

SP Exam Project

Mathias Andresen

May 12, 2021

Listing 1: CMakeLists.txt

```
1 cmake_minimum_required(VERSION 3.17)
2 project(sp_exam_project)
3
4 set(CMAKE_CXX_STANDARD 20)
5
6 add_library(
7     stochastic-simulation
8     library/simulation.cpp
9     library/simulation.cpp
10    library/SymbolTable.h
11    library/simulation_monitor.h
12    library/data.h
13    library/data.cpp
14    library/my-thread-pool.h
15 )
16
17 add_executable(sp_exam_project main.cpp vessels.h)
18
19 target_link_libraries(sp_exam_project PRIVATE stochastic-simulation)
```

Listing 2: main.cpp

```
1 #include <iostream>
2 #include "library/simulation.h"
3 #include <chrono>
4 #include "vessels.h"
5
6 using namespace StochasticSimulation;
7
8 class hospitalized_monitor: public simulation_monitor {
9 private:
10     double_t hospitalized_acc{0.0};
11     double_t last_time{0.0};
12 public:
13     size_t max_hospitalized{0};
14
15     void monitor(SimulationState &state) override {
16         auto currently_hospitalized = state.reactants.get("H").amount;
17
18         if (currently_hospitalized > max_hospitalized) {
19             max_hospitalized = currently_hospitalized;
20         }
21
22         hospitalized_acc += (currently_hospitalized * (state.time - last_time));
23
24         last_time = state.time;
25     }
26
27     double_t get_mean_hospitalized() const {
```

```

28     return (hospitalized_acc / last_time);
29 }
30 };
31
32 void simulate_covid() {
33     std::cout << "Simulating covid19 example with hospitalized monitor" << std::endl;
34     Vessel covid_vessel = seihr(10000);
35
36     std::cout << covid_vessel << std::endl;
37     covid_vessel.visualize_reactions("covid_graph.png");
38
39     std::cout << "reaction graph can be seen at: covid_graph.png" << std::endl;
40
41     hospitalized_monitor monitor{};
42
43     auto trajectory = covid_vessel.do_simulation(120, monitor);
44
45     std::cout << "Simulation done" << std::endl;
46     std::cout << "Max hospitalized: " << monitor.max_hospitalized << std::endl;
47     std::cout << "Mean hospitalized: " << monitor.get_mean_hospitalized() << std::endl;
48
49     std::cout << "Writing trajectory to csv file at covid_output.csv" << std::endl;
50     trajectory->write_csv("covid_output.csv");
51     std::cout << "Turn it into a graph using python ./draw_graph.py covid release" << std::endl;
52 }
53
54 void simulate_covid_multiple() {
55     std::cout << "Simulating covid19 example 30 times and calculating mean" << std::endl;
56     Vessel covid_vessel = seihr(10000);
57
58     auto trajectories = covid_vessel.do_multiple_simulations(110, 100);
59
60     std::cout << "Simulations done" << std::endl << "Computing mean trajectory" << std::endl;
61
62     auto mean = SimulationTrajectory::compute_mean_trajectory(trajectories);
63
64     std::cout << "Writing mean trajectory to csv file at covid_output_multiple.csv" << std::endl;
65     mean.write_csv("covid_output_multiple.csv");
66     std::cout << "Turn it into a graph using python ./draw_graph.py covid ↗
↪covid_output_multiple.csv" << std::endl;
67 }
68
69 void simulate_introduction() {
70     std::cout << "Simulating introduction example" << std::endl;
71     Vessel introduction_vessel = introduction(25, 50, 1, 0.001);
72     std::cout << introduction_vessel << std::endl;
73
74     introduction_vessel.visualize_reactions("intro_graph.png");
75
76     auto trajectory = introduction_vessel.do_simulation(400);
77
78     trajectory->write_csv("intro_output.csv");
79 }
80
81
82 void simulate_circadian() {
83     std::cout << "Simulating circadian rhythm example..." << std::endl;
84     Vessel oscillator = circadian_oscillator();
85
86     std::cout << oscillator << std::endl;
87     oscillator.visualize_reactions("cir_graph.png");

```

```

88
89     auto trajectory = oscillator.do_simulation(110);
90
91     std::cout << "Writing csv file..." << std::endl;
92     trajectory->write_csv("circadian_output.csv");
93 }
94
95 void simulate_circadian2() {
96     std::cout << "Simulating circadian rhythm alternative example..." << std::endl;
97     Vessel oscillator = circadian_oscillator2();
98
99     std::cout << oscillator << std::endl;
100    oscillator.visualize_reactions("cir2_graph.png");
101
102    auto trajectory = oscillator.do_simulation(110);
103
104    std::cout << "Writing csv file..." << std::endl;
105    trajectory->write_csv("circadian2_output.csv");
106 }
107
108 void benchmark() {
109     std::cout << "Benchmarking with circadian rhythm example (max_time=100)" << std::endl;
110
111     auto runs{30};
112
113     Vessel oscillator = circadian_oscillator();
114
115     unsigned long time_acc1{0};
116     for (int i = 0; i < runs; ++i) {
117         auto t0 = std::chrono::high_resolution_clock::now();
118         oscillator.do_simulation(100);
119         auto t1 = std::chrono::high_resolution_clock::now();
120
121         time_acc1 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
122     }
123     auto mean_time1 = time_acc1 / runs;
124     std::cout << "Simulation 1 mean time (nanoseconds): " << mean_time1 << std::endl;
125
126     unsigned long time_acc2{0};
127     for (int i = 0; i < runs; ++i) {
128         auto t0 = std::chrono::high_resolution_clock::now();
129         oscillator.do_simulation2(100);
130         auto t1 = std::chrono::high_resolution_clock::now();
131
132         time_acc2 += std::chrono::duration_cast<std::chrono::nanoseconds>(t1-t0).count();
133     }
134     auto mean_time2 = time_acc2 / runs;
135     std::cout << "Simulation 2 mean time (nanoseconds): " << mean_time2 << std::endl;
136 }
137
138 int main() {
139     simulate_covid();
140     // simulate_covid_multiple();
141
142     // simulate_introduction();
143     // simulate_circadian();
144     // simulate_circadian2();
145
146     // benchmark();
147 }

```

```

1 //
2 // Created by Mathias on 12-05-2021.
3 //
4
5 #ifndef SP_EXAM_PROJECT_VESSELS_H
6 #define SP_EXAM_PROJECT_VESSELS_H
7
8 #include "library/simulation.h"
9
10 using namespace StochasticSimulation;
11
12 Vessel seihr(uint32_t N)
13 {
14     auto v = Vessel{};
15     const auto eps = 0.0009; // initial fraction of infectious
16     const auto I0 = size_t(std::round(eps*N)); // initial infectious
17     const auto E0 = size_t(std::round(eps*N*15)); // initial exposed
18     const auto S0 = N-I0-E0; // initial susceptible
19     const auto R0 = 2.4; // basic reproductive number (initial, without lockdown etc)
20     const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
21     const auto gamma = 1.0 / 3.1; // recovery rate (I -> R) ~3.1 days
22     const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
23     const auto P_H = 0.9e-3; // probability of hospitalization
24     const auto kappa = gamma * P_H*(1.0-P_H); // hospitalization rate (I -> H)
25     const auto tau = 1.0/10.12; // recovery/death rate in hospital (H -> R) ~10.12 days
26
27     // Reactants
28     auto S = v("S", S0); // susceptible
29     auto E = v("E", E0); // exposed
30     auto I = v("I", I0); // infectious
31     auto H = v("H", 0); // hospitalized
32     auto R = v("R", 0); // removed/immune (recovered + dead)
33
34     // Reactions
35     v(S >=> E, I, beta/N);
36     v(E >=> I, alpha);
37     v(I >=> R, gamma);
38     v(I >=> H, kappa);
39     v(H >=> R, tau);
40
41     return v;
42 }
43
44 Vessel introduction(uint32_t A_start, uint32_t B_start, uint32_t D_amount, double_t lambda) {
45     auto v = Vessel{};
46     // Reactants
47     auto A = v("A", A_start);
48     auto B = v("B", B_start);
49     auto C = v("C", 0);
50     auto D = v("D", D_amount);
51     // Reactions
52     v(A + B * 2 >=> C, D, lambda);
53
54     return v;
55 }
56
57 /** direct encoding */
58 Vessel circadian_oscillator()
59 {
60     auto alphaA = 50.0;

