Optimization
Genetic Algorithm: A possible model for COVID-19

Arturo del Cerro Vergara (NIU:1593930) and Jorge Piqueras Marqués (NIU:1583634)

February 16, 2021

Master's degree in Modelling for Science and Engineering

Abstract

Taking a SIR model for the COVID-19 pandemic, in this project we have implemented a genetic algorithm that tries to find the individual that potentially could have started the pandemic. In order to do this we studied and built up a program based on genetic algorithms that, with the help of dynamic mutations, crossovers and elitism, achieves a reduction on the fitness of each generation converging into a local minima. For this reason, we give arguments to explain the non full convergence of the algorithm and we find some other combined methods that try to force the algorithm finding a better solution.

Contents

1 Introduction

1.1 Genetic Algorithm

The history of genetic algorithms started a long time ago in the 1950's, when scientists dwelled on the idea of implementing concepts of evolution and natural selection into computer science. Popularized by John Holland, this optimization algorithm, just like any other, attempts to find the optimum values of a set of variables, which maximize or minimize an heuristic, which in this case we call fitness function. In contrast to simulated annealing and other single solution based algorithms, genetic algorithms are population based, which means that many candidate solutions are presented during the search, making it more exploratory.

The main idea is to randomly produce an initial population of individuals, which are represented by a set of variables (that constitute the genotype) that are a candidate solution to the optimization problem. This genotype is composed by the genes, that is, each one of the variables, which are usually represented in binary. Then, mimicking a process of natural selection, the individuals that compute a better fitness value are selected to generate offspring. This is done by a process of recombination of the parents' genes, and sporadically, mutation. The resulting individuals become members of a new generation, and the process is repeated until the new population is the same size as the starting one. Given that we are choosing the fittest individuals to generate the offspring, each generation has a better mean fitness than the previous one, and the process can be repeated until this mean fitness is optimized, or the problem converges (most individuals have the same genotype).

The combination of the selection process; where the fittest individuals are more likely to reproduce, and therefore we focus the search on the regions of better fitness; and mutation, to avoid having a population of copies of well-fitted individuals; makes genetic algorithm a method that balances well exploratory and exploitatory strategies. This is the reason why GA are used to solve problems with big fitness landscape, and are chosen over other methods in many domains, from engineering or physics to computer science.

1.2 Project description

1.2.1 The model

The problem that we try to solve in this project is based on a model of a system of differential equations with 8 variables, that intends to describe the evolution of the COVID-19 epidemic. The system of differential equations is the following:

$$\begin{cases} \frac{\mathrm{d}S(t)}{\mathrm{d}t} & = -\lambda(t)S(t), \\ \frac{\mathrm{d}E(t)}{\mathrm{d}t} & = \lambda(t)S(t) - \sigma E(t), \\ \frac{\mathrm{d}I_1(t)}{\mathrm{d}t} & = \sigma(t)E(t) - \gamma_1 I_1(t), \\ \frac{\mathrm{d}A(t)}{\mathrm{d}t} & = \gamma_1(1-p)I_1(t) - (\kappa+\gamma_2)A(t), \\ \frac{\mathrm{d}A_d(t)}{\mathrm{d}t} & = \kappa A(t) - \gamma_2 A_d(t), \\ \frac{\mathrm{d}I_2(t)}{\mathrm{d}t} & = \gamma_1 p I_1(t) - (\alpha+\gamma_2)I_2(t), \\ \frac{\mathrm{d}Y(t)}{\mathrm{d}t} & = \alpha I_2(t) - (\delta+\gamma_2)Y(t), \\ \frac{\mathrm{d}R(t)}{\mathrm{d}t} & = \gamma_2\left(A(t) + A_d(t) + I_2(t) + Y(t)\right). \end{cases} \tag{1.2.1}$$

The 8 variables of the model are:

- $S(t)$: number of susceptibles to the infection.
- $E(t)$: number of individuals already exposed to infection.
- $R(t)$: number of recovered individuals.
- $I_1(t)$: number of presymptomatic infectious individuals.
- $A(t)$: number of almost asymptomatic infectious individuals, undetected by the health system.
- $A_d(t)$: number of almost asymptomatic infectious individuals, detected by the health system.
- $I_2(t)$: number of infected individuals with strong symptoms.
- $Y(t)$: number of infected individuals with serious symptoms.

The model has a total of 11 parameters, and also includes the strength of the infection at time t:

$$\lambda(t) = \beta \frac{\phi I_1(t) + A(t) + (1-\epsilon_I)\left[A_d(t) + I_2(t)\right] + (1-\epsilon_Y)Y(t)}{P}, \tag{1.2.2}$$

where P is the population size. The 11 parameters have the following meaning:

- β: average infectivity.
- $\phi < 1$: pre-symptomatic infectivity factor.
- ϵ_I: isolation effectiveness of strong cases.
- ϵ_Y: isolation effectiveness of serious cases.
- σ: rate of appearance of infectious ability.
- γ_1: rate of appearance of symptoms.
- γ_2: rate of recovery.
- κ: detectability rate of asymptomatic people.
- p: probability for strong symptoms (probability of the health system to clinically detect cases).

- α: rate of appearance of serious symptoms (i.e. rate at which individuals with strong symptoms become so seriously ill that they are in need of hospitalization and a fraction of those will require intensive care and respirators).
- δ: disease-induced mortality.

Under this assumption, the population size at time t, which approximately coincides with $P(0)$, is:

$$P(t) = S(t) + E(t) + I_1(t) + A(t) + A_d(t) + I_2(t) + Y(t) + R(t). \tag{1.2.3}$$

We supose that the population P and the variables $A_d(t)$, $I_2(t)$, $Y(t)$, $R(t)$ and $D(t)$ are known, and the rest of them are unknown. The objective of the project is; starting from a set of values for the 5 known variables, and for 100 days; finding the set of 11 parameters that best predicts the values of the variables for the 100 days, starting from a fixed initial condition.

1.2.2 Individuals

The first thing that we should do, is propose a way to store the genotype of the individuals that will be a possible solution to our problem. In our case, the genotype is composed by two genes: the initial conditions of the model variables $E(0)$, $I_1(t)$ and $A(t)$ (the other variables are given in the first row of the data table, or can be calculated with them), and also the 11 parameters. In order to store and manage the genotype of the individuals, we will create the following structure:

```
typedef struct {
  unsigned long IC[IC_GENES_NUMBER];  // Initial conditions. In this order:
  ↪  E(0), I1(0), A(0).
  unsigned long Pars[PARAMETERS_GENES_NUMBER]; //Genotype. In this order:
  ↪  beta,cphi, epsI, epsY, sigma, gamma1, gamma2, kappa, p, alpha, delta.
  double fitness;
} Individual;
```

As we can see, we also include the `fitness` variable to store the heuristic of a given individual. Notice how we set the genes to be of type `unsigned long`, as we should be ready to perform bitwise operations with them. To do this, we will store these integers in binary, and the precision to do so is described in 1. With this structure, we will be able to perform the cross-over and mutation operations on the genes, but we also need to store the parameters of a given individual as `double`, because we will use them to compute the solution of the system of differential equations. To do this, we define another structure, `ODE_Parameters`, that contains the 11 parameters in a suitable way to calculate the fitness.:

```
typedef struct{   //Store the 11 ODE parameters
  double beta, phi, epsI, epsY;
  double sigma;
  double gamma1, gamma2;
  double kappa;
  double p;
  double alpha;
  double delta;
  double PopSize;
} ODE_Parameters;
```

Also, we provide defined functions to switch between the two representations of the parameters inside the algorithm (from integer in binary to `double`).

