sP Exam Mini-Project

Nikolaj Kofod Krogh

June 14, 2024

Conclusion

I have included CMakeLists.txt that shows how to build the project and run the tests. It is included as the first listing.

As we can see from the benchmark it is much faster to run the simulation in parallel (22.1 s) compared to running on a single thread (89.7 s), because it is possible to use multiple threads and thereby distribute the load.

```
COVID-19 simulation took 89692.4 ms (89.7 seconds)
Parallel COVID-19 simulation took 22098.6 ms (22.1 seconds)
```

Figure 1: benchmark

```
COVID19 SEIHR_5822763
Peak hospitalized: 1180
```

Figure 2: peak hospitalized NDK

```
COVID19 SEIHR_589755
Peak hospitalized: 115
```

Figure 3: peak hospitalized NNJ

```
Average peak hospitalized over 100 simulations: 3.64
```

Figure 4: average peak over 100 simulations with a population size of 10000

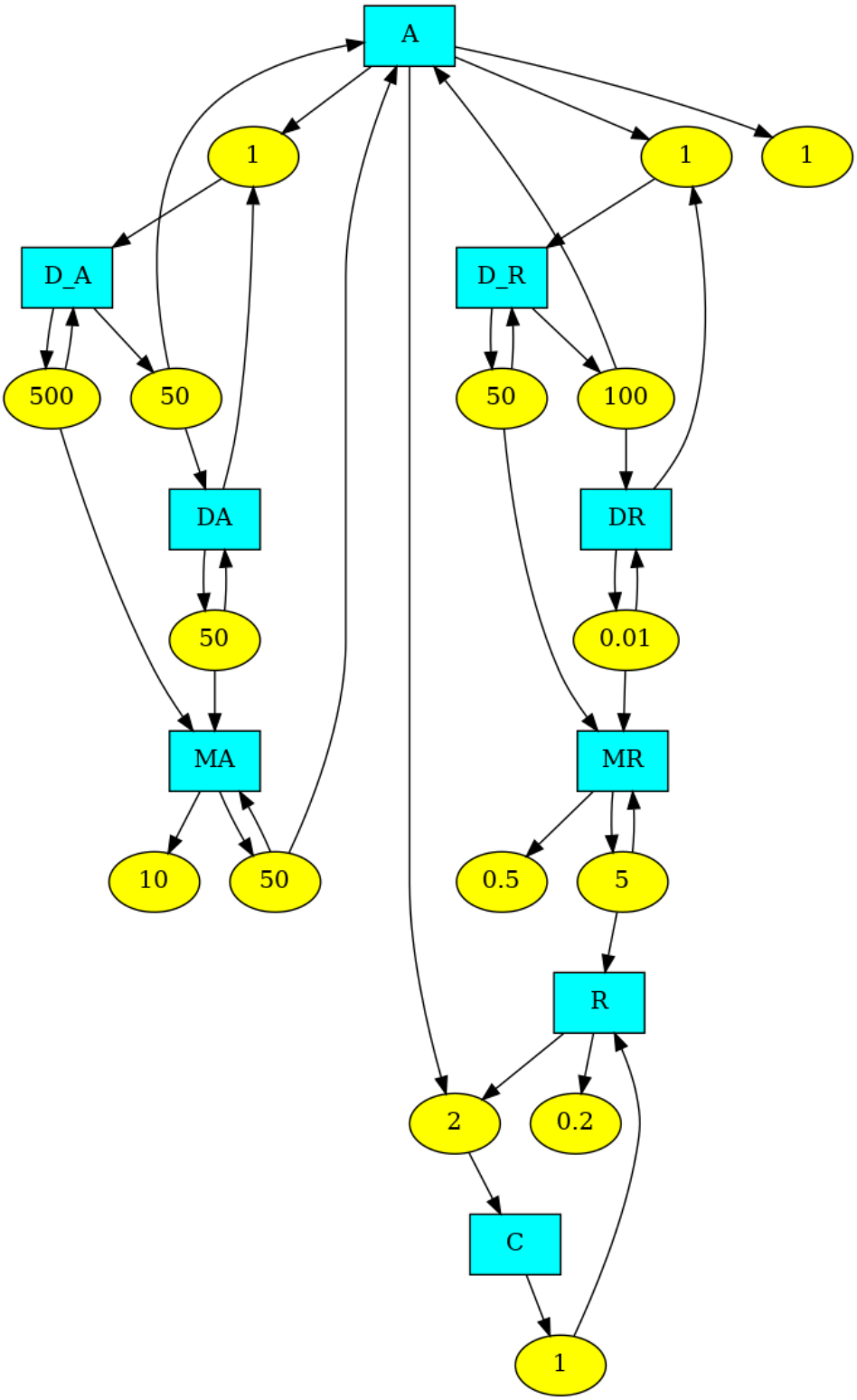

Figure 5: Circadian Rhythm tree

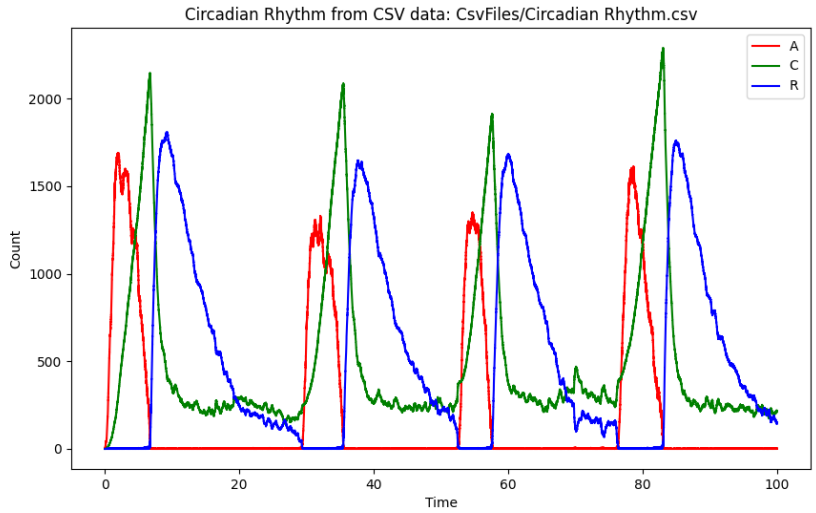

Figure 6: Circadian Rhythm in images

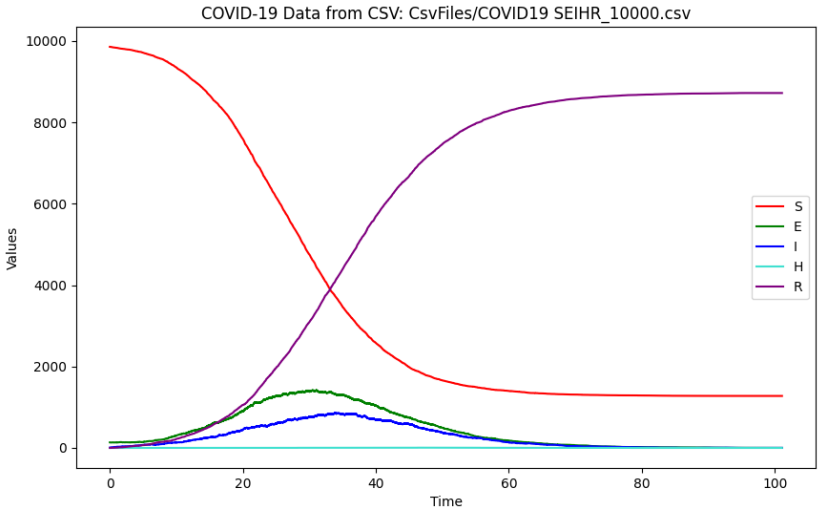

Figure 7: COVID19 population size 10000

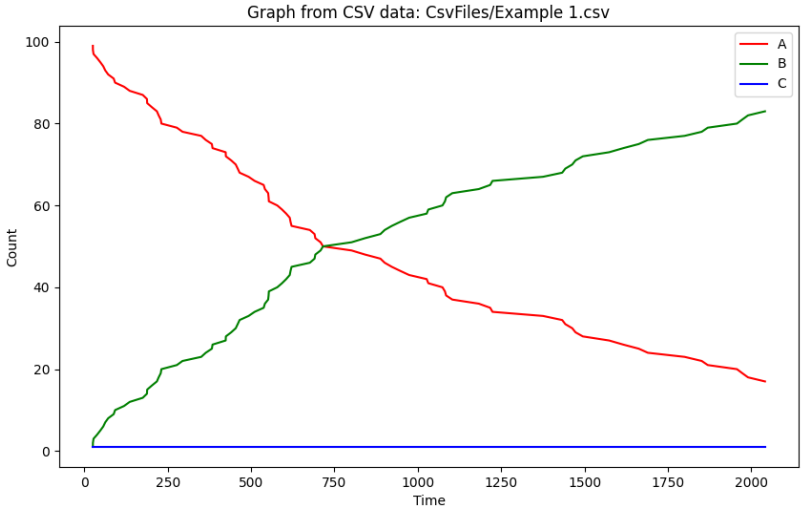

Figure 8: Example 1

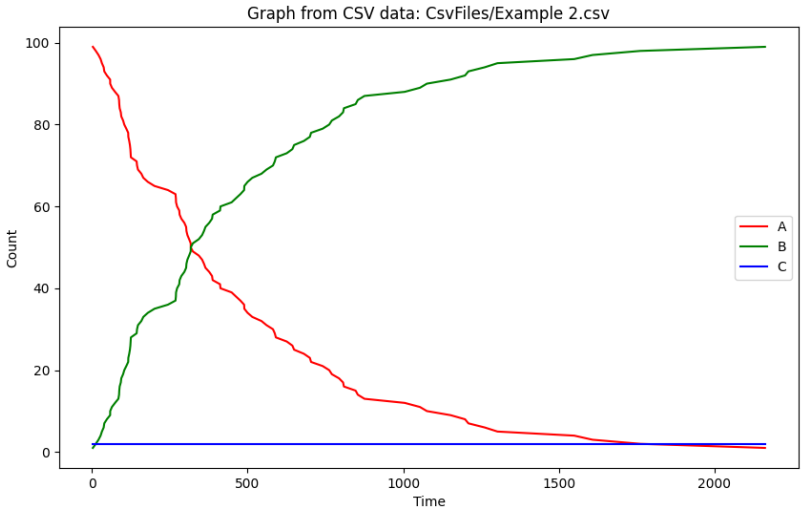

Figure 9: Example 2

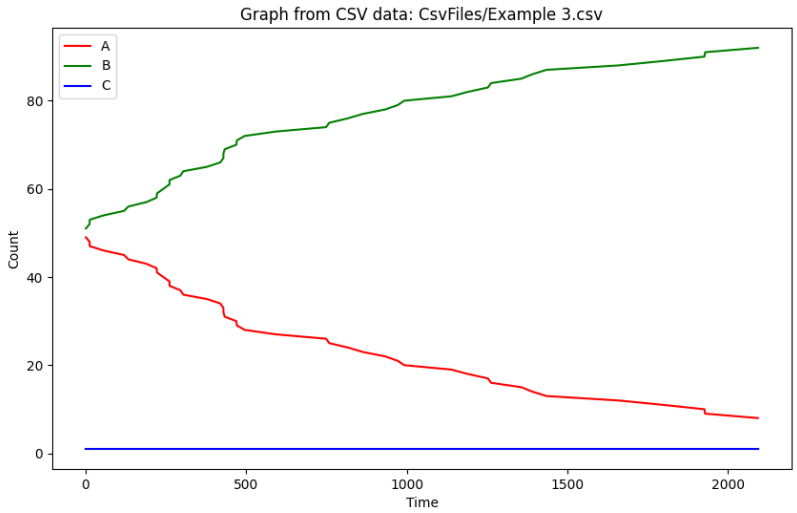

Figure 10: Example 3

Listing 1: ./CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(Exam)

set(CMAKE_CXX_STANDARD 20)
enable_testing()

# Locate GTest
find_package(GTest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

# Define common source files
set(COMMON_SRC
    Arrow.cpp
    Arrow.h
    benchmark.h
    benchmark.cpp
    Environment.cpp
    Environment.h
    Molecule.cpp
    Molecule.h
    ParallelSimulation.h
    Reaction.cpp
    Reaction.h
    Simulation.h
    SymbolTable.h
    Vessel.cpp
    Vessel.h
)

add_executable(Exam main.cpp ${COMMON_SRC})

# Define test source files
set(TEST_SRC
    Tests/SymbolTableTests.cpp
    Tests/VesselTests.cpp
)

# Create a test executable for each test source file
foreach(test_src ${TEST_SRC})
```

```
    get_filename_component(test_name ${test_src} NAME_WE)
    add_executable(${test_name} ${test_src} ${COMMON_SRC})
    target_link_libraries(${test_name} ${GTEST_LIBRARIES} ${GTEST_MAIN_LIBRARIES} pthread)
endforeach()
```

Listing 2: ./main.cpp

```
#include <iostream>
#include <thread>
#include <cmath>
#include "Vessel.h"
#include "Simulation.h"
#include "ParallelSimulation.h"
#include "benchmark.h"
#include <vector>


stochastic::Vessel simulation_circadian_rhythm() {
    const auto alphaA = 50;
    const auto alpha_A = 500;
    const auto alphaR = 0.01;
    const auto alpha_R = 50;
    const auto betaA = 50;
    const auto betaR = 5;
    const auto gammaA = 1;
    const auto gammaR = 1;
    const auto gammaC = 2;
    const auto deltaA = 1;
    const auto deltaR = 0.2;
    const auto deltaMA = 10;
    const auto deltaMR = 0.5;
    const auto thetaA = 50;
    const auto thetaR = 100;


    auto v = stochastic::Vessel{"Circadian Rhythm"};
    const auto env = v.environment();
    const auto DA = v.add("DA", 1);
    const auto D_A = v.add("D_A", 0);
    const auto DR = v.add("DR", 1);
    const auto D_R = v.add("D_R", 0);
    const auto MA = v.add("MA", 0);
    const auto MR = v.add("MR", 0);
    const auto A = v.add("A", 1);