```

```

61     auto alpha_A = 500.0;
62     auto alphaR = 0.01;
63     auto alpha_R = 50.0;
64     auto betaA = 50.0;
65     auto betaR = 5.0;
66     auto gammaA = 1.0;
67     auto gammaR = 1.0;
68     auto gammaC = 2.0;
69     auto deltaA = 1.0;
70     auto deltaR = 0.2;
71     auto deltaMA = 10.0;
72     auto deltaMR = 0.5;
73     auto thetaA = 50.0;
74     auto thetaR = 100.0;
75     auto v = Vessel{};
76     auto env = v.environment();
77     auto DA = v("DA", 1);
78     auto D_A = v("D_A", 0);
79     auto DR = v("DR", 1);
80     auto D_R = v("D_R", 0);
81     auto MA = v("MA", 0);
82     auto MR = v("MR", 0);
83     auto A = v("A", 0);
84     auto R = v("R", 0);
85     auto C = v("C", 0);
86     v(A + DA >= D_A, gammaA);
87     v(D_A >= DA + A, thetaA);
88     v(A + DR >= D_R, gammaR);
89     v(D_R >= DR + A, thetaR);
90     v(D_A >= MA + D_A, alpha_A);
91     v(DA >= MA + DA, alphaA);
92     v(D_R >= MR + D_R, alpha_R);
93     v(DR >= MR + DR, alphaR);
94     v(MA >= MA + A, betaA);
95     v(MR >= MR + R, betaR);
96     v(A + R >= C, gammaC);
97     v(C >= R, deltaA);
98     v(A >= env, deltaA);
99     v(R >= env, deltaR);
100    v(MA >= env, deltaMA);
101    v(MR >= env, deltaMR);
102    return v;
103 }
104
105 /** alternative encoding using catalysts */
106 Vessel circadian_oscillator2()
107 {
108     auto alphaA = 50.0;
109     auto alpha_A = 500.0;
110     auto alphaR = 0.01;
111     auto alpha_R = 50.0;
112     auto betaA = 50.0;
113     auto betaR = 5.0;
114     auto gammaA = 1.0;
115     auto gammaR = 1.0;
116     auto gammaC = 2.0;
117     auto deltaA = 1.0;
118     auto deltaR = 0.2;
119     auto deltaMA = 10.0;
120     auto deltaMR = 0.5;
121     auto thetaA = 50.0;

```

```

122     auto thetaR = 100.0;
123     auto v = Vessel{};
124     auto env = v.environment();
125     auto DA = v("DA", 1);
126     auto D_A = v("D_A", 0);
127     auto DR = v("DR", 1);
128     auto D_R = v("D_R", 0);
129     auto MA = v("MA", 0);
130     auto MR = v("MR", 0);
131     auto A = v("A", 0);
132     auto R = v("R", 0);
133     auto C = v("C", 0);
134     v(A + DA >= D_A, gammaA);
135     v(D_A >= DA + A, thetaA);
136     v(DR + A >= D_R, gammaR);
137     v(D_R >= DR + A, thetaR);
138     v(env >= MA, D_A, alpha_A);
139     v(env >= MA, DA, alphaA);
140     v(env >= MR, D_R, alpha_R);
141     v(env >= MR, DR, alphaR);
142     v(env >= A, MA, betaA);
143     v(env >= R, MR, betaR);
144     v(A + R >= C, gammaC);
145     v(C >= R, deltaA);
146     v(A >= env, deltaA);
147     v(R >= env, deltaR);
148     v(MA >= env, deltaMA);
149     v(MR >= env, deltaMR);
150     return v;
151 }
152
153 #endif //SP_EXAM_PROJECT_VESSELS_H

```

Listing 4: simulation.h

```

1  //
2  // Created by Mathias on 09-05-2021.
3  //
4
5  #ifndef SP_EXAM_PROJECT_SIMULATION_H
6  #define SP_EXAM_PROJECT_SIMULATION_H
7
8  #include <string>
9  #include <utility>
10 #include <vector>
11 #include <set>
12 #include <optional>
13 #include <map>
14 #include <numeric>
15 #include <random>
16 #include <algorithm>
17 #include <functional>
18 #include <sstream>
19 #include <fstream>
20 #include <chrono>
21 #include <thread>
22 #include <future>
23 #include <ranges>
24 #include "SymbolTable.h"
25 #include "simulation_monitor.h"
26 #include "data.h"
27

```

```

28 namespace StochasticSimulation {
29
30     using map_type = std::map<double_t, SimulationState>;
31     class SimulationTrajectory: public map_type {
32     private:
33         double_t largest_time{-1};
34         static double_t compute_interpolated_value(
35             const std::string& key,
36             SimulationState& s0,
37             SimulationState& s1,
38             double_t x);
39     public:
40         using map_type::map;
41
42         SimulationTrajectory(const SimulationTrajectory& val): map_type(val) {
43             largest_time = val.largest_time;
44         }
45
46         SimulationTrajectory(SimulationTrajectory&& rval): map_type(std::move(rval)) {
47             largest_time = std::move(rval.largest_time);
48         };
49
50         SimulationTrajectory& operator=(const SimulationTrajectory & val) {
51             map_type::operator=(val);
52             largest_time = val.largest_time;
53         };
54
55         SimulationTrajectory& operator=(SimulationTrajectory&& rval) {
56             map_type::operator=(std::move(rval));
57             largest_time = std::move(rval.largest_time);
58         };
59
60         static SimulationTrajectory ↵
61         ↵compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories);
62
63         void insert(SimulationState state) {
64             if (state.time > largest_time) {
65                 largest_time = state.time;
66             }
67
68             map_type::insert({state.time, std::move(state)});
69         }
70
71         void write_csv(const std::string& path);
72
73         double_t get_max_time() {
74             return largest_time;
75         }
76     };
77
78     class Vessel {
79     private:
80         std::vector<Reaction> reactions{};
81         SymbolTable<Reactant> reactants;
82     public:
83
84         Vessel() = default;
85
86         Vessel(const Vessel &val) {
87             reactions = val.reactions;
88             reactants = val.reactants;

```

```

88     }
89
90     Vessel (Vessel&& rval) {
91         reactions = std::move(rval.reactions);
92         reactants = std::move(rval.reactants);
93     };
94
95     Reactant& operator()(std::string name, size_t initial_amount) {
96         Reactant newReactant{std::move(name), initial_amount};
97
98         reactants.put(newReactant.name, newReactant);
99
100        return reactants.get(newReactant.name);
101    }
102
103    Reaction operator()(Reaction&& reaction, double_t rate) {
104        reaction.rate = rate;
105
106        reactions.push_back(reaction);
107
108        return reaction;
109    }
110
111    Reaction operator()(Reaction&& reaction, std::initializer_list<Reactant> catalysts, ↵
↵double rate) {
112        reaction.rate = rate;
113        reaction.catalysts = catalysts;
114
115        // Add to vessel reactions
116        reactions.push_back(reaction);
117
118        return reaction;
119    }
120
121    Reaction operator()(Reaction&& reaction, Reactant catalyst, double_t rate) {
122        reaction.rate = rate;
123        reaction.catalysts = {catalyst};
124
125        // Add to vessel reactions
126        reactions.push_back(reaction);
127
128        return reaction;
129    }
130
131
132    Reactant& environment() {
133        if (reactants.contains("__env__")) {
134            return reactants.get("__env__");
135        }
136
137        auto newReactant = Reactant("__env__", 0, 0);
138
139        reactants.put(newReactant.name, newReactant);
140
141        return reactants.get(newReactant.name);
142    }
143
144    void visualize_reactions(const std::string& filename);
145
146    std::shared_ptr<SimulationTrajectory> do_simulation2(double_t end_time, ↵
↵simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);

