

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

```
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 // Requirement 7 use of monitor
9 class hospitalized_monitor: public simulation_monitor {
10 private:
11     double_t hospitalized_acc{0.0};
12     double_t last_time{0.0};
13 public:
14     size_t max_hospitalized{0};
15
16     void monitor(SimulationState &state) override {
17         auto currently_hospitalized = state.reactants.get("H").amount;
18
19         if (currently_hospitalized > max_hospitalized) {
20             max_hospitalized = currently_hospitalized;
21         }
22
23         hospitalized_acc += (currently_hospitalized * (state.time - last_time));
24
25         last_time = state.time;
26     }
27 }
```

```

27
28     double_t get_mean_hospitalized() const {
29         return (hospitalized_acc / last_time);
30     }
31 };
32
33 void simulate_covid() {
34     std::cout << "Simulating covid19 example with hospitalized monitor" << std::endl;
35     Vessel covid_vessel = seihr(10000);
36
37     std::cout << covid_vessel << std::endl;
38     covid_vessel.visualize_reactions("covid_graph.png");
39
40     std::cout << "reaction graph can be seen at: covid_graph.png" << std::endl;
41
42     hospitalized_monitor monitor{};
43
44     auto trajectory = covid_vessel.do_simulation(120, monitor);
45
46     std::cout << "Simulation done" << std::endl;
47     std::cout << "Max hospitalized: " << monitor.max_hospitalized << std::endl;
48     std::cout << "Mean hospitalized: " << monitor.get_mean_hospitalized() << std::endl;
49
50     std::cout << "Writing trajectory to csv file at covid_output.csv" << std::endl;
51     trajectory->write_csv("covid_output.csv");
52     std::cout << "Turn it into a graph using python ./draw_graph.py covid release" << std::endl;
53 }
54
55 void simulate_covid_multiple() {
56     std::cout << "Simulating covid19 example 30 times and calculating mean" << std::endl;
57     Vessel covid_vessel = seihr(10000);
58
59     auto trajectories = covid_vessel.do_multiple_simulations(110, 100);
60
61     std::cout << "Simulations done" << std::endl << "Computing mean trajectory" << std::endl;
62
63     auto mean = SimulationTrajectory::compute_mean_trajectory(trajectories);
64
65     std::cout << "Writing mean trajectory to csv file at covid_output_multiple.csv" << std::endl;
66     mean.write_csv("covid_output_multiple.csv");
67     std::cout << "Turn it into a graph using python ./draw_graph.py covid ↗
68     ↪covid_output_multiple.csv" << std::endl;
69 }
70
71 void simulate_introduction() {
72     std::cout << "Simulating introduction example" << std::endl;
73     Vessel introduction_vessel = introduction(25, 50, 1, 0.001);
74     std::cout << introduction_vessel << std::endl;
75
76     introduction_vessel.visualize_reactions("intro_graph.png");
77
78     auto trajectory = introduction_vessel.do_simulation(400);
79
80     trajectory->write_csv("intro_output.csv");
81 }
82
83 void simulate_circadian() {
84     std::cout << "Simulating circadian rhythm example..." << std::endl;
85     Vessel oscillator = circadian_oscillator();
86

```

```

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

```

Listing 3: vessels.h

```

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         // Requirement 9 compute mean
61         static SimulationTrajectory ↵
62         ↪compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& trajectories);
63
64         void insert(SimulationState state) {
65             if (state.time > largest_time) {
66                 largest_time = state.time;
67             }
68
69             map_type::insert({state.time, std::move(state)});
70         }
71
72         // Requirement 6 output trajectory
73         void write_csv(const std::string& path);
74
75         double_t get_max_time() {
76             return largest_time;
77         }
78     };
79
80     // Requirement 1 operators for DSEL
81     class Vessel {
82     private:
83         std::vector<Reaction> reactions{};
84         SymbolTable<Reactant> reactants;
85     public:

```

```

86     Vessel() = default;
87
88     Vessel(const Vessel &val) {
89         reactions = val.reactions;
90         reactants = val.reactants;
91     }
92
93     Vessel (Vessel&& rval) {
94         reactions = std::move(rval.reactions);
95         reactants = std::move(rval.reactants);
96     };
97
98     Reactant& operator()(std::string name, size_t initial_amount) {
99         Reactant newReactant{std::move(name), initial_amount};
100
101         reactants.put(newReactant.name, newReactant);
102
103         return reactants.get(newReactant.name);
104     }
105
106     Reaction operator()(Reaction&& reaction, double_t rate) {
107         reaction.rate = rate;
108
109         reactions.push_back(reaction);
110
111         return reaction;
112     }
113
114     Reaction operator()(Reaction&& reaction, std::initializer_list<Reactant> catalysts, ↵
↵double rate) {
115         reaction.rate = rate;
116         reaction.catalysts = catalysts;
117
118         // Add to vessel reactions
119         reactions.push_back(reaction);
120
121         return reaction;
122     }
123
124     Reaction operator()(Reaction&& reaction, Reactant catalyst, double_t rate) {
125         reaction.rate = rate;
126         reaction.catalysts = {catalyst};
127
128         // Add to vessel reactions
129         reactions.push_back(reaction);
130
131         return reaction;
132     }
133
134
135     Reactant& environment() {
136         if (reactants.contains("__env__")) {
137             return reactants.get("__env__");
138         }
139
140         auto newReactant = Reactant("__env__", 0, 0);
141
142         reactants.put(newReactant.name, newReactant);
143
144         return reactants.get(newReactant.name);
145     }

