expected format") {
        Species A("A");
        Species B("B");

        Reaction r = A + B >> 0.02 >>= B + A;

        std::string expected = "A + B   >>  (0.02)  >>= B + A";
        CHECK(r.to_string() == expected);
    }
}
```

Listing 29: ./test/symbol_table_test.cpp

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <memory>
#include <doctest/doctest.h>
#include "../lib/include/symbol_table.hpp"

using namespace StochasticSimulation;

// Requirement 9: Implement unit tests (e.g. test symbol table methods, their failure cases,
↪pretty-printing reaction rules, etc).
TEST_CASE("SymbolTable basic functionality") {
    SymbolTable<std::string, int> table;

    SUBCASE("Add and get value") {
        table.add("foo", 42);
        CHECK(table.get("foo") == 42);
    }

    SUBCASE("Duplicate key throws") {
        table.add("bar", 123);
        CHECK_THROWS_AS(table.add("bar", 456), std::runtime_error);
    }

    SUBCASE("Missing key throws on get") {
        CHECK_THROWS_AS(table.get("missing"), std::out_of_range);
    }

    SUBCASE("Contains works correctly") {
        CHECK_FALSE(table.contains("x"));
        table.add("x", 7);
        CHECK(table.contains("x"));
    }

    SUBCASE("getValues returns all values") {
        table.add("a", 1);
        table.add("b", 2);
        auto values = table.getValues();
        CHECK(values.size() == 2);
        CHECK(std::find(values.begin(), values.end(), 1) != values.end());
        CHECK(std::find(values.begin(), values.end(), 2) != values.end());
```

```
    }
}
```

Listing 30: ./benchmark/CMakeLists.txt

```
include("../cmake/benchmark.cmake")

add_executable(benchmarks benchmarks.cpp
        ../lib/include/debug_print.hpp)
target_link_libraries(benchmarks PRIVATE
        benchmark::benchmark
        stochastic-simulation
)
```

Listing 31: ./bin/src/CMakeLists.txt

```
set(CMAKE_AUTOMOC ON)
find_package(Qt6 COMPONENTS Core Gui Widgets Charts REQUIRED)

add_executable(sp_exam_project main.cpp
        ../../lib/include/debug_print.hpp)

target_include_directories(sp_exam_project
        PRIVATE
        ${CMAKE_SOURCE_DIR}/lib/include
        ${CMAKE_SOURCE_DIR}/lib/examples
)

find_package(Qt6 COMPONENTS Core Gui Widgets Charts REQUIRED)

target_link_libraries(sp_exam_project
        PRIVATE
        stochastic-simulation
        Qt6::Core
        Qt6::Gui
        Qt6::Widgets
        Qt6::Charts
        benchmark
)

target_link_libraries(stochastic-simulation
        PRIVATE Qt6::Core Qt6::Gui Qt6::Widgets Qt6::Charts
)

target_compile_features(sp_exam_project PRIVATE cxx_std_20)
```

Listing 32: ./cmake/benchmark.cmake

```
# Downloads and compiles Google Benchmark
include(FetchContent)
set(FETCHCONTENT_QUIET ON)
set(FETCHCONTENT_UPDATES_DISCONNECTED ON)

set(BENCHMARK_ENABLE_TESTING OFF CACHE BOOL "Enable testing of the benchmark library.")
set(BENCHMARK_ENABLE_EXCEPTIONS ON CACHE BOOL "Enable the use of exceptions in the benchmark library.")
set(BENCHMARK_ENABLE_LTO OFF CACHE BOOL "Enable link time optimisation of the benchmark library.")
set(BENCHMARK_USE_LIBCXX OFF CACHE BOOL "Build and test using libc++ as the standard library.")
set(BENCHMARK_ENABLE_WERROR OFF CACHE BOOL "Build Release candidates with -Werror.")
set(BENCHMARK_FORCE_WERROR OFF CACHE BOOL "Build Release candidates with -Werror regardless of compiler issues.")
set(BENCHMARK_ENABLE_INSTALL OFF CACHE BOOL "Enable installation of benchmark. (Projects embedding benchmark may want to turn this OFF.)")
```

```
set(BENCHMARK_ENABLE_DOXYGEN OFF CACHE BOOL "Build documentation with Doxygen.")
set(BENCHMARK_INSTALL_DOCS OFF CACHE BOOL "Enable installation of documentation.")
set(BENCHMARK_DOWNLOAD_DEPENDENCIES ON CACHE BOOL "Allow the downloading and in-tree building of unmet dependencies")
set(BENCHMARK_ENABLE_GTEST_TESTS OFF CACHE BOOL "Enable building the unit tests which depend on gtest")
set(BENCHMARK_USE_BUNDLED_GTEST OFF CACHE BOOL "Use bundled GoogleTest. If disabled, the find_package(GTest) will be used.")
FetchContent_Declare(googlebenchmark
        GIT_REPOSITORY https://github.com/google/benchmark.git
        GIT_TAG v1.8.3    # or "main" for latest
        GIT_SHALLOW TRUE  # download specific revision only (git clone --depth 1)
        GIT_PROGRESS TRUE # show download progress in Ninja
        USES_TERMINAL_DOWNLOAD TRUE)
FetchContent_MakeAvailable(googlebenchmark)

message(STATUS "!!! Benchmark comparison requires python3 and `pip install scipy` !!!")
set(benchmark_cmp python3 ${googlebenchmark_SOURCE_DIR}/tools/compare.py)
```

