The program implemented in the second part of the project is a pandemic simulator: given the information about the features of a population and the characteristics of the disease, it is possible to recreate a simulation of a pandemic in a space where the entities are allowed to move with a random speed and random directions.

The original idea that we had in mind was to create a space in which the entities would not move around at a constant random velocity but according to a vector field that would have been generated through a Perlin noise generator. Unfortunately, the implementation of the Perlin generator turned out to be more challenging than we thought, and the improvements we made were not as good as we hoped. We included both scripts in the directory: the one without the Perlin noise generator in the `pandemic_simulation_no_perlin` folder and the one that has it in `pandemic_simulation_perlin`.

Both of these programs can be complied with simultaneously using the `CMakelists.txt` file. To optimize the execution and the compilation of the code it is recommended to use the command:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug -S . -B build
```

The two executables that come out are one named `simulation` and the other `perlin_simulation` both of which accept the same input:

- The beta factor: it represents the probability that a sick entity would infect a susceptible when they are at a certain distance;
- The gamma factor: it represents the inverse of the recovery time (a gamma factor of 0.1 tells us that an infected entity would take $1/0.1 = 10$ days to recover);
- The number of entities;
- The dimensions of the window;
- The distance needed to infect;
- The size of the entities in the simulation;
- The amount of entities infected at day 0;

In the simulation, the beta factor, the gamma factor, and the number of entities are the essentials inputs that must be provided to the program to run correctly, while the last four variables are already preset but still changeable from the user. In particular, the window's dimensions are initialized to a 1500 by 1500 pixels square, the distance needed for an entity to infect another is the square root of 50, the number of infected entities is set at 1, and the size of their radius is 3 pixels. The user must give these variables at runtime. To make the input clearer, we used the Lyra library. To every

one of the parameters is assigned a command that must be followed from the input value.

```
OPTIONS, ARGUMENTS:
  -?, -h, --help
  -s, --sizeBall, --sb <Size Ball>
Radius of a single ball in the simulation. It must be a
positive integer;

  -x, --width <Window's Width>
Width of the window in which the balls move during
the simulation. It must be a positive ineger;

  -y, --height <Window's High>
Heigh of the window in which the balls move during
the simulation. It must be a positive integer;

  -d, --distance <Distancing>
Distance necessary to infect other people. It must be a
positive integer;

  -b, --beta <Beta
Beta factor - the infection factor. It must be a value
between 0 an 1;

  -g, --gamma <Gamma>
Gamma factor - the recover factor, ususlly it is the invers
of the time (in days) necessary to recover. It must be
a value between 0 an 1;

  -p, --population <Population>
Number of entities in the simulation. It must be a positive
integer;

  -i, --infected <Infected>
Number of sick entities that can spread out the disease at
day 0. It must be greater than 0 but smaller than the total
number of entities.
```

This is the replica of what the terminal would print if the user gives `-h` as a parameter at runtime. Here are listed all the characteristics and the interval of definition of each one of the variables.

For the parameters that can assume a positive integer value, there are also some limitations on how big they can get: the size of an entity cannot be greater than the module of the sum of the window's dimensions; the window's dimensions cannot be greater than the size of the screen on which it is working; the value of the radius of infectivity of an entity, can only be as great as half of the mean of the window's dimensions, but still if it is too big than all the entities will get infected almost immediately and the simulation loses its meaning; the number of entities in the simulation can only be as many as the area of the window divided by one hundred (if the computer of the user can handle this task).

The `simulation` program consists of a window in which a series of entities, represented by small circles, are free to move around with a random velocity which module square must be between 0.8 and 1 pixel per second (this interval comes from experimental data that show a good approximation with the sir model) (Giuse se vuoi qui puoi inserire il valore della varianza con i dati che ti mandero con il programma fixato !!CANCELLARE QUESTO MESSAGGIO!!)

The position of each entity at day zero is randomly assigned. At this time, a ball can only be found in one of two states: infected, if it is the or one of the "patient zero", or not infected (or susceptible). In the first case, a ball would have a rad color and a red ring around, showing the infection area, while in the second case, the ball would have a light blue color. The passing of time can be monitored on the terminal, that prints out some basic information about the course of the pandemic: the current day, the current amount of susceptible, the current amount of infected, and the current amount of recovered. The recovered are the last step of the evolution of the infection (in this model, entities of this state contains both the death and the actual recovered) When the infected counter gets at zero, the simulation automatically stops and closes. It is also possible to stop the simulation before its end by just clicking the close button on the window. After the end of the simulation, there will be available a file `.dat` that contains the data of the pandemic simulated: it consists of a table composed of four columns each showing the day, the ammount of susceptibles, infected, recovered, and as many rows as the amount of days that simulation lasted.

To implement such a model, we based all the code on the graphic library SFML, so it is crucial to have it installed on the machine that compiles the source.

The first of the two classes in the code is Variables. It is the container of the information needed for the program to work correctly. The private part of the class contains the fundamental variables of the system, initialized using the member function `parse_variables`, implemented using the Lyra library. The initialization uses seven set functions which work as the gateway between the information given by the user and the simulation. Each of them contains a condition function that checks if the parameter respects the interval of the definition of the variable and returns an error if it does not. Some of them are methods some are not. The difference is whether or not they use the private information of the class. To provide the private variables to other functions outside the class we also included seven get functions that return each one of the values.

The heart beating of the simulation is the Balls class which contains all the methods needed to create, move, check, and count each entity. The private part of the class contains a random number generator, a continuative seed generator, the two fundamental variables beta and gamma, and the vector of entities, named balls.