    const auto R = v.add("R", 0);
    const auto C = v.add("C", 0);
    v.add((A + DA) >> gammaA >>= D_A);
    v.add(D_A >> thetaA >>= DA + A);
    v.add((A + DR) >> gammaR >>= D_R);
    v.add(D_R >> thetaR >>= DR + A);
    v.add(D_A >> alpha_A >>= MA + D_A);
    v.add(DA >> alphaA >>= MA + DA);
    v.add(D_R >> alpha_R >>= MR + D_R);
    v.add(DR >> alphaR >>= MR + DR);
    v.add(MA >> betaA >>= MA + A);
    v.add(MR >> betaR >>= MR + R);
    v.add((A + R) >> gammaC >>= C);
    v.add(C >> deltaA >>= R);
    v.add(A >> deltaA >>= env);
    v.add(R >> deltaR >>= env);
    v.add(MA >> deltaMA >>= env);
    v.add(MR >> deltaMR >>= env);
```

```
56
57
58      return v;
59  };
60
61  stochastic::Vessel simulation_covid19(uint32_t population_size) {
62      auto v = stochastic::Vessel{"COVID19 SEIHR_" + std::to_string(population_size)};
63      const auto eps = 0.0009; // initial fraction of infectious
64      const auto I0 = size_t(std::round(eps * population_size)); // initial infectious
65      const auto E0 = size_t(std::round(eps * population_size * 15)); // initial exposed
66      const auto S0 = population_size - I0 - E0; // initial susceptible
67      const auto R0 = 2.4; // initial basic reproductive number
68      const auto alpha = 1.0 / 5.1; // incubation rate (E -> I) ~5.1 days
69      const auto gamma = 1.0 / 3.1; // recovery rate (I -> R) ~3.1 days
70      const auto beta = R0 * gamma; // infection/generation rate (S+I -> E+I)
71      const auto P_H = 0.9e-3; // probability of hospitalization
72      const auto kappa = gamma * P_H * (1.0 - P_H); // hospitalization rate (I -> H)
73      const auto tau = 1.0 / 10.12; // removal rate in hospital (H -> R) ~10.12 days
74      const auto S = v.add("S", S0); // susceptible
75      const auto E = v.add("E", E0); // exposed
76      const auto I = v.add("I", I0); // infectious
77      const auto H = v.add("H", 0); // hospitalized
78      const auto R = v.add("R", 0); // removed/immune (recovered + dead)
79      v.add((S + I) >> beta / population_size >>= E + I); // susceptible becomes exposed by infectious
80      v.add(E >> alpha >>= I); // exposed becomes infectious
81      v.add(I >> gamma >>= R); // infectious becomes removed
82      v.add(I >> kappa >>= H); // infectious becomes hospitalized
83      v.add(H >> tau >>= R); // hospitalized becomes removed
84
85      return v;
86
87  }
88
89  stochastic::Vessel simulation_example1() {
90      auto v = stochastic::Vessel{"Example 1"};
91      const auto env = v.environment();
92      const auto A = v.add("A", 100);
93      const auto B = v.add("B", 0);
94      const auto C = v.add("C", 1);
95      const auto lambda = 0.001;
96      v.add((A + C) >> lambda >>= B + C);
97      return v;
98  }
99
100 stochastic::Vessel simulation_example2() {
101     auto v = stochastic::Vessel{"Example 2"};
102     const auto env = v.environment();
103     const auto A = v.add("A", 100);
104     const auto B = v.add("B", 0);
105     const auto C = v.add("C", 2);
106     const auto lambda = 0.001;
107     v.add((A + C) >> lambda >>= B + C);
108
109
110     return v;
111 }
112
113 stochastic::Vessel simulation_example3() {
114     auto v = stochastic::Vessel{"Example 3"};
115     const auto env = v.environment();
116     const auto A = v.add("A", 50);
```