Listing 33: ./cmake/clang-format.cmake

```
file(GLOB_RECURSE ALL_SOURCE_FILES "../src/*.cpp" "../src/*.hpp" "../lib/*.cpp" "../lib/*.hpp" "../test/*.cpp")

# Runs the clang-format on all the source files
execute_process(
        COMMAND /usr/bin/clang-format -style=file -i ${ALL_SOURCE_FILES}
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
)
```

Listing 34: ./cmake/doctest.cmake

```
# Downloads and compiles DocTest unit testing framework
include(FetchContent)
set(FETCHCONTENT_QUIET ON)
set(FETCHCONTENT_UPDATES_DISCONNECTED ON)

set(DOCTEST_WITH_TESTS OFF CACHE BOOL "Build tests/examples")
set(DOCTEST_WITH_MAIN_IN_STATIC_LIB ON CACHE BOOL "Build a static lib for doctest::doctest_with_main")
set(DOCTEST_NO_INSTALL OFF CACHE BOOL "Skip the installation process")
set(DOCTEST_USE_STD_HEADERS OFF CACHE BOOL "Use std headers")

FetchContent_Declare(doctest
        GIT_REPOSITORY https://github.com/doctest/doctest.git
        GIT_TAG v2.4.11   # "main" for latest
        GIT_SHALLOW TRUE  # download specific revision only (git clone --depth 1)
        GIT_PROGRESS TRUE # show download progress in Ninja
        USES_TERMINAL_DOWNLOAD TRUE)
FetchContent_MakeAvailable(doctest)
```

Listing 35: ./cmake/sanitizers.cmake

```
# Downloads and compiles DocTest unit testing framework
include(FetchContent)
set(FETCHCONTENT_QUIET ON)
set(FETCHCONTENT_UPDATES_DISCONNECTED ON)

set(DOCTEST_WITH_TESTS OFF CACHE BOOL "Build tests/examples")
set(DOCTEST_WITH_MAIN_IN_STATIC_LIB ON CACHE BOOL "Build a static lib for doctest::doctest_with_main")
set(DOCTEST_NO_INSTALL OFF CACHE BOOL "Skip the installation process")
```

```
set(DOCTEST_USE_STD_HEADERS OFF CACHE BOOL "Use std headers")

FetchContent_Declare(doctest
        GIT_REPOSITORY https://github.com/doctest/doctest.git
        GIT_TAG v2.4.11   # "main" for latest
        GIT_SHALLOW TRUE  # download specific revision only (git clone --depth 1)
        GIT_PROGRESS TRUE # show download progress in Ninja
        USES_TERMINAL_DOWNLOAD TRUE)
FetchContent_MakeAvailable(doctest)
```

Listing 36: ./CMakeLists.txt

```
cmake_minimum_required(VERSION 3.30)
project(sp_exam_project)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

include("cmake/sanitizers.cmake")

enable_testing() # Keep this if you have your own unit tests (like in 'test')

add_subdirectory(lib/src)

add_subdirectory(bin/src)

add_subdirectory(test)

add_subdirectory(benchmark)
```