```



```

147
148         std::shared_ptr<SimulationTrajectory> do_simulation(double_t end_time, ↵
↵simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
149
150         std::vector<std::shared_ptr<SimulationTrajectory>> do_multiple_simulations(double_t ↵
↵end_time, size_t simulations_to_run);
151
152         friend std::ostream& operator<<(std::ostream& s, const Vessel& vessel);
153     };
154
155
156
157 }
158
159 #endif //SP_EXAM_PROJECT_SIMULATION_H

```

Listing 5: simulation.cpp

```

1 //
2 // Created by Mathias on 09-05-2021.
3 //
4
5 #include <iostream>
6 #include <utility>
7 #include "simulation.h"
8
9 namespace StochasticSimulation {
10
11
12     std::ostream &operator<<(std::ostream &s, const Vessel &vessel) {
13         s << "{" << std::endl;
14         for (const auto& reaction: vessel.reactions) {
15             s << "\t" << reaction;
16             if (&reaction != &vessel.reactions.back()) {
17                 s << ",";
18             }
19             s << std::endl;
20         }
21         return s << "}";
22     }
23
24     void Vessel::visualize_reactions(const std::string& filename) {
25         std::stringstream str;
26         SymbolTable<std::string> node_map{};
27
28         str << "digraph {" << std::endl;
29
30         auto i = 0;
31         for (auto& reactant: reactants.getMap()) {
32             if (reactant.second.name != "__env__") {
33                 node_map.put(reactant.second.name, "s" + std::to_string(i));
34
35                 str << node_map.get(reactant.second.name)
36                     << "[label=\"" << reactant.second.name << ↵
↵"\" , shape=\"box\" , style=\"filled\" , fillcolor=\"cyan\" ];\" << std::endl;
37                 i++;
38             }
39         }
40
41         i = 0;
42         for (auto& reaction: reactions) {
43             std::string reaction_node{"r" + std::to_string(i)};

```

```

44
45     str << reaction_node << "[label=\"" << reaction.rate <<
↳"\", shape=\"oval\", style=\"filled\", fillcolor=\"yellow\"];\" << std::endl;
46     if (reaction.catalysts.has_value()) {
47         for (auto& catalyst: reaction.catalysts.value()) {
48             str << node_map.get(catalyst.name) << " -> " << reaction_node << "
↳[arrowhead=\"tee\"];\" << std::endl;
49         }
50     }
51     for (auto& reactant: reaction.from) {
52         if (reactant.name != "__env__") {
53             str << node_map.get(reactant.name) << " -> " << reaction_node << ";\";" <<
↳std::endl;
54         }
55     }
56     for (auto& product: reaction.to) {
57         if (product.name != "__env__") {
58             str << reaction_node << " -> " << node_map.get(product.name) << ";\";" <<
↳std::endl;
59         }
60     }
61     i++;
62 }
63
64 str << "}";
65
66 std::ofstream dotfile;
67 dotfile.open(filename + ".dot");
68 dotfile << str.str();
69 dotfile.close();
70
71 std::stringstream command_builder;
72 command_builder << "dot -Tpng -o " << filename << " " << filename << ".dot";
73 system(command_builder.str().c_str());
74 }
75
76
77 std::shared_ptr<SimulationTrajectory> Vessel::do_simulation2(double_t end_time,
↳simulation_monitor &monitor) {
78     SimulationTrajectory trajectory{};
79     double_t t{0};
80
81     auto thread_id = std::this_thread::get_id();
82     auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
83     std::default_random_engine engine(epoch * (std::hash<std::thread::id>{})(thread_id));
84
85     // Insert initial state
86     trajectory.insert(SimulationState{reactants, t});
87
88     while (t <= end_time) {
89         for (Reaction& reaction: reactions) {
90             // New: using new compute delay function
91             reaction.compute_delay2(trajectory.at(t), engine);
92         }
93
94         auto r = reactions.front();
95
96         // Select Reaction with min delay which is not -1
97         for (auto& reaction: reactions) {
98             if (reaction.delay == -1) {
99                 continue;

```

```

100         } else if (reaction.delay < r.delay) {
101             r = reaction;
102         } else if (r.delay == -1) {
103             r = reaction;
104         }
105     }
106
107     // Stop if we have no reactions to do, thus r.delay == -1
108     if (r.delay == -1) {
109         break;
110     }
111
112     auto& last_state = trajectory.at(t);
113
114     t += r.delay;
115
116     SimulationState state{last_state.reactants, t};
117
118     if (
119         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
120         →state.reactants.get(e.name).amount >= e.required;}) &&
121         (
122             !r.catalysts.has_value() ||
123             std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
124         →[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
125         )
126     ) {
127         for (auto& reactant: r.from) {
128             state.reactants.get(reactant.name).amount -= reactant.required;
129         }
130         for (auto& reactant: r.to) {
131             state.reactants.get(reactant.name).amount += reactant.required;
132         }
133     }
134
135     trajectory.insert(std::move(state));
136
137     monitor.monitor(trajectory.at(t));
138
139     return std::make_shared<SimulationTrajectory>(std::move(trajectory));
140 }
141
142 std::shared_ptr<SimulationTrajectory> Vessel::do_simulation(double_t end_time,
143 →simulation_monitor &monitor) {
144     SimulationTrajectory trajectory{};
145     double_t t{0};
146
147     auto thread_id = std::this_thread::get_id();
148     auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
149     std::default_random_engine engine(epoch * (std::hash<std::thread::id>{})(thread_id));
150
151     // Insert initial state
152     trajectory.insert(SimulationState{reactants, t});
153
154     while (t <= end_time) {
155         for (Reaction& reaction: reactions) {
156             reaction.compute_delay(trajectory.at(t), engine);
157         }
158
159         auto r = reactions.front();

```

```

158
159 // Select Reaction with min delay which is not -1
160 for (auto& reaction: reactions) {
161     if (reaction.delay == -1) {
162         continue;
163     } else if (reaction.delay < r.delay) {
164         r = reaction;
165     } else if (r.delay == -1 && reaction.delay != -1) {
166         r = reaction;
167     }
168 }
169
170 // Stop if we have no reactions to do, thus r.delay == -1
171 if (r.delay == -1) {
172     break;
173 }
174
175 auto& last_state = trajectory.at(t);
176
177 t += r.delay;
178
179 SimulationState state{last_state.reactants, t};
180
181 if (
182     std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
183     ↪state.reactants.get(e.name).amount >= e.required;}) &&
184     (
185         !r.catalysts.has_value() ||
186         std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
187         ↪[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
188     ) {
189     for (auto& reactant: r.from) {
190         state.reactants.get(reactant.name).amount -= reactant.required;
191     }
192     for (auto& reactant: r.to) {
193         state.reactants.get(reactant.name).amount += reactant.required;
194     }
195 }
196
197 trajectory.insert(std::move(state));
198
199 monitor.monitor(trajectory.at(t));
200 }
201
202 return std::make_shared<SimulationTrajectory>(std::move(trajectory));
203 }
204
205 std::vector<std::shared_ptr<SimulationTrajectory>>
206 Vessel::do_multiple_simulations(double_t end_time, size_t simulations_to_run) {
207     std::vector<std::shared_ptr<SimulationTrajectory>> result{};
208     result.reserve(simulations_to_run);
209
210     auto cores = std::thread::hardware_concurrency();
211     int jobs = std::min(simulations_to_run, (cores - 1));
212     auto simulations_per_job = simulations_to_run / jobs;
213
214     auto futures =
215     ↪std::vector<std::future<std::vector<std::shared_ptr<SimulationTrajectory>>>>{};
216
217     auto lambda = [&vessel = *this, &end_time](size_t to_run){