		Phenotype		Genotype	
		upper limit	precision	upper limit	Factor from genotype to phenotype
Initial Conditions	$E(0)$, $I_1(0)$, $A(0)$	P	10^{-3}	2^{30}	$\frac{1}{1000}$
Parameters	β	1	10^{-12}	2^{40}	2^{-40}
	ϕ	1	10^{-6}	2^{20}	2^{-20}
	ε_I	1	10^{-3}	2^{10}	2^{-10}
	ε_Y	1	10^{-3}	2^{10}	2^{-10}
	σ	1	10^{-6}	2^{20}	2^{-20}
	γ_1	1	10^{-6}	2^{20}	2^{-20}
	γ_2	1	10^{-6}	2^{20}	2^{-20}
	κ	1	10^{-3}	2^{10}	2^{-10}
	p	1	10^{-6}	2^{20}	2^{-20}
	α	1	10^{-12}	2^{40}	2^{-40}
	δ	1	10^{-12}	2^{40}	2^{-40}

Figure 1: Summary of the upper limit and precision used to store the genotype (in the `individual` structure) and the phenotype (in the `ODE_Parameters` structure), which are chosen to match common sense values of the genes.

1.2.3 Fitness function

In order to estimate how close an individual is to the solution, we need to define an heuristic that we call the fitness function. As we previously explained, our goal is to find an individual with a set of parameters which can model best the epidemic, that means that with its 11 parameters and the initial condition for the starting day, we should be able to solve the differential equations to get the value of the variables for the 100 next days, and then, the closest this values are to the ones on the real data table, the better fitness an individual should have. Considering this, it makes sense to define a fitness that uses the norm of the values

in the predicted data by the parameters, and the real data. For instance, we might define the fitness function as:

$$\sum_{t=1}^{100} t\left[\left(A_d(t)-\overline{A_d}(t)\right)^2+\left(I_2(t)-\overline{I_2}(t)\right)^2+\left(Y(t)-\overline{Y}(t)\right)^2+\left(R(t)-\overline{R}(t)\right)^2+\left(D(t)-\overline{D}(t)\right)^2\right], \tag{1.2.4}$$

where the over-bared variables ($\overline{A_d}(t)$ and so on) are the real data values from the epidemic, and the non-bared variables are the predicted by the model parameters for each day t. This way, the fittest individuals will be the ones with a lower fitness function, as that will mean that the are better at predicting the real evolution of the epidemic. By 1.2.4 we are giving more importance to the last days than to the first ones to ensure the individual reaches a consistent solution for the problem.

2 The code

In this section, we will explain the code that we implemented to attempt to solve the optimization problem, including some parts of the code and describing the logic behind the algorithm and the different techniques used to encode the selection, crossover and mutation of the genetic algorithm. We will also include an alternate hybrid version of the algorithm that also implements a descent method, as an attempt to improve the results.

2.1 Storing the real data and initializing the population

The first thing the algorithm does, is to retain the table of real data that we use to calculate the fitness. In order to make the process easier and avoid hard-coding, we keep the data inside a `.txt` file named **`Genetic_data.txt`**. Then, we use a C language file named `read_data`, where we keep the functions `Get_data` and `Data_from_txt`, which read the text file line by line and store the variables for each day. Also, the format that we use to store the data is also a structure, called `DataForFitting`:

```
typedef struct{
        double PopSize;
        unsigned long N_days;
        double Data_Time_Series[Ndays_Time_Series][Nvar_Time_Series];
} DataForFitting;
```

This structure contains the population size `PopSize`, the number of days `N_days`, and a 2x2 matrix `Data_Time_Series`, containing the data (the rows are each one of the days, and the columns are the 5 variables). Then, once the

Get_data function is called, it takes a pointer to a globally declared DataForFitting structure, named **TheData**, and inside it, it calls the **Data_from_txt** function, which opens the text file and stores the data on the Data_Time_Series matrix of the TheData structure.

This structure, once created, is not modified during the algorithm. Although, we also need to store the values of the variables that the model calculates making a prediction with the parameters of an individual. We will create a DataForFitting structure each time we calculate the fitness of an individual, in order to store the prediction and compare it to the TheData structure.

The next thing we do to start up the algorithm is creating the initial population. We need to choose the population size and use the **create_Population** function, which fills up a vector of individuals (individual pointer) named population. For each individual inside the population, the **create_Individual** function is called:

```
double uniform(){
    return (double)rand() / (double)RAND_MAX ;
}

Individual create_Individual(){
    Individual ind;

    ind.IC[0]    = floor(uniform()*pow(2,30));
    ind.IC[1]    = floor(uniform()*pow(2,30));
    ind.IC[2]    = floor(uniform()*pow(2,30));

    ind.Pars[0]  = floor(uniform()*pow(2,40));
    ind.Pars[1]  = floor(uniform()*pow(2,20));
    ind.Pars[2]  = floor(uniform()*pow(2,10));
    ind.Pars[3]  = floor(uniform()*pow(2,10));
    ind.Pars[4]  = floor(uniform()*pow(2,20));
    ind.Pars[5]  = floor(uniform()*pow(2,20));
    ind.Pars[6]  = floor(uniform()*pow(2,20));
    ind.Pars[7]  = floor(uniform()*pow(2,10));
    ind.Pars[8]  = floor(uniform()*pow(2,20));
    ind.Pars[9]  = floor(uniform()*pow(2,40));
    ind.Pars[10] = floor(uniform()*pow(2,40));

    ind.fitness  = MAXDOUBLE;

    return ind;
}

Individual * create_Population(int population_size){
    Individual * population = malloc(sizeof(Individual) * population_size);
    for(int i = 0; i < population_size; i++){
        *(population + i ) = create_Individual();
    }
    return population;
}
```

As we see, the initial conditions and the parameters are initialised taking a random `double` value normalized to 1, and then renormalized to the upper limit of the genes, described in Figure 1. The floor function is called to transform intro a rounded down integer in binary. Also, the fitness is initialized to the value of the maximum representable double number.