Listing 37: ./lib/include/CMakeLists.txt

```
```

Listing 38: ./lib/src/CMakeLists.txt

```
set(CMAKE_AUTOMOC ON) # enables Qt's MOC auto-processing
# set(CMAKE_INCLUDE_CURRENT_DIR ON)

# Build the core library
add_library(stochastic-simulation STATIC
        simulator.cpp
        reaction.cpp
        state.cpp
        charter.cpp
        ../examples/multi_threading.cpp
        ../examples/circadian_rhythm.cpp
        ../examples/covid-19.cpp
        ../examples/exponential_decay.cpp
        ../src/trajectory_chart_widget.cpp
        ../include/trajectory_chart_widget.hpp
        ../include/charter.hpp
        ../include/multithreading.hpp
        ../include/debug_print.hpp
        ../../bin/src/utils.hpp
        ../examples/circadian_rhythm.hpp
)


target_include_directories(stochastic-simulation
        PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/../include
```

```
        ${CMAKE_CURRENT_SOURCE_DIR}/../examples
)

find_package(Qt6 COMPONENTS Core Gui Widgets Charts REQUIRED)

target_link_libraries(stochastic-simulation
        PRIVATE Qt6::Core Qt6::Gui Qt6::Widgets Qt6::Charts
)
```

Listing 39: ./test/CMakeLists.txt

```
include("../cmake/doctest.cmake")

add_executable(symbol_table_test symbol_table_test.cpp)
target_link_libraries(symbol_table_test PRIVATE doctest::doctest_with_main stochastic-simulation)
add_test(NAME symbol_table_test COMMAND symbol_table_test)

add_executable(reaction_test reaction_test.cpp)
target_link_libraries(reaction_test PRIVATE doctest::doctest_with_main stochastic-simulation)
add_test(NAME reaction_test COMMAND reaction_test)
```

Benchmark Results

Table 1: Benchmark results for 50 iterations and 100 runs (Processor: AMD Ryzen 9 5950X 16-Core)

Benchmark	Time (ms)	CPU (ms)	Iterations	endTime
seihr_single_core	6936	6910	50	100
seihr_multi_core	976	4.69	50	100
circadian_rhythm_single_core_100_runs	24519	24432	50	48
circadian_rhythm_single_core_100_runs_optimized	24512	24421	50	48
circadian_rhythm_multi_core_100_runs	4383	14.1	50	48
exponential_decay_single_core	8.77	8.75	50	1500
exponential_decay_multi_core	8.07	5.00	50	1500

Pretty printing of reactions

```
Circadian Rhythm:
A + DA  >>  (1.00)  >>= D_A
D_A     >> (50.00)  >>= DA + A
A + DR  >>  (1.00)  >>= D_R
D_R     >> (100.00) >>= DR + A
D_A     >> (500.00) >>= MA + D_A
DA      >> (50.00)  >>= MA + DA
D_R     >> (50.00)  >>= MR + D_R
DR      >>  (0.01)  >>= MR + DR
MA      >> (50.00)  >>= MA + A
MR      >>  (5.00)  >>= MR + R
A + R   >>  (2.00)  >>= C
C       >>  (1.00)  >>= R
A       >>  (1.00)  >>= env
R       >>  (0.20)  >>= env
MA      >> (10.00)  >>= env
MR      >>  (0.50)  >>= env
```

