SP Exam Project

Mathias Andresen

May 13, 2021

Compiled with Visual Studio 2019 Community (version 16) in Clion. Using bundled CMake 3.17.5.

Listing 1: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.17)
project(sp_exam_project)

set(CMAKE_CXX_STANDARD 20)

add_library(
    stochastic-simulation
    library/simulation.cpp
    library/simulation.cpp
    library/SymbolTable.h
    library/simulation_monitor.h
    library/data.h
    library/data.cpp
    library/my-thread-pool.h
)

add_executable(sp_exam_project main.cpp vessels.h)

target_link_libraries(sp_exam_project PRIVATE stochastic-simulation)
```

Listing 2: main.cpp

```
#include <iostream>
#include "library/simulation.h"
#include <chrono>
#include "vessels.h"

using namespace StochasticSimulation;

// Requirement 7 use of monitor
class hospitalized_monitor: public simulation_monitor {
private:
    double_t hospitalized_acc{0.0};
    double_t last_time{0.0};
public:
    size_t max_hospitalized{0};

    void monitor(SimulationState &state) override {
        auto currently_hospitalized = state.reactants.get("H").amount;

        if (currently_hospitalized > max_hospitalized) {
            max_hospitalized = currently_hospitalized;
        }

        hospitalized_acc += (currently_hospitalized * (state.time - last_time));

        last_time = state.time;
    }
```

```cpp

    double_t get_mean_hospitalized() const {
        return (hospitalized_acc / last_time);
    }
};

void simulate_covid() {
    std::cout << "Simulating covid19 example with hospitalized monitor" << std::endl;
    Vessel covid_vessel = seihr(10000);

    std::cout << covid_vessel << std::endl;
    covid_vessel.visualize_reactions("covid_graph.png");

    std::cout << "reaction graph can be seen at: covid_graph.png" << std::endl;

    hospitalized_monitor monitor{};

    auto trajectory = covid_vessel.do_simulation(120, monitor);

    std::cout << "Simulation done" << std::endl;
    std::cout << "Max hospitalized: " << monitor.max_hospitalized << std::endl;
    std::cout << "Mean hospitalized: " << monitor.get_mean_hospitalized() << std::endl;

    std::cout << "Writing trajectory to csv file at covid_output.csv" << std::endl;
    trajectory->write_csv("covid_output.csv");
    std::cout << "Turn it into a graph using python ./draw_graph.py covid release" << std::endl;
}

void simulate_covid_multiple() {
    std::cout << "Simulating covid19 example 30 times and calculating mean" << std::endl;
    Vessel covid_vessel = seihr(10000);

    auto trajectories = covid_vessel.do_multiple_simulations(110, 100);

    std::cout << "Simulations done" << std::endl << "Computing mean trajectory" << std::endl;

    auto mean = SimulationTrajectory::compute_mean_trajectory(trajectories);

    std::cout << "Writing mean trajectory to csv file at covid_output_multiple.csv" << std::endl;
    mean.write_csv("covid_output_multiple.csv");
    std::cout << "Turn it into a graph using python ./draw_graph.py covid covid_output_multiple.csv" << std::endl;
}

void simulate_introduction() {
    std::cout << "Simulating introduction example" << std::endl;
    Vessel introduction_vessel = introduction(25, 50, 1, 0.001);
    std::cout << introduction_vessel << std::endl;

    introduction_vessel.visualize_reactions("intro_graph.png");

    auto trajectory = introduction_vessel.do_simulation(400);

    trajectory->write_csv("intro_output.csv");
}


void simulate_circadian() {
    std::cout << "Simulating circadian rhythm example..." << std::endl;
    Vessel oscillator = circadian_oscillator();

    std::cout << oscillator << std::endl;
    oscillator.visualize_reactions("cir_graph.png");

    auto trajectory = oscillator.do_simulation(110);

    std::cout << "Writing csv file..." << std::endl;
    trajectory->write_csv("circadian_output.csv");
}

void simulate_circadian2() {
    std::cout << "Simulating circadian rhythm alternative example..." << std::endl;
    Vessel oscillator = circadian_oscillator2();

    std::cout << oscillator << std::endl;
    oscillator.visualize_reactions("cir2_graph.png");

    auto trajectory = oscillator.do_simulation(110);

    std::cout << "Writing csv file..." << std::endl;
    trajectory->write_csv("circadian2_output.csv");
}

void benchmark() {
    std::cout << "Benchmarking with circadian rhythm example (max_time=100)" << std::endl;

    auto runs{30};

    Vessel oscillator = circadian_oscillator();

    unsigned long time_acc1{0};
    for (int i = 0; i < runs; ++i) {
        auto t0 = std::chrono::high_resolution_clock::now();
        oscillator.do_simulation(100);
        auto t1 = std::chrono::high_resolution_clock::now();

        time_acc1 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
    }
    auto mean_time1 = time_acc1 / runs;
    std::cout << "Simulation 1 mean time (nanoseconds): " << mean_time1 << std::endl;

    unsigned long time_acc2{0};
    for (int i = 0; i < runs; ++i) {
        auto t0 = std::chrono::high_resolution_clock::now();
        oscillator.do_simulation2(100);
        auto t1 = std::chrono::high_resolution_clock::now();

        time_acc2 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
    }
    auto mean_time2 = time_acc2 / runs;
    std::cout << "Simulation 2 mean time (nanoseconds): " << mean_time2 << std::endl;
}

int main() {
    simulate_covid();
//    simulate_covid_multiple();

//    simulate_introduction();
//    simulate_circadian();
//    simulate_circadian2();

//    benchmark();
```

```
}
```

Listing 3: vessels.h

```cpp
//
// Created by Mathias on 12-05-2021.
//

#ifndef SP_EXAM_PROJECT_VESSELS_H
#define SP_EXAM_PROJECT_VESSELS_H

#include "library/simulation.h"

using namespace StochasticSimulation;

Vessel seihr(uint32_t N)
{
    auto v = Vessel{};
    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; // basic reproductive number (initial, without lockdown etc)
    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; // recovery/death rate in hospital (H -> R) ~10.12 days

    // Reactants
    auto S = v("S", S0); // susceptible
    auto E = v("E", E0); // exposed
    auto I = v("I", I0); // infectious
    auto H = v("H", 0); // hospitalized
    auto R = v("R", 0); // removed/immune (recovered + dead)

    // Reactions
    v(S >>= E, I, beta/N);
    v(E >>= I, alpha);
    v(I >>= R, gamma);
    v(I >>= H, kappa);
    v(H >>= R, tau);

    return v;
}

Vessel introduction(uint32_t A_start, uint32_t B_Start, uint32_t D_amount, double_t lambda) {
    auto v = Vessel{};
    // Reactants
    auto A = v("A", A_start);
    auto B = v("B", B_Start);
    auto C = v("C", 0);
    auto D = v("D", D_amount);
    // Reactions
    v(A + B * 2 >>= C, D, lambda);

    return v;
}

/** direct encoding */
Vessel circadian_oscillator()
```

```
{
    auto alphaA = 50.0;
    auto alpha_A = 500.0;
    auto alphaR = 0.01;
    auto alpha_R = 50.0;
    auto betaA = 50.0;
    auto betaR = 5.0;
    auto gammaA = 1.0;
    auto gammaR = 1.0;
    auto gammaC = 2.0;
    auto deltaA = 1.0;
    auto deltaR = 0.2;
    auto deltaMA = 10.0;
    auto deltaMR = 0.5;
    auto thetaA = 50.0;
    auto thetaR = 100.0;
    auto v = Vessel{};
    auto env = v.environment();
    auto DA = v("DA", 1);
    auto D_A = v("D_A", 0);
    auto DR = v("DR", 1);
    auto D_R = v("D_R", 0);
    auto MA = v("MA", 0);
    auto MR = v("MR", 0);
    auto A = v("A", 0);
    auto R = v("R", 0);
    auto C = v("C", 0);
    v(A + DA >>= D_A, gammaA);
    v(D_A >>= DA + A, thetaA);
    v(A + DR >>= D_R, gammaR);
    v(D_R >>= DR + A, thetaR);
    v(D_A >>= MA + D_A, alpha_A);
    v(DA >>= MA + DA, alphaA);
    v(D_R >>= MR + D_R, alpha_R);
    v(DR >>= MR + DR, alphaR);
    v(MA >>= MA + A, betaA);
    v(MR >>= MR + R, betaR);
    v(A + R >>= C, gammaC);
    v(C >>= R, deltaA);
    v(A >>= env, deltaA);
    v(R >>= env, deltaR);
    v(MA >>= env, deltaMA);
    v(MR >>= env, deltaMR);
    return v;
}