2.2 Fitness calculation. Integrating the ODEs with `RKF78.c`

The calculation of the fitness from the individual's genes is not direct. The fitness is calculated with Equation 1.2.4 using the structure with the predicted variables of the model for the 100 days, but not with the parameters. In order to solve the system of differential equations and therefore obtain the predicted counts for an individual, we make use of a Runge-Kutta-Fehlberg algorithm the code of which is provided in a **`RKF78.c`** file. The functions that are implemented to control the calculation of the fitness are mostly given by the project supervisors, so we will not add their codes. The process can be summarised as follows:

1. **`CoreModelVersusDataQuadraticError`** is a function that takes a pointer to an individual and a void pointer to the structure of real data. The first thing it does is declare a `DataForFitting`-type struct named **`ThePrediction`** that is initialized only on the first row of the matrix, where we set the initial conditions. Then, it creates a vector **`xt[CoreModelDIM]`** which contains all the variables that the integrator uses as initial condition: the IC inside the individual, and the first row of the data table. Then, it also creates a `ODE_Parameters`-type structure to store the model parameters of the individuals, but as doubles, so that they have the right format for the integrator.

2. **`GeneratePredictionFromIndividual`** is then called inside the previous function. Inside it, a loop iterates over the total number of days, using the **`RKF78sys`** function. In summary, this function uses the initial conditions in the `xt` vector and the parameters `ODE_pars`, update the `xt` vector after every iteration, with the variables of the model at time $t+1$. The functions returns an status if an error occurs, otherwise, the $t+1$ row in the `ThePrediction` structure is filled with the values in `xt`. This repeats until the total number of days (100 in our case) is reached, and the prediction matrix is filled. In addition, the `RKF78sys` function needs a void **`Coremodel`** function as an input, which contains the differential equations of the system in terms of the parameters and the model variables.

3. After the `GeneratePredictionFromIndividual` is finished, the structure with the prediction is filled with the data and we can use it to calculate the fitness of the individual using Equation 1.2.4:

```
double fitn=0; double sum=0;
for (int i=0; i<Ndays_Time_Series; i++){
    for (int j=0; j<Nvar_Time_Series; j++){
    sum += pow(ThePrediction.Data_Time_Series[i][j] -
    ↪  TDfF->Data_Time_Series[i][j],2);
    }
    fitn += (i+1)*sum;
    sum = 0;
}
ind->fitness = fitn;
```

Following this process, every time the `CoreModelVersusDataQuadraticError` function is called for an individual, its fitness is calculated and stored.

2.3 Main algorithm

The following code is the part of the `main.c` file that contains the core of the genetic algorithm. The outermost `while` loop repeats the algorithm until the maximum number of desired generations is reached. On each generation, a new population, that we call `next_generation`, is created from offspring of the individuals of the previous generation. Elitism will be explained in the next section. Inside the `for` loop marked with Ⓐ, the individuals of the new generation are created. It iterates until half of the population size because two individuals are produced on each iteration. Two parents are selected using a tournament selection method (explained later), and then we produce the offspring with one point crossover. The crossover function has a probability of 75%, so eventually, the parents will be copied to the next generation without any modification. Afterwards, mutation is applied for the resulting individuals. The probability of mutation is dynamic, which means that it changes during the run, and the method to set it is not trivial, some of the different ways will be discussed on the next sections.

Finally, the fitness of the offspring is evaluated with the `CoreModelVersusDataQuadraticError` function. The `while` loop around the whole process controls if the individuals that are created are not feasible, which happens when the `RKF78Sys` function returns an error and the fitness value is set to `MAX_DOUBLE`. If at least one of the two individuals is feasible, we keep both, and if both of them are infeasible, we repeat the process of creation until one of them is feasible, ensuring a population with at least 50% of feasible individuals. The reason why we choose to move into the next pair of individuals, even if one of the two is unfeasible, is that we trust that, with the presence of mutation, it will make the algorithm more exploratory.

```
int generation = 0;
double fitness_goal = 100;
```

```
while (generation <= max_num_generation && population->fitness >
↪  fitness_goal){
  Individual * next_generation = NULL;
  if ((next_generation = (Individual*)
  ↪  malloc(population_size*sizeof(Individual))) == NULL)
      Exit_Error("When allocating memory for the next generation", 6);

  fprintf(output,"\n\n------------- Generation Number %i -------------
  ↪  \n",generation);

  double infeasibles = 0;

  int best = random_in(0,population_size-1),second_best =
  ↪  random_in(0,population_size-1);  //Elitism: the two best individuals
  ↪  go to the next generation
  double fittest_val = infeasible_value;
  for (int i=0;i < population_size; i++){
      if (((population+i)->fitness) < fittest_val) {fittest_val =
      ↪  (population+i)->fitness; second_best = best; best = i;}
  }


  ↪  OnePointCrossover(population+best,(population+second_best),(next_generation),(next_gene
  CoreModelVersusDataQuadraticError((next_generation),TheData);
  CoreModelVersusDataQuadraticError((next_generation+1),TheData);


(*) for(int i = 1;i < population_size/2; i++)
    {
      double prob_mutation;
      int is_feasible = 1;
      while (is_feasible){ // Controls the proportion of infeasible
      ↪  individuals
        // Selection for crossover
        Individual *parent1 = population + tournament_selection(population,
        ↪  population_size);
        Individual *parent2 = population + tournament_selection(population,
        ↪  population_size);
        OnePointCrossover(parent1, parent2, (next_generation + 2*i),
        ↪  (next_generation + 2*i + 1),0.75);

        for (int child = 0; child <= 1; child++) {
          // Dynamic decreasing mutation
          prob_mutation = (next_generation + 2*i +
          ↪  child)->fitness/(pow(10,12) + (next_generation + 2*i +
          ↪  child)->fitness);
          Mutation((next_generation + 2*i + child), prob_mutation);

          CoreModelVersusDataQuadraticError((next_generation + 2*i +
          ↪  child),TheData);

          if (((next_generation + 2*i + child)->fitness <
          ↪  infeasible_value)) {
            CGDescent((next_generation + 2*i + child),TheData); //
            ↪  Conjugate Gradient Descent
```

```
            SA((next_generation + 2*i + child),TheData); // Simulated
            ↪ Annealing
          }
        }

        if (((next_generation + 2*i)->fitness < infeasible_value) ||
        ↪ ((next_generation + 2*i + 1)->fitness < infeasible_value))
        ↪ is_feasible = 0;

      }
    }

    Individual * aux = population;
    population = next_generation;
    free(aux);

    double fittest = DBL_MAX;
    double mean = 0;
    for (int j = 0;j<population_size;j++){
        if ((population+j)->fitness < infeasible_value){mean  = mean +
        ↪ (population+j)->fitness;}
        else infeasibles++;
        if ((population+j)->fitness < fittest) {fittest =
        ↪ (population+j)->fitness;}
        }

    fprintf(output,"\nFittest individual fitness value = %f\n",fittest);
    fprintf(output,"%f,\n", fittest);
    fprintf(output,"Mean fitness of the feasibles in population =
    ↪ %f\n",mean/(population_size-infeasibles));
    fprintf(output,"Proportion of infeasibles individuals in generation %i =
    ↪ %f\n",generation,infeasibles/population_size);

    generation++;
  }
```

The final part of the main algorithm, outside of the outermost while loop, copies the next generation into the starting population so that the next iteration can happen. After this, we calculate the mean value of the fitness among the individuals in the generation, the fittest individual's fitness and the proportion of infeasible individuals, and print them.