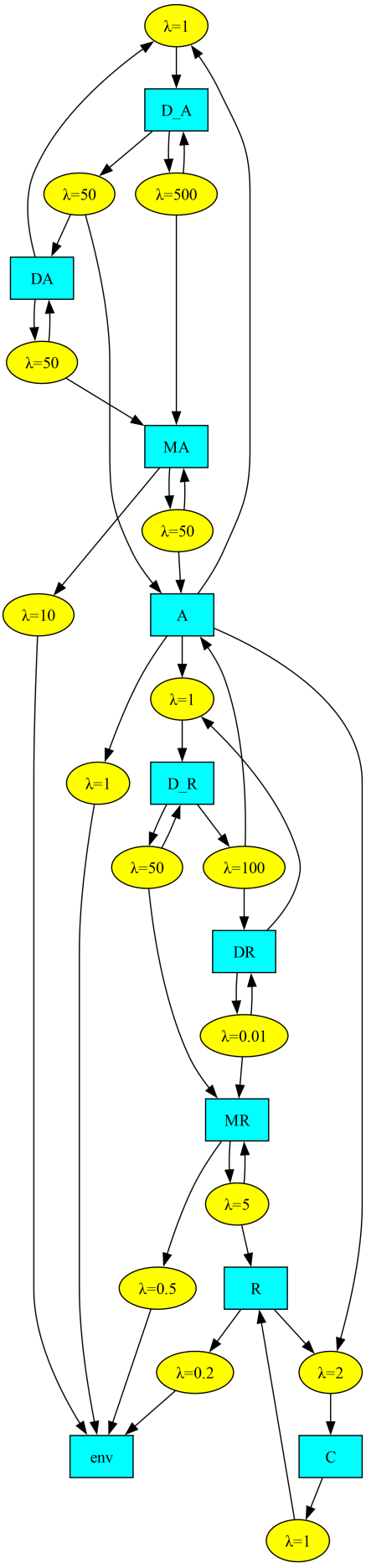

Figure 1: ./graphs/circadian_rhythm.png

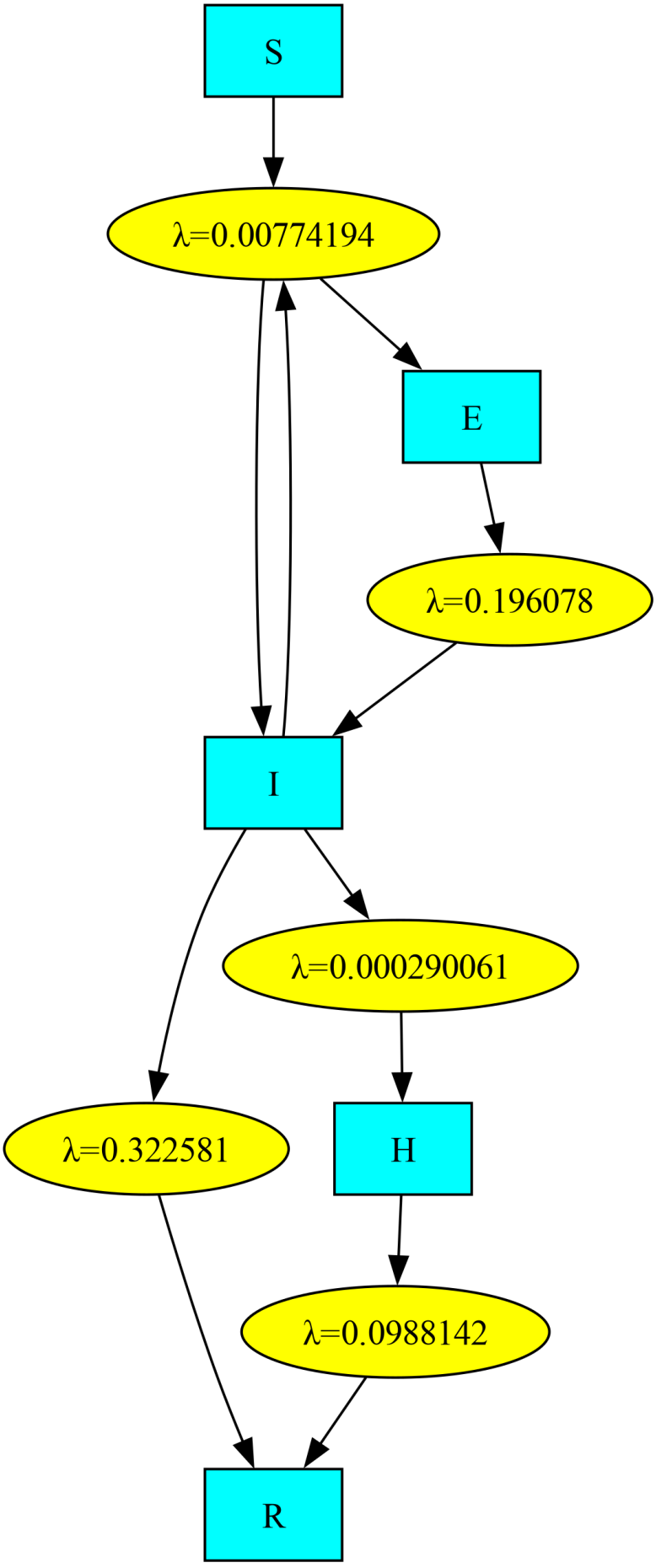

