A C++ implementation of the SIR model and beyond

Simone Coli Giuseppe Sguera

Abstract

Within this project we implement a program that uses the SIR model to simulate the outbreak of a pandemy. The program can also operate backwards and estimate the SIR parameters of a given set of data by using the least squares method.

As variations on the theme we implement a forecasting program based on the logistic model, as well as a real-time graphic simulation of the spreading of a flu among a closed group of people.

All the code is written in C++.

External libraries needed to properly run the program (some of them must be installed onto the user's computer):

- Lyra;
- SFML;
- Doctest;
- CMake (optional);

The work is divided among three directories:

mandatory_SIR_model It contains the headers and source code of the main program.

logistic_model It contains the headers and the source code of the logistic model program.

graphic_simulation It contains the headers and the source code of the graphic simulation program.

Plus a directory, `files`, with the .clang-format and other files.

More about the SIR model at [1], [2], [3].

Contents

1 Mandatory SIR model

The code is spread over many translation units.

Classes are defined in headers, methods and free functions are declared in headers and defined in the source code. The file `pandemy.test.cpp` contains tests source code while `root_pandemy.C` is a root macro which can help to graphically visualize the program output.

To compile the program is suggested the use of CMake:

```
$ cmake -S . -B build/
$ make sir_model    # the program executable
$ make sir_model.t # the tests executable
```

otherwise you can use your favourite compiler. For example:

```
# to generate the program executable
$ g++ State.cpp Virus.cpp Pandemy.cpp Least_Squares.cpp \
Print.cpp Parser.cpp main.cpp -o sir_model
# to generate the tests executable
$ g++ State.cpp Virus.cpp Pandemy.cpp Least_Squares.cpp \
pandemy.test.cpp -o sir_model.t
```

Using CMake will compile with `-Wall -Wextra -fsanitize=address` options enabled.

1.1 Purpose

There are two modes the program can work in: the sir simulation mode and the fitting mode.

While running in the sir simulation mode, given the initial state of a population (number of susceptibles, infected and removed) and the parameters of the pandemic (beta and gamma values), the program simulates the developing of the epidemy by numerically resolving a system of three ODEs and prints on standard output the day-by-day count of susceptibles, infected and removed from day 1 until the day choosen by the user.

While running in the fitting mode, given a file containing the day-by-day count of susceptibles, infected and removed, the program estimates the parameters of file pandemic by means of the least squares method [6] and prints them on standard output. Data must be provided in the format `day S I R`, with blank spaces separating the values.

In the directory `files` of this archive there are some pdfs of some fitting made during the debugging and the tesitng.

1.2 Running the program

First you must choose whether to run the sir simulation mode or to run the fitting mode. The two choices are mutually exclusives. If no choice is made, the program will do nothing.

```
$ ./sir_model sir # runs the sir simulation mode
$ ./sir_model fit #runs the fitting mode
$ ./sir_model.t #executes tests
```

The other options (such as the initial state of the population or the file containing the data to fit) are entered from command line by using the specific token.

For more detailed instructions use the help command:

```
$ ./sir_model --help # general help
$ ./sir_model sir --help # sir simulation mode help
$ ./sir_model fit --help # fitting mode help
```

If gnuplot is installed, it is possible to use it to graphically visualize the program output. Alternatively, if ROOT is installed, it is possible to load the macro and execute it.

```
$ ./sir_model sir -b 0.8 -g 0.3 -S 900 -I 100 -R 0 -d 30 > dati.dat
$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 8    last modified 2019-12-01
```

```
        Copyright (C) 1986-1993, 1998, 2004, 2007-2019
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> plot "dati.dat" u 1:2 w l, "dati.dat" u 1:3 w l, \
 "dati.dat" u 1:4 w l
```

or

```
$ ./sir_model sir -b 0.8 -g 0.3 -S 900 -I 100 -R 0 -d 30 > dati.dat
$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.22/00                        https://root.cern |
  | (c) 1995-2020, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for linuxx8664gcc on Jun 14 2020, 15:54:05                 |
  | From tags/v6-22-00@v6-22-00                                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
   ------------------------------------------------------------------

root [0] .L root_pandemy.C
root [1] pandemy("dati.dat")
```

1.3 Classes and methods

The main classes of the program are:

Pandemy The class core of the program. It contains all the data of the pandemic to simulate (the state and the parameters). An object of type *Pandemy* it's initialized by giving it as arguments a *State* object and a *Virus* object (see below).

Methods of the class:

progression It takes as argument the duration in days of the simulation and returns a vector containing the day-by-day count of susceptibles, infected and removed.

get_data It takes as argument the name of the file containing the data to fit with the fitting mode and returns a vector containing the data acquired.

get_state get_virus They respectively return the *State* and the *Virus* objects *Pandemy* has been initialized with. This methods are not used in the main program, they are needed only for testing.