2.3.1 Elitism

For every generation (only relevant after the first generation), before the marked `for` loop starts and fills up the next generation population, we carry out a procedure with which the two first offspring individuals of the next generation are produced using elitism. We search among the original population for the individuals with best fitness and once we find them, they are sent to the crossover function with probability 0, which means that they are added into the next generation without being modified. This is a very important feature in the algorithm

that ensures that the best solution of the problem does not increase its fitness from one generation to the next one, thus helping the algorithm converge.

2.3.2 Tournament selection

The method we use to select the individuals that will produce the offspring for the next generation is a tournament selection. The `tournament_selection` function is the following:

```
int tournament_selection(Individual * population,int population_size){
    int first_fighter  = floor(uniform()*population_size);
    int second_fighter = floor(uniform()*population_size);
    return population[first_fighter].fitness <
    ↪  population[second_fighter].fitness ? first_fighter : second_fighter;
}
```

When the function is called, it takes a population of individuals and creates two random integers normalized to the population size. These integers, `first_fighter` and `second_fighter` work as the index of to individuals of the population which are selected randomly. Then, we compare their fitness value and we use the ternary conditional operator to return the fittest individual's index. The individual that "won" the tournament, will be used to create the offspring together with the winner of a second selection. This method of selecting the parent individuals is an easy way to add a bias on the algorithm towards the fittest individuals, so that the ones with the lowest fitness are more likely to be retain their genes for the next generation.

2.3.3 Crossover and mutation

When generating the offspring, we use one point crossover to mix up the genes of the parents. As it is explained for instance on [6], in this type of crossover, we choose a random position of the chromosome string and cut the parents in two parts. Then the remaining tail part of one parent is moved into the other parent. The function used in the code for this procedure is:

```
void OnePointCrossover(Individual* parent1, Individual* parent2, Individual*
↪  child1, Individual* child2, double prob){
    Copy_Individual(parent1, child1);
    Copy_Individual(parent2, child2);

    if(uniform() < prob){
      unsigned char len = 8*sizeof(unsigned int);
      unsigned char d = uniform()*(len-1) + 1, di = len - d;
      int par = random_in(0,PARAMETERS_GENES_NUMBER-1);
```

```
      child1->Pars[par] = (( parent1->Pars[par] >> d) << d) | ((
      ↪  parent2->Pars[par] << di) >> di);
      child2->Pars[par] = (( parent2->Pars[par] >> d) << d) | ((
      ↪  parent1->Pars[par] << di) >> di);
    }
}
```

First, it copies each parameter from the parents into the childs. Then, a random position in the gene string is chosed, named `d`. The bitwise operators [4] that are used, are the `>>` and `<<` operators, to shift bits to the right and to the left, and the `|` operator, which is used to join the first cut section to the tail of the other parent. The crossover function also takes as an input a crossover probability, which is usually in the range $p_c \epsilon (0.6, 1)$. Generally, we set it to be $p_c = 0.75$, and $p_c = 0$ in the elitism. Crossover is the most important operation to quickly increase the search space of the algorithm.

On the other hand, we also perform mutation on the child individuals. There are many ways to apply mutation, it is possible to set a very low probability for which each one of the genes can be altered, or also set a slighly higher probability to mutate only a random gene each time. We choose the first option, and the function we implement is:

```
void Mutation(Individual* ind, double prob) {
  if(uniform() < prob){
    int i = random_in(0,PARAMETERS_GENES_NUMBER-1);
    int x;
    if (i==10 || i == 9 || i == 0)
        {x = random_in(0,pow(2,20))*random_in(0,pow(2,20));}
    else if (i == 1 || i == 4 || i == 5 || i == 6 || i == 8)
        {x = random_in(0,pow(2,20));}
    else
        {x = random_in(0,pow(2,10));}

    ind->Pars[i] ^= (1U << x);
  }
}
```

The choice of the probabilities of crossover and mutation, p_c and p_m, are crucial for the success of the algorithm. In general, p_m should be a very small number, between 0.001 and 0.2 as a strangely big case. It is better to set a dynamic probability instead of a fixed value, but whether it is better to start low in the early generations and increase in the final ones, or the other way around, is an open debate for the experts, as [8] remarks. Mutation is important to make sure that no point in the search space is left out, so it makes sense to have a higher mutation probability in the starting generations to make the algorithm more exploratory,

and avoid it at the end so that convergence is faster. But also, mutation can be key at the final generations, because the algorithm might run into a local optimum so mutation can control that the final population isn't just made of copies of a very fit but suboptimal individual. This dilemma leads to think that p_m should depend on the fitness value. In the Result section, we will explain the different methods we experimented with and show the results obtained.

2.4 Model additions

As we will discuss in the results section, genetic algorithm is not able to solve this problem by itself, giving incorrect results and falling into local minima. For this reason, we attempt to implement a hybrid model which complements the GA with a different optimization method. The basic idea is to apply the complementary optimization function at the end of the offspring, so that the fitness of the new generation individuals is boosted and the solution is reached faster and more easily.

2.4.1 Conjugate gradient descent

As we show in Figure 2, in this section we describe a code that implements conjugate gradient descent at the end of the creation of the generation offspring to improve the quality of the individuals.

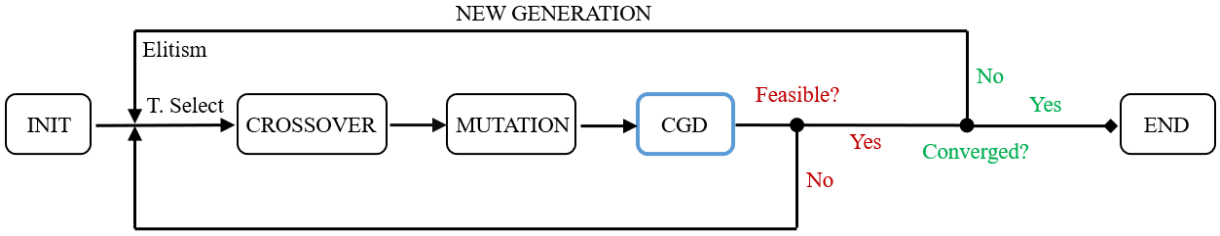

Figure 2: Overall scheme of the main algorithm of an hybrid model that complements GA with conjugate gradient descent.