/** alternative encoding using catalysts */
Vessel circadian_oscillator2()
{
    auto alphaA = 50.0;
    auto alpha_A = 500.0;
    auto alphaR = 0.01;
    auto alpha_R = 50.0;
    auto betaA = 50.0;
    auto betaR = 5.0;
    auto gammaA = 1.0;
    auto gammaR = 1.0;
    auto gammaC = 2.0;
    auto deltaA = 1.0;
    auto deltaR = 0.2;
    auto deltaMA = 10.0;
    auto deltaMR = 0.5;
    auto thetaA = 50.0;
    auto thetaR = 100.0;
    auto v = Vessel{};
    auto env = v.environment();
    auto DA = v("DA", 1);
    auto D_A = v("D_A", 0);
    auto DR = v("DR", 1);
    auto D_R = v("D_R", 0);
    auto MA = v("MA", 0);
    auto MR = v("MR", 0);
    auto A = v("A", 0);
    auto R = v("R", 0);
    auto C = v("C", 0);
    v(A + DA >>= D_A, gammaA);
    v(D_A >>= DA + A, thetaA);
    v(DR + A >>= D_R, gammaR);
    v(D_R >>= DR + A, thetaR);
    v(env >>= MA, D_A, alpha_A);
    v(env >>= MA, DA, alphaA);
    v(env >>= MR, D_R, alpha_R);
    v(env >>= MR, DR, alphaR);
    v(env >>= A, MA, betaA);
    v(env >>= R, MR, betaR);
    v(A + R >>= C, gammaC);
    v(C >>= R, deltaA);
    v(A >>= env, deltaA);
    v(R >>= env, deltaR);
    v(MA >>= env, deltaMA);
    v(MR >>= env, deltaMR);
    return v;
}

#endif //SP_EXAM_PROJECT_VESSELS_H
```

Listing 4: simulation.h

```
//
// Created by Mathias on 09-05-2021.
//

#ifndef SP_EXAM_PROJECT_SIMULATION_H
#define SP_EXAM_PROJECT_SIMULATION_H

#include <string>
#include <utility>
#include <vector>
#include <set>
#include <optional>
#include <map>
#include <numeric>
#include <random>
#include <algorithm>
#include <functional>
#include <sstream>
#include <fstream>
#include <chrono>
#include <thread>
#include <future>
#include <ranges>
#include "SymbolTable.h"
#include "simulation_monitor.h"
```

```
#include "data.h"

namespace StochasticSimulation {

    using map_type = std::map<double_t, SimulationState>;
    class SimulationTrajectory: public map_type {
    private:
        double_t largest_time{-1};
        static double_t compute_interpolated_value(
                const std::string& key,
                SimulationState& s0,
                SimulationState& s1,
                double_t x);
    public:
        using map_type::map;

        SimulationTrajectory(const SimulationTrajectory& val): map_type(val) {
            largest_time = val.largest_time;
        }

        SimulationTrajectory(SimulationTrajectory&& rval): map_type(std::move(rval)) {
            largest_time = std::move(rval.largest_time);
        };

        SimulationTrajectory& operator=(const SimulationTrajectory & val) {
            map_type::operator=(val);
            largest_time = val.largest_time;
        };

        SimulationTrajectory& operator=(SimulationTrajectory&& rval) {
            map_type::operator=(std::move(rval));
            largest_time = std::move(rval.largest_time);
        };

        // Requirement 9 compute mean
        static SimulationTrajectory compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories);

        void insert(SimulationState state) {
            if (state.time > largest_time) {
                largest_time = state.time;
            }

            map_type::insert({state.time, std::move(state)});
        }

        // Requirement 6 output trajectory
        void write_csv(const std::string& path);

        double_t get_max_time() {
            return largest_time;
        }
    };

    // Requirement 1 operators for DSEL
    class Vessel {
    private:
        std::vector<Reaction> reactions{};
        SymbolTable<Reactant> reactants;
    public:

        Vessel() = default;

        Vessel(const Vessel &val) {
            reactions = val.reactions;
            reactants = val.reactants;
        }

        Vessel (Vessel&& rval) {
            reactions = std::move(rval.reactions);
            reactants = std::move(rval.reactants);
        };

        Reactant& operator()(std::string name, size_t initial_amount) {
            Reactant newReactant{std::move(name), initial_amount};

            reactants.put(newReactant.name, newReactant);

            return reactants.get(newReactant.name);
        }

        Reaction operator()(Reaction&& reaction, double_t rate) {
            reaction.rate = rate;

            reactions.push_back(reaction);

            return reaction;
        }

        Reaction operator()(Reaction&& reaction, std::initializer_list<Reactant> catalysts,   ↙
↪double rate) {
            reaction.rate = rate;
            reaction.catalysts = catalysts;

            // Add to vessel reactions
            reactions.push_back(reaction);

            return reaction;
        }

        Reaction operator()(Reaction&& reaction, Reactant catalyst, double_t rate) {
            reaction.rate = rate;
            reaction.catalysts = {catalyst};

            // Add to vessel reactions
            reactions.push_back(reaction);

            return reaction;
        }


        Reactant& environment() {
            if (reactants.contains("__env__")) {
               return reactants.get("__env__");
            }

            auto newReactant = Reactant("__env__", 0, 0);

            reactants.put(newReactant.name, newReactant);

            return reactants.get(newReactant.name);
        }

        // Requirement 2 network graph
        void visualize_reactions(const std::string& filename);

        // Requirement 10 optimized algorithm
        std::shared_ptr<SimulationTrajectory> do_simulation2(double_t end_time, simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);

        // Requirement 4 simulation
        std::shared_ptr<SimulationTrajectory> do_simulation(double_t end_time, simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);

        // Requirement 8 parallelization
        std::vector<std::shared_ptr<SimulationTrajectory>> do_multiple_simulations(double_t end_time, size_t simulations_to_run);

        // Requirement 2 pretty print
        friend std::ostream& operator<<(std::ostream& s, const Vessel& vessel);
    };



}

#endif //SP_EXAM_PROJECT_SIMULATION_H
```

Listing 5: simulation.cpp

```
//
// Created by Mathias on 09-05-2021.
//

#include <iostream>
#include <utility>
#include "simulation.h"

namespace StochasticSimulation {


    // Requirement 2
    std::ostream &operator<<(std::ostream &s, const Vessel &vessel) {
        s << "{" << std::endl;
        for (const auto& reaction: vessel.reactions) {
            s << "\t" << reaction;
            if (&reaction != &vessel.reactions.back()) {
                s << ",";
            }
            s << std::endl;
        }
        return s << "}";
    }