```

```

216     auto simulations = std::vector<std::shared_ptr<SimulationTrajectory>>{};
217     simulations.reserve(to_run);
218
219     auto new_vessel = Vessel(vessel);
220
221     for (int i = 0; i < to_run; ++i) {
222         simulations.push_back(new_vessel.do_simulation(end_time));
223     }
224
225     return simulations;
226 };
227
228 for (int i = 0; i < jobs; ++i) {
229     futures.push_back(std::async(std::launch::async, lambda, simulations_per_job));
230 }
231 auto missing_simulations = simulations_to_run - (simulations_per_job * jobs);
232 if (missing_simulations != 0) {
233     futures.push_back(std::async(std::launch::async, lambda, missing_simulations));
234 }
235
236 for (auto& future: futures) {
237     auto future_result = future.get();
238     for (auto& res: future_result) {
239         result.push_back(std::move(res));
240     }
241 }
242
243 return result;
244 }
245
246 double_t SimulationTrajectory::compute_interpolated_value(const std::string& key, ↵
↵SimulationState& s0, SimulationState& s1, double_t x) {
247     return
248         s0.reactants.get(key).amount
249         + ((
250             ( (double_t) s1.reactants.get(key).amount - (double_t) ↵
↵s0.reactants.get(key).amount) /
251             ( s1.time - s0.time )
252             ) * (x - s0.time));
253 }
254
255 SimulationTrajectory ↵
↵SimulationTrajectory::compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& ↵
↵trajectories) {
256     auto average_delay = trajectories.front()->get_max_time() / trajectories.front()->size();
257
258     // Get a list of all keys
259     std::vector<std::string> reactant_keys{};
260     for (auto& reactant: trajectories.front()->begin()->second.reactants) {
261         reactant_keys.push_back(reactant.second.name);
262     }
263
264     // Find upper bound for mean trajectory
265     double_t upper_bound{-1.0};
266     for (auto& trajectory: trajectories) {
267         if (upper_bound == -1.0 || trajectory->get_max_time() < upper_bound) {
268             upper_bound = trajectory->get_max_time();
269         }
270     }
271
272     SimulationTrajectory mean_trajectory{};

```

```

273
274     for (auto& trajectory: trajectories) {
275         auto iterator = trajectory->begin();
276         SimulationState& s0 = iterator->second;
277         iterator++;
278         SimulationState& s1 = iterator->second;
279
280         double_t t{0};
281
282         while((t + average_delay) <= upper_bound) {
283             if (t >= s0.time) {
284                 if (t <= s1.time) {
285                     if (!mean_trajectory.contains(t)) {
286                         mean_trajectory.insert((SimulationState{{}}, t));
287                     }
288
289                     for (auto& key: reactant_keys) {
290                         auto interpolated_value = ↵
↵SimulationTrajectory::compute_interpolated_value(key, s0, s1, t);
291
292                         auto& table = mean_trajectory.at(t).reactants;
293
294                         if (!table.contains(key)) {
295                             Reactant reactant{key, 0.0};
296                             table.put(key, reactant);
297                         }
298
299                         table.get(key).amount += interpolated_value;
300                     }
301
302                     t += average_delay;
303                 } else {
304                     s0 = s1;
305                     iterator++;
306
307                     if (iterator != trajectory->end()) {
308                         s1 = iterator->second;
309                     } else {
310                         break;
311                     }
312                 }
313             }
314         }
315     }
316
317     for (double_t i = 0; (i + average_delay) <= upper_bound; i += average_delay) {
318         for (auto& key: reactant_keys) {
319             mean_trajectory.at(i).reactants.get(key).amount /= trajectories.size();
320         }
321     }
322
323     return std::move(mean_trajectory);
324 }
325
326 void SimulationTrajectory::write_csv(const std::string &path) {
327     std::ofstream csv_file;
328     csv_file.open(path);
329
330     auto reactants = at(0).reactants;
331
332     for (auto& reactant : reactants) {

```

```

333         csv_file << reactant.second.name << ",";
334     }
335     csv_file << "time" << std::endl;
336
337     for (auto& state : *this) {
338         for (auto& reactant: reactants) {
339             csv_file << state.second.reactants.get(reactant.second.name).amount << ",";
340         }
341         csv_file << state.second.time << std::endl;
342     }
343
344     csv_file.close();
345 }
346 }

```

Listing 6: data.h

```

1  //
2  // Created by Mathias on 11-05-2021.
3  //
4
5  #ifndef SP_EXAM_PROJECT_DATA_H
6  #define SP_EXAM_PROJECT_DATA_H
7
8  namespace StochasticSimulation {
9      class SimulationState;
10     struct Reaction;
11     class ReactantCollection;
12
13     struct Reactant {
14         std::string name;
15         double_t amount; // double to allow for mean values
16         size_t required{1};
17
18         Reactant(std::string name, size_t initial_amount):
19             name(std::move(name)),
20             amount(initial_amount)
21         {}
22
23         Reactant(std::string name, double_t initial_amount):
24             name(std::move(name)),
25             amount(initial_amount)
26         {}
27
28         Reactant(std::string name, size_t initial_amount, size_t required):
29             name(std::move(name)),
30             amount(initial_amount),
31             required(required)
32         {}
33
34         ~Reactant() = default;
35
36         Reactant(const Reactant& a) {
37             name = a.name;
38             amount = a.amount;
39             required = a.required;
40         }
41
42         Reactant(Reactant&& a) {
43             name = std::move(a.name);
44             amount = std::move(a.amount);
45             required = std::move(a.required);

```

```

46     }
47
48     Reactant& operator=(Reactant&& a) {
49         name = std::move(a.name);
50         amount = std::move(a.amount);
51         required = std::move(a.required);
52
53         return *this;
54     }
55
56     Reactant& operator=(const Reactant& a) {
57         name = a.name;
58         amount = a.amount;
59         required = a.required;
60
61         return *this;
62     }
63
64     Reaction operator>=(Reactant other);
65
66     Reaction operator>=(ReactantCollection other);
67
68     ReactantCollection operator+(const Reactant& other);
69
70     bool operator<(const Reactant& other) const;
71
72     Reactant operator*(size_t req) {
73         required = req;
74         return *this;
75     }
76
77 };
78
79 class ReactantCollection: public std::set<Reactant> {
80 public:
81     using std::set<Reactant>::set;
82     Reaction operator>=(Reactant other);
83     Reaction operator>=(ReactantCollection other);
84 };
85
86 class Reaction {
87 public:
88     std::set<Reactant> from;
89     std::set<Reactant> to;
90     std::optional<std::vector<Reactant>> catalysts;
91     double_t rate{};
92     double_t delay{-1};
93
94     Reaction(std::set<Reactant> from, std::set<Reactant> to):
95         from(from),
96         to(to)
97     {}
98
99     Reaction(std::set<Reactant> from, std::set<Reactant> to, std::initializer_list<Reactant>
→catalysts, double rate):
100         from(from),
101         to(to),
102         catalysts(catalysts),
103         rate(rate)
104     {}
105