As in any CGD algorithm, every iteration, the test solution is updated following the expression:

$$x_{\mathrm{k}+1} = x_{\mathrm{k}} + \alpha_{\mathrm{k}} d_{\mathrm{k}}, \tag{2.4.1}$$

where d_{k} is a descent direction and α_k is the step size, selected with a line search backtracker function. The gradient direction is also updated every iteration as follows:

$$d_k = -\nabla f(x_k) + \beta_k d_{k-1}, \tag{2.4.2}$$

where $\nabla f(x_k)$ is the gradient of the heuristic function, and β_k is a parameter that is chosen such that the directions are mutually orthogonal to their scalar product. We will use the Polak-Ribière formula, and compute:

$$\beta_k = max\left(0, \beta^{PR}\right) = max\left(0, \frac{\nabla f(x_{k+1})^T \left(\nabla f(x_{k+1}) - \nabla f(x_k)\right)}{\nabla f(x_k)^T \nabla f(x_k)}\right).$$

In this problem, we have already stated that the function we are trying to optimize is the fitness 1.2.4, which as we know, is technically not a function of the ODE parameters, but of the predicted values of the model variables. Still, the test solution x_k, is the vector (or structure) of the ODE parameters, so again, we have to imagine that the fitness function $f(x_k)$ is a function that takes the parameters and computes the fitness. Considering this, computing the gradient of this function is not obvious, as there is no analytical function that depends on the parameters. What we do is compute the numerical derivative, using forward difference, as to avoid negative values of the parameters, calculating each of the components of the gradient with:

$$\frac{\partial f(x)}{\partial x_i} \simeq \frac{f(x + h_i) - f(x)}{h}. \tag{2.4.3}$$

In order to calculate $f(x+h_i)$ we compute the fitness with the ODE parameters x, but adding an h $<<$ increment to the i^{th} parameter. The code we use to compute the gradient is:

```
void gradient(Individual *ind, void *TheData, double grad[]){
  Individual *aux_ind;
  if ((aux_ind = (Individual*) malloc(sizeof(Individual))) == NULL)
  ↪  Exit_Error("When allocating memory for individual", 7);
  DataForFitting *TDfF = (DataForFitting *) TheData;
  double h = 0.000001;
  int is_feasible = 0;


  for (int par = 0;par<PARAMETERS_GENES_NUMBER;par++){
    Copy_Individual(ind,aux_ind);
    ODE_Parameters ODE_pars = {crom2HSPar(aux_ind->Pars[0]),
    ↪  crom2Par(aux_ind->Pars[1]), crom2LSPar(aux_ind->Pars[2]),
    ↪  crom2LSPar(aux_ind->Pars[3]), crom2Par(aux_ind->Pars[4]),
    ↪  crom2Par(aux_ind->Pars[5]), crom2Par(aux_ind->Pars[6]),
    ↪  crom2LSPar(aux_ind->Pars[7]), crom2Par(aux_ind->Pars[8]),
    ↪  crom2HSPar(aux_ind->Pars[9]), crom2HSPar(aux_ind->Pars[10]),
    ↪  TDfF->PopSize};

    if (par == 0)       ODE_pars.beta += h;
    else if (par == 1)  ODE_pars.phi += h;
    else if (par == 2)  ODE_pars.epsI += h;
    else if (par == 3)  ODE_pars.epsY += h;
    else if (par == 4)  ODE_pars.sigma += h;
    else if (par == 5)  ODE_pars.gamma1 += h;
    else if (par == 6)  ODE_pars.gamma2 += h;
    else if (par == 7)  ODE_pars.kappa += h;
    else if (par == 8)  ODE_pars.p += h;
```

```
    else if (par == 9)  ODE_pars.alpha += h;
    else if (par == 10) ODE_pars.delta += h;


    is_feasible = evaluate(aux_ind, TheData, ODE_pars);
    grad[par] = (ind->fitness - aux_ind->fitness)/(h);
  }
}
```

Inside the `CGDescent` function, we use the `evaluate` function to calculate the fitness of a given individual and parameters, and the function `alpha_search` to backtrack the optimal value of the step size parameter. With this, the code for the main function of the descend is:

```
void CGDescent(Individual *ind, void *TheData){
  Individual *aux_ind;
  if ((aux_ind = (Individual*) malloc(sizeof(Individual))) == NULL)
  ↪  Exit_Error("When allocating memory in CGDescent", 11);
  Copy_Individual(ind,aux_ind);

  double beta, alpha = 1;
  double grad[PARAMETERS_GENES_NUMBER], prev_grad[PARAMETERS_GENES_NUMBER],
         d[PARAMETERS_GENES_NUMBER], s[PARAMETERS_GENES_NUMBER];

  gradient(ind,TheData,grad);


  for (int par = 0;par<PARAMETERS_GENES_NUMBER;par++){ d[par] = -grad[par];}
  alpha = alpha_search(ind, TheData, d);
  for (int par = 0;par<PARAMETERS_GENES_NUMBER;par++){
    ind->Pars[par] += alpha*d[par]; s[par] = d[par];}

  int iter_max = 10;
  for (int iter = 0; iter < iter_max; iter++){
    for (int par = 0;par<PARAMETERS_GENES_NUMBER;par++){
        prev_grad[par] = grad[par];}
    gradient(ind, TheData,grad);
    beta = get_beta(grad,prev_grad);
    for (int par = 0;par<PARAMETERS_GENES_NUMBER;par++){
        s[par] = d[par] + beta*s[par];}
    alpha = alpha_search(ind, TheData, d);
    ind->Pars[0]  += alpha*Par2HSDiscPar(s[0]);
    ind->Pars[1]  += alpha*Par2DiscPar(s[1]);
    ind->Pars[2]  += alpha*Par2LSDiscPar(s[2]);
    ind->Pars[3]  += alpha*Par2HSDiscPar(s[3]);
    ind->Pars[4]  += alpha*Par2DiscPar(s[4]);
    ind->Pars[5]  += alpha*Par2DiscPar(s[5]);
    ind->Pars[6]  += alpha*Par2DiscPar(s[6]);
    ind->Pars[7]  += alpha*Par2LSDiscPar(s[7]);
    ind->Pars[8]  += alpha*Par2DiscPar(s[8]);
```

```
    ind->Pars[9]  += alpha*Par2HSDiscPar(s[9]);
    ind->Pars[10] += alpha*Par2HSDiscPar(s[10]);
  }
}
```

2.4.2 Simulated annealing

Inspired in annealing in metals, simulated annealing is an algorithm for finding a global optimum for a given function. In our case we want to minimize the fitness function described in 1.2.4. Simulated annealing takes an individual and its parameters and makes a perturbation on them through coordinates search optimization. Then it measures the fitness of this perturbed individual. The algorithm accepts this perturbation if the fitness is reduced or it rejects the perturbation on the contrary. In order to avoid falling into local minimum, the algorithm accepts with some probability that depends on the current fitness value a 'bad' perturbation that increases the fitness. This process is repeated a fixed number of times with the philosophy that, with a large enough time, the algorithm will find a global solution for the problem.