```
    const auto B = v.add("B", 50);
    const auto C = v.add("C", 1);
    const auto lambda = 0.001;
    v.add((A + C) >> lambda >>= B + C);

    return v;
}

void single_simulation_test() {
    auto covid19 = simulation_covid19(10000);
    auto circadian_rhythm = simulation_circadian_rhythm();
    auto example1 = simulation_example1();
    auto example2 = simulation_example2();
    auto example3 = simulation_example3();
    std::string path = stochastic::Simulation::assign_unique_filename(covid19.name);
    stochastic::Simulation::simulate(path, covid19, [](stochastic::Vessel &v, double
↪current_time) {}, 100);
}

void parallel_simulation_test() {
    auto covid19 = simulation_covid19(10000);
    auto circadian_rhythm = simulation_circadian_rhythm();
    auto example1 = simulation_example1();
    auto example2 = simulation_example2();
    auto example3 = simulation_example3();
    auto observer = [](stochastic::Vessel &v, double current_time) {};
    stochastic::ParallelSimulation::parallelize_simulations(
            {example1, example2, example3}, observer, 2000);
}

void peak_hospitalization_NNJ_NDK() {

    auto observer = [](stochastic::Vessel &vessel, double current_time) {
        static int max_hospitalized = 0;
        int current_hospitalized = vessel.molecules.get("H")->quantity;
        if (current_hospitalized > max_hospitalized) {
            max_hospitalized = current_hospitalized;
        }
        if (current_time > 100) {
            std::cout << vessel.name << "\nPeak hospitalized: " << max_hospitalized << std::endl;
        }
    };
    auto NDK = 5822763;
    auto NNJ = 589755;
    std::vector<int> population_sizes = {NNJ};
    for (int population_size: population_sizes) {
        auto vessel = simulation_covid19(population_size);
        stochastic::Simulation::simulate("", vessel, observer, 100, false);
    }
}

void average_peak_hospitalized_over_100_simulations() {
    int sum_max_hospitalized = 0;
    int number_of_simulations = 100;

    // Observer to keep track of the maximum number of hospitalized
    auto observer = [&sum_max_hospitalized](stochastic::Vessel &vessel, double current_time) {
        static int max_hospitalized = 0;
        int current_hospitalized = vessel.molecules.get("H")->quantity;
        if (current_hospitalized > max_hospitalized) {
            max_hospitalized = current_hospitalized;
```

```
        }
        if (current_time > 100) {
            sum_max_hospitalized += max_hospitalized;
            max_hospitalized = 0; // reset for next simulation
        }
    };

    std::vector<stochastic::Vessel> vessels;
    for (int i = 0; i < number_of_simulations; ++i) {
        vessels.push_back(simulation_covid19(10000));
    }
    stochastic::ParallelSimulation::parallelize_simulations(vessels, observer, 100, false);
    double average_peak_hospitalized = static_cast<double>(sum_max_hospitalized) /
↪number_of_simulations;
    std::cout << "Average peak hospitalized over " << number_of_simulations << " simulations: "
↪<< average_peak_hospitalized
              << std::endl;
}

void pretty_print_circadian_rhythm() {
    const auto alphaA = 50;
    const auto alpha_A = 500;
    const auto alphaR = 0.01;
    const auto alpha_R = 50;
    const auto betaA = 50;
    const auto betaR = 5;
    const auto gammaA = 1;
    const auto gammaR = 1;
    const auto gammaC = 2;
    const auto deltaA = 1;
    const auto deltaR = 0.2;
    const auto deltaMA = 10;
    const auto deltaMR = 0.5;
    const auto thetaA = 50;
    const auto thetaR = 100;


    auto v = stochastic::Vessel{"Circadian_Rhythm"};
    const auto env = v.environment();
    const auto DA = v.add("DA", 1);
    const auto D_A = v.add("D_A", 0);
    const auto DR = v.add("DR", 1);
    const auto D_R = v.add("D_R", 0);
    const auto MA = v.add("MA", 0);
    const auto MR = v.add("MR", 0);
    const auto A = v.add("A", 1);
    const auto R = v.add("R", 0);
    const auto C = v.add("C", 0);
    v.add((A + DA) >> gammaA >>= D_A);
    v.add(D_A >> thetaA >>= DA + A);
    v.add((A + DR) >> gammaR >>= D_R);
    v.add(D_R >> thetaR >>= DR + A);
    v.add(D_A >> alpha_A >>= MA + D_A);
    v.add(DA >> alphaA >>= MA + DA);
    v.add(D_R >> alpha_R >>= MR + D_R);
    v.add(DR >> alphaR >>= MR + DR);
    v.add(MA >> betaA >>= MA + A);
    v.add(MR >> betaR >>= MR + R);
    v.add((A + R) >> gammaC >>= C);
    v.add(C >> deltaA >>= R);
    v.add(A >> deltaA >>= env);
```

```
    v.add(R >> deltaR >>= env);
    v.add(MA >> deltaMA >>= env);
    v.add(MR >> deltaMR >>= env);

    v.pretty_print();
    v.get_digraph();
}

void symbol_table_test() {
    // Create a SymbolTable instance
    stochastic::SymbolTable<std::string, double> symbolTable;

    // Insert some key-value pairs
    symbolTable.insert("DA", 1);
    symbolTable.insert("D_A", 0);
    symbolTable.insert("DR", 1);
    symbolTable.insert("D_R", 0);
    symbolTable.insert("MA", 0);
    symbolTable.insert("MR", 0);
    symbolTable.insert("A", 1);
    symbolTable.insert("R", 0);
    symbolTable.insert("C", 0);

    // Check if a key exists
    if (symbolTable.contains("DA")) {
        std::cout << "DA exists in the symbol table." << std::endl;
    } else {
        std::cout << "DA does not exist in the symbol table." << std::endl;
    }

    // Get the value associated with a key
    double value = symbolTable.get("DR");
    std::cout << "The value of DR is: " << value << std::endl;

    // Try to insert a key that already exists
    symbolTable.insert("DA", 1);

    // Update the value associated with a key
    symbolTable.update("DA", 2);

    // Update the value associated with a key that does not exist
    symbolTable.update("test", 2);

}

void benchmark() {
    // Make 100 vessels for single simulation
    std::vector<stochastic::Vessel> single_simulation_vessels;
    for (int i = 0; i < 100; ++i) {
        single_simulation_vessels.push_back(simulation_covid19(10000));
    }
    // Make 100 vessels for parallel simulation
    std::vector<stochastic::Vessel> parallel_simulation_vessels;
    for (int i = 0; i < 100; ++i) {
        parallel_simulation_vessels.push_back(simulation_covid19(10000));
    }

    stochastic::Benchmark benchmark;

    benchmark.start("COVID-19 simulation");
    for (int i = 0; i < 100; ++i) {
```

```
297
298         stochastic::Simulation::simulate("", single_simulation_vessels[i], [](stochastic::Vessel
    ↪&v, double current_time) {}, 100,
299                                             false);
300     }
301     benchmark.stop("COVID-19 simulation");
302
303     benchmark.start("Parallel COVID-19 simulation");
304     auto observer = [](stochastic::Vessel &v, double current_time) {
305     };
306     stochastic::ParallelSimulation::parallelize_simulations(parallel_simulation_vessels,
    ↪observer, 100, false);
307     benchmark.stop("Parallel COVID-19 simulation");
308
309     benchmark.report();
310 }
311
312 int main() {
313
314 //    symbol_table_test();
315 //    pretty_print_circadian_rhythm();
316 //    single_simulation_test();
317 //    parallel_simulation_test();
318 //    peak_hospitalization_NNJ_NDK();
319 //    average_peak_hospitalized_over_100_simulations();
320 //    benchmark();
321     return 0;
322 }
```

Listing 3: ./Molecule.h

