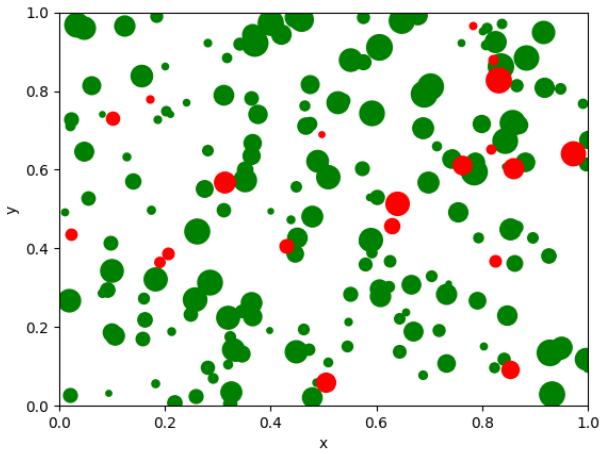
Eksternistyczne zaliczenie ćwiczeń z Programowania 2021/2022

Program musi się kompilować i działać na komputerach OKWF

Proszę o przesłanie rozwiązania w postaci archiwum o nazwie *Imie_Nazwisko.zip* na platformę Kampus2 `https://kampus-student2.ckc.uw.edu.pl/course/view.php?id=10286` do 24 kwietnia.

Symulacja pandemii

Celem projektu jest napisanie uproszczonej symulacji rozwoju pandemii.
W mieście o kształcie kwadratu o rozmiarze L $\times$ L znajduje sie n osób reprezentowanych na rysunkach przez kółka o rożnym kolorze i promieniu. Kolor symbolizuje stan osoby: czerwony - osoba chora, zielony - osoba zdrowa, niebieski - osoba, która była chora ale wyzdrowiała. Promień kulki charakteryzuje to jak blisko trzeba do danej osoby podejść, żeby się zarazić. Przykładowa sytuacja początkowa przedstawiona jest na rysunku 1.
Zakładamy, że osoby mogą poruszać się tylko ruchem jednostajnym, prostoliniowym, a w momencie dotar-



Rysunek 1: Przykładowy wygląd miasta (L=1) w chwili początkowej

cia do granic miasta zawracają i poruszają się po takiej drodze i z taką prędkością jak kulka po odbiciu od ściany. Osoby chore zarażają osoby zdrowe (kolor zmienia się na czerwony), gdy znajdują sie odpowiednio blisko. Spotkanie osób ze sobą nie wpływa na ich tor ruchu. Po upływie czasu t (`recoveryTime`) osoby chore zdrowieją (kolor zmienia się na niebieski).
Przy pomocy symulacji należy stworzyć animację pokazującą ruch osób oraz ich stan zdrowia (kolor) w czasie.

Głównym elementem programu jest pętla po kolejnych chwilach czasu, w której wykonywane są następujące działania:

- wykonanie przemieszczenia każdej z osób zgodnie ze wzorem:

$$\vec{r_i}(t + \Delta t) = \vec{r_i}(t) + \vec{v_i}\Delta t$$

- sprawdzanie czy osoba dotarła do granic miasta (przykładowo warunek na granicy $x = L$: $x + R \geqslant L$, a na granicy $x = 0$: $x - R \leqslant 0$) i wektor prędkości musi być skierowany w stronę granicy.

- po dotarciu do granicy zmiana prędkości osoby zgodnie z formułą:

$$\vec{v}_i' = \vec{v}_i - 2(\vec{v}_i \cdot \hat{n})(\hat{n})$$

(składowa prędkości prostopadła do granicy, zmienia po dotarciu do krańca miasta kierunek na przeciwny) Wektor $\vec{n}$ jest wersorem prostopadłym do linii granicy, skierowanym na zewnątrz.