```



```

146
147 // Requirement 2 network graph
148 void visualize_reactions(const std::string& filename);
149
150 // Requirement 10 optimized algorithm
151 std::shared_ptr<SimulationTrajectory> do_simulation2(double_t end_time, ↵
↵simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
152
153 // Requirement 4 simulation
154 std::shared_ptr<SimulationTrajectory> do_simulation(double_t end_time, ↵
↵simulation_monitor& monitor = EMPTY_SIMULATION_MONITOR);
155
156 // Requirement 8 parallelization
157 std::vector<std::shared_ptr<SimulationTrajectory>> do_multiple_simulations(double_t ↵
↵end_time, size_t simulations_to_run);
158
159 // Requirement 2 pretty print
160 friend std::ostream& operator<<(std::ostream& s, const Vessel& vessel);
161 };
162
163
164
165 }
166
167 #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 // Requirement 2
13 std::ostream &operator<<(std::ostream &s, const Vessel &vessel) {
14     s << "{" << std::endl;
15     for (const auto& reaction: vessel.reactions) {
16         s << "\t" << reaction;
17         if (&reaction != &vessel.reactions.back()) {
18             s << ",";
19         }
20         s << std::endl;
21     }
22     return s << "}";
23 }
24
25 // Requirement 2
26 void Vessel::visualize_reactions(const std::string& filename) {
27     std::stringstream str;
28     SymbolTable<std::string> node_map{};
29
30     str << "digraph {" << std::endl;
31
32     auto i = 0;
33     for (auto& reactant: reactants.getMap()) {
34         if (reactant.second.name != "__env__") {

```

```

35         node_map.put(reactant.second.name, "s" + std::to_string(i));
36
37         str << node_map.get(reactant.second.name)
38             << "[label=\"" << reactant.second.name <<
    ↪ "\" ,shape=\"box\" ,style=\"filled\" ,fillcolor=\"cyan\" ];\" << std::endl;
39         i++;
40     }
41 }
42
43 i = 0;
44 for (auto& reaction: reactions) {
45     std::string reaction_node{"r" + std::to_string(i)};
46
47     str << reaction_node << "[label=\"" << reaction.rate <<
    ↪ "\" ,shape=\"oval\" ,style=\"filled\" ,fillcolor=\"yellow\" ];\" << std::endl;
48     if (reaction.catalysts.has_value()) {
49         for (auto& catalyst: reaction.catalysts.value()) {
50             str << node_map.get(catalyst.name) << " -> " << reaction_node << "
    ↪
    ↪ "[arrowhead=\"tee\" ];\" << std::endl;
51         }
52     }
53     for (auto& reactant: reaction.from) {
54         if (reactant.name != "__env__") {
55             str << node_map.get(reactant.name) << " -> " << reaction_node << " ,\" <<
    ↪
    ↪ std::endl;
56         }
57     }
58     for (auto& product: reaction.to) {
59         if (product.name != "__env__") {
60             str << reaction_node << " -> " << node_map.get(product.name) << " ,\" <<
    ↪
    ↪ std::endl;
61         }
62     }
63
64     i++;
65 }
66
67 str << "}";
68
69 std::ofstream dotfile;
70 dotfile.open(filename + ".dot");
71 dotfile << str.str();
72 dotfile.close();
73
74 std::stringstream command_builder;
75 command_builder << "dot -Tpng -o " << filename << " " << filename << ".dot";
76 system(command_builder.str().c_str());
77 }
78
79 // Requirement 10 alternative simulation
80 std::shared_ptr<SimulationTrajectory> Vessel::do_simulation2(double_t end_time,
    ↪
    ↪ simulation_monitor &monitor) {
81     SimulationTrajectory trajectory{};
82     double_t t{0};
83
84     auto thread_id = std::this_thread::get_id();
85     auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
86     std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
87
88     // Insert initial state
89     trajectory.insert(SimulationState{reactants, t});

```

```

90
91 while (t <= end_time) {
92     for (Reaction& reaction: reactions) {
93         // New: using new compute delay function
94         reaction.compute_delay2(trajjectory.at(t), engine);
95     }
96
97     auto r = reactions.front();
98
99     // Select Reaction with min delay which is not -1
100    for (auto& reaction: reactions) {
101        if (reaction.delay == -1) {
102            continue;
103        } else if (reaction.delay < r.delay) {
104            r = reaction;
105        } else if (r.delay == -1) {
106            r = reaction;
107        }
108    }
109
110    // Stop if we have no reactions to do, thus r.delay == -1
111    if (r.delay == -1) {
112        break;
113    }
114
115    auto& last_state = trajjectory.at(t);
116
117    t += r.delay;
118
119    SimulationState state{last_state.reactants, t};
120
121    if (
122        std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
123        ↪state.reactants.get(e.name).amount >= e.required;}) &&
124        (
125            !r.catalysts.has_value() ||
126            std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
127            ↪[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
128        )
129    ) {
130        for (auto& reactant: r.from) {
131            state.reactants.get(reactant.name).amount -= reactant.required;
132        }
133        for (auto& reactant: r.to) {
134            state.reactants.get(reactant.name).amount += reactant.required;
135        }
136
137        trajjectory.insert(std::move(state));
138
139        monitor.monitor(trajjectory.at(t));
140    }
141
142    return std::make_shared<SimulationTrajectory>(std::move(trajjectory));
143 }
144
145 // Requirement 4 simulation
146 std::shared_ptr<SimulationTrajectory> Vessel::do_simulation(double_t end_time,
147 ↪simulation_monitor &monitor) {
148     SimulationTrajectory trajjectory{};
149     double_t t{0};