Figure 2: ./graphs/covid.png

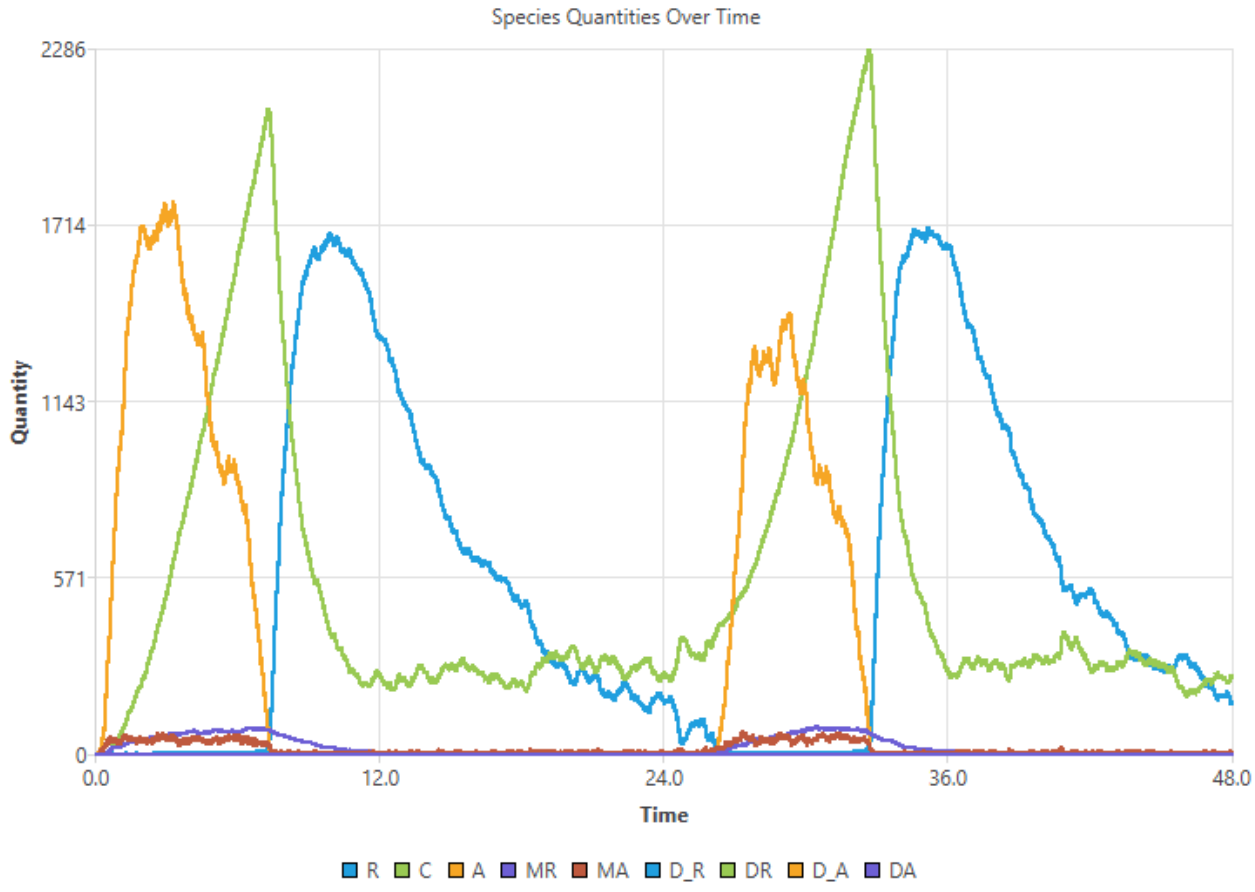

Figure 3: ./graphs/circadian.png

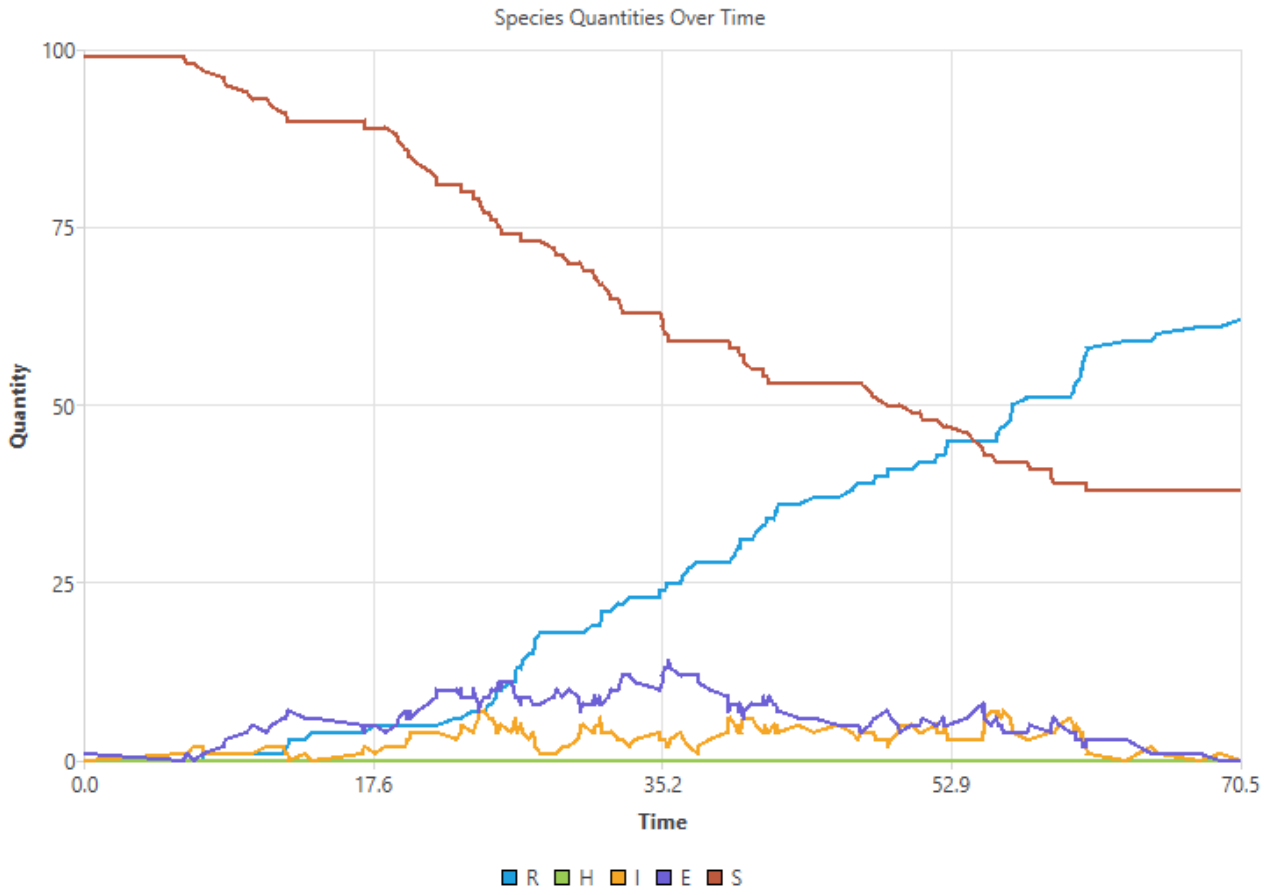

Figure 4: ./graphs/covid_sim.png