In order to implement this simulated annealing algorithm, we create the `SA` function that takes every individual of each new population and tries to improve its fitness by at most a 70%. However, and in order to not enter in an infinite loop, the maximum number of iterations the full simulated annealing algorithm is done for each individual is 10 times. This number is so low because of computational issues, a thing that will be discussed later. Besides, the probability for accepting a 'bad' step for trying to escape a local minima is in an exponential form depending on the current perturbation fitness as documented in [9]. Below is described the code and the specifications of the simulated annealing algorithm.

```
void SA(Individual *ind, void *TheData){
  Individual *aux_ind;
  if ((aux_ind = (Individual*) malloc(sizeof(Individual))) == NULL)
  ↪  Exit_Error("When allocating memory in SA", 12);
  DataForFitting *TDfF = (DataForFitting *) TheData;

  double accept_perturbation;
  double goal = ind->fitness*7/10;
  int iters = 0, iters_max = 10;

  while (ind->fitness > goal && iters < iters_max){
      for (int par = 0; par<PARAMETERS_GENES_NUMBER;par++){
          Copy_Individual(ind,aux_ind);

          ODE_Parameters ODE_aux_pars = { crom2HSPar(aux_ind->Pars[0]),
          crom2Par(aux_ind->Pars[1]),   crom2LSPar(aux_ind->Pars[2]),
          crom2LSPar(aux_ind->Pars[3]), crom2Par(aux_ind->Pars[4]),
```

```
            crom2Par(aux_ind->Pars[5]),   crom2Par(aux_ind->Pars[6]),
            crom2LSPar(aux_ind->Pars[7]), crom2Par(aux_ind->Pars[8]),
            crom2HSPar(aux_ind->Pars[9]), crom2HSPar(aux_ind->Pars[10]),
            TDfF->PopSize };

            if      (par == 0)  ODE_aux_pars.beta   += random_in(-1,1)*uniform();
            else if (par == 1)  ODE_aux_pars.phi    += random_in(-1,1)*uniform();
            else if (par == 2)  ODE_aux_pars.epsI   += random_in(-1,1)*uniform();
            else if (par == 3)  ODE_aux_pars.epsY   += random_in(-1,1)*uniform();
            else if (par == 4)  ODE_aux_pars.sigma  += random_in(-1,1)*uniform();
            else if (par == 5)  ODE_aux_pars.gamma1 += random_in(-1,1)*uniform();
            else if (par == 6)  ODE_aux_pars.gamma2 += random_in(-1,1)*uniform();
            else if (par == 7)  ODE_aux_pars.kappa  += random_in(-1,1)*uniform();
            else if (par == 8)  ODE_aux_pars.p      += random_in(-1,1)*uniform();
            else if (par == 9)  ODE_aux_pars.alpha  += random_in(-1,1)*uniform();
            else if (par == 10) ODE_aux_pars.delta  += random_in(-1,1)*uniform();

            evaluate(aux_ind, TheData, ODE_aux_pars);

            accept_perturbation =
              pow(euler, -(aux_ind->fitness - ind->fitness)/ind->fitness);

            if ((aux_ind->fitness < infeasible_value) && ((aux_ind->fitness <
            ↪  ind->fitness) || (uniform() < accept_perturbation) )){
              Copy_Individual(aux_ind,ind);
            }
        }
        iters++;
    }
    free(aux_ind);
}
```

One of the features of this algorithm to be forementioned are the perturbations in each individual. This perturbations consists in a 1D-random walk with a random step of max distance 1 in the positive or negative direction. This step is fixed very carefully because of the fragility when changing parameters for the Runge-Kutta algorithm.

The hybrid implementation of GA with SA is equivalent in terms of structure to the one we showed in Figure 2, just substituting the `CGDescent` function with the `SA` one.

2.5 Running the code

In order to run the code just execute the following command:

```
gcc -g -Wall -O3 -o main main.c my_functions.c RKF78.c read_data.c -lm
```

It is very important to keep in mind that the `Genetic_data.txt` which contains the table of real variable values, must be in the same folder as the `.c` files. Once this command is executed the program starts asking the desired number of individuals in each iteration and the maximum number of generations in case the algorithm doesn't find a solution earlier. The output of the algorithm will be printed in a txt named `output_genetic` that contains information about each generation: fittest individual, mean fitness of the feasible individuals and proportion of infeasible individuals. When the algorithm ends it informs about the time and final results. We give an example below.

```
------------- Generation Number 50 -------------

Fittest individual fitness value                    = 3596381695036.875000
Mean fitness of the feasibles in population         = 35274830821473.500000
Proportion of infeasibles individuals               = 0.120000
```

In the case where the algorithm reaches a solution it gives the parameters for the differential equation.

3 Results

In this section, we will explain the different tests and attempts we made to study the behaviour of the code. First, we will start with the basic GA implementation, discussing the importance of population size and the mutation rate, and then we will explain the results of the hybrid implementation with CGD and SA. Different sizes of populations have been tested for this problem in order to see how dependent the algorithm is to the number of individuals in the population. In Figure 3 are shown the results for population sizes of 10, 32, 50 and 200 until 50 generations long.

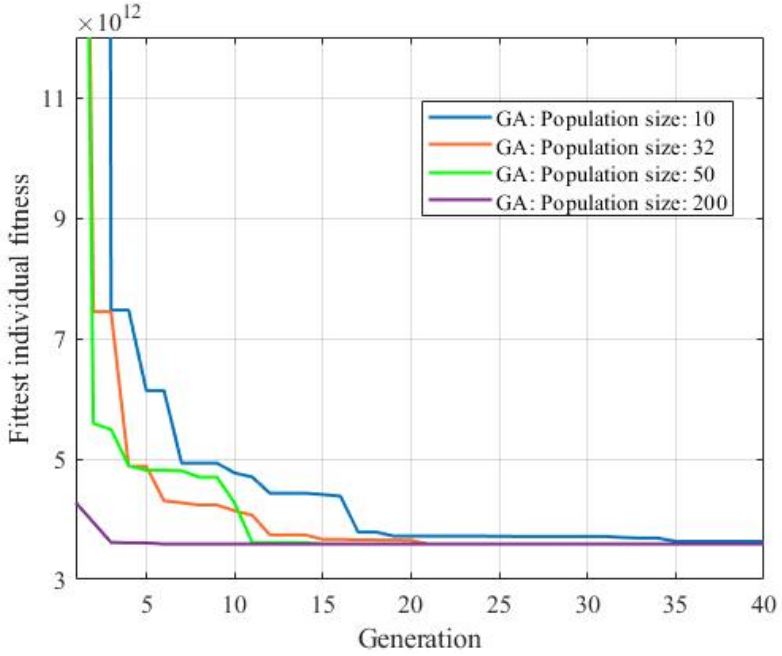

Figure 3: Results of the basic implementation of the GA. We show the fittest individual's fitness value over the generations, and for different sizes of the population. As it is expected, the algorithm converges faster with bigger population sizes, as the search space is also bigger (more solutions are proposed every generation). In all cases, the algorithm converges to what seems a local minima.