    // Requirement 2
    void Vessel::visualize_reactions(const std::string& filename) {
        std::stringstream str;
        SymbolTable<std::string> node_map{};

        str << "digraph {" << std::endl;

        auto i = 0;
        for (auto& reactant: reactants.getMap()) {
            if (reactant.second.name != "__env__") {
                node_map.put(reactant.second.name, "s" + std::to_string(i));

                str << node_map.get(reactant.second.name)
                    << "[label=\"" << reactant.second.name << "\",shape=\"box\",style=\"filled\",fillcolor=\"cyan\"];" << std::endl;
                i++;
            }
        }

        i = 0;
        for (auto& reaction: reactions) {
            std::string reaction_node{"r" + std::to_string(i)};

            str << reaction_node << "[label=\"" << reaction.rate << "\",shape=\"oval\",style=\"filled\",fillcolor=\"yellow\"];" << std::endl;
            if (reaction.catalysts.has_value()) {
                for (auto& catalyst: reaction.catalysts.value()) {
                    str << node_map.get(catalyst.name) << " -> " << reaction_node << "[arrowhead=\"tee\"];" << std::endl;
                }
            }
            for (auto& reactant: reaction.from) {
                if (reactant.name != "__env__") {
                    str << node_map.get(reactant.name) << " -> " << reaction_node << ";" << std::endl;
                }
            }
            for (auto& product: reaction.to) {
                if (product.name != "__env__") {
                    str << reaction_node << " -> " << node_map.get(product.name) << ";" << std::endl;
                }
            }

            i++;
        }

        str << "}";

        std::ofstream dotfile;
        dotfile.open(filename + ".dot");
        dotfile << str.str();
        dotfile.close();

        std::stringstream command_builder;
        command_builder << "dot -Tpng -o " << filename << " " << filename << ".dot";
        system(command_builder.str().c_str());
    }

    // Requirement 10 alternative simulation
    std::shared_ptr<SimulationTrajectory> Vessel::do_simulation2(double_t end_time, simulation_monitor &monitor) {
        SimulationTrajectory trajectory{};
        double_t t{0};

        auto thread_id = std::this_thread::get_id();
        auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
        std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));

        // Insert initial state
        trajectory.insert(SimulationState{reactants, t});

        while (t <= end_time) {
            for (Reaction& reaction: reactions) {
                // New: using new compute delay function
                reaction.compute_delay2(trajectory.at(t), engine);
            }

            auto r = reactions.front();

            // Select Reaction with min delay which is not -1
            for (auto& reaction: reactions) {
                if (reaction.delay == -1) {
                    continue;
                } else if (reaction.delay < r.delay) {
                    r = reaction;
                } else if (r.delay == -1) {
                    r = reaction;
                }
            }

            // Stop if we have no reactions to do, thus r.delay == -1
            if (r.delay == -1) {
                break;
            }

            auto& last_state = trajectory.at(t);

            t += r.delay;

            SimulationState state{last_state.reactants, t};

            if (
                    std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;}) &&
                    (
                            !r.catalysts.has_value() ||
                            std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(), [&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
                    )
                    ) {
                for (auto& reactant: r.from) {
                    state.reactants.get(reactant.name).amount -= reactant.required;
                }
                for (auto& reactant: r.to) {
                    state.reactants.get(reactant.name).amount += reactant.required;
                }
            }

            trajectory.insert(std::move(state));

            monitor.monitor(trajectory.at(t));
        }

        return std::make_shared<SimulationTrajectory>(std::move(trajectory));
    }

    // Requirement 4 simulation
    std::shared_ptr<SimulationTrajectory> Vessel::do_simulation(double_t end_time, simulation_monitor &monitor) {
        SimulationTrajectory trajectory{};
        double_t t{0};
```

```cpp
        auto thread_id = std::this_thread::get_id();
        auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
        std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));

        // Insert initial state
        trajectory.insert(SimulationState{reactants, t});

        while (t <= end_time) {
            for (Reaction& reaction: reactions) {
                reaction.compute_delay(trajectory.at(t), engine);
            }

            auto r = reactions.front();

            // Select Reaction with min delay which is not -1
            for (auto& reaction: reactions) {
                if (reaction.delay == -1) {
                    continue;
                } else if (reaction.delay < r.delay) {
                    r = reaction;
                } else if (r.delay == -1 && reaction.delay != -1) {
                    r = reaction;
                }
            }

            // Stop if we have no reactions to do, thus r.delay == -1
            if (r.delay == -1) {
                break;
            }

            auto& last_state = trajectory.at(t);

            t += r.delay;

            SimulationState state{last_state.reactants, t};

            if (
                    std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;}) &&
                    (
                            !r.catalysts.has_value() ||
                            std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(), [&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
                    )
                    ) {
                for (auto& reactant: r.from) {
                    state.reactants.get(reactant.name).amount -= reactant.required;
                }
                for (auto& reactant: r.to) {
                    state.reactants.get(reactant.name).amount += reactant.required;
                }
            }

            trajectory.insert(std::move(state));

            monitor.monitor(trajectory.at(t));
        }

        return std::make_shared<SimulationTrajectory>(std::move(trajectory));
    }

    // Requirement 8 multiple at same time
    std::vector<std::shared_ptr<SimulationTrajectory>>
    Vessel::do_multiple_simulations(double_t end_time, size_t simulations_to_run) {
        std::vector<std::shared_ptr<SimulationTrajectory>> result{};
        result.reserve(simulations_to_run);

        auto cores = std::thread::hardware_concurrency();
        int jobs = std::min(simulations_to_run, (cores - 1));
        auto simulations_per_job = simulations_to_run / jobs;

        auto futures = std::vector<std::future<std::vector<std::shared_ptr<SimulationTrajectory>>>>{};

        auto lambda = [&vessel = *this, &end_time](size_t to_run){
            auto simulations = std::vector<std::shared_ptr<SimulationTrajectory>>{};
            simulations.reserve(to_run);

            auto new_vessel = Vessel(vessel);

            for (int i = 0; i < to_run; ++i) {
                simulations.push_back(new_vessel.do_simulation(end_time));
            }

            return simulations;
        };

        for (int i = 0; i < jobs; ++i) {
            futures.push_back(std::async(std::launch::async, lambda, simulations_per_job));
        }
        auto missing_simulations = simulations_to_run - (simulations_per_job * jobs);
        if (missing_simulations != 0) {
            futures.push_back(std::async(std::launch::async, lambda, missing_simulations));
        }

        for (auto& future: futures) {
            auto future_result = future.get();
            for (auto& res: future_result) {
                result.push_back(std::move(res));
            }
        }

        return result;
    }

    // Private function for calculation interpolated value
    double_t SimulationTrajectory::compute_interpolated_value(const std::string& key, SimulationState& s0, SimulationState& s1, double_t x) {
        return
            s0.reactants.get(key).amount
            + ((
                   ( (double_t) s1.reactants.get(key).amount - (double_t) s0.reactants.get(key).amount) /
                   ( s1.time - s0.time )
               ) * (x - s0.time));
    }

    // Requirement 9 compute mean trajectory
    SimulationTrajectory SimulationTrajectory::compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories) {
        auto average_delay = trajectories.front()->get_max_time() / trajectories.front()->size();

        // Get a list of all keys
        std::vector<std::string> reactant_keys{};
        for (auto& reactant: trajectories.front()->begin()->second.reactants) {
            reactant_keys.push_back(reactant.second.name);
        }