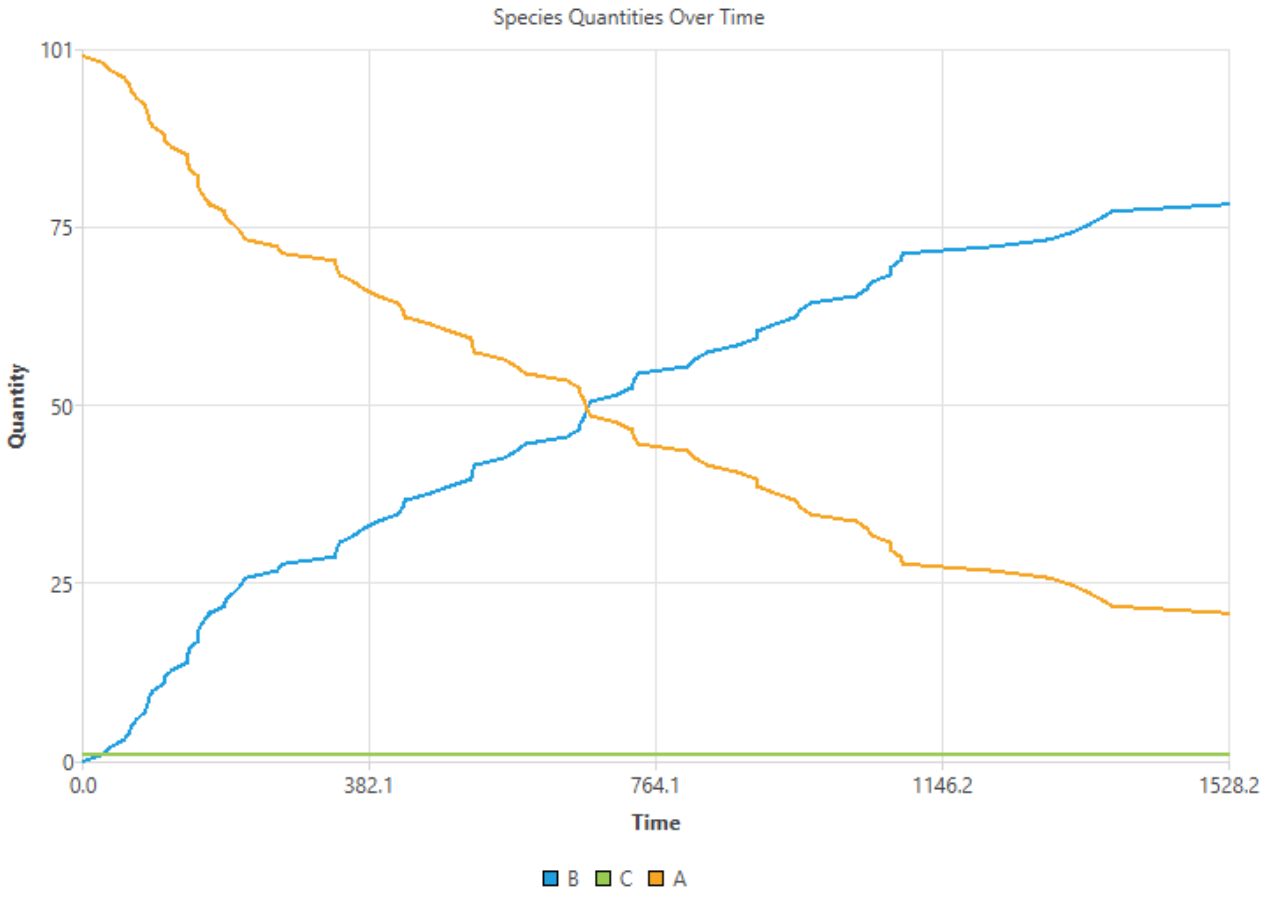

Figure 5: ./graphs/exp_dec_a.png

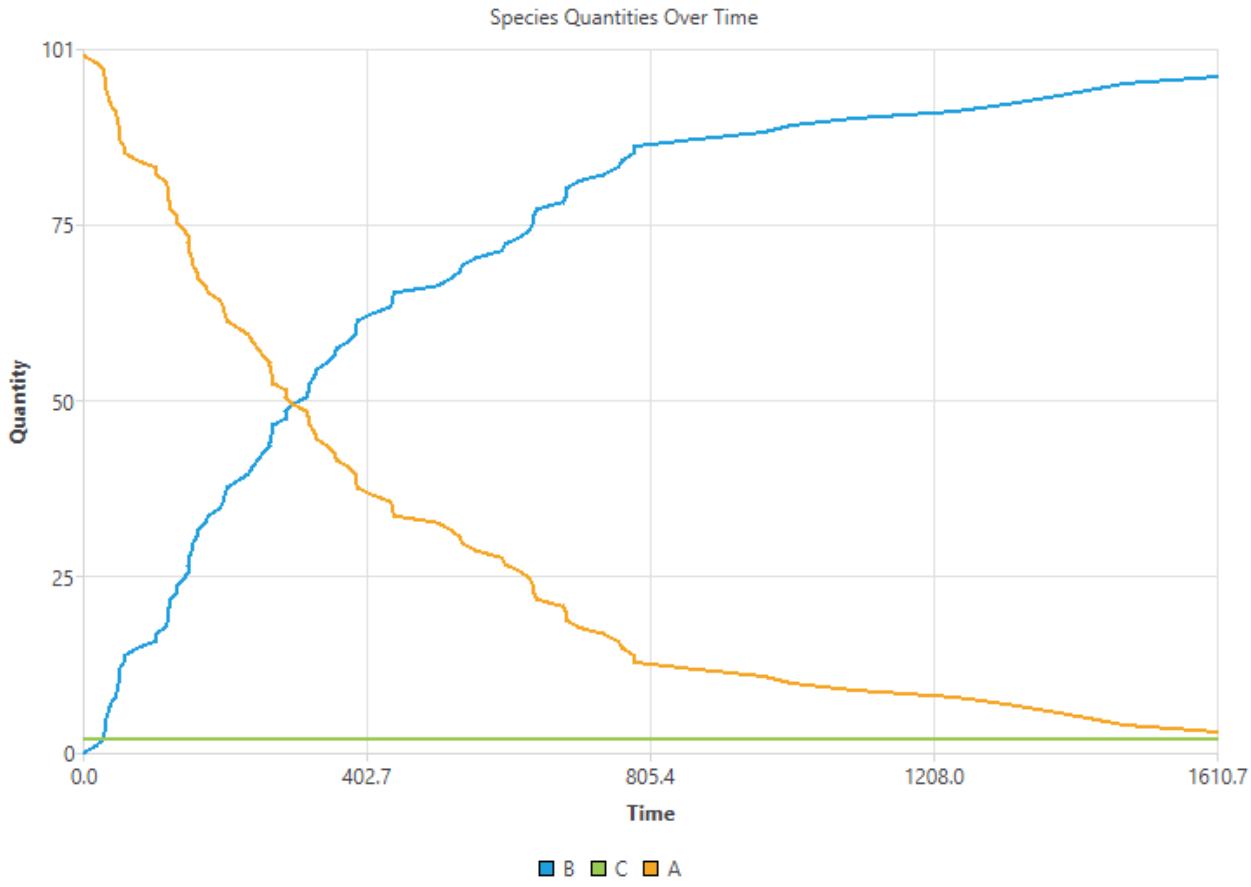