```

```

148
149 auto thread_id = std::this_thread::get_id();
150 auto epoch = std::chrono::system_clock::now().time_since_epoch().count();
151 std::default_random_engine engine(epoch * (std::hash<std::thread::id>{}(thread_id)));
152
153 // Insert initial state
154 trajectory.insert(SimulationState{reactants, t});
155
156 while (t <= end_time) {
157     for (Reaction& reaction: reactions) {
158         reaction.compute_delay(trajectory.at(t), engine);
159     }
160
161     auto r = reactions.front();
162
163     // Select Reaction with min delay which is not -1
164     for (auto& reaction: reactions) {
165         if (reaction.delay == -1) {
166             continue;
167         } else if (reaction.delay < r.delay) {
168             r = reaction;
169         } else if (r.delay == -1 && reaction.delay != -1) {
170             r = reaction;
171         }
172     }
173
174     // Stop if we have no reactions to do, thus r.delay == -1
175     if (r.delay == -1) {
176         break;
177     }
178
179     auto& last_state = trajectory.at(t);
180
181     t += r.delay;
182
183     SimulationState state{last_state.reactants, t};
184
185     if (
186         std::all_of(r.from.begin(), r.from.end(), [&state](const Reactant& e){return
187         ↪state.reactants.get(e.name).amount >= e.required;}) &&
188         (
189             !r.catalysts.has_value() ||
190             std::all_of(r.catalysts.value().begin(), r.catalysts.value().end(),
191             ↪[&state](const Reactant& e){return state.reactants.get(e.name).amount >= e.required;})
192         )
193     ) {
194         for (auto& reactant: r.from) {
195             state.reactants.get(reactant.name).amount -= reactant.required;
196         }
197         for (auto& reactant: r.to) {
198             state.reactants.get(reactant.name).amount += reactant.required;
199         }
200
201         trajectory.insert(std::move(state));
202
203         monitor.monitor(trajectory.at(t));
204     }
205
206     return std::make_shared<SimulationTrajectory>(std::move(trajectory));
207 }


```

```

207
208 // Requirement 8 multiple at same time
209 std::vector<std::shared_ptr<SimulationTrajectory>>
210 Vessel::do_multiple_simulations(double_t end_time, size_t simulations_to_run) {
211     std::vector<std::shared_ptr<SimulationTrajectory>> result{};
212     result.reserve(simulations_to_run);
213
214     auto cores = std::thread::hardware_concurrency();
215     int jobs = std::min(simulations_to_run, (cores - 1));
216     auto simulations_per_job = simulations_to_run / jobs;
217
218     auto futures = ↵
↪std::vector<std::future<std::vector<std::shared_ptr<SimulationTrajectory>>>>{};
219
220     auto lambda = [&vessel = *this, &end_time](size_t to_run){
221         auto simulations = std::vector<std::shared_ptr<SimulationTrajectory>>{};
222         simulations.reserve(to_run);
223
224         auto new_vessel = Vessel(vessel);
225
226         for (int i = 0; i < to_run; ++i) {
227             simulations.push_back(new_vessel.do_simulation(end_time));
228         }
229
230         return simulations;
231     };
232
233     for (int i = 0; i < jobs; ++i) {
234         futures.push_back(std::async(std::launch::async, lambda, simulations_per_job));
235     }
236     auto missing_simulations = simulations_to_run - (simulations_per_job * jobs);
237     if (missing_simulations != 0) {
238         futures.push_back(std::async(std::launch::async, lambda, missing_simulations));
239     }
240
241     for (auto& future: futures) {
242         auto future_result = future.get();
243         for (auto& res: future_result) {
244             result.push_back(std::move(res));
245         }
246     }
247
248     return result;
249 }
250
251 // Private function for calculation interpolated value
252 double_t SimulationTrajectory::compute_interpolated_value(const std::string& key, ↵
↪SimulationState& s0, SimulationState& s1, double_t x) {
253     return
254         s0.reactants.get(key).amount
255         + ((
256             ( (double_t) s1.reactants.get(key).amount - (double_t) ↵
↪s0.reactants.get(key).amount) /
257             ( s1.time - s0.time )
258             ) * (x - s0.time));
259 }
260
261 // Requirement 9 compute mean trajectory
262 SimulationTrajectory ↵
↪SimulationTrajectory::compute_mean_trajectory(std::vector<std::shared_ptr<SimulationTrajectory>>& ↵
↪trajectories) {

```

```

263     auto average_delay = trajectories.front()->get_max_time() / trajectories.front()->size();
264
265     // Get a list of all keys
266     std::vector<std::string> reactant_keys{};
267     for (auto& reactant: trajectories.front()->begin()->second.reactants) {
268         reactant_keys.push_back(reactant.second.name);
269     }
270
271     // Find upper bound for mean trajectory
272     double_t upper_bound{-1.0};
273     for (auto& trajectory: trajectories) {
274         if (upper_bound == -1.0 || trajectory->get_max_time() < upper_bound) {
275             upper_bound = trajectory->get_max_time();
276         }
277     }
278
279     SimulationTrajectory mean_trajectory{};
280
281     for (auto& trajectory: trajectories) {
282         auto iterator = trajectory->begin();
283         SimulationState& s0 = iterator->second;
284         iterator++;
285         SimulationState& s1 = iterator->second;
286
287         double_t t{0};
288
289         while((t + average_delay) <= upper_bound) {
290             if (t >= s0.time) {
291                 if (t <= s1.time) {
292                     if (!mean_trajectory.contains(t)) {
293                         mean_trajectory.insert((SimulationState{{}}, t));
294                     }
295
296                     for (auto& key: reactant_keys) {
297                         auto interpolated_value = 
298                         ↪SimulationTrajectory::compute_interpolated_value(key, s0, s1, t);
299
300                         auto& table = mean_trajectory.at(t).reactants;
301
302                         if (!table.contains(key)) {
303                             Reactant reactant{key, 0.0};
304                             table.put(key, reactant);
305                         }
306
307                         table.get(key).amount += interpolated_value;
308                     }
309
310                     t += average_delay;
311                 } else {
312                     s0 = s1;
313                     iterator++;
314
315                     if (iterator != trajectory->end()) {
316                         s1 = iterator->second;
317                     } else {
318                         break;
319                     }
320                 }
321             }
322         }