```
class Balls{
  std::random_device rd{};
  std::default_random_engine rnd{rd()};
  std::vector<Ball> ball{};
  float beta;
  float gamma;

  ...

};

```

The balls vector contains the `sf::Circles`, which are objects of the SFML library, the state of the entity (Sus, Inf, Rec), its velocity, and an integer value that allows remembering the day in which the entity got infected. When a Balls object is declared, the values of the beta and the gamma factor must be given to the class' constructor. The initialization of the rest of the variables occurs in the `add_balls` member function. Here the balls vector gets filled in with the number of entities taken as input, each with the same radius and characteristics but different velocity and initial position. These last two properties of the entities get assigned using two other member functions: `random_speed` and `random_position`, both using a random number generator. The first generates two numbers from a uniform double distribution between -1 and 1; one represents the velocity on the x-axis, the other the velocity on the y-axis, with the condition that the module square of their sum must be greater than 0.8 and smaller than 1. The second also generates two random numbers, but from a uniform integer distribution between zero and the window's size; one is the position on the x-axis, while the other is the position on the y-axis.

To print the circles we declared in the main file a `sf::Renderwindow` and created a loop that refreshes the image 60 times per second. Inside this loop, we implemented the functions that make the simulation. All the entities are displayed simultaneously using the `drw_balls` member function that prints every one of the circles at each frame. Every frame, the position of every entity is also updated using the function `move_balls` which uses the method `sf::move` that takes the entities' velocities as the parameter and move the entity in the direction of the velocity vector. Every circle changes its position 60 times per second, and while it does, another member function, called `check_collision`, checks the distance between one entity and all the others for every one of them. If that value is smaller than the one given in Variables as infectivity radius and the entity has an infected status, it calls the function `probability_of_infection` that uses the random number generator to get a number between zero and one. If this number is smaller than the beta factor, the function returns true, otherwise false. If the output of `probability_of_infection` is true and the status of the other entity is susceptible, then it turns infected. The program uses a clock,

declared in the main file, which every second add one day to the counter. When an entity gets infected, it saves the day in its information. Using the removed member function, after $1/\gamma$ days its status gets updated to Recovered. The function that unrolls the task of updating all the information that comes from the state of an entity is `check_status`.

```
public:
  Balls(float b, float g):
  beta{b},
  gamma{g}
  {}
  sf::Vector2f random_speed();
  void bounce_off_the_wall(sf::RenderWindow const &w1, int
    const i);
  void move_balls(sf::RenderWindow &window);
  void draw_balls(sf::RenderWindow &window) const;
  int probability_of_infection();
  void check_collision(int const day, Variables v);
  bool count_balls(sf::Clock &c, int &d, std::ofstream &write)
    const;
  void check_status(Variables const& v);
  sf::Vector2u random_position(Variables const& v);
  void add_balls(Variables const& v);
  void removed(int const day);

```

The window in which the circles move is not unlimited. When one of the circles gets to one of the window's "walls" it bounces back with a velocity characterized by the same module but opposite orientation (as the principle of conservation of momentum suggests) To know how many entities are in one state we implemented the `count_balls` function. This function prints out every second the amount of susceptible, infected, and recovered, both on the terminal and on a file `.dat`. While testing the simulator we noticed that even if we were changing the beta factor, the entities kept infecting each other with the same probability. The problem was that an entity needs some time to get out of another's infectivity radius, even if very small, the program does 60 calculations per second, which means that every second it generates 60 random numbers if two circles are in contact. The solution to this problem was to separate the refresh rate of the frame and the loop that contains `check_collision`. We added a new clock in the main file named check that checks the "collisions" every 1/6 seconds, this way two entities must share the same area for at least that amount of time to contract the disease. Thinking about it, also in the real world, even if the probability of being infected is 100%, a healthy person needs to share the same area of a sick one for a certain amount of time to be sure of getting the disease.

The Perlin simulation has a source code that is almost the same as the one just presented. The only difference between the two models is that here the speed of the entities is not constant but changes depending on the position of each circle. In this variation, we added a new header called speed

that contains three additional functions. We also added one in the class Balls regulating the changing of speed, and a Perlin noise library. The only new function in Balls is `change_speed`. This member function is used in the same loop of `check_collision`. Because of its heavy calculations doing it 60 times per second would require a lot of computing power. It checks the position of every entity and gives them a new velocity based on the information that gets from `perlin_speed`. This function is the main function of the speed library and is the function that returns the new velocity depending on th current position. The Perlin noise consists of random numbers generator which generates values that are consistant with the one generated previously. Practically the perlin noise looks more like a contiuos function, while an ordinary random generator is more like a jumping function.

In this case the perlinNoise function in the perlinNoise library takes the dimensions of the box and gives as output a scalar value. Because the output of the Perlin function is a scalar between zero and one, while the velocity of an entity is a two-dimensional vector we needed to get a conversion. To do so we created the function `vector_converter` which takes as a parameter a scalar value between zero and one, converts it in an angle, and returns a vector that contains its sine and cosine values, obtaining the desired type.