- sprawdzanie czy osoby znalazły się tak blisko siebie, że osoba zdrowa mogła zostać zakażona (zakażenie osoby i zachodzi wtedy, gdy druga osoba (j) jest chora i gdy odległość między nimi jest mniejsza lub równa od sumy ich promieni):

$$|\vec{r}_i - \vec{r}_j| \leqslant R_i + R_j$$

- Po czasie t (`recoveryTime`) zachodzi zmiana stanu osoby chorej z czerwonego na niebieski, czyli osoba jest zdrowa, ale po przebytej chorobie. Każdy ozdrowieniec ma wysoką odporność i nie może się ponownie zarazić.

Organizacja kodu: W archiwum z rozwiązaniem musi być plik tekstowy o nazwie `README`, z zapisanym pełnym poleceniem do kompilacji, wyjaśnieniami dotyczącymi uruchamiania i działania programu i przykładem/przykładami jak kod można uruchomić na komputerach w OKWF.
Projekt musi być podzielony na klasy, a kod każdej klasy na deklarację w pliku nagłówkowym **Klasa.h**, oraz implementację w pliku **Klasa.cpp**, **Klasa.cc** lub **Klasa.C**, gdzie słowo `Klasa` trzeba zastąpić nazwą danej klasy.

Proszę pamiętać o czytelności kodu źródłowego – między innymi o jego odpowiednim sformatowaniu (wcięcia).

Funkcja `main()` powinna być bardzo krótka (powinien być w niej powoływany do życia obiekt klasy `City` i uruchamiane funkcje na rzecz tego obiektu).

Wymagania (na 50%):
W mieście znajdują się trzy osoby: dwie zdrowe (zielone) i jedna chora (czerwona). Rozmiar miasta (bok kwadratu L) wynosi 0.25, `recoveryTime` $rt = 0.5$, krok czasowy $dt = 0.02$, liczba iteracji $nIter = 100$.
Poniższa tabela przedstawia zestawienie parametrów dla wszystkich osób.

x	y	v_x	v_y	promień	kolor
0.1	0.2	0.1	0.2	0.02	green
0.1	0.1	0.5	0.3	0.05	green
0.2	0.1	0.3	0.6	0.03	red

Powyższe wartości wpisane są na stałe do programu.
Program musi zawierać następujące klasy:

- `Person` - klasa reprezentująca pojedynczą osobę. Klasa powina zawierać zmienne prywatne: położenie `double x, y`, prędkość `double vx, vy`, promień `double radius`, kolor `std::string color` oraz `double timeR`. Zmienna timeR przechowuje czas od momentu zachorowania danej osoby.

  Klasa powinna mieć konstruktor oraz metody typu `set` do ustawiania wartości zmiennych i metody typu `get` do ich odczytywania.

- `City` - klasa opisująca miasto. Powinna zawierać zmienne prywatne: pojemnik `vector<Person> people`, `int nIter` oraz `double dt, boxSize, recoveryTime`:

  - `nIter` – liczba iteracji;
  - `dt` – długość kroku czasowego;
  - `recoveryTime` – czas po którym chory wraca do zdrowia (w jednostkach - krokach czasowych).

  Klasa powinna też mieć co najmniej jeden konstruktor.

  W klasie `City` powinny być prywatne metody:

  - `movePeople` – wykonująca przemieszczenie wszystkich osób w danym kroku czasowym;

– `cityBoundary` – sprawdzająca dla pojedynczej osoby, czy dotarła do granic miasta i w razie zaistnienia takiej sytuacji modyfikujaca prędkość tej osoby; metoda uruchamiana na końcu funkcji `movePeople`.

– `infection` – sprawdzająca dla wszystkich osób po kolei czy zaszło spotkanie pomiędzy dwoma osobami, a jeśli tak i osoba zdrowa spotkała się z chorą, funkcja powinna zmieniać stan osoby zdrowej na chorą;

– `increaseTimeR` – zwiększająca wszystkim chorym czas od początku choroby o wartość kroku czasowego `dt`; w przypadku gdy minął czas `recoveryTime` osoba chora zmienia się w zdrową (kolor czerwony zmienia się w niebieski).