        // Find upper bound for mean trajectory
        double_t upper_bound{-1.0};
        for (auto& trajectory: trajectories) {
            if (upper_bound == -1.0 || trajectory->get_max_time() < upper_bound) {
                upper_bound = trajectory->get_max_time();
            }
        }

        SimulationTrajectory mean_trajectory{};

        for (auto& trajectory: trajectories) {
            auto iterator = trajectory->begin();
            SimulationState& s0 = iterator->second;
            iterator++;
            SimulationState& s1 = iterator->second;

            double_t t{0};

            while((t + average_delay) <= upper_bound) {
                if (t >= s0.time) {
                    if (t <= s1.time) {
                        if (!mean_trajectory.contains(t)) {
                            mean_trajectory.insert((SimulationState{{}, t}));
                        }

                        for (auto& key: reactant_keys) {
                            auto interpolated_value =  ⤦
↪SimulationTrajectory::compute_interpolated_value(key, s0, s1, t);

                            auto& table = mean_trajectory.at(t).reactants;

                            if (!table.contains(key)) {
                                Reactant reactant{key, 0.0};
                                table.put(key, reactant);
                            }

                            table.get(key).amount += interpolated_value;
                        }

                        t += average_delay;
                    } else {
                        s0 = s1;
                        iterator++;

                        if (iterator != trajectory->end()) {
                            s1 = iterator->second;
                        } else {
                            break;
                        }
                    }
                }
            }
        }

        for (double_t i = 0; (i + average_delay) <= upper_bound; i += average_delay) {
            for (auto& key: reactant_keys) {
                mean_trajectory.at(i).reactants.get(key).amount /= trajectories.size();
            }
        }

        return std::move(mean_trajectory);
    }

    // Requirement 6 output to csv which can then be turned into a graph via python script
    void SimulationTrajectory::write_csv(const std::string &path) {
        std::ofstream csv_file;
        csv_file.open(path);

        auto reactants = at(0).reactants;

        for (auto& reactant : reactants) {
            csv_file << reactant.second.name << ",";
        }
        csv_file << "time" << std::endl;

        for (auto& state : *this) {
            for (auto& reactant: reactants) {
                csv_file << state.second.reactants.get(reactant.second.name).amount << ",";
            }
            csv_file << state.second.time << std::endl;
        }

        csv_file.close();
    }
}
```

Listing 6: data.h

```
//
// Created by Mathias on 11-05-2021.
//

#ifndef SP_EXAM_PROJECT_DATA_H
#define SP_EXAM_PROJECT_DATA_H

namespace StochasticSimulation {
    class SimulationState;
    struct Reaction;
    class ReactantCollection;

    struct Reactant {
        std::string name;
        double_t amount; // double to allow for mean values
        size_t required{1};

        Reactant(std::string name, size_t initial_amount):
                name(std::move(name)),
                amount(initial_amount)
        {}

        Reactant(std::string name, double_t initial_amount):
                name(std::move(name)),
                amount(initial_amount)
        {}

```

```cpp
        Reactant(std::string name, size_t initial_amount, size_t required):
                name(std::move(name)),
                amount(initial_amount),
                required(required)
        {}

        ~Reactant() = default;

        Reactant(const Reactant& a) {
            name = a.name;
            amount = a.amount;
            required = a.required;
        }

        Reactant(Reactant&& a) {
            name = std::move(a.name);
            amount = std::move(a.amount);
            required = std::move(a.required);
        }

        Reactant& operator=(Reactant&& a) {
            name = std::move(a.name);
            amount = std::move(a.amount);
            required = std::move(a.required);

            return *this;
        }

        Reactant& operator=(const Reactant& a) {
            name = a.name;
            amount = a.amount;
            required = a.required;

            return *this;
        }

        // Requirement 1 operator for DSEL
        Reaction operator>>=(Reactant other);

        // Requirement 1 operator for DSEL
        Reaction operator>>=(ReactantCollection other);

        // Requirement 1 operator for DSEL
        ReactantCollection operator+(const Reactant& other);

        bool operator<(const Reactant& other) const;

        // Requirement 1 operator for DSEL
        Reactant operator*(size_t req) {
            required = req;
            return *this;
        }

    };

    class ReactantCollection: public std::set<Reactant> {
    public:
        using std::set<Reactant>::set;
        // Requirement 1 operator for DSEL
        Reaction operator>>=(Reactant other);
        // Requirement 1 operator for DSEL
        Reaction operator>>=(ReactantCollection other);
    };

    class Reaction {
    public:
        std::set<Reactant> from;
        std::set<Reactant> to;
        std::optional<std::vector<Reactant>> catalysts;
        double_t rate{};
        double_t delay{-1};

        Reaction(std::set<Reactant> from, std::set<Reactant> to):
                from(from),
                to(to)
        {}

        Reaction(std::set<Reactant> from, std::set<Reactant> to, std::initializer_list<Reactant> catalysts, double rate):
                from(from),
                to(to),
                catalysts(catalysts),
                rate(rate)
        {}

        Reaction(std::set<Reactant> from, std::set<Reactant> to, double rate):
                from(from),
                to(to),
                catalysts{},
                rate(rate)
        {}

        void compute_delay(SimulationState& state, std::default_random_engine& engine);
        void compute_delay2(SimulationState& state, std::default_random_engine& engine);

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

    struct SimulationState {
    public:
        SymbolTable<Reactant> reactants;
        double_t time;

        SimulationState(SymbolTable<Reactant> reactants, double_t time):
            reactants{reactants},
            time{time}
        {};

        SimulationState(const SimulationState&) = default;
        SimulationState(SimulationState&&) = default;

        SimulationState& operator=(const SimulationState &) = default;
        SimulationState& operator=(SimulationState&&) = default;

        ~SimulationState() = default;

        friend std::ostream &operator<<(std::ostream &, const SimulationState &);
    };

}

#endif //SP_EXAM_PROJECT_DATA_H
```

Listing 7: data.cpp

```cpp
//
// Created by Mathias on 11-05-2021.
//

#include <iostream>
#include <utility>
#include "simulation.h"

namespace StochasticSimulation {
    Reaction Reactant::operator>>=(StochasticSimulation::Reactant other) {
        return Reaction{{*this}, {std::move(other)}};
    }

    Reaction Reactant::operator>>=(ReactantCollection other) {
        return Reaction{{*this}, std::move(other)};
    }

    ReactantCollection Reactant::operator+(const Reactant& other) {
        return ReactantCollection{*this, other};
    }

    bool Reactant::operator<(const Reactant& other) const {
        return name < other.name;
    }

    Reaction ReactantCollection::operator>>=(Reactant other) {
        return Reaction{*this, {std::move(other)}};
    }