```
1  #pragma once
2
3  #include <string>
4  #include "Reaction.h"
5
6  namespace stochastic {
7
8      class Reaction;
9
10     class Molecule {
11     public:
12         std::string name;
13         double quantity;
14         std::string graphviz_tag;
15
16         Molecule(const std::string &name, double quantity);
17
18         ~Molecule() = default;
19
20         void operator-=(double quantity);
21
22         void operator+=(double quantity);
23
24         Reaction operator+(const Molecule &molecule) const;
25
26         Reaction operator+(const Reaction &reaction);
27
28         Reaction operator>>(double reaction_rate) const;
29     };
30
31 } // stochastic
```

Listing 4: ./Molecule.cpp

```cpp
#include "Molecule.h"

namespace stochastic {
    Molecule::Molecule(const std::string &name, double quantity) {
        this->name = name;
        this->quantity = quantity;
    }

    /**
     * ------------------  Exercise 1 ------------------
     * @brief Overloads the '-=' operator to decrease the quantity of the molecule.
     *
     * @param q The amount by which the quantity of the molecule should be decreased.
     */
    void Molecule::operator-=(double q) {
        Molecule::quantity -= q;
    };

    /**
     * ------------------  Exercise 1 ------------------
     * @brief Overloads the '+=' operator to increase the quantity of the molecule.
     *
     * @param q The amount by which the quantity of the molecule should be increased.
     */
    void Molecule::operator+=(double q) {
        Molecule::quantity += q;
    }

     /**
     * ------------------  Exercise 1 ------------------
     * @brief Overloads the '+' operator to create a reaction with two molecules as reactants.
     *
     * This function creates a new reaction and adds the current molecule.
     *
     * @param molecule The molecule that reacts with the current molecule.
     * @return The created reaction with the two molecules as reactants.
     */
    Reaction Molecule::operator+(const Molecule &molecule) const {
        auto r = Reaction();
        // dereference the current molecule and add it to the reactants of the reaction
        r.reactants.push_back(*this);
        r.reactants.push_back((molecule));
        return r;
    }

    /**
     * ------------------  Exercise 1 ------------------
     * @brief Overloads the '+' operator to create a reaction with the current molecule and the reactants of another reaction.
     *
     * This function creates a new reaction and adds the current molecule and the reactants of the reaction.
     *
     * @param reaction The reaction whose reactants will react with the current molecule.
     * @return The created reaction with the current molecule and the reactants of the passed reaction as reactants.
     */
    Reaction Molecule::operator+(const Reaction &reaction) {
```

```
        auto r = Reaction();
        r.reactants.push_back(*this);
        for (auto reactant: reaction.reactants) {
            r.reactants.push_back(reactant);
        }

        return r;
    }

    /**
     * ------------------  Exercise 1 -----------------
     * @brief Overloads the '>>' operator to create a reaction with the current molecule as a reactant and a specified rate.
     *
     * This function creates a new reaction and adds the current molecule as a reactant.
     *
     * @param rate The rate at which the current molecule reacts.
     * @return The created reaction with the current molecule as a reactant and the specified rate.
     */
    Reaction Molecule::operator>>(double rate) const {
        auto reaction = Reaction();
        reaction.reactants.push_back(*this);
        reaction.rate = rate;
        return reaction;
    }
} // stochastic
```

Listing 5: ./Reaction.h

```
#pragma once

#include <string>
#include <vector>
#include "Molecule.h"
#include "Environment.h"

namespace stochastic {

    // Required to avoid circular dependencies
    class Molecule;

    // Use enum to check if we should add the reactants to the left or right side of the reaction
    typedef enum {
        left,
        right
    } reaction_side;

    class Reaction {
    public:
        std::string name;
        double rate;
        reaction_side side = left;
        std::vector<Molecule> products;
        std::vector<Molecule> reactants;
        double delay;

        Reaction(const std::string &name, double rate, const std::vector<Molecule> &reactants,
                 const std::vector<Molecule> &products);

        Reaction();

        ~Reaction() = default;
```

```
34
35          Reaction operator>>(double reaction_rate);
36
37          Reaction operator+(const Molecule &molecule);
38
39          Reaction operator>>=(const Molecule &products);
40
41          Reaction operator>>=(const Environment &env);
42
43          Reaction operator>>=(const Reaction &reaction);
44
45      };
46
47
48  } // stochastic
```

Listing 6: ./Reaction.cpp

```
1   #include "Reaction.h"
2
3   namespace stochastic {
4       Reaction::Reaction(const std::string &name, double rate, const std::vector<Molecule> &r,
5                          const std::vector<Molecule> &p) {
6           this->name = name;
7           this->rate = rate;
8           this->reactants = r;
9           this->products = p;
10      }
11
12      Reaction::Reaction() = default;
13
14
15      /**
16       * ------------------  Exercise 1 ------------------
17       * @brief Overloads the '>>' operator to set the rate of the reaction.
18       *
19       * The rate of a reaction is a measure of how quickly the reactants turn into products.
20       *
21       * @param r The new rate of the reaction.
22       * @return The modified reaction with the updated rate.
23       */
24      Reaction Reaction::operator>>(double r) {
25          Reaction::rate = r;
26          return *this;
27      }
28
29
30      /**
31       * ------------------  Exercise 1 ------------------
32       * @brief Overloads the '+' operator to add a molecule to the reaction.
33       *
34       * This function adds a molecule to either the reactants or the products of the reaction,
35       * depending on the current side. If the current side is 'left', the molecule is added to
    ↪the reactants.
36       * If the current side is 'right', the molecule is added to the products.
37       *
38       * @param m The molecule to be added to the reaction.
39       * @return The modified reaction with the added molecule.
40       */
41      Reaction Reaction::operator+(const Molecule &m) {
42          if (Reaction::side == left) {
43              Reaction::reactants.push_back(m);
```

```
44          } else {
45              Reaction::products.push_back(m);
46          }
47          return *this;
48      }
49
50      /**
51       * ------------------  Exercise 1 ------------------
52       * @brief Overloads the '>>=' operator to add a product molecule to the reaction.
53       *
54       * This function sets the current side to 'right' and adds a molecule to the products of the reaction.
55       *
56       * @param p The product molecule to be added to the reaction.
57       * @return The modified reaction with the added product molecule.
58       */
59      Reaction Reaction::operator>>=(const Molecule &p) {
60          Reaction::side = right;
61          Reaction::products.push_back(p);
62          return *this;
63      }
64
65      /**
66       * ------------------  Exercise 1 ------------------
67       * @brief Overloads the '>>=' operator to clear the products of the reaction.
68       *
69       * This function clears all the product molecules from the reaction when an environment is passed.
70       *
71       * @param env The environment, which when passed, triggers the clearing of the products.
72       * @return The modified reaction with the cleared products.
73       */
74      Reaction Reaction::operator>>=(const Environment &env) {
75          this->products.clear();
76          return *this;
77      }
78
79      /**
80       * ------------------  Exercise 1 ------------------
81       * @brief Overloads the '>>=' operator to replace the products of the current reaction with the reactants of another reaction.
82       *
83       * This function replaces the products of the current reaction with the reactants of the passed reaction.
84       *
85       * @param reaction The reaction whose reactants will replace the products of the current reaction.
86       * @return The modified reaction with the replaced products.
87       */
88      Reaction Reaction::operator>>=(const Reaction &reaction) {
89          Reaction::products = reaction.reactants;
90          return *this;
91      }
92  } // stochastic
```

Listing 7: ./Environment.h

```
1   #pragma once
2
3   namespace stochastic {
4
5       class Environment {
```

```
    private:

    public:
        Environment();

        ~Environment() = default;
    };

} // stochastic
```

Listing 8: ./Environment.cpp

```
#include "Environment.h"

namespace stochastic {
    /**
     * @brief Environment should be empty because the reactions decay into the environment
     */
    Environment::Environment() = default;

} // stochastic
```

Listing 9: ./Vessel.h

```
#pragma once

#include "Reaction.h"
#include "SymbolTable.h"
#include "Arrow.h"

namespace stochastic {

    class Vessel {
    public:
        stochastic::SymbolTable<std::string, Molecule *> molecules;
        std::vector<Reaction> reactions;
        std::string name;

        Vessel(const std::string &name);

        ~Vessel() = default;


        Molecule &add(const std::string &name, double quantity);

        void add(const Reaction &reaction);

        Environment environment();

        void pretty_print();

        void get_digraph();

        void assign_tags(SymbolTable <std::string, std::string> &table, int &counter);

        std::vector<Arrow> create_arrows(const SymbolTable <std::string, std::string> &table);

        void write_to_file(const std::vector<Arrow> &arrows, const SymbolTable <std::string,
↪std::string> &table);
    };
} // stochastic
```

Listing 10: ./Vessel.cpp