Ponadto w klasie `City` powinny być metody:

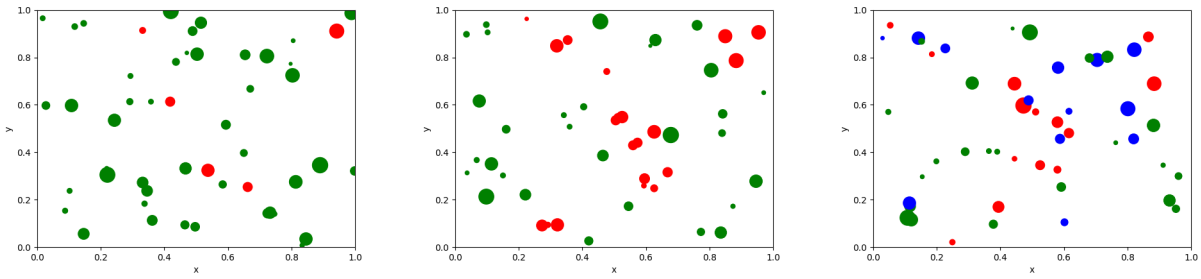
– `evolution` – wykonująca pętlę po iteracjach;

– `round` – wykonująca wszystkie działania w ramach jednej iteracji: `movePeople`, `infection`, `increaseTimeR`.

Klasa powinna mieć co najmniej jeden konstruktor oraz metody typu `set` do ustawiania wartości zmiennych i metody typu `get` do ich odczytywania.

Graficzne przedstawienie rozwoju pandemii:

Efektem działania programu będą rysunki przedstawiające ewolucję pandemii, czyli położenie osób na płaszczyznie XY w poszczególnych chwilach czasu, z zaznaczeniem aktualnego stanu (kolor) każdej z osób. Rysunki muszą być zapisane w formacie `png`, w folderze o nazwie `plots`, w plikach o nazwach **frame_ABCD.png**, np. frame_0001.png. Przykładowe stop-klatki z ewolucji miasta widoczne są na rysunku: 2.



Rysunek 2: Przykładowe stop-klatki z ewolucji miasta. Parametry początkowe jak w `randomConfiguration`. Konfiguracje w chwilach: t=0, środek i koniec przedziału czasowego.

Do wykorzystania jest gotowa klasa `Plotter` (pliki `Plotter.h` i Plotter.cpp) do tworzenia rysunków przy użyciu nakładki `matplotlibcpp.h`[1] i biblioteki `python`, którą trzeba dołączyć na etapie kompilacji programu. Wersja działająca w OKWF:
`g++ *.cpp -I/usr/include/python3.6m -lpython3.6m`

Klasa `Plotter` w obecnej postaci zawiera metody publiczne:

– `plot` – wykonującą rysunek reprezentujący stan miasta w danej chwili czasu.

– `makeAnimation` – łączącą wszystkie rysunki w animację (plik w formacie gif) poprzez wywołanie komendy systemowej `convert` (do użycia na komputerach OKWF). W serwisie `repl.it` (i nie tylko) można alternatywnie użyć dołączonego skryptu `mergeFrames.py`.

```
convert frame*.png animation.gif
```

lub

```
python3 mergeFrames.py
```

[1] https://github.com/lava/matplotlib-cpp

Skrypt `mergeFrames.py` zadziała poprawnie tylko wtedy, gdy rysunki w formacie png będą znajdowały się w folderze plots.
Polecenia systemowe można uruchomić z wnętrza programu za pomocą polecenia `system` z biblioteki `cstdlib`, np.: `system("cd plots");`

Wymagania na 75%:

Wymagania takie jak na 50% i dodatkowo:
w mieście może być dowolna liczba osób. Można je zaludniać osobami o losowych początkowych wartościach położenia i prędkości lub wczytywać początkowe parametry z pliku. Program powinien pytać czy losujemy parametry osób, uruchamiamy wersję testową (ustawienia jak w wymaganiach na 50%) czy wczytujemy dane z pliku. W każdej z klas (`Person` i `City`) powinny być co najmniej dwa konstruktory.