    Reaction ReactantCollection::operator>>=(ReactantCollection other) {
        return Reaction{*this, std::move(other)};
    }

    std::ostream &operator<<(std::ostream &s, const Reaction &reaction) {
        s << "{ ";
        for (const auto& reactant: reaction.from) {
            s << reactant.name << "+";
        }
        s << "\b" << " >>= ";
        if (reaction.catalysts.has_value()) {
            s << "(";
            for (const auto& catalyst: reaction.catalysts.value()) {
                s << catalyst.name << "+";
            }
            s << "\b" << ") ";
        }
        for (const auto& reactant: reaction.to) {
            s << reactant.name << "+";
        }
        s << "\b";

        return s << " - " << reaction.rate << " }";
    }

    void Reaction::compute_delay2(SimulationState& state, std::default_random_engine &engine) {
        size_t reactant_amount{1};
        size_t catalyst_amount{1};

        for (const Reactant& reactant: from) {
            auto amount = reactant.name == "__env__" ? 1 :   ↙
↪state.reactants.get(reactant.name).amount;
            reactant_amount *= amount;
        }
        // New: check if amount is 0 already
        if (reactant_amount == 0) {
            delay = -1;
            return;
        }

        if (catalysts.has_value()) {
            for (auto& catalyst: catalysts.value()) {
                catalyst_amount *= state.reactants.get(catalyst.name).amount;
            }
        }

        double_t rate_k = rate * reactant_amount * catalyst_amount;

        if (rate_k > 0) {
            delay = std::exponential_distribution<double_t>(rate_k)(engine);
        } else {
            delay = -1;
        }
    }

    void Reaction::compute_delay(SimulationState& state, std::default_random_engine &engine) {
        size_t reactant_amount{1};
        size_t catalyst_amount{1};

        for (const Reactant& reactant: from) {
            auto amount = reactant.name == "__env__" ? 1 :   ↙
↪state.reactants.get(reactant.name).amount;
            reactant_amount *= amount;
        }
        if (catalysts.has_value()) {
            for (auto& catalyst: catalysts.value()) {
                catalyst_amount *= state.reactants.get(catalyst.name).amount;
            }
        }

        double_t rate_k = rate * reactant_amount * catalyst_amount;

        if (rate_k > 0) {
            delay = std::exponential_distribution<double_t>(rate_k)(engine);
        } else {
            delay = -1;
        }
    }

    std::ostream &operator<<(std::ostream &s, const SimulationState& state) {
        s << "{" << std::endl
            << "time: " << state.time << "," << std::endl
            << "reactants: {" << std::endl;
        for(auto& pair: state.reactants) {
            s << pair.first << ": " << pair.second.amount << "," << std::endl;
        }

        s << "}";
        return s;
    }
}
```

Listing 8: simulation_monitor.h

```
//
// Created by Mathias on 11-05-2021.
//

#ifndef SP_EXAM_PROJECT_SIMULATION_MONITOR_H
#define SP_EXAM_PROJECT_SIMULATION_MONITOR_H

#include <functional>
#include "data.h"

namespace StochasticSimulation {

    // Requirement 7 classes for generic system state monitors
    class simulation_monitor {
    public:
        virtual void monitor(SimulationState& state) = 0;
    };


    class empty_simulation_monitor: public simulation_monitor {
        void monitor(SimulationState &state) override {
          return;
        };
    };

    static auto EMPTY_SIMULATION_MONITOR = empty_simulation_monitor{};

    class basic_simulation_monitor: public simulation_monitor {
    private:
        const std::function<void(SimulationState&)> monitor_function;
    public:
        basic_simulation_monitor(const std::function<void(SimulationState&)>& monitor_func):
            simulation_monitor{},
            monitor_function{monitor_func}
        {}

        void monitor(SimulationState& state) override {
            monitor_function(state);
        }
    };
}

#endif //SP_EXAM_PROJECT_SIMULATION_MONITOR_H
```

Listing 9: SymbolTable.h

```
//
// Created by Mathias on 10-05-2021.
//

#ifndef SP_EXAM_PROJECT_SYMBOLTABLE_H
#define SP_EXAM_PROJECT_SYMBOLTABLE_H

#include <unordered_map>
#include <string>
#include <stdexcept>
#include <utility>
#include <iterator>

namespace StochasticSimulation {
    // Requirement 3 generic symbol table
    struct SymbolTableException : public std::exception
    {
        std::string message;
    public:
        explicit SymbolTableException(std::string message): message(std::move(message))
        {}

        [[nodiscard]] const char* what() const override
        {
            return message.c_str();
        }
    };

    template<typename T>
    class SymbolTable {
        using map_type = std::unordered_map<std::string, T>;
    private:
        map_type map{};
    public:
        using iterator = typename map_type::iterator;
        using const_iterator = typename map_type::const_iterator;

        SymbolTable<T>() = default;

        SymbolTable<T>(const SymbolTable<T>& a) {
            map = a.map;
        };

        SymbolTable<T>(SymbolTable<T>&& a) {
            map = std::move(a.map);
        };

        ~SymbolTable() = default;

        SymbolTable<T>& operator=(const SymbolTable& a) {
            map = a.map;
            return *this;
        };

        SymbolTable<T>& operator=(SymbolTable&& a) {
            map = std::move(a.map);
            return *this;
        };

        void put(const std::string& key, T value) {
            if (!map.contains(key)) {
                map.insert_or_assign(key, value);
            } else {
                throw SymbolTableException("Key " + key + " already used");
            }
        }

        T& get(const std::string& key) {
            try {
                return map.at(key);
            } catch (std::out_of_range& e) {
                throw SymbolTableException("Key " + key + " was not found");
            }
        }

        const T& get(const std::string& key) const {
            try {
                return map.at(key);
            } catch (std::out_of_range& e) {
                throw SymbolTableException("Key " + key + " was not found");
            }
        }

        bool contains(const std::string& key) {
            return map.contains(key);
        }

        std::unordered_map<std::string, T> getMap() {
            return map;
        }

        iterator begin() {
            return map.begin();
        }

        iterator end() {
            return map.end();
        }

        const_iterator begin() const {
            return map.begin();
        }

        const_iterator end() const {
            return map.end();
        }

    };
}

#endif //SP_EXAM_PROJECT_SYMBOLTABLE_H
```

Listing 10: Results

```
Pretty-print Reactions:

Introduction:
{
     { A+B >>= (D) C - 0.001 }
}

Covid:
{
          { S >>= (I) E - 7.74194e-05 },
          { E >>= I - 0.196078 },
          { I >>= R - 0.322581 },
          { I >>= H - 0.000290061 },
          { H >>= R - 0.0988142 }
}

Carcadian Rythm:
{
          { A+DA >>= D_A - 1 },
          { D_A >>= A+DA - 50 },
          { A+DR >>= D_R - 1 },
          { D_R >>= A+DR - 100 },
          { D_A >>= D_A+MA - 500 },
          { DA >>= DA+MA - 50 },
          { D_R >>= D_R+MR - 50 },
          { DR >>= DR+MR - 0.01 },
          { MA >>= A+MA - 50 },
          { MR >>= MR+R - 5 },
          { A+R >>= C - 2 },
          { C >>= R - 1 },
          { A >>= __env__ - 1 },
          { R >>= __env__ - 0.2 },
          { MA >>= __env__ - 10 },
          { MR >>= __env__ - 0.5 }
}

Carcadian Rythm alternative:
{
          { A+DA >>= D_A - 1 },
          { D_A >>= A+DA - 50 },
          { A+DR >>= D_R - 1 },
          { D_R >>= A+DR - 100 },
          { __env__ >>= (D_A) MA - 500 },
          { __env__ >>= (DA) MA - 50 },
          { __env__ >>= (D_R) MR - 50 },
          { __env__ >>= (DR) MR - 0.01 },
          { __env__ >>= (MA) A - 50 },
          { __env__ >>= (MR) R - 5 },
          { A+R >>= C - 2 },
          { C >>= R - 1 },
          { A >>= __env__ - 1 },
          { R >>= __env__ - 0.2 },
          { MA >>= __env__ - 10 },
          { MR >>= __env__ - 0.5 }
}


Example output from monitoring hospitalized (not the one on the graph)
Max hospitalized: 3
Mean hospitalized: 0.551814