```cpp
#include <iostream>
#include <set>
#include <fstream>
#include "Vessel.h"
#include "Arrow.h"

namespace stochastic {
    Vessel::Vessel(const std::string &name) {
        this->name = name;
    }

    Environment Vessel::environment() {
        return Environment();
    }

    /**
     * -----------------  Exercise 3 -----------------
     * @brief Adds a molecule to the vessel.
     *
     * This function creates a new molecule with the given name and quantity, adds it to the vessel's collection of molecules, and returns a reference to the molecule.
     *
     * @param name The name of the molecule to be added.
     * @param quantity The quantity of the molecule to be added.
     * @return A reference to the newly created molecule.
     */

    Molecule &Vessel::add(const std::string &name, double quantity) {
        auto m = new Molecule(name, quantity);
        molecules.insert(name, m);
        return *m;
    }

    /**
     * -----------------  Exercise 3 -----------------
     * @brief Adds a reaction to the vessel.
     * @param reaction The reaction to be added.
     */
    void Vessel::add(const Reaction &reaction) {
        reactions.push_back(reaction);
    }

    /**
     * -----------------  Exercise 2 -----------------
     * @brief Prints a formatted representation of the reactions in the vessel.
     *
     * For each reaction, it prints the names of the reactants, the reaction rate, and the names of the products.
     * If a reactant or product is not the last one in its list, a plus sign is printed after its name.
     * If there are no products for a reaction, it prints 'none'.
     */
    void Vessel::pretty_print() {

        // Iterate over each reaction
        for (const auto &reaction: reactions) {
            // Iterate over each reactant in the reaction
            for (auto &reactant: reaction.reactants) {

                std::cout << reactant.name;
```

```
58
59                  if (&reactant != &reaction.reactants.back()) {
60                      std::cout << " + ";
61                  }
62              }
63
64              // Print the reaction rate
65              std::cout << " -> " << reaction.rate << " -> ";
66
67              // Check if there are any products in the reaction
68              if (reaction.products.empty()) {
69                  std::cout << "none";
70              } else {
71                  // If there are products, print each one
72                  for (size_t i = 0; i < reaction.products.size(); ++i) {
73                      std::cout << reaction.products[i].name;
74                      if (i != reaction.products.size() - 1) {
75                          std::cout << " + ";
76                      }
77                  }
78              }
79              std::cout << std::endl;
80          }
81      }
82
83
84      /**
85       * ------------------  Exercise 2 ------------------
86       * @brief Assigns unique tags to each molecule involved in the reactions.
87       *
88       * For each reactant and product in the reaction, it checks if the molecule's name is already in the symbol table.
89       * If it's not, it inserts the molecule's name into the symbol table with a unique tag ("s" followed by a counter).
90       * The counter is incremented after each insertion.
91       *
92       * @param table A reference to the symbol table where the molecule names and their corresponding tags are stored.
93       * @param counter A reference to the counter used to generate unique tags for the molecules.
94       */
95      void Vessel::assign_tags(SymbolTable<std::string, std::string> &table, int &counter) {
96          for (auto &r: reactions) {
97              for (auto &m: r.reactants) {
98                  if (!table.contains(m.name)) {
99                      table.insert(m.name, ("s" + std::to_string(counter++)));
100                 }
101             }
102             for (auto &m: r.products) {
103                 if (!table.contains(m.name)) {
104                     table.insert(m.name, ("s" + std::to_string(counter++)));
105                 }
106             }
107         }
108     }
109
110     /**
111      * ------------------  Exercise 2 ------------------
112      * @brief Creates a vector of Arrows representing the reactions in the vessel.
113      *
114      * For each reaction, it creates an Arrow object.
115      * For each reactant and product in the reaction, it adds their corresponding tags from the
```

```
↪symbol table to the source and target of the Arrow, respectively.
116       * It also sets the rate of the Arrow to the rate of the reaction.
117       * The created Arrow is then added to a vector of Arrows, which is returned at the end.
118       *
119       * @param table A reference to the symbol table where the molecule names and their   ↙
↪corresponding tags are stored.
120       * @return A vector of Arrows representing the reactions in the vessel.
121       */
122      std::vector<Arrow> Vessel::create_arrows(const SymbolTable<std::string, std::string> &table) {
123          std::vector<Arrow> arrows;
124          for (const auto &r: reactions) {
125              Arrow arrow = {};
126              for (const auto &reactant: r.reactants) {
127                  arrow.source.push_back(table.get(reactant.name));
128              }
129              for (const auto &product: r.products) {
130                  arrow.target.push_back(table.get(product.name));
131              }
132              arrow.rate = r.rate;
133              arrows.push_back(arrow);
134          }
135          return arrows;
136      }
137
138      /**
139       * -----------------  Exercise 2 -----------------
140       * @brief Writes the reactions represented by Arrows to a .dot file.
141       *
142       * The function opens a file with the name of the vessel and writes the .dot representation   ↙
↪of the reactions to it.
143       * For each molecule in the vessel, it writes a line in the .dot file with the molecule's   ↙
↪tag, name, and color.
144       * For each Arrow in the vector, it writes a line in the .dot file with the Arrow's tag,   ↙
↪rate, source, and target.
145       * The function then closes the file and generates a .png file from the .dot file using the   ↙
↪'dot' command.
146       *
147       * @param arrows A vector of Arrows representing the reactions in the vessel.
148       * @param table A reference to the symbol table where the molecule names and their   ↙
↪corresponding tags are stored.
149       */
150      void Vessel::write_to_file(const std::vector<Arrow> &arrows, const SymbolTable<std::string,   ↙
↪std::string> &table) {
151          std::ofstream file;
152          std::string path = "/home/krogh/CLionProjects/cpp/Exam/" + name + ".dot";
153          file.open(path);
154          if (!file) {
155              std::cerr << "Unable to open file for writing.\n";
156              return;
157          }
158
159          file << "digraph {\n";
160          int counter = 0;
161          // Create a set to store unique tags
162          std::set<std::string> u_sets;
163          for (const auto &r: reactions) {
164              for (const auto &reactant: r.reactants) {
165                  auto pair = u_sets.insert(table.get(reactant.name));
166                  if (pair.second) {
167                      file << table.get(reactant.name) << "[label=\"" << reactant.name
168                           << R"(",shape="box",style="filled",fillcolor="cyan"];)" << '\n';
```

```
169                 }
170             }
171
172             for (const auto &product: r.products) {
173                 auto pair = u_sets.insert(table.get(product.name));
174                 if (pair.second) {
175                     file << table.get(product.name) << "[label=\"" << product.name
176                          << R",shape="box",style="filled",fillcolor="cyan"];)" << '\n';
177                 }
178             }
179
180         }
181
182         for (const auto &arrow: arrows) {
183             file << "r" << counter << "[label=\"" << arrow.rate
184                  << R",shape="oval",style="filled",fillcolor="yellow"];)"
185                  << '\n';
186             for (const auto &src: arrow.source) {
187                 file << src << " -> " << "r" << counter << ";\n";
188             }
189
190             for (const auto &target: arrow.target) {
191                 file << "r" << counter << " -> " << target << ";\n";
192             }
193             counter++;
194         }
195
196         file << "}\n";
197         file.close();
198
199         std::string command =
200                 "dot -Tpng " + path + " -o /home/krogh/CLionProjects/cpp/Exam/images/" +
201                 name + "_tree" + ".png";
202         system(command.c_str());
203     }
204
205     /**
206      * ------------------  Exercise 2 ------------------
207      * @brief Generates a directed graph representation of the reactions in the vessel.
208      *
209      * This function first creates a symbol table and a counter for assigning unique tags to the molecules.
210      * It then calls the assign_tags function to assign unique tags to each molecule involved in the reactions.
211      * After that, it calls the create_arrows function to create a vector of Arrows representing the reactions in the vessel.
212      * Finally, it calls the write_to_file function to write the reactions represented by the Arrows to a .dot file.
213      */
214     void Vessel::get_digraph() {
215
216         SymbolTable<std::string, std::string> table;
217         int counter = 0;
218         assign_tags(table, counter);
219         std::vector<Arrow> arrows = create_arrows(table);
220         write_to_file(arrows, table);
221     }
222
223
224 } // stochastic
```

Listing 11: ./Tests/VesselTests.cpp

```cpp
#include <gtest/gtest.h>
#include <filesystem>
#include "../Vessel.h"

using namespace stochastic;
/**
 * Entire file is exercise 9
 */

TEST(VesselTest, AddMolecule) {
    Vessel vessel("TestVessel");
    Molecule &molecule = vessel.add("H2O", 10.0);
    ASSERT_EQ(molecule.name, "H2O");
    ASSERT_EQ(molecule.quantity, 10.0);
}