```

```

323     for (double_t i = 0; (i + average_delay) <= upper_bound; i += average_delay) {
324         for (auto& key: reactant_keys) {
325             mean_trajectory.at(i).reactants.get(key).amount /= trajectories.size();
326         }
327     }
328 }
329
330 return std::move(mean_trajectory);
331 }
332
333 // Requirement 6 output to csv which can then be turned into a graph via python script
334 void SimulationTrajectory::write_csv(const std::string &path) {
335     std::ofstream csv_file;
336     csv_file.open(path);
337
338     auto reactants = at(0).reactants;
339
340     for (auto& reactant : reactants) {
341         csv_file << reactant.second.name << ",";
342     }
343     csv_file << "time" << std::endl;
344
345     for (auto& state : *this) {
346         for (auto& reactant: reactants) {
347             csv_file << state.second.reactants.get(reactant.second.name).amount << ",";
348         }
349         csv_file << state.second.time << std::endl;
350     }
351
352     csv_file.close();
353 }
354 }

```

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     // Requirement 1 operator for DSEL
65     Reaction operator>=(Reactant other);
66
67     // Requirement 1 operator for DSEL
68     Reaction operator>=(ReactantCollection other);
69
70     // Requirement 1 operator for DSEL
71     ReactantCollection operator+(const Reactant& other);
72
73     bool operator<(const Reactant& other) const;
74
75     // Requirement 1 operator for DSEL
76     Reactant operator*(size_t req) {
77         required = req;
78         return *this;
79     }
80
81 };
82
83 class ReactantCollection: public std::set<Reactant> {
84 public:
85     using std::set<Reactant>::set;
86     // Requirement 1 operator for DSEL
87     Reaction operator>=(Reactant other);
88     // Requirement 1 operator for DSEL

```



```

89     Reaction operator>>=(ReactantCollection other);
90 };
91
92 class Reaction {
93 public:
94     std::set<Reactant> from;
95     std::set<Reactant> to;
96     std::optional<std::vector<Reactant>> catalysts;
97     double_t rate{};
98     double_t delay{-1};
99
100     Reaction(std::set<Reactant> from, std::set<Reactant> to):
101         from(from),
102         to(to)
103     {}
104
105     Reaction(std::set<Reactant> from, std::set<Reactant> to, std::initializer_list<Reactant> ↗
↪catalysts, double rate):
106         from(from),
107         to(to),
108         catalysts(catalysts),
109         rate(rate)
110     {}
111
112     Reaction(std::set<Reactant> from, std::set<Reactant> to, double rate):
113         from(from),
114         to(to),
115         catalysts{},
116         rate(rate)
117     {}
118
119     void compute_delay(SimulationState& state, std::default_random_engine& engine);
120     void compute_delay2(SimulationState& state, std::default_random_engine& engine);
121
122     friend std::ostream &operator<<(std::ostream &s, const Reaction &reaction);
123 };
124
125 struct SimulationState {
126 public:
127     SymbolTable<Reactant> reactants;
128     double_t time;
129
130     SimulationState(SymbolTable<Reactant> reactants, double_t time):
131         reactants{reactants},
132         time{time}
133     {};
134
135     SimulationState(const SimulationState&) = default;
136     SimulationState(SimulationState&&) = default;
137
138     SimulationState& operator=(const SimulationState &) = default;
139     SimulationState& operator=(SimulationState&&) = default;
140
141     ~SimulationState() = default;
142
143     friend std::ostream &operator<<(std::ostream &, const SimulationState &);
144 };
145
146 }
147
148 #endif //SP_EXAM_PROJECT_DATA_H

```

```

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 : ↵

```

```

61     state.reactants.get(reactant.name).amount;
62         reactant_amount *= amount;
63     }
64     // New: check if amount is 0 already
65     if (reactant_amount == 0) {
66         delay = -1;
67         return;
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 : ↗
90         state.reactants.get(reactant.name).amount;
91         reactant_amount *= amount;
92     }
93     if (catalysts.has_value()) {
94         for (auto& catalyst: catalysts.value()) {
95             catalyst_amount *= state.reactants.get(catalyst.name).amount;
96         }
97     }
98
99     double_t rate_k = rate * reactant_amount * catalyst_amount;
100
101     if (rate_k > 0) {
102         delay = std::exponential_distribution<double_t>(rate_k)(engine);
103     } else {
104         delay = -1;
105     }
106 }
107
108 std::ostream &operator<<(std::ostream &s, const SimulationState& state) {
109     s << "{" << std::endl
110         << "time: " << state.time << "," << std::endl
111         << "reactants: {" << std::endl;
112     for(auto& pair: state.reactants) {
113         s << pair.first << ": " << pair.second.amount << "," << std::endl;
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
13     // Requirement 7 classes for generic system state monitors
14     class simulation_monitor {
15     public:
16         virtual void monitor(SimulationState& state) = 0;
17     };
18
19
20     class empty_simulation_monitor: public simulation_monitor {
21     void monitor(SimulationState &state) override {
22         return;
23     };
24 };
25
26 static auto EMPTY_SIMULATION_MONITOR = empty_simulation_monitor{};
27
28     class basic_simulation_monitor: public simulation_monitor {
29     private:
30         const std::function<void(SimulationState&)> monitor_function;
31     public:
32         basic_simulation_monitor(const std::function<void(SimulationState&)>& monitor_func):
33             simulation_monitor{},
34             monitor_function{monitor_func}
35         {}
36
37         void monitor(SimulationState& state) override {
38             monitor_function(state);
39         }
40     };
41 }
42
43 #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 // Requirement 3 generic symbol table
17 struct SymbolTableException : public std::exception
18 {
19     std::string message;
20 public:
21     explicit SymbolTableException(std::string message): message(std::move(message))
22     {}
23
24     [[nodiscard]] const char* what() const override
25     {
26         return message.c_str();
27     }
28 };
29
30 template<typename T>
31 class SymbolTable {
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) {
42         map = a.map;
43     };
44
45     SymbolTable<T>(SymbolTable<T>&& a) {
46         map = std::move(a.map);
47     };
48
49     ~SymbolTable() = default;
50
51     SymbolTable<T>& operator=(const SymbolTable& a) {
52         map = a.map;
53         return *this;
54     };
55
56     SymbolTable<T>& operator=(SymbolTable&& a) {
57         map = std::move(a.map);
58         return *this;
59     };
60
61     void put(const std::string& key, T value) {
62         if (!map.contains(key)) {
63             map.insert_or_assign(key, value);
64         } else {
65             throw SymbolTableException("Key " + key + " already used");
66         }
67     }
68
69     T& get(const std::string& key) {
70         try {
71             return map.at(key);
72         } catch (std::out_of_range& e) {
73             throw SymbolTableException("Key " + key + " was not found");
74         }
75     }

```