Benchmarks:
Benchmarking with circadian rhythm example (max_time=100) (30 times each)
Simulation 1 mean time (nanoseconds): 117913398
Simulation 2 mean time (nanoseconds): 17683541
```

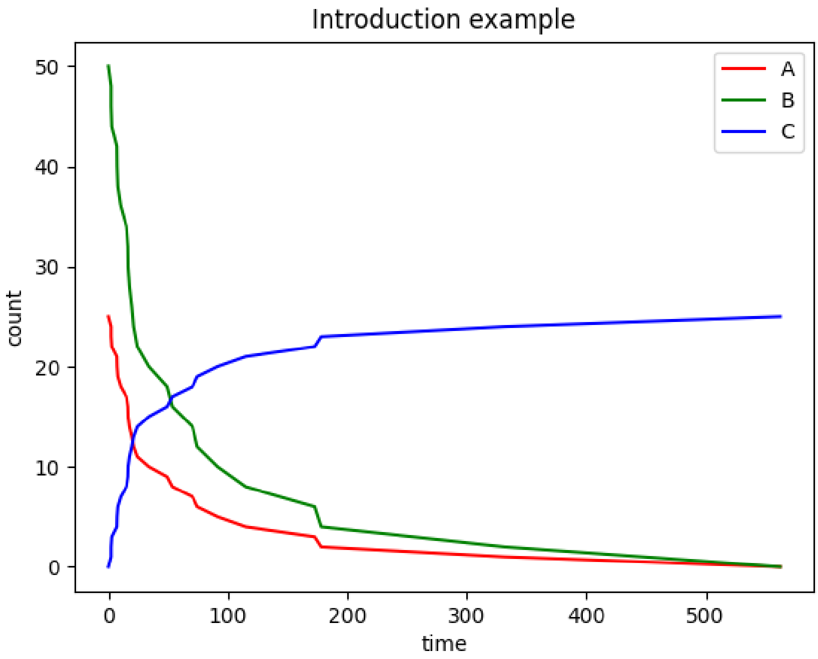

Figure 1: A(0)=25, B(0)=50, D=1

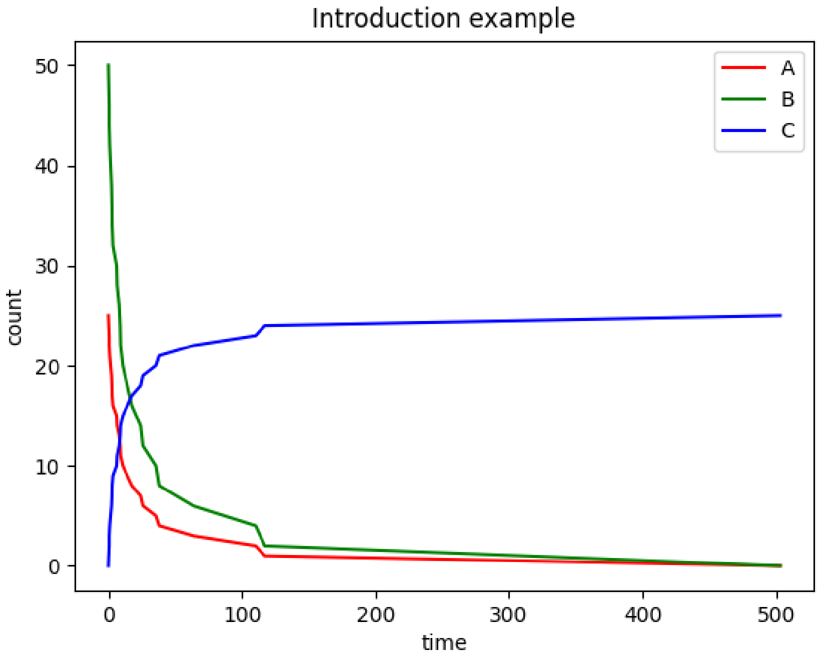

Figure 2: A(0)=25, B(0)=50, D=2