Least_Squares The class that deals with the fitting part. By taking as argument of initialization a vector containing the data to fit, its member functions implement the least squares method (from now on LSM) and the compute of the chi square.

Methods of the class:

Chi_Square It returns the chi square of set of data relatively to a theoretical prevision.

get_parameters After appyling the LSM, it returns a *Virus* object initialized with the estimated parameters of the data fitted. The decimal precision of the fitting is given as argument of the method.

Minor data structures - but not less important - are:

State A simple struct to store all the information of the state of the population, such as the number of people belonging to each compartment. `i_sigma` is a value used to compute the chi square and represents the uncertainty associated with the number of infected.

Virus Same as the *State* struct, it contains the information of the disease causing the pandemic, that are the transmission rate (β value) and the inverse of the infection period (γ value).

Parser Struct that handles the parsing of the command line. To this scope, it has been used Lyra. The parser checks the correctness of the input format. No checks are made on the input values, except for the duration of the simulation and the decimal precision of the fit. All the other value checks are performed inside their specific class constructor.

For both the *State* and *Virus* struct is defined the `operator==`, which turns to be useful for testing purposes.

Not all the code is gathered in data structures and classes, there are some free functions too. The choice not to define another class is due to their marginal role in the program. They are not related, in fact, with the SIR model nor with the LSM. Those free functions are:

copy_round A simple but effective function to round the members of a *State* object. It is used when printing the output of the progression method.

print A function that prints on standard output a table with the day-by-day count of susceptibles, infected and removed. It is disabled.

print_simple A function that prints on standard output the row day-by-day count of susceptibles, infected and removed. The print format is `day S I R`. The datavalue separator is a blank space, but it can be substituted with a comma to print in csv format.

1.4 Limits of the program

The assignment of this project provides that both the transmission parameters β and γ must be between zero and one.

This puts a very heavy limit on the forecasting capability of the program. The beta factor is defined as the inverse of the time interval between two contacts, which means it represents the number of conctacts per unit of time. Taking as unit an interval of 1 day, imposing $\beta \in [0; 1]$, prevents the program to simulate the outbreak of pandemic with more than 1 encounter/day, or to fit data with a ratio $I(t+1)/I(t)$ greater than 2 (which is most of the real cases).

Another limit to the fitting power of the program lies in the SIR model itself. The amount of people passing from one compartment to another are deeply linked to each other and to the number of people in the compartments. For example, the number of new cases at a time t, is detemined by how many susceptibles and infected there are and by how many people are passing in the removed group at t.

This isn't much of a problem when simulating an epidemic or when fitting data that contains information on all the poopulation. But in most of the real cases, it is known only the number of infected and removed, because those are the only mesurable. So a guess on the total size of the population must be made by the user, resulting in an inaccurate fit.

More limitations are due to the constant value of beta and the assumption that the number of all the infected is known at every time t, which does not reflect reality. However considering also this particual cases is far beyond the goal of the program.

You can find more data and more fitting at KaldarrostaJazz, which will be kept updated to implement more models.

2 Logistic model

As for the previous program, the code is spread over many translation units, with classes defined in headers and methods and free functions declared in headers and defined in the source code. The file `logistic_test.cpp` contains tests source code while `root_logistic.C` is a root macro which can help to graphically visualize the program output. `converter.cpp` is a small program to help converting data into the correct format. `class_acquisition_data_test` is a file containing the data used to run the tests, this data have been taken from the Protezione Civile Italiana github page, they are the data of the first

40 recorded days of COVID-19 pandemic in Italy (form March 1st 2020 to April 10th 2020).

To compile the program is suggested the use of CMake:

```
$ cmake -S . -B build/
$ make logistic_model    # the program executable
$ make logistic_test.t # the tests executable
$ make converter # the data converter
```

otherwise you can use your favourite compiler. For example:

```
# to generate the program executable
$ g++ logistic_model.cpp main.cpp -o sir_model
# to generate the tests executable
$ g++ logistic_model.cpp logistic_test.cpp -o sir_model.t
# to generate the data converter executable
$ g++ converter.cpp -o converter
```

Using CMake will compile with `-Wall -Wextra -fsanitize=address` options enabled.

2.1 Purpose

The program has the aim of overcoming the limits of the SIR fitting model previously explained.

As studied by M. Batista in [4], the logistic model can be applied to the start of a pandemic to make forecastings on the maximum outbreak of the disease.