Dodatkowe metody:

- `testConfiguration` – Konfiguracja testowa (parametry jak w wymaganiach na 50%).
- `randomConfiguration` – Konfiguracja losowo generowana: składowe położenia i prędkości 50 osób są losowane z rozkładu płaskiego: $x_0, y_0 \in [0, L]$, $v_{x0}, v_{y0} \in [-0.5, 0.5]$; 90% osób jest zdrowych (kolor zielony), a 10% jest chora (kolor czerwony). Do każdej osoby przypisany jest promień, losowany z rozkładu płaskiego: $r \in [0.01, 0.05]$. Rozmiar miasta (bok kwadratu L) wynosi 1., `recoveryTime` $rt = 0.7$, krok czasowy $dt = 0.02$, liczba iteracji $nIter = 50$.
- `readConfiguration` – wczytująca początkowe parametry (dla dowolnej liczby osób) z pliku o nazwie `input_configuration.txt` i wypełniająca `vector<Person>`. Metoda powinna zwracać status wczytywania np. w postaci zmiennej logicznej typu `bool`. W przypadku problemów z plikiem (stan strumienia `fail() == true`) powinna wypisać informację na ekranie i zaproponować inną metodę zaludnienia miasta (wartości losowe, albo testowe jak opisane wyżej).
  Każda linia pliku wejściowego musi zawierać następujące informacje o jednej osobie: współrzędne położenia i prędkości w chwili początkowej, promień i kolor.
  Plik z parametrami jak dla wersji testowej wygląda tak:

```
0.1 0.2  0.1 0.2  0.02  green
0.1 0.1  0.5 0.3  0.05  green
0.2 0.1  0.3 0.6  0.03  red
```

Wymagania na 100%:

Wymagania takie jak na 75% i dodatkowo:

działanie programu musi być sterowane przez parametry wczytywane z linii poleceń (argumenty main) oraz dane z ostatniej iteracji mogą być zapisywane w pliku.

Dodatkowa metoda:

- `parseParameters(int argc, char* argv [])` – odczytywanie parametrów z linii poleceń. Program musi być zabezpieczony przed uruchomieniem w niepoprawny sposób, np. bez żadnego parametru. W takiej sytuacji powinien przerywać działanie i wypisywać jakie są możliwe opcje. W przypadku nie podania wszystkich opcji, powinny być używane standardowe wartości parametrów i komunikat jakie są to wartości powinien być wypisywany na ekranie.

Parametry linii poleceń (powinna być możliwość wczytywania ich w dowolnej kolejności):

- `--nIter` – liczba kroków czasowych w symulacji;
- `--dt` – długość kroku czasowego;
- `--nPeople` – liczba osób;
- `--recoveryTime` – czas potrzebny na wyzdrowienie;

– `--input` *typ* - wybór sposobu wprowadzania informacji o osobach. Możliwe wartości parametru *typ*:
  1. `random` - składowe położenia i prędkości osób są losowane z rozkładu płaskiego;
  2. `test` - konfiguracja służąca do testowania poprawności działania programu (parametry jak w wymaganiach na 50%);
  3. `file` – wczytywanie początkowej konfiguracji z pliku o nazwie `input_configuration.txt`.
– `--output` - przyjmuje wartości `true` lub `false` i kontroluje zapisywanie do pliku o nazwie `output_configuration.txt` końcowej konfiguracji w takim samym formacie jak w przypadku pliku wejściowego z początkową konfiguracją.
– `--doFrames` - przyjmuje wartości `true` lub `false` i kontroluje zapisywanie klatek i tworzenie końcowej animacji.

Przykładowe uruchomienie programu w wariancie na 100%:

```
./CovidSimulation --nIter 200 --dt 0.02 --nPeople 20 --input random  --recoveryTime 1.2
```