```

76
77     const T& get(const std::string& key) const {
78         try {
79             return map.at(key);
80         } catch (std::out_of_range& e) {
81             throw SymbolTableException("Key " + key + " was not found");
82         }
83     }
84
85     bool contains(const std::string& key) {
86         return map.contains(key);
87     }
88
89     std::unordered_map<std::string, T> getMap() {
90         return map;
91     }
92
93     iterator begin() {
94         return map.begin();
95     }
96
97     iterator end() {
98         return map.end();
99     }
100
101     const_iterator begin() const {
102         return map.begin();
103     }
104
105     const_iterator end() const {
106         return map.end();
107     }
108
109 };
110 }
111
112 #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 }
}

```

Circadian 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

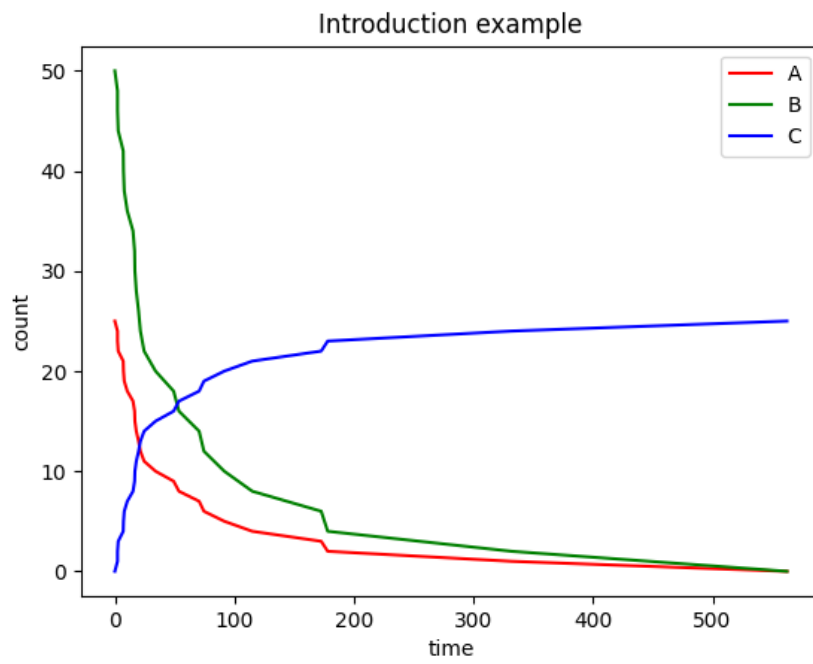


Figure 1: $A(0)=25$, $B(0)=50$, $D=1$

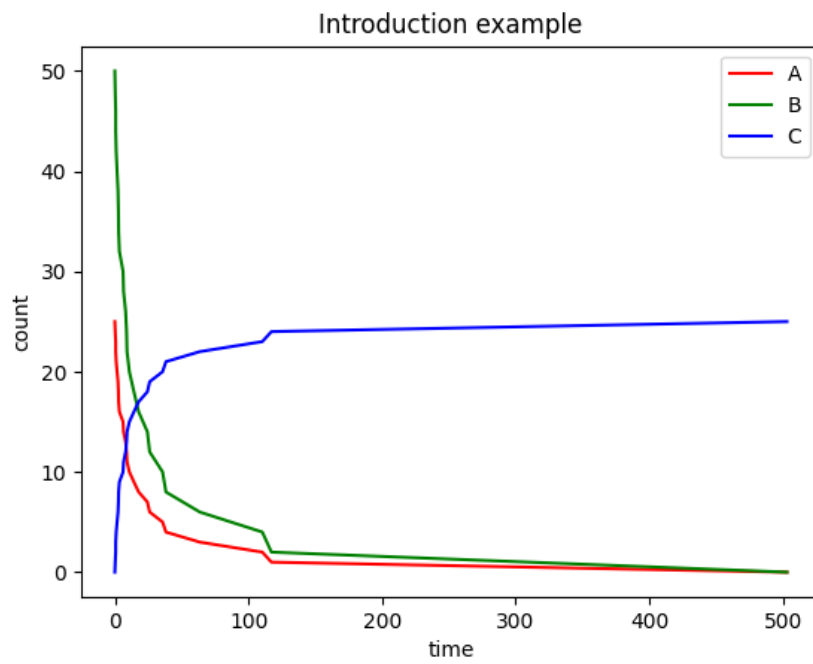


Figure 2: $A(0)=25$, $B(0)=50$, $D=2$

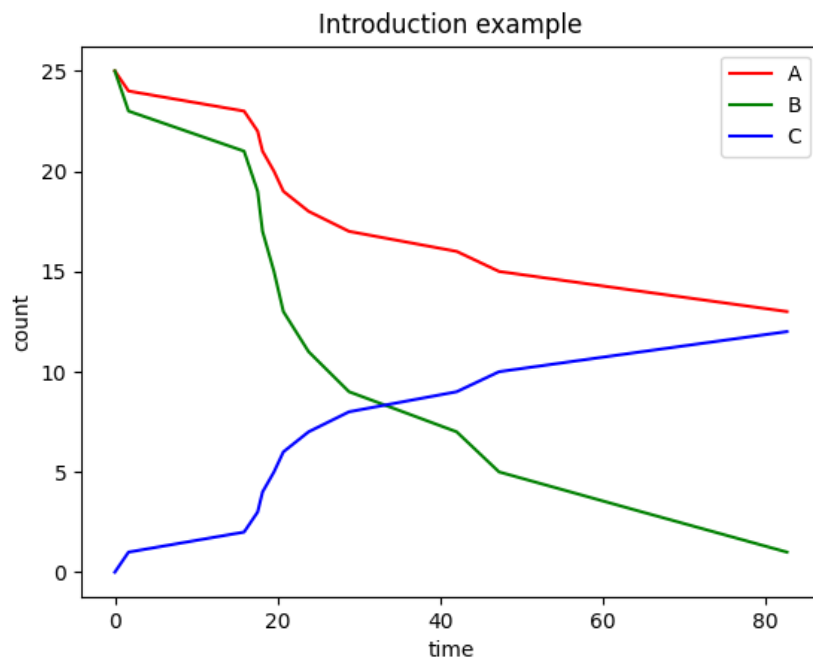


Figure 3: $A(0)=25$, $B(0)=25$, $D=1$

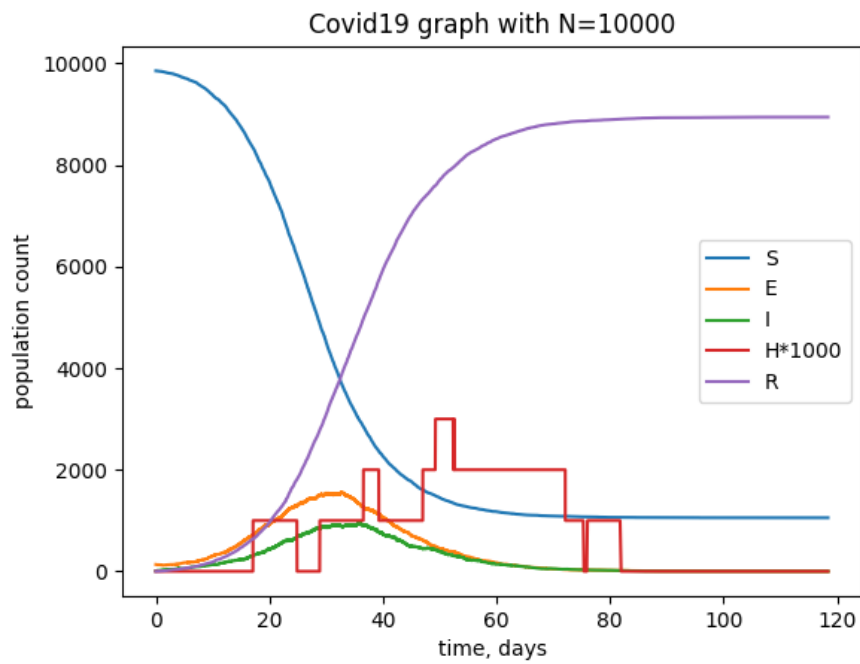


Figure 4: Sample covid trajectory

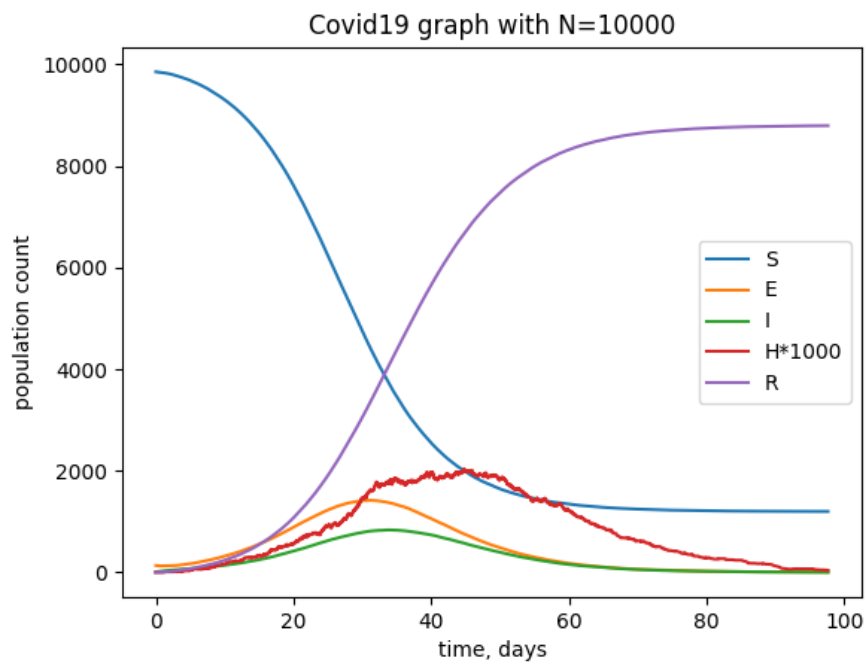


Figure 5: Mean covid trajectory of 30 simulations