As it can be seen, all of these different experiments end up reaching a relatively low fitness value: they all go down into a value of approximately $f(x_k) = 3.5 \cdot 10^{12}$. Assuming there exists a solution for the parameters of the differential equation that gives a 0 or nearby value, it seems that our algorithm reaches and gets stuck into a local minima. Neither the high-probability mutation, nor crossovers between individuals, nor larger number of generations accomplish a better result. Which could be the reasons for this 'bad' solution the algorithm gives? We have multiple possible answers for this questions. The first one and more simple (and also less likely) is that this really might be the best solution of a global minima. On the other hand, the second possibility is that the algorithm doesn't escape from the local minima due to the fragility of the Runge-Kutta method, which is extremely

sensitive to variances in the input parameters and returns infeasible individuals that cannot contribute to improve the generation for a better fitness value. The fact that after tunning the mutation and crossover rate, generation size, and other key hyperparameters of the model, the algorithm still fails, seems to indicate that there is a problem with the way that we define the fitness and how we evaluate it. A good GA needs to have an effective method to calculate the fitness and to make mutations on the individuals to cover a wide search space. In our case, it seems that the calculation of the fitness (Runge-Kutta function) is not as regular as it should be. Finally, the third answer is that maybe this algorithm cannot reach any better solution by itself so it needs the help from other optimization algorithms. The results for this last answer are studied later.

Despite the fact that the algorithm does not reach the desired 0 fitness value, it works in some ways. With the help of the elitism idea it maintains or improves its fittest individual and seems to be very quick to reach the local minima: it only needs 10 to 15 generations to reach it when having a reasonable number of individuals. For a extremely large of them the algorithm turns into a random search algorithm, not the subject of this work. And for a small size of population it is seen in Figure 3 that it takes more iterations for the algorithm to reach the local minima but finally ends in it.

Finally, for the population sizes tested above, the computational times for 10, 32, 50 and 200 respectively are: 12.8 seconds, 0.0256 seconds per individual; 56.82 seconds, 0.035 seconds per individual; 40 seconds, 0.016 seconds per individual; and 279 seconds, 0.0279 seconds per individual.

3.1 Mutation probability

One of the most important choices for the algorithm is how to set the probability of mutation, p_m. As we explained in previous sections, experts [8] still ponder over whether the probability should increase as the generations go on, or decrease, agreeing on the fact that the probability should always be dynamic. Decreasing dynamic probability will help the convergence of the algorithm, while maintaining the exploratory character of the algorithm in the early stages. Meanwhile, increasing dynamic probability is helpful to make sure that in the last stages of the algorithm, the method does not get stuck into local optima, as it keeps producing individuals that are slightly different to a suboptimal fittest. Nevertheless, what seems to be a common knowledge, is that mutation probability should be kept in the range: $p_m \epsilon [0, 0.2]$, with 0.2 being an unusually big value. We runned the code for the simple GA implementation for three different expressions of p_m:

1. **Dynamic decreasing probability**: The mutation rate depends on the fitness $f(x_k)$ of the individual, being higher the bigger the fitness is. To achieve this we add the Elliot sigmoid function, that is soft enough. We add the value 10^{12} in the denominator because it is of the order of magnitude of the converging fitness and because we have the previous knowledge about the non-convergence to zero values for the fitness. If this genetic algorithm gave a final fitness value of 0, the factor of the Elliot sigmoid function would

be 10^6. The philosophy is to keep the mutation rate high in the early stages, to make the algorithm very exploratory, and then decrease it as the fitness starts converging. Once the fitness goes below this order of magnitude, p_m goes to 0, which might help the algorithm converge.

$$p_m = \frac{f(x_k)}{f(x_k) + 10^{12}}. \tag{3.1.1}$$

2. **Dynamic increasing probability**: The mutation range depends on the generation number the individual is part of, g, and g_{max} is the constant maximum number of generations we choose to run. The number a and b control the range in which the probability changes.

$$p_m = a + b\left(\frac{g - g_{max}}{g_{max}}\right). \tag{3.1.2}$$

   An increasing probability makes sense because mutation can play a big role expanding the search space on the late stages of the algorithm, where the population might have fallen into a local minimum.

In Figure 4 we show the evolution of the fittest individual's fitness value along the generation, fixing $g_{max} = 50$ and a population size of 50. In blue, the case with decreasing mutation rate given by Equation 3.1.1, and in orange and green with increasing mutation rate given by 2, with parameters $(a, b) = (0.55, 0.35)$ and $(0.2, 0.2)$ respectively.

It is hard to evaluate and compare the three methods, specially when the problem still converges to the wrong local minimum, and mutation is an operation that involves randomness, but by looking at the results, it seems like choosing a probability based on the fitness value is a reasonable method, and also, that high values of p_m help the model converge faster. This last part is not usually a common trait for GAs, as the probability is kept very close to 0, but in our case, it seems like, as the algorithm can't really find an optimal solution, high values of the mutation rate do not hinder the results.

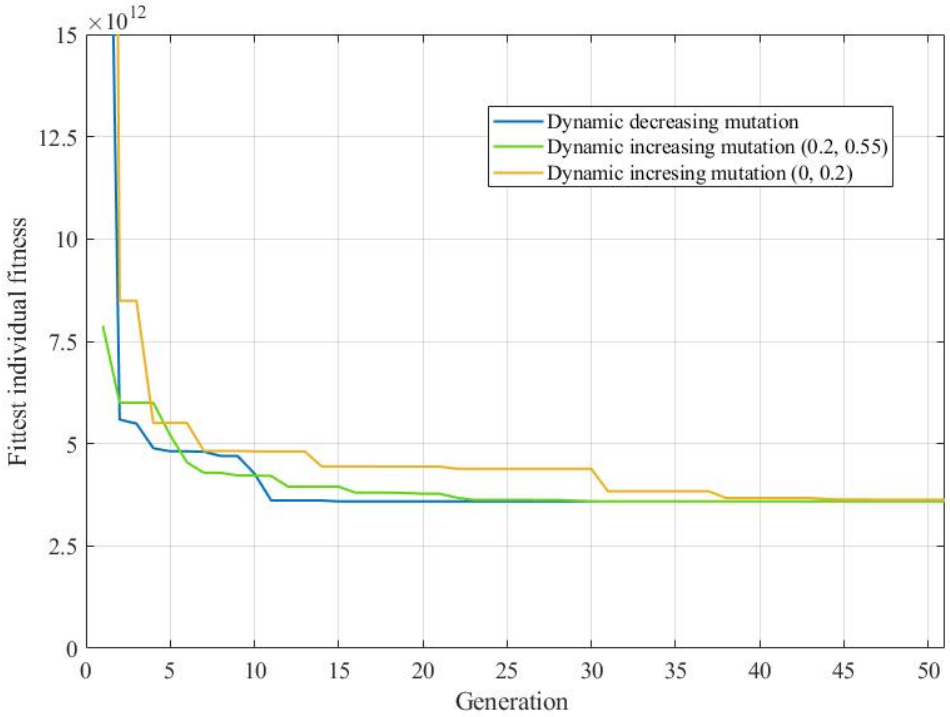

Figure 4: Evolution of the fittest individual fitness value over 50 generations, for different models of the mutation probability, as explained in Section 3.1.