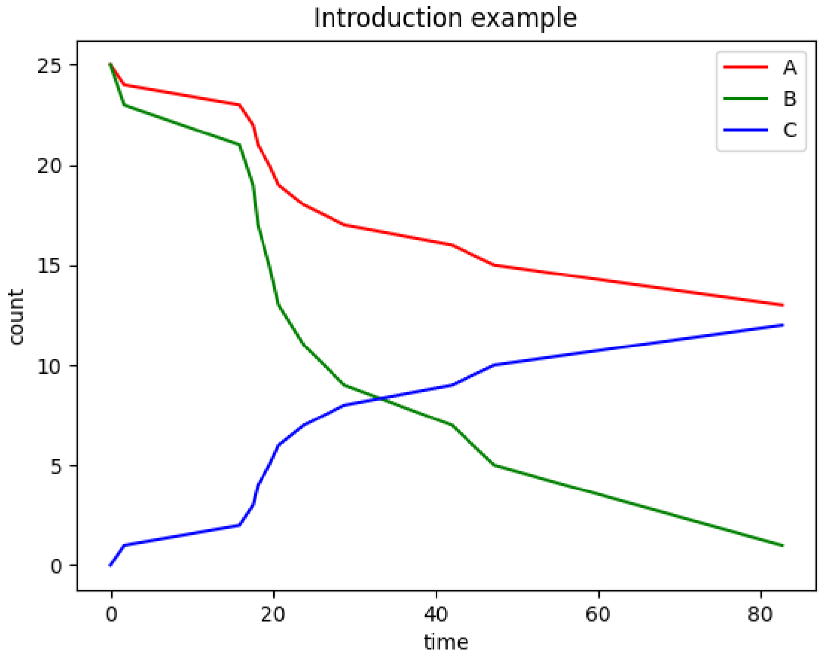

Figure 3: A(0)=25, B(0)=25, D=1

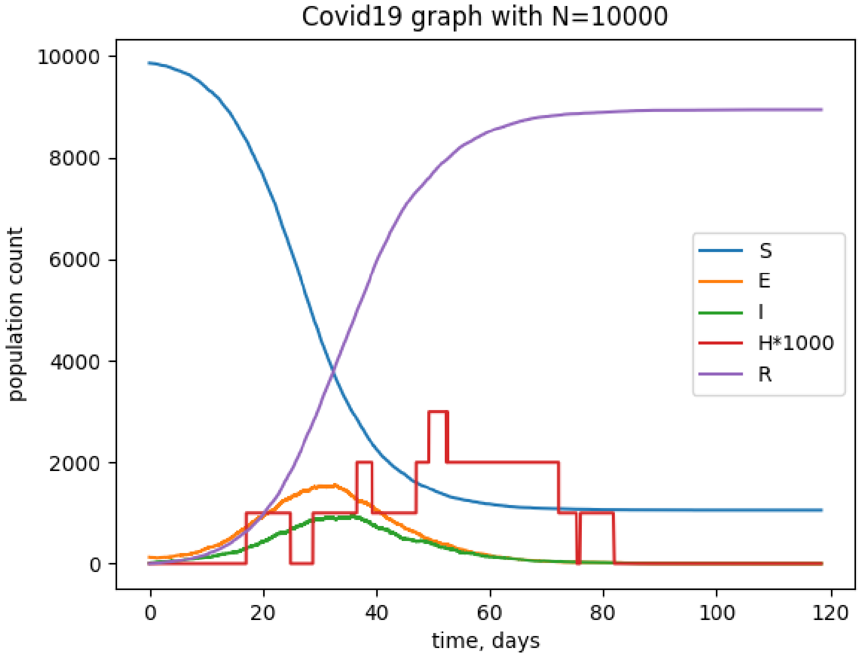

Figure 4: Sample covid trajectory

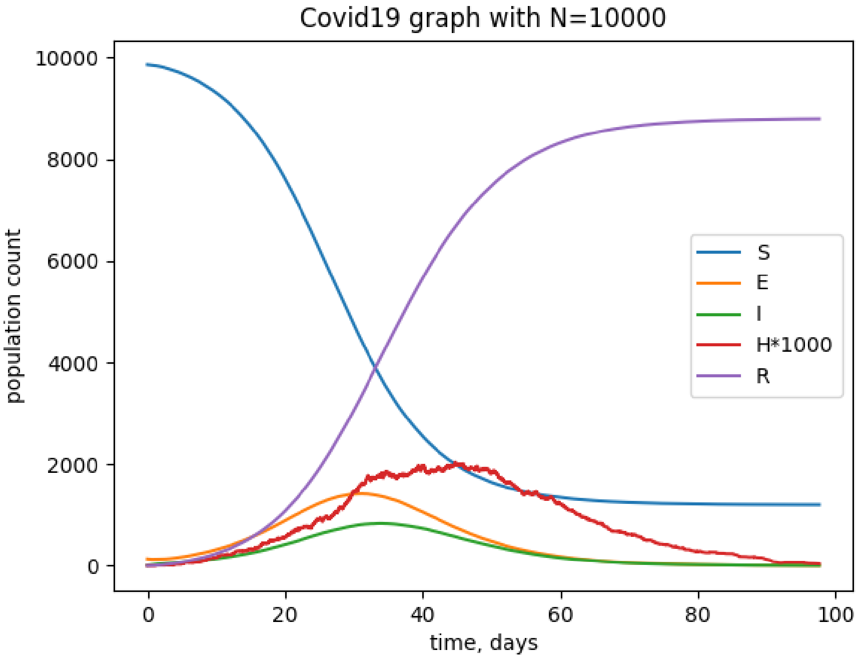

Figure 5: Mean covid trajectory of 30 simulations

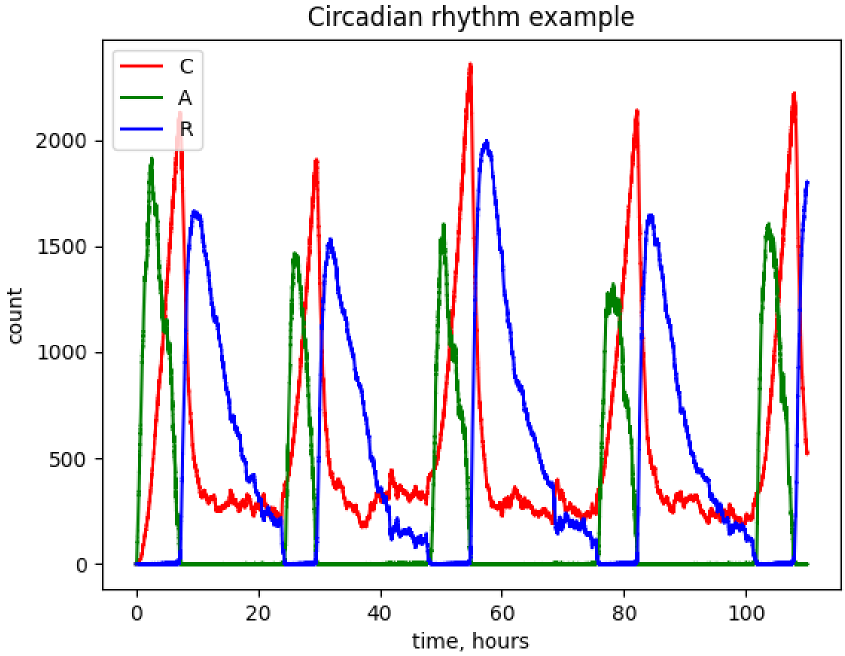

Figure 6: Sample circadian rythm trajectory