TEST(VesselTest, AddReaction) {
    Vessel vessel("TestVessel");
    Reaction reaction;
    reaction.reactants.push_back(Molecule("H2O", 10.0));
    reaction.products.push_back(Molecule("H2", 5.0));
    reaction.products.push_back(Molecule("O2", 5.0));
    vessel.add(reaction);
    ASSERT_EQ(vessel.reactions.size(), 1);
}

TEST(VesselTest, PrettyPrintEmptyVessel) {
    stochastic::Vessel vessel("TestVessel");
    testing::internal::CaptureStdout();
    vessel.pretty_print();
    std::string output = testing::internal::GetCapturedStdout();
    EXPECT_EQ(output, "");
}

TEST(VesselTest, PrettyPrintVesselWithSingleReactionNoProducts) {
    stochastic::Vessel vessel("TestVessel");
    stochastic::Reaction reaction;
    reaction.reactants.push_back(stochastic::Molecule("H2O", 2));
    reaction.rate = 1.0;
    vessel.add(reaction);
    testing::internal::CaptureStdout();
    vessel.pretty_print();
    std::string output = testing::internal::GetCapturedStdout();
    EXPECT_EQ(output, "H2O -> 1 -> none\n");
}

TEST(VesselTest, PrettyPrintVesselWithSingleReactionWithProducts) {
    stochastic::Vessel vessel("TestVessel");
    stochastic::Reaction reaction;
    reaction.reactants.push_back(stochastic::Molecule("H2O", 2));
    reaction.products.push_back(stochastic::Molecule("H2", 1));
    reaction.products.push_back(stochastic::Molecule("O2", 1));
    reaction.rate = 1.0;
    vessel.add(reaction);
    testing::internal::CaptureStdout();
    vessel.pretty_print();
    std::string output = testing::internal::GetCapturedStdout();
    EXPECT_EQ(output, "H2O -> 1 -> H2 + O2\n");
}

```

```
61 TEST(VesselTest, PrettyPrintVesselWithMultipleReactions) {
62     stochastic::Vessel vessel("TestVessel");
63     stochastic::Reaction reaction1;
64     reaction1.reactants.push_back(stochastic::Molecule("H2O", 2));
65     reaction1.products.push_back(stochastic::Molecule("H2", 1));
66     reaction1.products.push_back(stochastic::Molecule("O2", 1));
67     reaction1.rate = 1.0;
68     vessel.add(reaction1);
69     stochastic::Reaction reaction2;
70     reaction2.reactants.push_back(stochastic::Molecule("H2", 1));
71     reaction2.reactants.push_back(stochastic::Molecule("O2", 1));
72     reaction2.products.push_back(stochastic::Molecule("H2O", 2));
73     reaction2.rate = 2.0;
74     vessel.add(reaction2);
75     testing::internal::CaptureStdout();
76     vessel.pretty_print();
77     std::string output = testing::internal::GetCapturedStdout();
78     EXPECT_EQ(output, "H2O -> 1 -> H2 + O2\nH2 + O2 -> 2 -> H2O\n");
79 }
80
81 TEST(VesselTest, AssignTagsAssignsUniqueTags) {
82     Vessel vessel("TestVessel");
83     Reaction reaction;
84     reaction.reactants.push_back(Molecule("H2O", 10.0));
85     reaction.products.push_back(Molecule("H2", 5.0));
86     reaction.products.push_back(Molecule("O2", 5.0));
87     vessel.add(reaction);
88     SymbolTable<std::string, std::string> table;
89     int counter = 0;
90     vessel.assign_tags(table, counter);
91     ASSERT_EQ(table.get("H2O"), "s0");
92     ASSERT_EQ(table.get("H2"), "s1");
93     ASSERT_EQ(table.get("O2"), "s2");
94 }
95
96 TEST(VesselTest, CreateArrowsCreatesCorrectNumberOfArrows) {
97     Vessel vessel("TestVessel");
98     Reaction reaction;
99     reaction.reactants.push_back(Molecule("H2O", 10.0));
100    reaction.products.push_back(Molecule("H2", 5.0));
101    reaction.products.push_back(Molecule("O2", 5.0));
102    vessel.add(reaction);
103    SymbolTable<std::string, std::string> table;
104    int counter = 0;
105    vessel.assign_tags(table, counter);
106    std::vector<Arrow> arrows = vessel.create_arrows(table);
107    ASSERT_EQ(arrows.size(), 1);
108 }
109
110 TEST(VesselTest, WriteToFileCreatesFile) {
111     Vessel vessel("TestVessel");
112     Reaction reaction;
113     reaction.reactants.push_back(Molecule("H2O", 10.0));
114     reaction.products.push_back(Molecule("H2", 5.0));
115     reaction.products.push_back(Molecule("O2", 5.0));
116     vessel.add(reaction);
117     SymbolTable<std::string, std::string> table;
118     int counter = 0;
119     vessel.assign_tags(table, counter);
120     std::vector<Arrow> arrows = vessel.create_arrows(table);
121     vessel.write_to_file(arrows, table);
```

```
122     ASSERT_TRUE(std::filesystem::exists("/home/krogh/CLionProjects/cpp/Exam/TestVessel.dot"));
123 }
```

Listing 12: ./Arrow.h

```
1  #pragma once
2
3  #include <vector>
4  #include <string>
5
6  namespace stochastic {
7
8      class Arrow {
9      public:
10         std::vector<std::string> source;
11         std::vector<std::string> target;
12         double rate;
13
14         Arrow(std::vector<std::string> src, std::vector<std::string> tgt, double r);
15
16         Arrow() = default;
17
18         ~Arrow() = default;
19     };
20
21 } // stochastic
```

Listing 13: ./Arrow.cpp

```
1  #include "Arrow.h"
2
3  namespace stochastic {
4
5      /**
6       * ------------------  Exercise 2 ------------------
7       * @brief Constructs a new Arrow object.
8       *
9       * The Arrow object represents a reaction in the system, with source molecules, target
↪molecules, and a reaction rate.
10      *
11      * @param src A vector of source molecules' Graphviz tags, represented as strings.
12      * @param tgt A vector of target molecules' Graphviz tags, represented as strings.
13      * @param r The rate of the reaction, represented as a double.
14      */
15     Arrow::Arrow(std::vector<std::string> src, std::vector<std::string> tgt, double r) {
16
17         // Iterating over the source vector and adding each source to the Arrow's source vector
18         for (auto &s: src) {
19             this->source.push_back(s);
20         }
21
22         // Iterating over the target vector and adding each target to the Arrow's target vector
23         for (auto &t: tgt) {
24             this->target.push_back(t);
25         }
26
27         // Setting the rate of the Arrow (reaction)
28         this->rate = r;
29     }
30
31 } // stochastic
```

Listing 14: ./SymbolTable.h

```cpp
#pragma once

#include <map>
#include <vector>

namespace stochastic {

    template<typename K, typename V>
    class SymbolTable {
    private:
        std::map<K, V> table;
    public:
        SymbolTable() = default;

        ~SymbolTable() = default;


       /**
        * ------------------  Exercise 3 ------------------
        * @brief Inserts a key-value pair into the symbol table.
        *
        * This function attempts to insert a given key-value pair into the symbol table.
        * If the key already exists in the table, it prints a message indicating that the symbol
↪already exists.
        *
        * @param key The key to insert into the symbol table.
        * @param value The value to associate with the given key.
        */
        void insert(K key, V value) {
            // If it is pointing to the end of the table, the key does not exist in the table
            if (table.find(key) != table.end()) {
                std::cout << "Symbol already exists in the table" << std::endl;
            } else {
                table[key] = value;
            }
        }

        /**
         * ------------------  Exercise 3 ------------------
         * @brief Retrieves the value associated with the given key from the symbol table.
         *
         * This function attempts to find the given key in the symbol table.
         * If the key is found, it returns the associated value.
         * If the key is not found, it prints an error message .
         *
         * @param key The key to search for in the symbol table.
         * @return The value associated with the given key.
         */
        V get(const K &key) const {
            if (table.find(key) == table.end()) {
                std::cout << "Symbol not found in the table" << std::endl;
            }
            return table.find(key)->second;
        }

        /**
         * ------------------  Exercise 3 ------------------
         * @brief Checks if the symbol table contains the given key.
         *
         * @param key The key to check for in the symbol table.
```

```
         * @return True if the key exists in the table, false otherwise.
         */
        bool contains(K key) {
            return table.find(key) != table.end();
        }