This program implements the logistic model following Batista's directives and fits a set of data returning the parameters of the corresponding logistic curve. The curve parameters are connected to some charateristics of the pandemic. In particular:

- The K factor is the maximum number of infected that will be reached during the pandemic.
- The A factor is linked with the initial number of infected at time 0 by the relation $I(0) = \frac{k}{1+A}$.
- The r factor is the transmission rate. It is related with the $R(t)$ of the pandemic: $R(t) = \frac{1+Ae^{-rt}}{1+Ae^{-r(t+1)}}$.

To estimate the values and minimize the variance it is used the steepest descent method [7], and so an initial guess must be made (as Batista explains in his work). The user MUST NOT make the guess, but instead has to choose two points of the data (the k-th and the m-th) which will be used by the program to make the initial guess. The two points must be choosen in a

way that, if N are the days of pandemic recoreded in the data $k \approx N$ and $m \approx k/2$.

On standard output the program prints some information about the fit:

- the initial guess (if found);
- the estimated parameters;
- the variance;
- the day with the maximum number of daily cases;

If asked, the program can olso make a forecast for a specific day after N.

More about the logistic model applied to the growth of a population at [2], [5].

2.2 Running the program

To know how to run the program enter the help command

```
# logistic_model help
$ ./logistic_model --help
# coverter help
$ ./converter --help
```

if the `-d <value>` option has been enabled the program will make a forecast for the <value>th day and will create a `.dat` file with the day-by-day count of the cases. It is possible then to load the root macro `root_logistic.C` and graphically visualize the prevision of the model:

```
$ ./build/logistic_model -d 45 -m 14 -k 37 -a 0.0005 -f matteo_l.dat
An initial guess was found:
K(0) = 155.031 A(0) = 118.131 r(0) = 0.152266
The series converged after 129 iterations
Estimated parameters:
K = 163.733| A = 118.073| r = 0.152267
Estimated infected at t = 0: 1.37506
The maximum daily cases day was day 31.3352 from the data day one
Variance = 72.4682
Total cases expected at day 45 from the first day of the data = 145.561
$ root
   ------------------------------------------------------------------
  | Welcome to ROOT 6.22/00                        https://root.cern |
  | (c) 1995-2020, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for linuxx8664gcc on Jun 14 2020, 15:54:05                 |
  | From tags/v6-22-00@v6-22-00                                      |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
```

```
root [0] .L root_logistic.C
root [1] print_fit("matteo_l.dat")
Info in <TCanvas::Print>: pdf file canva.pdf has been created
```

2.3 The classes and their methods

The data converter program is very simple and its code is entirely in `converter.cpp`. It reads the choosen file and creates a new one with the data formatted in the proper way. The file to read and the one to write must be given as command-line arguments. It is also possible to convert only a part of the data, by using the option `-d <value>`. For example to convert only the first 35 days: `-d 35`. To open and interact with the files it has been used the `fstream` library, while to parse the command line Lyra. The program is already configured to covert data formatted in the `day S I R` form, as the one printed by the sir model simulators. It is possible to modify this if needed (e.g. to read a file which shows only the number of infected and removed).

The implementation of the logistic model is made up by only three classes.

Logistic The class that implements the logistic model and computes the number of cases at a time t. The constructor takes as argument an `std::array` of three doubles, each of them corresponding to a parameter of the curve. `cases_t` and `dcases_t` do pretty much the same job, one calculates the number of total infected at t, the other calculates the daily number of infected at t. The method `cases_grad` computes the gradient of the logistic function and is used by the steepest descent algorithm. `log` creates a `.dat` file containing the day-by-day count of both the total infected and the daily infected.

Acquisition It handles the acquisition of data from external files.The file must be given to the program already formatted by using the converter. The name of the methods are self-explanatory: `Data` returns the total-cases vector, `dData` returns the vector of the daily cases.

Fit The main dish of the code. The *Fit* class does the hard job of estimating the parameters. To initialize a *Fit* object must be given as arguments:

- the m data index;
- the k data index;
- the convergence factor α (indicated in the code with the letter a);
- the name of the file to fit;

The major mathods of the class are:

initial_guess which returns the initial guess for the K, A, r parameters using the m-th value and the k-th value of the data;

steepest_descent which takes as argument the initial guess and returns the estimates of the parameters after running the steepest descent algorithm (from now on SD).

The `steepest_descent` function needs another argument to work, which is the *delta* vector (in the code it is an `std::array`). This vector contains the quantity by which the parameters are modified (in positive or in negative) after every iteration.

The other methods of the class concurr to the SD algorithm:

- variance computes the variance between the data and the predicted values;
- std_dev computes the standard deviation between the data and the predicted values;
- var_grad is the function that calculates the gradient of the variance;

The parsing of the command line is done by Lyra and the code is in the `main.cpp` and doesn't do any checks on the correctness of the values, which instead are inspected in the *Fit* constructor and in the functions.