```



```

106     Reaction(std::set<Reactant> from, std::set<Reactant> to, double rate):
107         from(from),
108         to(to),
109         catalysts{},
110         rate(rate)
111     {}
112
113     void compute_delay(SimulationState& state, std::default_random_engine& engine);
114     void compute_delay2(SimulationState& state, std::default_random_engine& engine);
115
116     friend std::ostream &operator<<(std::ostream &s, const Reaction &reaction);
117 };
118
119 struct SimulationState {
120 public:
121     SymbolTable<Reactant> reactants;
122     double_t time;
123
124     SimulationState(SymbolTable<Reactant> reactants, double_t time):
125         reactants{reactants},
126         time{time}
127     {};
128
129     SimulationState(const SimulationState&) = default;
130     SimulationState(SimulationState&&) = default;
131
132     SimulationState& operator=(const SimulationState &) = default;
133     SimulationState& operator=(SimulationState&&) = default;
134
135     ~SimulationState() = default;
136
137     friend std::ostream &operator<<(std::ostream &, const SimulationState &);
138 };
139
140 }
141
142 #endif //SP_EXAM_PROJECT_DATA_H

```

Listing 7: data.cpp

```

1  //
2  // Created by Mathias on 11-05-2021.
3  //
4
5  #include <iostream>
6  #include <utility>
7  #include "simulation.h"
8
9  namespace StochasticSimulation {
10     Reaction Reactant::operator>=(StochasticSimulation::Reactant other) {
11         return Reaction{*this, {std::move(other)}};
12     }
13
14     Reaction Reactant::operator>=(ReactantCollection other) {
15         return Reaction{*this, std::move(other)};
16     }
17
18     ReactantCollection Reactant::operator+(const Reactant& other) {
19         return ReactantCollection{*this, other};
20     }
21
22     bool Reactant::operator<(const Reactant& other) const {

```

```

23     return name < other.name;
24 }
25
26 Reaction ReactantCollection::operator>=(Reactant other) {
27     return Reaction{*this, {std::move(other)}};
28 }
29
30 Reaction ReactantCollection::operator>=(ReactantCollection other) {
31     return Reaction{*this, std::move(other)};
32 }
33
34 std::ostream &operator<<(std::ostream &s, const Reaction &reaction) {
35     s << "{ ";
36     for (const auto& reactant: reaction.from) {
37         s << reactant.name << "+";
38     }
39     s << "\b" << " >=";
40     if (reaction.catalysts.has_value()) {
41         s << "(";
42         for (const auto& catalyst: reaction.catalysts.value()) {
43             s << catalyst.name << "+";
44         }
45         s << "\b" << ") ";
46     }
47     for (const auto& reactant: reaction.to) {
48         s << reactant.name << "+";
49     }
50     s << "\b";
51
52     return s << " - " << reaction.rate << " }";
53 }
54
55 void Reaction::compute_delay2(SimulationState& state, std::default_random_engine &engine) {
56     size_t reactant_amount{1};
57     size_t catalyst_amount{1};
58
59     for (const Reactant& reactant: from) {
60         auto amount = reactant.name == "__env__" ? 1 : ↗
↪state.reactants.get(reactant.name).amount;
61         reactant_amount *= amount;
62     }
63     // New: check if amount is 0 already
64     if (reactant_amount == 0) {
65         delay = -1;
66         return;
67     }
68
69     if (catalysts.has_value()) {
70         for (auto& catalyst: catalysts.value()) {
71             catalyst_amount *= state.reactants.get(catalyst.name).amount;
72         }
73     }
74
75     double_t rate_k = rate * reactant_amount * catalyst_amount;
76
77     if (rate_k > 0) {
78         delay = std::exponential_distribution<double_t>(rate_k)(engine);
79     } else {
80         delay = -1;
81     }
82 }

```

```

83
84 void Reaction::compute_delay(SimulationState& state, std::default_random_engine &engine) {
85     size_t reactant_amount{1};
86     size_t catalyst_amount{1};
87
88     for (const Reactant& reactant: from) {
89         auto amount = reactant.name == "__env__" ? 1 : 2
→state.reactants.get(reactant.name).amount;
90         reactant_amount *= amount;
91     }
92     if (catalysts.has_value()) {
93         for (auto& catalyst: catalysts.value()) {
94             catalyst_amount *= state.reactants.get(catalyst.name).amount;
95         }
96     }
97
98     double_t rate_k = rate * reactant_amount * catalyst_amount;
99
100    if (rate_k > 0) {
101        delay = std::exponential_distribution<double_t>(rate_k)(engine);
102    } else {
103        delay = -1;
104    }
105 }
106
107 std::ostream &operator<<(std::ostream &s, const SimulationState& state) {
108     s << "{" << std::endl
109         << "time: " << state.time << "," << std::endl
110         << "reactants: {" << std::endl;
111     for(auto& pair: state.reactants) {
112         s << pair.first << ": " << pair.second.amount << "," << std::endl;
113     }
114
115     s << "}";
116     return s;
117 }
118 }

```

Listing 8: simulation_monitor.h

```

1 //
2 // Created by Mathias on 11-05-2021.
3 //
4
5 #ifndef SP_EXAM_PROJECT_SIMULATION_MONITOR_H
6 #define SP_EXAM_PROJECT_SIMULATION_MONITOR_H
7
8 #include <functional>
9 #include "data.h"
10
11 namespace StochasticSimulation {
12     class simulation_monitor {
13     public:
14         virtual void monitor(SimulationState& state) = 0;
15     };
16
17
18     class empty_simulation_monitor: public simulation_monitor {
19     public:
20         void monitor(SimulationState &state) override {
21             return;
22         }
23     };
24 }

```

```

23
24 static auto EMPTY_SIMULATION_MONITOR = empty_simulation_monitor{};
25
26 class basic_simulation_monitor: public simulation_monitor {
27 private:
28     const std::function<void(SimulationState&)> monitor_function;
29 public:
30     basic_simulation_monitor(const std::function<void(SimulationState&)>& monitor_func):
31         simulation_monitor{},
32         monitor_function{monitor_func}
33     {}
34
35     void monitor(SimulationState& state) override {
36         monitor_function(state);
37     }
38 };
39 }
40
41 #endif //SP_EXAM_PROJECT_SIMULATION_MONITOR_H

```

Listing 9: SymbolTable.h

```

1 //
2 // Created by Mathias on 10-05-2021.
3 //
4
5 #ifndef SP_EXAM_PROJECT_SYMBOLTABLE_H
6 #define SP_EXAM_PROJECT_SYMBOLTABLE_H
7
8 #include <unordered_map>
9 #include <string>
10 #include <stdexcept>
11 #include <utility>
12 #include <iterator>
13
14 namespace StochasticSimulation {
15
16     struct SymbolTableException : public std::exception
17     {
18         std::string message;
19     public:
20         explicit SymbolTableException(std::string message): message(std::move(message))
21         {}
22
23         [[nodiscard]] const char* what() const override
24         {
25             return message.c_str();
26         }
27     };
28
29     template<typename T>
30     class SymbolTable {
31     public:
32         using map_type = std::unordered_map<std::string, T>;
33     private:
34         map_type map{};
35     public:
36         using iterator = typename map_type::iterator;
37         using const_iterator = typename map_type::const_iterator;
38
39         SymbolTable<T>() = default;
40
41         SymbolTable<T>(const SymbolTable<T>& a) {