       /**
        * ------------------  Exercise 3 ------------------
        * @brief Retrieves all the values in the symbol table.
        *
        * This function iterates over all the key-value pairs in the symbol table,
        * and collects all the values into a vector. The order of the values in the
        * vector is the same as the order of the key-value pairs in the symbol table.
        *
        * @return A vector containing all the values in the symbol table.
        */
        std::vector<V> values() {
            std::vector<V> values;
            for (const auto &pair: table) {
                values.push_back(pair.second);
            }
            return values;
        }


        /**
         * ------------------  Exercise 3 ------------------
         * @brief Updates the value associated with a given key in the symbol table.
         *
         * If the key already exists in the table, it updates the value.
         * If the key does not exist, it prints a message indicating that the key was not found.
         *
         * @param key The key for which to update the value.
         * @param value The new value to associate with the given key.
         */
        void update(K key, V value) {
            if (table.find(key) != table.end()) {
                table[key] = value;
            } else {
                std::cout << ("Key not found in SymbolTable") << std::endl;
            }
        }


    };

} // stochastic
```

Listing 15: ./Tests/SymbolTableTests.cpp

```
#include <gtest/gtest.h>
#include "../SymbolTable.h"

/**
 * Entire file is exercise 9
 */

TEST(SymbolTableTest, InsertAndRetrieveValue) {
    stochastic::SymbolTable<int, std::string> table;
    table.insert(1, "one");
    EXPECT_EQ(table.get(1), "one");
```

```
}

TEST(SymbolTableTest, InsertDuplicateKey) {
    stochastic::SymbolTable<int, std::string> table;
    table.insert(1, "one");
    table.insert(1, "duplicate");
    EXPECT_EQ(table.get(1), "one");
}

TEST(SymbolTableTest, RetrieveNonExistentKey) {
    stochastic::SymbolTable<int, std::string> table;
    std::string default_value;
    EXPECT_EQ(table.get(1), default_value);

}

TEST(SymbolTableTest, CheckKeyExists) {
    stochastic::SymbolTable<int, std::string> table;
    table.insert(1, "one");
    EXPECT_TRUE(table.contains(1));
    EXPECT_FALSE(table.contains(2));
}

TEST(SymbolTableTest, RetrieveAllValues) {
    stochastic::SymbolTable<int, std::string> table;
    table.insert(1, "one");
    table.insert(2, "two");
    std::vector<std::string> values = table.values();
    EXPECT_EQ(values.size(), 2);
    EXPECT_EQ(values[0], "one");
    EXPECT_EQ(values[1], "two");
}

TEST(SymbolTableTest, UpdateValue) {
    stochastic::SymbolTable<int, std::string> table;
    table.insert(1, "one");
    table.update(1, "updated");
    EXPECT_EQ(table.get(1), "updated");
}

TEST(SymbolTableTest, UpdateNonExistentKey) {
    stochastic::SymbolTable<int, std::string> table;
    table.update(1, "updated");
    std::string defaultValue;
    EXPECT_EQ(table.get(1), defaultValue);
}
```

Listing 16: ./Simulation.h

```
#pragma once

#include <string>
#include <filesystem>
#include <fstream>
#include <random>
#include "Vessel.h"

namespace stochastic {

    class Simulation {
    private:
       /**
```

```
14          * ------------------  Exercise 4 ------------------
15          * @brief Computes the delay for a given reaction in a vessel.
16          *
17          * If any reactant quantity is zero, the function returns infinity, i.e. reaction cannot
    ↪occur.
18          * Otherwise, it calculates the product of the quantities of all reactants, multiplies it
    ↪by the reaction rate,
19          * and uses this to generate a random delay time from an exponential distribution.
20          *
21          * @param vessel The vessel in which the reaction is occurring.
22          * @param reaction The reaction for which the delay is being computed.
23          * @return The computed delay time for the reaction.
24          */
25          static double compute_delay(stochastic::Vessel &vessel, stochastic::Reaction &reaction) {
26              double product = 1;
27              // Iterate over all reactants in the reaction
28              for (const auto &reactant: reaction.reactants) {
29                  // If any reactant quantity is zero, return infinity
30                  if (vessel.molecules.get(reactant.name)->quantity <= 0) {
31                      return std::numeric_limits<double>::infinity();
32                  }
33                  // Multiply the product by the quantity of the current reactant
34                  product *= vessel.molecules.get(reactant.name)->quantity;
35              }
36              // Calculate the rate of the reaction
37              double rate = reaction.rate * product;
38              // Initialize a random number generator
39              std::random_device rd;
40              std::mt19937 gen(rd());
41              // Initialize an exponential distribution with the calculated rate
42              std::exponential_distribution<> d(rate);
43              // Return a random delay time from the exponential distribution
44              return d(gen);
45          }
46
47
48          /**
49           * ------------------  Exercise 4 ------------------
50           * @brief Checks if a reaction can occur in a given vessel.
51           *
52           * @param vessel The vessel in which the reaction is supposed to occur.
53           * @param reaction The reaction that is supposed to occur.
54           * @return A boolean value indicating whether the reaction can occur (true) or not (false).
55           */
56          static bool can_react(Vessel &vessel, Reaction &reaction) {
57              for (const auto &reactant: reaction.reactants) {
58                  if (vessel.molecules.get(reactant.name)->quantity <= 0) {
59                      return false;
60                  }
61              }
62              return true;
63          }
64
65
66          /**
67           * ------------------  Exercise 4 ------------------
68           * @brief Performs a reaction in a given vessel.
69           *
70           * It iterates over the reactants of the reaction, decreasing their quantity in the
    ↪vessel by one.
71           * Then, it iterates over the products of the reaction, increasing their quantity in the
```

```
↪vessel by one.
72           *
73           * @param vessel The vessel in which the reaction is occurring.
74           * @param reaction The reaction that is being performed.
75           */
76          static void perform_reaction(Vessel &vessel, Reaction &reaction) {
77              for (const auto &reactant: reaction.reactants) {
78                  vessel.molecules.get(reactant.name)->quantity -= 1;
79              }
80              for (const auto &product: reaction.products) {
81                  vessel.molecules.get(product.name)->quantity += 1;
82              }
83          }
84
85
86      public:
87
88          /**
89           * ------------------  Exercise 4 + 7 (Observer) -----------------
90           * @brief Simulates the reactions in a vessel over a given period of time.
91           *
92           * The CSV file includes the time and the quantity of each molecule at each time step.
93           * At each time step, it finds the reaction with the smallest delay and performs that ↙
↪reaction.
94           * The function continues until the total time has reached the end time specified by the ↙
↪user.
95           *
96           * @tparam Observer The type of the observer function that is called at each time step.
97           * @param path The path to the CSV file where the results will be written.
98           * @param vessel The vessel in which the reactions are occurring.
99           * @param observer The observer function that is called at each time step.
100          * @param end_time The end time for the simulation.
101          * @param output_to_file A boolean value indicating whether the results should be ↙
↪written to a file.
102          */
103         template<class Observer>
104         static void simulate(const std::string &path, Vessel &vessel, Observer observer, double ↙
↪end_time,
105                              bool output_to_file = false) {
106             std::ofstream file(path);
107             if (output_to_file) {
108                 file << "Time"; // Write the header for the time column
109                 for (const auto &molecule: vessel.molecules.values()) {
110                     file << "," << molecule->name; // Write the headers for the molecule columns
111                 }
112                 file << std::endl;
113             }
114
115             double time = 0;
116             while (time <= end_time) {
117                 Reaction min_delay_reaction;
118                 double min_delay = std::numeric_limits<double>::infinity();
119
120                 // Find the reaction with the smallest delay
121                 for (auto &reaction: vessel.reactions) {
122                     reaction.delay = compute_delay(vessel, reaction);
123                     if (reaction.delay < min_delay) {
124                         // Update the minimum delay and the corresponding reaction
125                         min_delay = reaction.delay;
126                         min_delay_reaction = reaction;
127                     }
```

```
                }

                time += min_delay;
                if (can_react(vessel, min_delay_reaction)) {
                    perform_reaction(vessel, min_delay_reaction);
                }

                observer(vessel, time);
                if (output_to_file) {
                    file << time;
                    for (const auto &molecule: vessel.molecules.values()) {
                        file << "," << molecule->quantity; // Write the quantity of each molecule to the file
                    }
                    file << std::endl;
                }
            }
            if (output_to_file) {
                file.close(); // Close the file
            }
        }