The *delta* vector is initialized in `main.cpp` as an `std::array`.

To store the values of the three parameters K, A, r it has been used an `std::array` object instead of a struct because a struct wasn't necessary. The maximum number of iteration to search for the parameters is 10000 and it is initialized as a global variable.

2.4 Limits of the program

Although the logistic model gets over the biggest problem of the SIR model fitting, it is not exempt from having limitations - it still is an idealization.

The biggest restriction is the incapability of predicting the entire course of the pandemic. Since the logistic function converges to a maximum, the logistic model can make forecastings and fitting only in the first part of the pandemic, when the contagions are still growing. So it is impossible to predict the dedflation of the disease or the rate at which the infections will decrease.

Moreover, the logistic model is very precise and reliable on short-term forecasts (max 5 days after the last day of the data), but tends to be more inaccurate when asked to make prediction further predictions.

Another heavy limitation is the importance of the initial guess. Choosing wrong m and k values can lead to very bad fittings, with a big variance.

There are algorithms that bypass this problem, but they need lot of computing power to make estimates in a decent time. However there is a WOP to implement also those algorithms.

It is a metter of costs and benefits: not to need the absolute knowledge of the state of the entire population, or to make prediction for the deflating part of the pandemic.

We have fitted Italy and China data of the early stage COVID-19 pandemic and we are pride to say that the program does a very good job. In the `files` directory you can find those fittings. The forecasted values where in agreement with the real values in the range of the error (the square root of the variance).

3 Graphic simulation

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`.

3.1 Running the program

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 7 pixels, 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).

3.2 Charateristics of the program

The `simulation` program consists of a window in which a series of entities, represented by small circles, are free to move around with equal speed and random direction.

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 is one of the "patient zero", or not infected (susceptible). In the first case, the ball is red with a red ring around, showing the infection area, while in the second case, the ball would be of a light blue color. The passing of time can be monitored on the terminal, that prints out some basic information about the progression of the pandemic: current day, current amount of susceptible, current amount of infected, and current amount of recovered. The recovered are the last step of the evolution of the infection (in this model this compartment contains both the deads and the actual recovered). When the infected counter gets to 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.

3.3 Classes and methods

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 equal to 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.

3.4 The Perlin simulator

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.

You can find more code at JustSimone.

3.5 The concretness of the model

This model is an idealization of a real situation, so it make some assumptions a priori that must be justified a posteriori.

In particular we made the assumprion that the space is homogeneous (the entities are uniformally distributed) and isotropic (the sum of all the velocity is equal to zero). Basically we have made the same assumption of the molecular chaos in statistical mechanics.

Those assumptions are in agreement with the assumptions of the SIR model and so we expect the two models to agree.

So they do. We have collected data from this program and then we have fitted them with the sir fitting program and we got very positive results. For example we fitted the `03_ball_pandemy.dat` (see below) and we got a chi square of 360.056, which converted to reduced chi square is $\tilde{\chi}^2 = \frac{360.056}{343} = 1.06$, meaning there is a strong agreement between data and theory (343 comes from $115 \cdot 3 - 2$). The graphic result of the fit is `03_ball_pandemy_fit.pdf`.

We have included some set of data and the relative fits in the `data_and_pdfs` folder under the name "ball_pandemy" or "bp". Feel free to try on your own and test the program by fitting it with the fitting mode of the sir model at section 1.

4 Testing and conclusions

In addition to the test written in the source code, it is interesting to cross test the programs.

Simulating a pandemic with either the SIR or the graphic simulator will produce data that fit perfectly with both the fitting programs. A friend and colleague of ours Matteo Bonacini has implement a pandemic simulator based on AI. We have had fun fitting the data he provided us and the result is a wonderful agreement between his and our model, proving that either both of them are right, either are wrong (e.g. see here). However, the capability of our models to fit the the real-world data, makes more probable the first option.

You can find some of those fitting in the `data_and_pdfs` folder of the `files` directory. All the files related to Matteo's project go under the name "matteo*".

We hope you will find our work as fun as we had and that you share the same love for data analysis as we do.

References

[1] https://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology.

[2] Gaeta, Giuseppe (2009), *Modelli Matematici in Biologia*, Springer.

[3] E. Sadurní, G. Luna-Acosta (2021), *Exactly solvable SIR models, their extension and their application to sensitive pandemic forecasting*, Springer Nature.

[4] Batista, Milan (2020), *Estimation of the final size of the coronavirus epidemic by the logistic model.*

[5] https://en.wikipedia.org/wiki/Logistic_function.

[6] Fornasini, Paolo (2008), *The Uncertainty in Physical Measurements*, Springer.

[7] https://en.wikipedia.org/wiki/Method_of_steepest_descent.