Figure 6: ./graphs/exp_dec_b.png

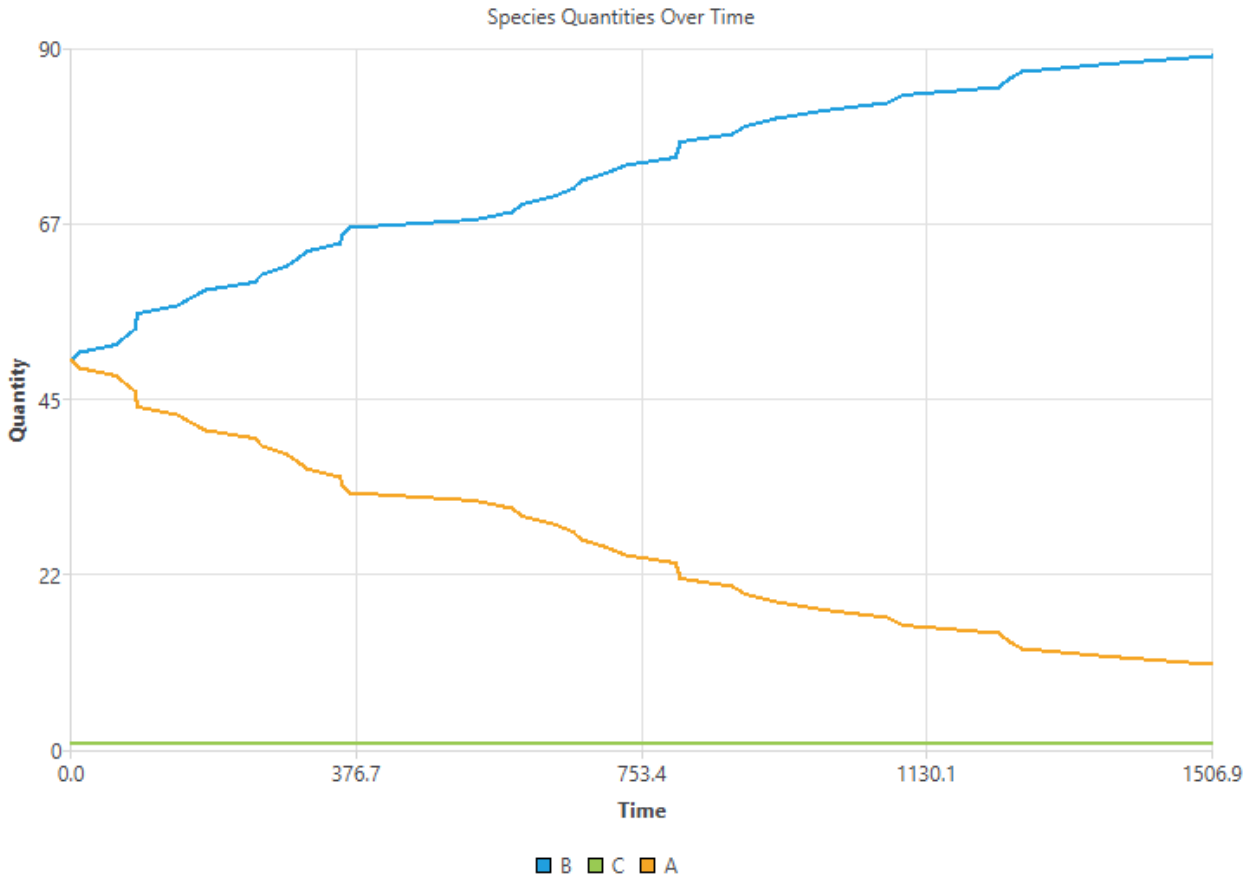

Figure 7: ./graphs/exp_dec_c.png