        /**
         * ------------------  Exercise 4 ------------------
         * @brief Assigns a unique filename for a given name.
         *
         * It appends the name to a predefined path and checks if a file with that name already exists.
         * If a file with that name exists, it appends a counter to the name and increments the counter until it finds a filename that does not exist.
         * Once a unique filename is found, it creates an empty file with that name and returns the full path to the file.
         *
         * @param name The base name for the file.
         * @return The full path to the newly created file.
         */
        static std::string assign_unique_filename(const std::string &name) {
            int counter = 1;
            std::string originalPath = "/home/krogh/CLionProjects/cpp/Exam/CsvFiles/" + name + ".csv";
            while (std::filesystem::exists(originalPath)) {
                originalPath =
                        "/home/krogh/CLionProjects/cpp/Exam/CsvFiles/" + name + "_" + std::to_string(counter) + ".csv";
                counter++;
            }
            std::ofstream file;
            file.open(originalPath);
            file.close();
            return originalPath;
        }


    };
} // stochastic
```

Listing 17: ./ParallelSimulation.h

```
#pragma once

namespace stochastic {
```

```
4
5       class ParallelSimulation {
6       public:
7           /**
8            * ------------------  Exercise 8 ------------------
9            * @brief Runs simulations in parallel for multiple vessels.
10           *
11           * This function runs simulations for multiple vessels in parallel. Each simulation is
↪run in a separate thread.
12           * If the ‘output_to_file‘ parameter is ‘true‘, a unique filename is assigned to each
↪vessel and the results of the simulation are written to a file.
13           * The function waits for all simulations to finish before returning.
14           *
15           * @tparam Observer The type of the observer function that is called at each time step
↪in the simulation.
16           * @param vessels A vector of vessels for which simulations will be run.
17           * @param observer The observer function that is called at each time step in the simulation.
18           * @param simulation_time The total time for which the simulation will be run.
19           * @param output_to_file A boolean value indicating whether the results of the
↪simulation should be written to a file.
20           */
21          template<class Observer>
22          static void parallelize_simulations(std::vector<Vessel> vessels, Observer observer,
↪double simulation_time, bool output_to_file = false) {
23              std::vector<std::thread> threads;
24              std::string path;
25              // For each vessel, assign a unique filename and start a simulation in a new thread.
26              for (auto &v: vessels) {
27                  if (output_to_file) {
28                      path = Simulation::Simulation::assign_unique_filename(v.name);
29                  }
30                  threads.push_back(std::thread(Simulation::simulate<Observer>, path, std::ref(v),
↪observer, simulation_time, output_to_file));
31              }
32
33              // Wait for all simulations to finish.
34              for (auto &thread: threads) {
35                  thread.join();
36              }
37          }
38      };
39
40  } // stochastic
```

Listing 18: ./benchmark.cpp

```
1   #include <iostream>
2   #include <iomanip>
3   #include "benchmark.h"
4
5   namespace stochastic {
6       /**
7    * ------------------  Exercise 10 ------------------
8    * @brief Starts a new benchmark.
9    *
10   * This function starts a new benchmark with the given name. It records the current time as the
↪start time of the benchmark.
11   * The benchmark is represented as a ‘time_point‘ object, which is added to the ‘time_points‘
↪vector.
12   *
13   * @param name The name of the benchmark. This name is used to identify the benchmark when
↪stopping it and reporting its duration.
```

```
 */
    void Benchmark::start(const std::string &name) {
        time_point tp;
        tp.name = name;
        tp.start = std::chrono::high_resolution_clock::now();
        time_points.push_back(tp);
    }

    /**
 * -----------------  Exercise 10 -----------------
 * @brief Stops a running benchmark.
 *
 * This function stops a running benchmark with the given name. It records the current time as the stop time of the benchmark.
 * The benchmark is represented as a ‘time_point‘ object, which is updated in the ‘time_points‘ vector.
 *
 * @param name The name of the benchmark. This name is used to identify the benchmark when stopping it and reporting its duration.
 */
    void Benchmark::stop(const std::string &name) {
        for (auto &tp: time_points) {
            if (tp.name == name) {
                tp.stop = std::chrono::high_resolution_clock::now();
                return;
            }
        }
    }

/**
 * -----------------  Exercise 10 -----------------
 * @brief Reports the duration of all benchmarks.
 *
 * This function iterates over all the ‘time_point‘ objects in the ‘time_points‘ vector, calculates the duration of each benchmark,
 * and prints the name of the benchmark and its duration in milliseconds and seconds to the standard output.
 */
    void Benchmark::report() {
        for (const auto &tp: time_points) {
            std::chrono::duration<double, std::milli> duration = tp.stop - tp.start;
            std::cout << tp.name << " took " << duration.count() << " ms ("
                      << std::fixed << std::setprecision(1) << duration.count()/1000 << " seconds)" << std::endl;

        }
    }
}
```

Listing 19: ./benchmark.h

```
#pragma once

#include <vector>
#include <chrono>
#include <string>

namespace stochastic
{

    class Benchmark
    {
```

```
    public:
        void start(const std::string &name);

        void stop(const std::string &name);

        void report();

    private:
        struct time_point
        {
            std::string name;
            std::chrono::high_resolution_clock::time_point start;
            std::chrono::high_resolution_clock::time_point stop;
        };
        std::vector<time_point> time_points;
    };

}
```

Listing 20: ./CsvFiles/graphical_simulation.py

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

"""
------------------  Exercise 6 ------------------
"""


def plot_example(file_path):
    # Load the data
    data = pd.read_csv(file_path)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(data['Time'], data['A'], label='A', color='red')
    plt.plot(data['Time'], data['B'], label='B', color='green')
    plt.plot(data['Time'], data['C'], label='C', color='blue')

    # Add labels and title
    plt.xlabel('Time')
    plt.ylabel('Count')
    plt.title('Graph from CSV data: ' + file_path)
    plt.legend()

    # Ensure the directory exists
    graphs_dir = 'images/'
    if not os.path.exists(graphs_dir):
        os.makedirs(graphs_dir)

    # Save the plot to a PNG file
    plt.savefig('images/' + file_path.split('/')[-1].replace('.csv', '') + '.png', format='png')


def plot_circadian_rhythm(file_path):
    # Load the data
    data = pd.read_csv(file_path)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(data['Time'], data['A'], label='A', color='red')
```

```
    plt.plot(data['Time'], data['C'], label='C', color='green')
    # plt.plot(data['Time'], data['DA'], label='DA', color='gray')
    # plt.plot(data['Time'], data['DR'], label='DR', color='purple')
    # plt.plot(data['Time'], data['D_A'], label='D_A', color='orange')
    # plt.plot(data['Time'], data['D_R'], label='D_R', color='yellow')
    # plt.plot(data['Time'], data['MA'], label='MA', color='pink')
    # plt.plot(data['Time'], data['MR'], label='MR', color='brown')
    plt.plot(data['Time'], data['R'], label='R', color='blue')

    # Add labels and title
    plt.xlabel('Time')
    plt.ylabel('Count')
    plt.title('Circadian Rhythm from CSV data: ' + file_path)
    plt.legend()

    # Ensure the directory exists
    graphs_dir = 'images/'
    if not os.path.exists(graphs_dir):
        os.makedirs(graphs_dir)

    # Save the plot to a PNG file
    plt.savefig('images/' + file_path.split('/')[-1].replace('.csv', '') + '.png', format='png')


def plot_covid19(file_path):
    # Load the data
    data = pd.read_csv(file_path)

    # Plot the data
    plt.figure(figsize=(10, 6))
    plt.plot(data['Time'], data['S'], label='S', color='red')
    plt.plot(data['Time'], data['E'], label='E', color='green')
    plt.plot(data['Time'], data['I'], label='I', color='blue')
    plt.plot(data['Time'], data['H'], label='H', color='turquoise')
    plt.plot(data['Time'], data['R'], label='R', color='purple')

    # Add labels and title
    plt.xlabel('Time')
    plt.ylabel('Values')
    plt.title('COVID-19 Data from CSV: ' + file_path)
    plt.legend()

    # Ensure the directory exists
    graphs_dir = 'images/'
    if not os.path.exists(graphs_dir):
        os.makedirs(graphs_dir)

    # Save the plot to a PNG file
    plt.savefig('images/' + file_path.split('/')[-1].replace('.csv', '') + '.png', format='png')


def main():
    # Call the function for each file
    plot_example('CsvFiles/Example 1.csv')
    plot_example('CsvFiles/Example 2.csv')
    plot_example('CsvFiles/Example 3.csv')
    plot_circadian_rhythm('CsvFiles/Circadian Rhythm.csv')
    plot_covid19('CsvFiles/COVID19 SEIHR_10000.csv')


if __name__ == "__main__":
```

```
103     main()
```