```

```

41     map = a.map;
42 };
43
44 SymbolTable<T>(SymbolTable<T>&& a) {
45     map = std::move(a.map);
46 };
47
48 ~SymbolTable() = default;
49
50 SymbolTable<T>& operator=(const SymbolTable& a) {
51     map = a.map;
52     return *this;
53 };
54
55 SymbolTable<T>& operator=(SymbolTable&& a) {
56     map = std::move(a.map);
57     return *this;
58 };
59
60 void put(const std::string& key, T value) {
61     if (!map.contains(key)) {
62         map.insert_or_assign(key, value);
63     } else {
64         throw SymbolTableException("Key " + key + " already used");
65     }
66 }
67
68 T& get(const std::string& key) {
69     try {
70         return map.at(key);
71     } catch (std::out_of_range& e) {
72         throw SymbolTableException("Key " + key + " was not found");
73     }
74 }
75
76 const T& get(const std::string& key) const {
77     try {
78         return map.at(key);
79     } catch (std::out_of_range& e) {
80         throw SymbolTableException("Key " + key + " was not found");
81     }
82 }
83
84 bool contains(const std::string& key) {
85     return map.contains(key);
86 }
87
88 std::unordered_map<std::string, T> getMap() {
89     return map;
90 }
91
92 iterator begin() {
93     return map.begin();
94 }
95
96 iterator end() {
97     return map.end();
98 }
99
100 const_iterator begin() const {
101     return map.begin();