3.2 Hybrid methods results

The bad results from the basic implementation of the genetic algorithm motivated us to try an hybrid model, using the codes that were described on Section 2.4. The results and the take-aways from the usage of CGD and SA can be summarized as follows:

- **Conjugate Gradient Descent**: This powerful optimization method was added to the code with the expectation that it would be able to improve the fitness on each execution so that the solution would be pushed away from the local minima. Although, we have found that even though it sounds great on paper, it is very difficult to implement due to the fact that on every step, we need to calculate the gradient numerically, and then change the parameters using a descent direction. When we apply this update on the parameters, and use them to calculate the new fitness, the sensitiveness of the Runge-Kutta function makes the individuals infeasible and therefore it is hard to find a step that improves the fitness. Also, the calculation of the fitness is very sensitive to small changes on even just one of the parameters, so the gradient components can be so big that the descent direction doesn't ever improve the heuristic. All of this makes the CGD implementation very computationally expensive, and it doesn't achieve any improvements on the basic GA.

- **Simulated annealing**: Simulated annealing is also expected to improve the fitness after every execution, but this time, we don't have to worry about computing any derivatives, and that's why we implemented the method. As we show in Figure 5, the SA function works, improving the fitness a lot after each individual is created, which makes the algorithm very fast to converge. By the scale it might seem that the SA fittest individual value curve is constant, but it isn't, it is decreasing but orders of magnitude lower than the basic GA implementation. As for the mean fitness values, we see the expected peaks due to mutations specially in the early stages, but it is generally also lower in the SA implementation.

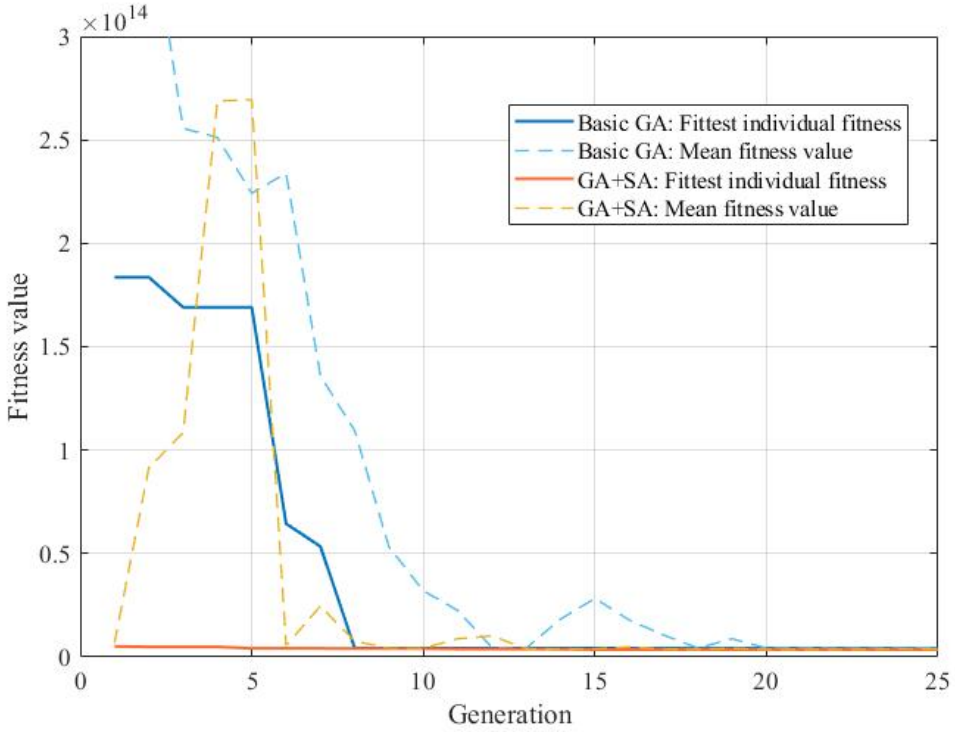

Figure 5: Comparison of the evolution of the fittest individual' fitness value and the mean fitness, for the basic GA implementation and the SA hybrid model.

Even though simulated annealing helps improve the convergence, it does not change the overall result, as the local minima is not avoided. But doing an overall analysis SA with GA algorithm takes a huge amount of time: 495 seconds, 1.17s per individual; while GA takes only 5.5 seconds, 0.013s per individual, a huge difference, this is why we only run the code with SA for population size of 14 individuals and 30 generations. This problem will be discussed in the conclusions below.

4 Conclusions

In this project a genetic algorithm has been developed, tested and modified in order to try solving the problem of finding the seed individual that started the COVID-19 through a the SIR model. A set of functions and functionalities have been worked out for some possible characteristics of the genetic algorithm: elitism, tournament selection, mutation, crossover and different sizes of populations and generations while the population was controlled to avoid infeasible individuals. We tested our genetic algorithm and found that the algorithm does not seem to converge into a 0 fitness value but into what appears to be a local minima. In order to avoid this local minima we tried a hybrid model mixing genetic algorithm with conjugate gradient descent or simulated annealing and we discovered that due to the delicacy of the Runge-Kutta method, the CGD did not work, neither the SA due to memory and computational time issues.

However in this work we have explored, tested and discused showing up and comparing a considerable amount of results gathering the possibilities that a genetic algorithm could take into account, not only for this particular problem but for many of them.

References

[1] Alsedà, L. (2021). Optimisation. Master's degree in Modlling for Science and Engineering. http://mat.uab.cat/~alseda/MasterOpt/index.html. [Online; accesed 12-February-2021].

[2] Alsedà, L. (2021). Optimisation. Genetic algorithms: A possible model for COVID-19. https://mat.uab.cat/~alseda/MasterOpt/GA-COVID-CM.pdf. [Online; accessed 12-February-2021].

[3] Dziedzic, J. (2009). Introduction to genetic algorithms and their applications. https://mat.uab.cat/~alseda/MasterOpt/Dziedzic.GA_intro.pdf. [Online; accessed 12-February-2021].

[4] Summit, S. (1999). Bitwise operators. https://www.eskimo.com/~scs/cclass/int/sx4ab.html. [Online; accessed 12-February-2021].

[5] Katoch, S., Chauhan, S.S. and Kumar, V. A review on genetic algorithm: past, present, and future. Multimed Tools Appl (2020). https://doi.org/10.1007/s11042-020-10139-6

[6] Beasley, David, Bull, David R. and Martin, Ralph Robert (1993). An overview of genetic algorithms: Part 1, fundamentals. University Computing 15 (2) , pp. 56-69. http://orca.cf.ac.uk/id/eprint/64436

[7] Beasley, David, Bull, David R. and Martin, Ralph Robert (1993). An overview of genetic algorithms: Part 2, research topics. University Computing 15 (4) , pp. 170-181. http://orca.cf.ac.uk/id/eprint/64433

[8] R.N. Greenwell, J.E. Angus, M. Finck (1995). Optimal mutation probability for genetic algorithms. Mathematical and Computer Modelling, Volume 21, Issue 8. Pages 1-11, ISSN 0895-7177, https://doi.org/10.1016/0895-7177(95)00035-Z.

[9] Akella, P. Simulated Annealing. https://mat.uab.cat/~alseda/MasterOpt/ComprehensiveSimulatedAnnealing.pdf