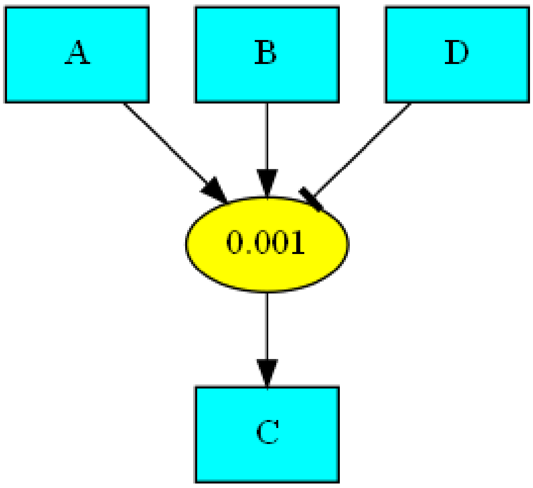

Figure 7: Intro reaction graph

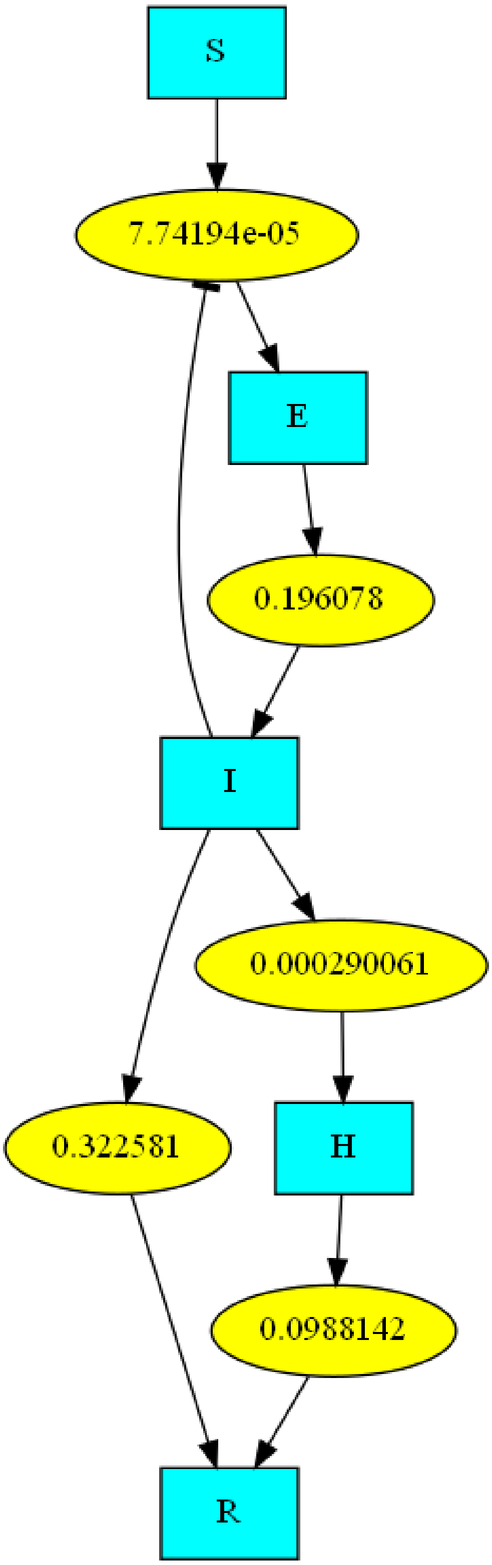

Figure 8: Covid reaction graph

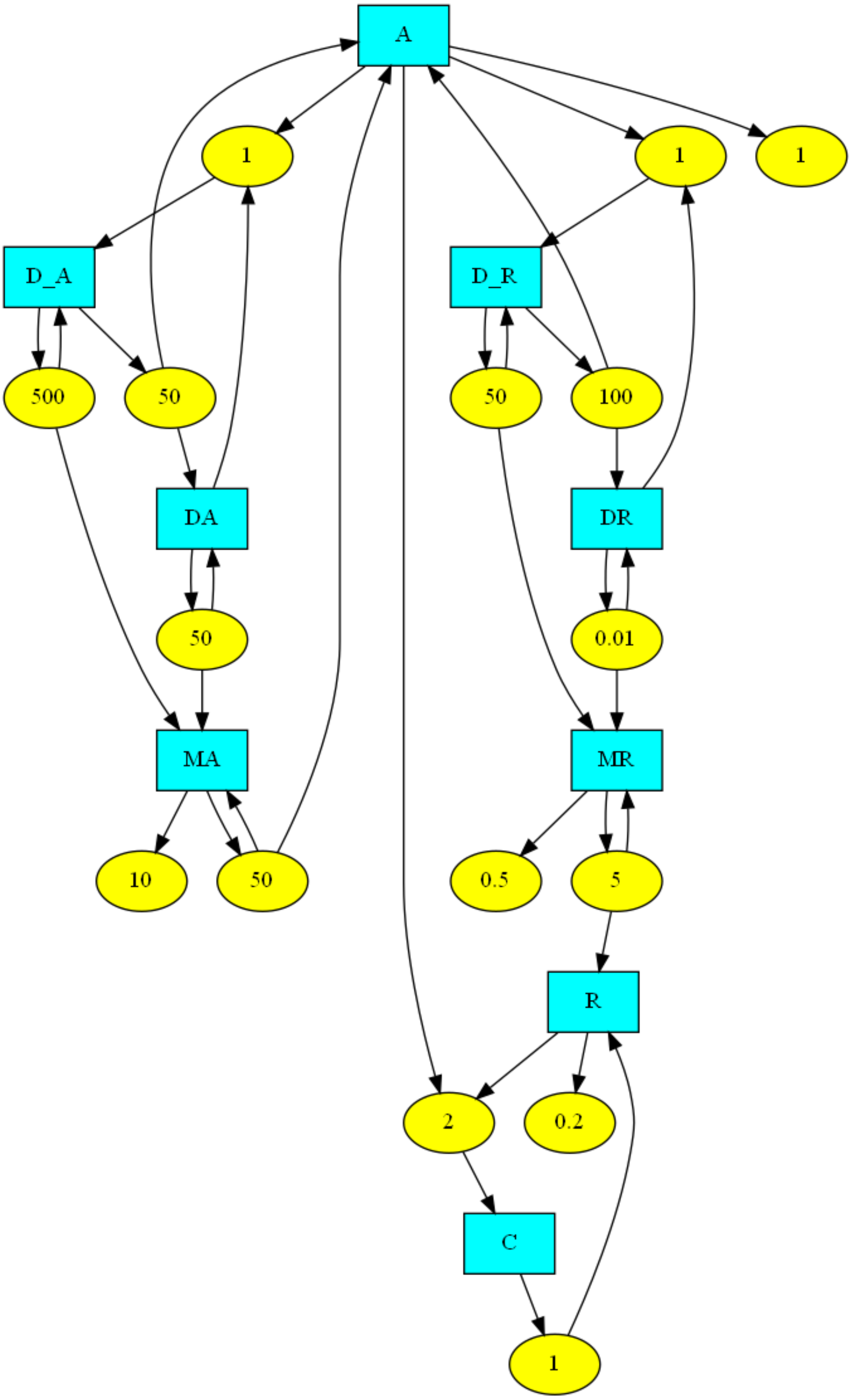

Figure 9: Circadian reaction graph

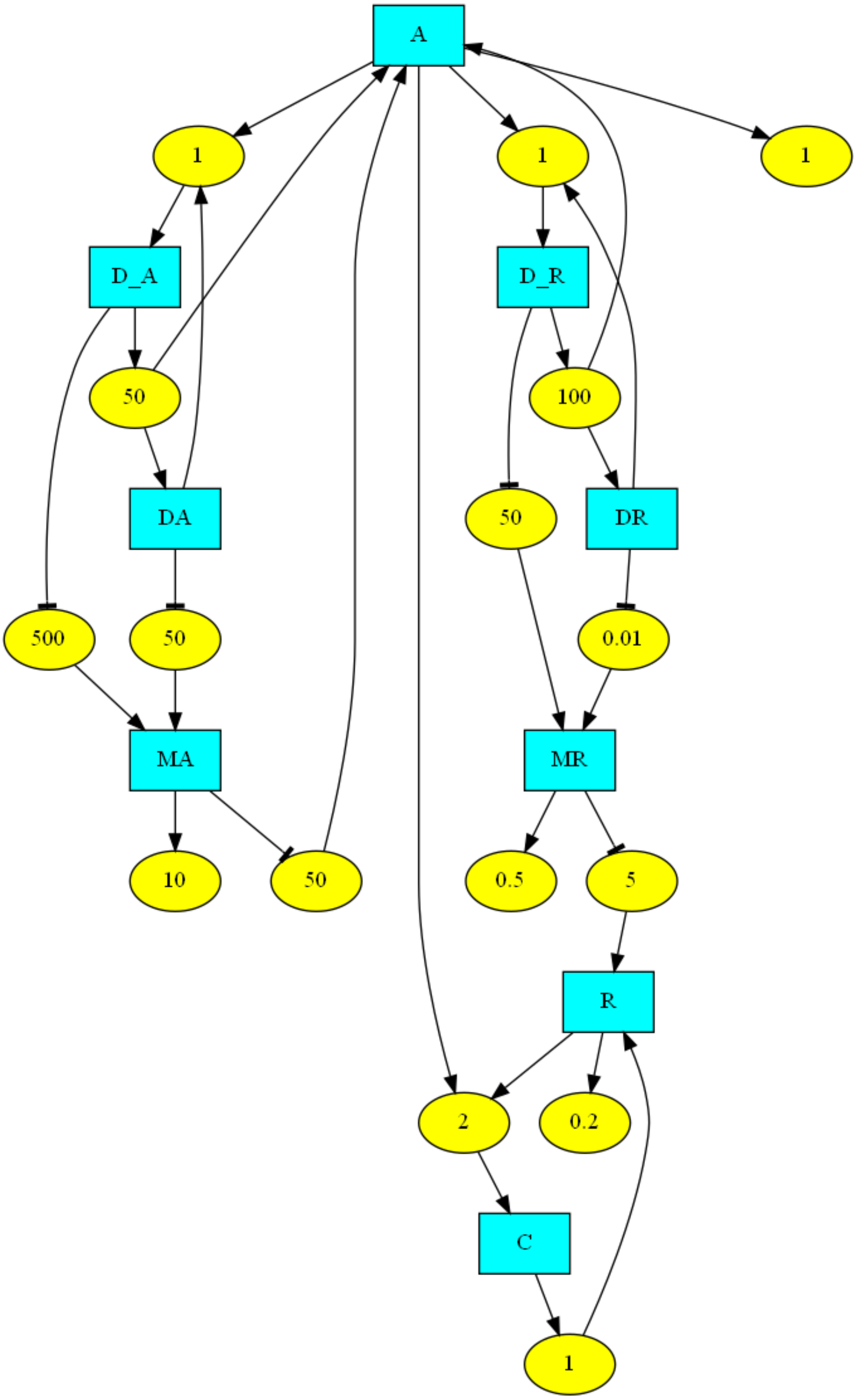

Figure 10: Circadian alternative reaction graph