```

```

102     }
103
104     const_iterator end() const {
105         return map.end();
106     }
107
108 };
109 }
110
111 #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

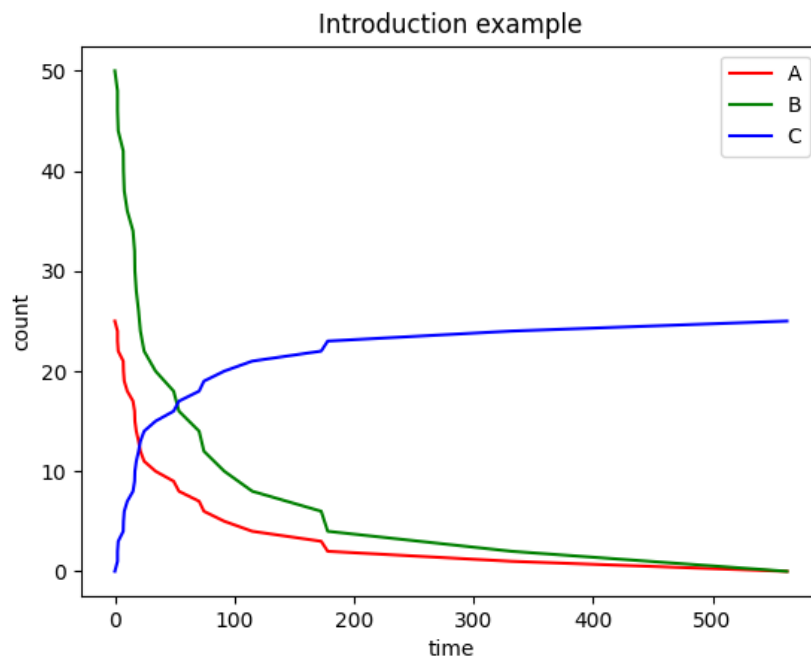


Figure 1: $A(0)=25$, $B(0)=50$, $D=1$

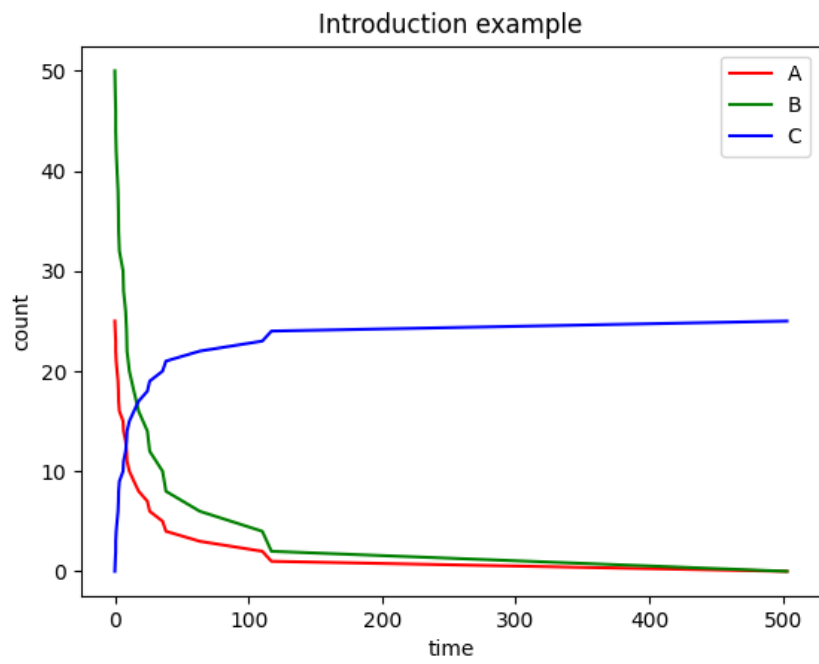


Figure 2: $A(0)=25$, $B(0)=50$, $D=2$

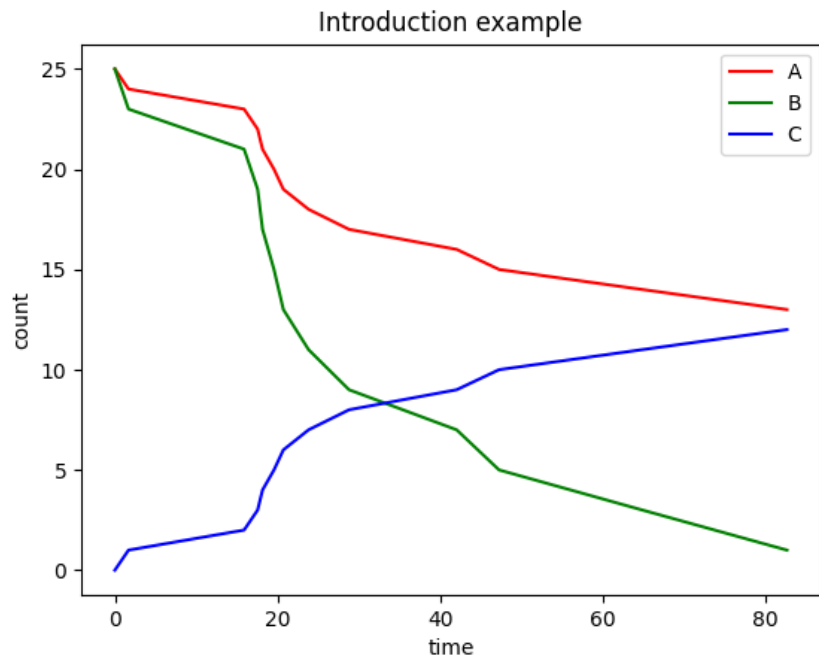


Figure 3: $A(0)=25$, $B(0)=25$, $D=1$

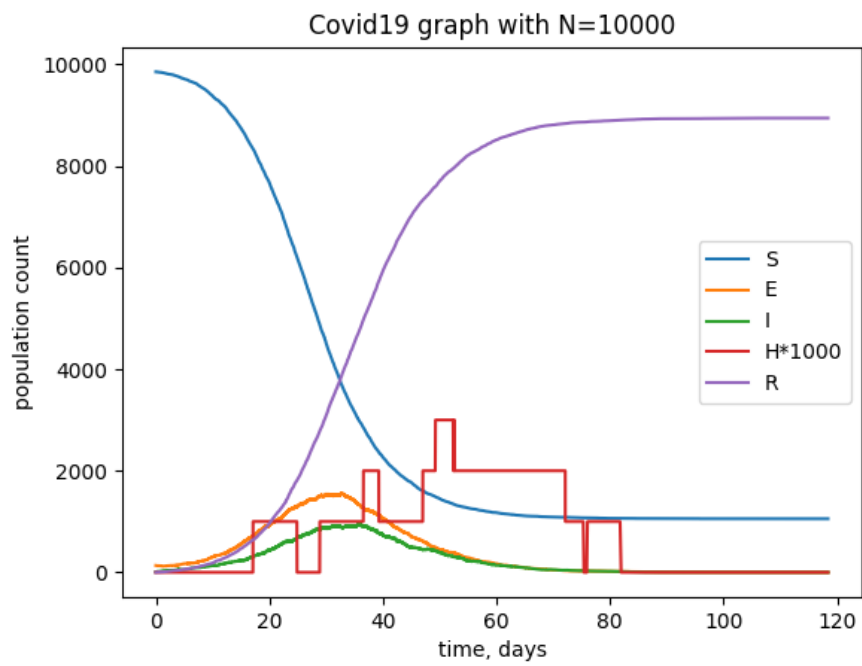


Figure 4: Sample covid trajectory