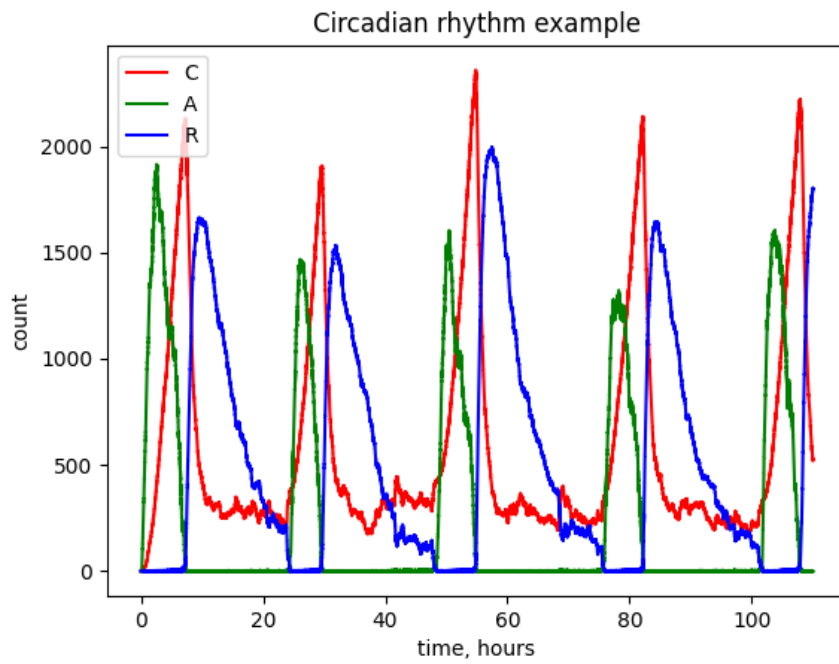


Figure 6: Sample circadian rythm trajectory

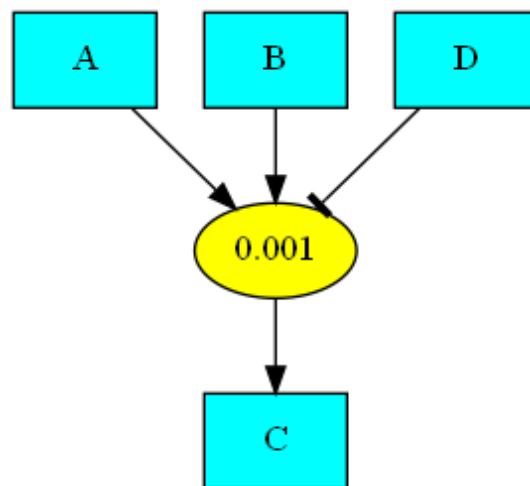


Figure 7: Intro reaction graph

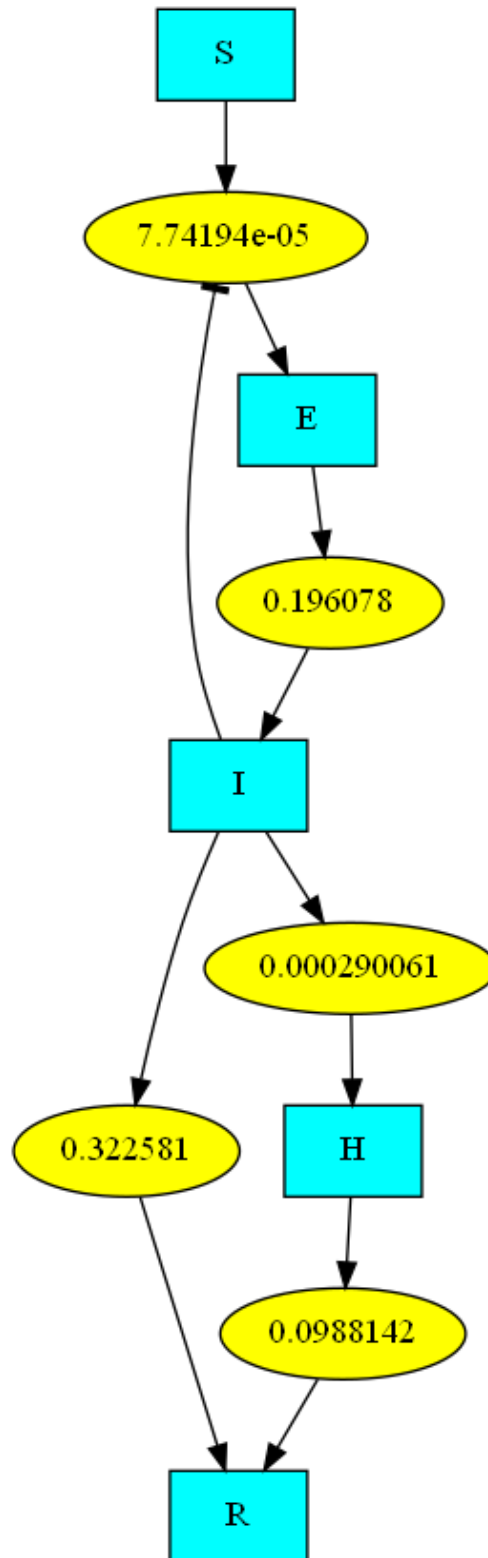


Figure 8: Covid reaction graph

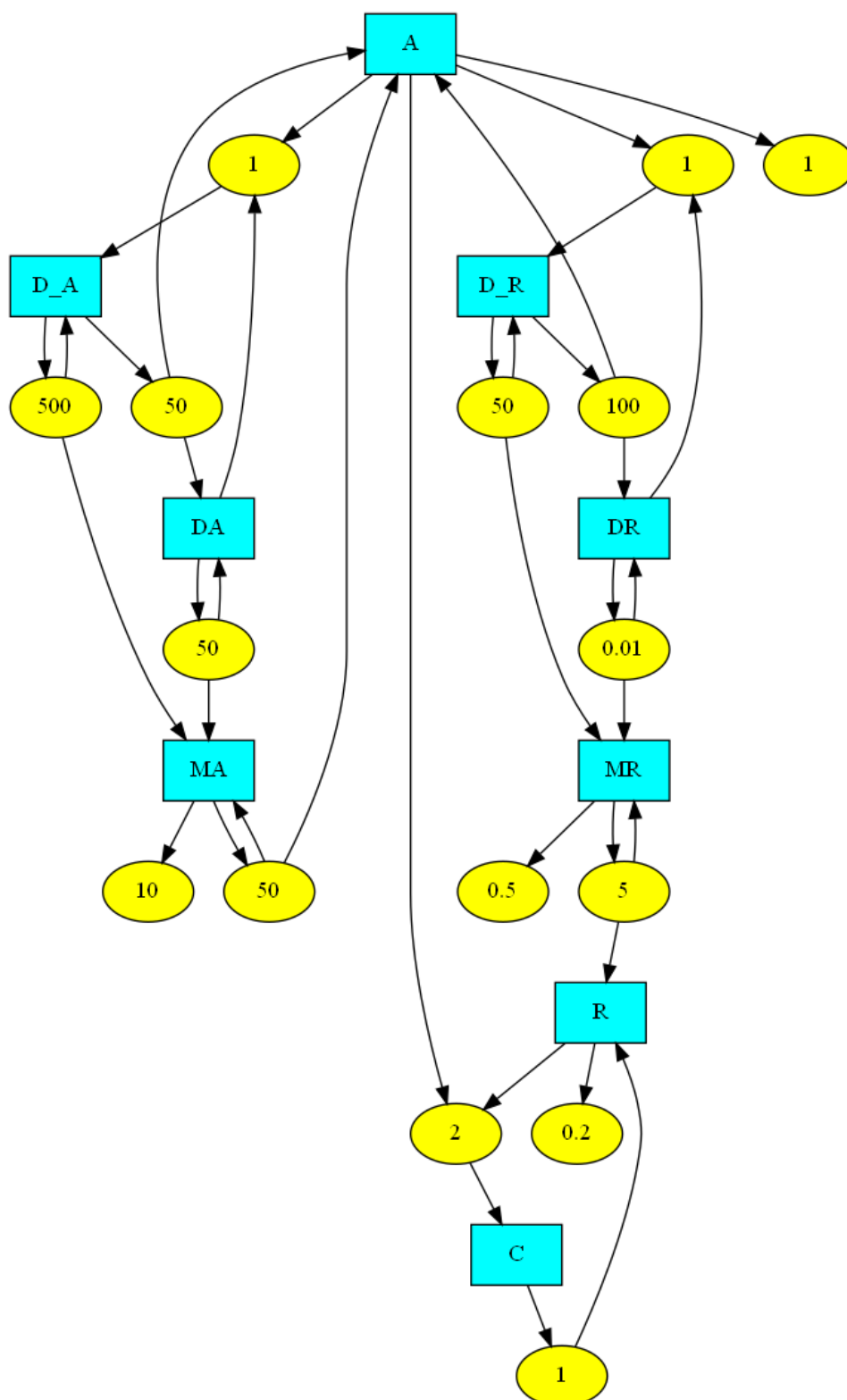


Figure 9: Circadian reaction graph

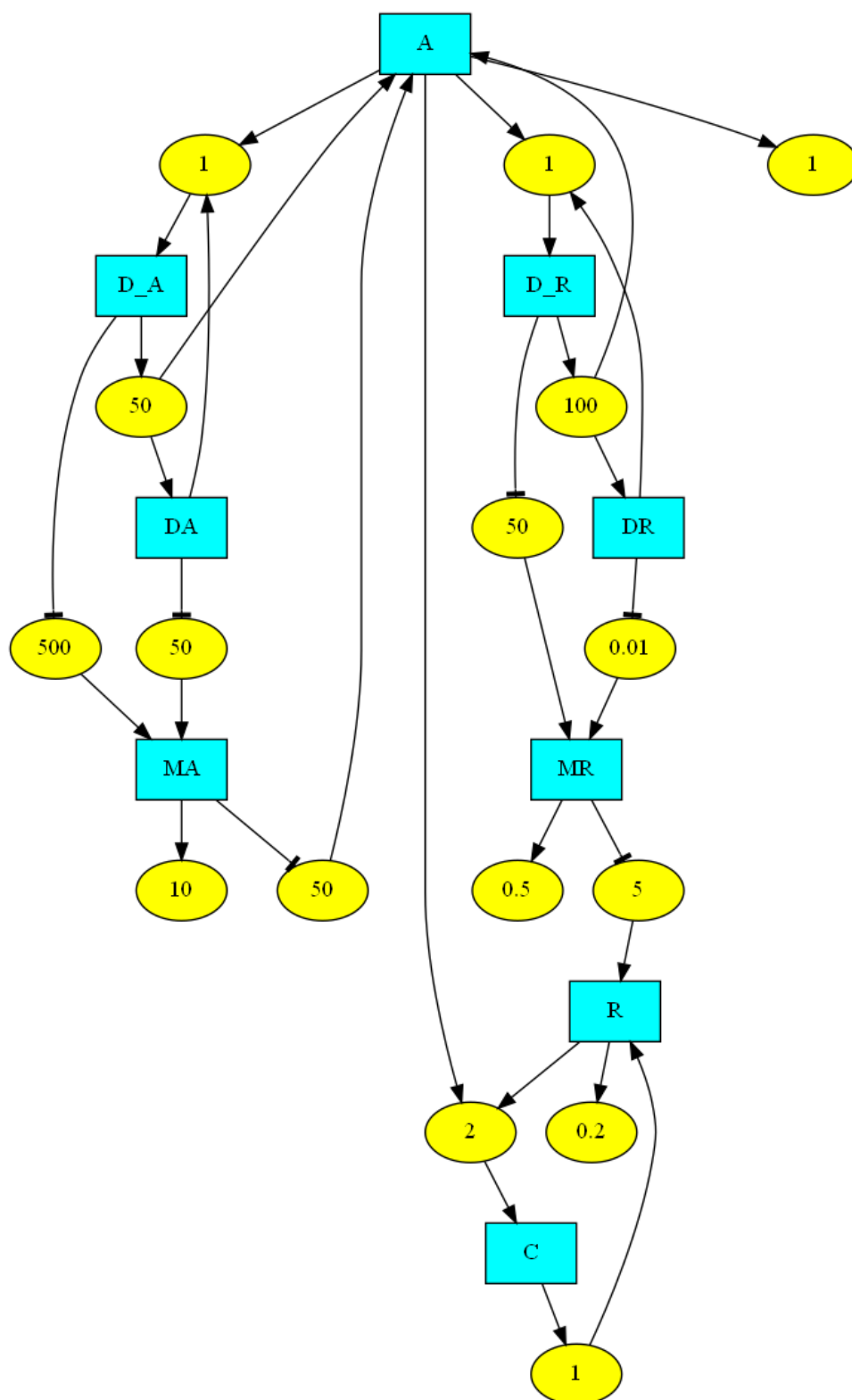


Figure 10: Circadian alternative reaction graph