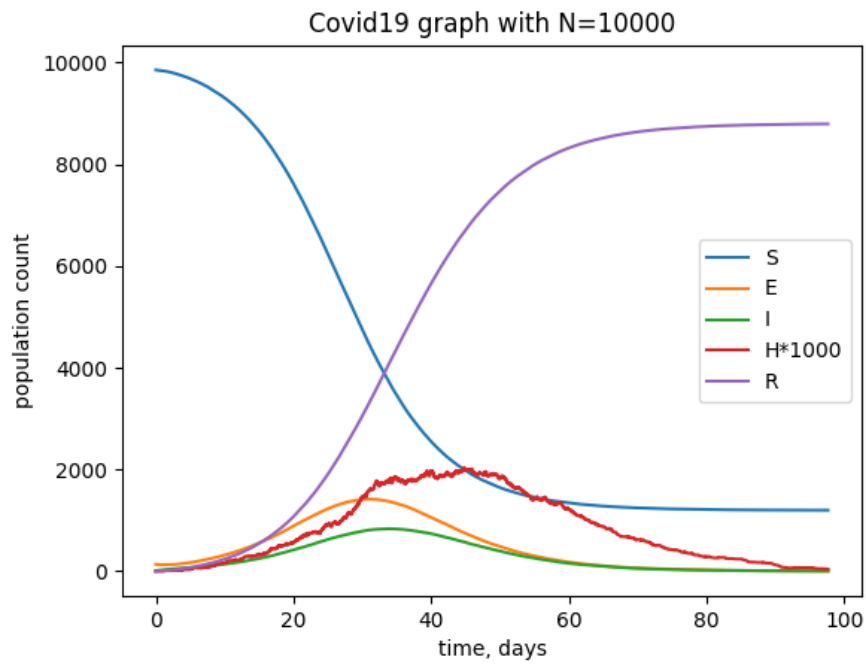


Figure 5: Mean covid trajectory of 30 simulations

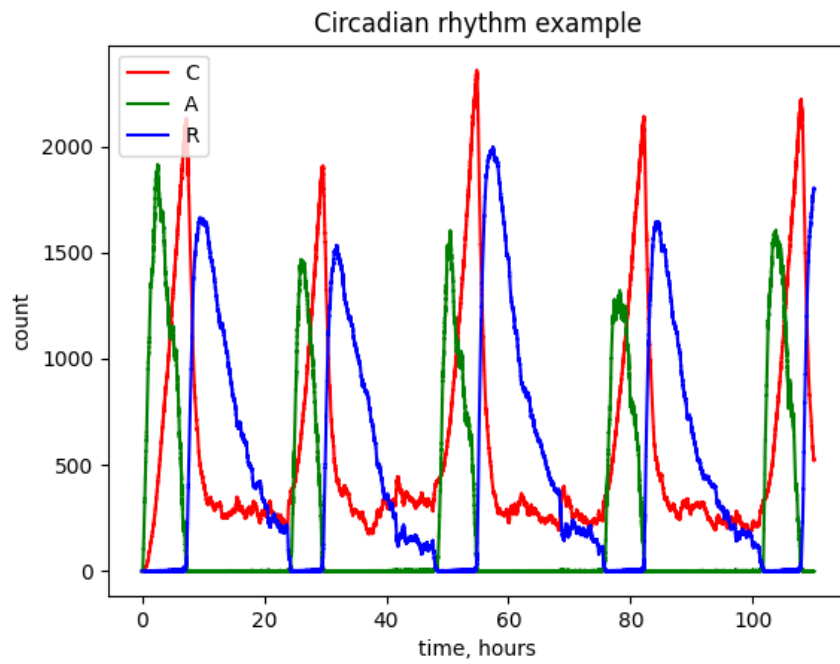


Figure 6: Sample circadian rhythm trajectory

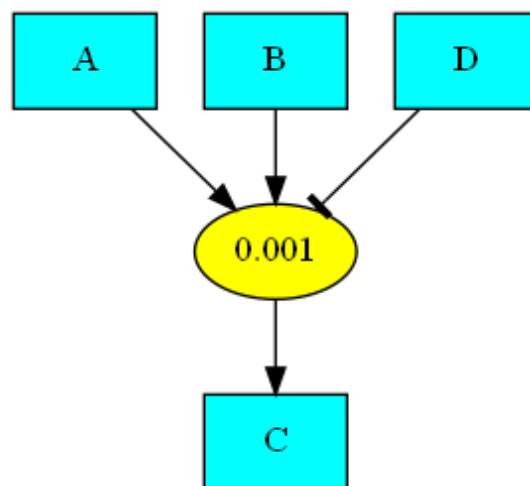


Figure 7: Intro reaction graph

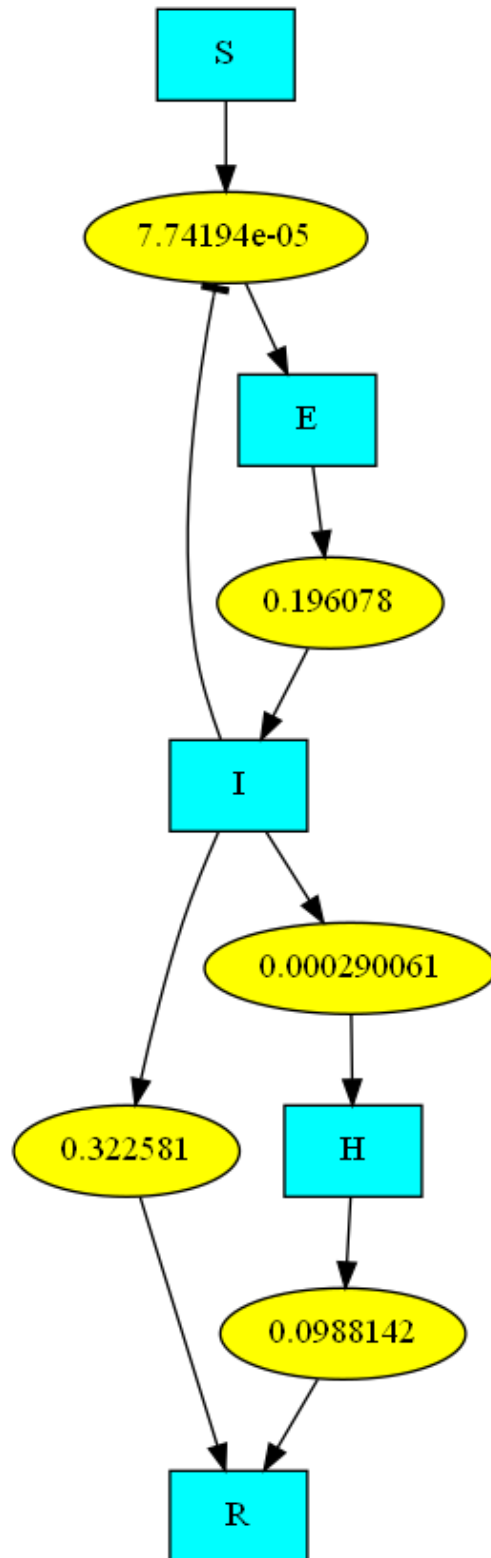


Figure 8: Covid reaction graph

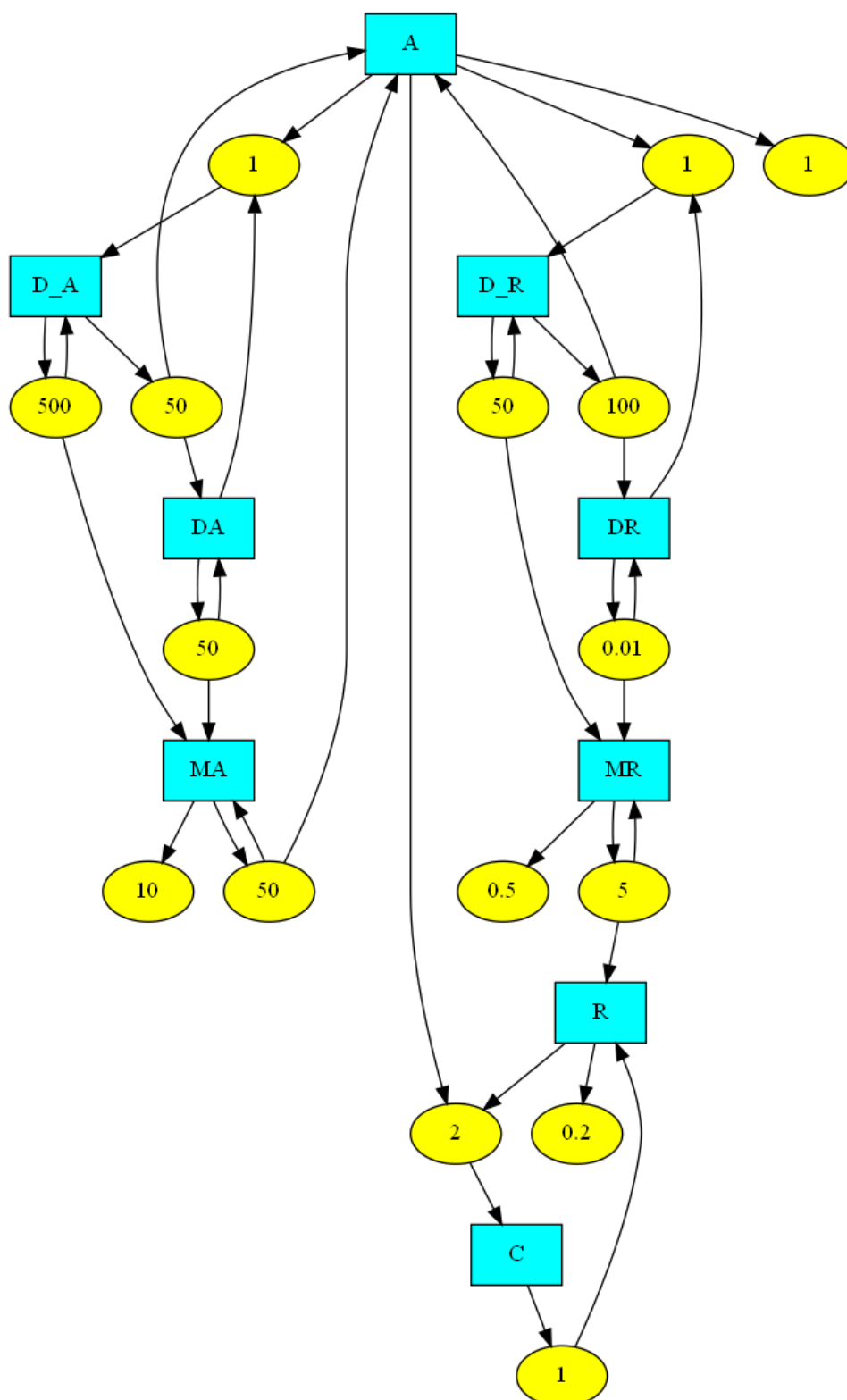


Figure 9: Circadian reaction graph

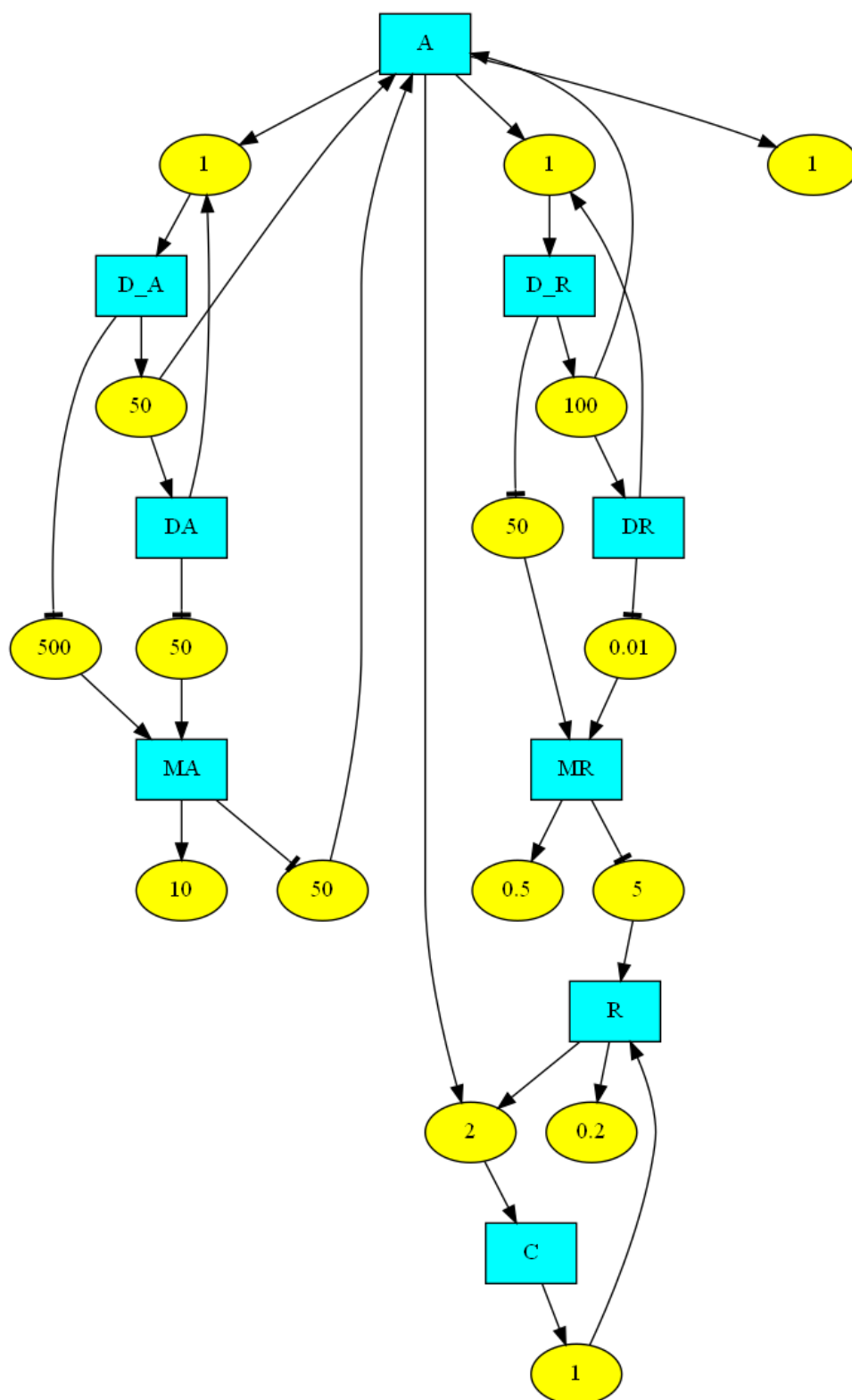


Figure 10: Circadian alternative reaction graph