GENETIC ALGORITHM OPTIMIZATION

Fitting a possible COVID-19 model

Juan Carlos Barroso Ruiz 1388269

Oriol Fernández Serracanta 1426251

February 16, 2021

1 Introduction

The goal of this project is to use a genetic algorithm to fit a set of parameters defining the temporal evolution of a COVID-19 pandemic. This is done by analyzing which of the different parameters combinations fits better the temporal series of different magnitudes we are provided with.
Of course, the way in which these objective-parameters are related to the temporal series of the magnitudes is by a system of differential equations, which we also know. The differential equation system is what is called a deterministic SEIR model in epidemiology. It is deterministic since the differential equations involved have no stochastic term in them.

$$\begin{aligned}
\frac{dS}{dt} &= -\lambda(t)S(t) \\
\frac{dE}{dt} &= \lambda(t)S(t) - \sigma E(t) \\
\frac{dI_1}{dt} &= \sigma E(t) - \gamma_1 I_1(t) \\
\frac{dA}{dt} &= \gamma_1(1-p)I_1(t) - (\kappa+\gamma_2)A(t) \\
\frac{dA_d}{dt} &= \kappa A(t) - \gamma_2 A_d(t) \\
\frac{dI_2}{dt} &= \gamma_1 p_1 I_1(t) - (\alpha+\gamma_2)I_2(t) \\
\frac{dY}{dt} &= \alpha I_2(t) - (\delta+\gamma_2)Y(t) \\
\frac{dR}{dt} &= \gamma_2(A(t) + A_d(t) + I_2(t) + Y(t))
\end{aligned} \tag{1}$$

We are then provided with the temporal evolution of the mean of the variables: A_d, I_2, Y, R and D. This data is used to asses whether a certain combination of parameters is likely to produce the expected temporal series, that is, whether the solution to that particular ODE system is similar to the one described by the temporal series. The way to asses that is to quantify the level of likelihood by a parameter called "fitness". Intuitively what we want is that the curve drawn by a given variable is similar to the one obtained after solving the ODE system with a particular choice of parameters. Now, the set of parameters we are asked to fix the value of are the following:

$$(E(0), I_1(0), A(0)) \quad \text{and} \quad (\beta, \phi, \epsilon_I, \epsilon_Y, \sigma, \gamma_1, \gamma_2, \kappa, p, \alpha, \delta)$$

If we are successful, we will be able to get the best choice of parameters in order to reproduce the temporal data provided initially.

2 The algorithm

2.1 Qualitative explanation of the algorithm

As we highlighted before the idea is to implement a genetic algorithm to this problem which will output the best possible choice of parameters for our model given the data. The idea behind such algorithm is as follows. We first generate a random set of possible choices for the parameters. These possible choices are considered to be individuals in a population, each representing a particular choice of parameters. Then, this individuals can mix among each other and give rise to a new generation of possible parameter's choices. However, we restrict the maximum number of individuals in a population, furthermore, we also restrict those who are eligible to become part of the new generation by evaluating its fitness to the data. This process is performed iteratively until some individual fulfills the desired threshold for the fitness (which is up to us).
All in all the process can be summed up in the following steps:

1. Initialize a population randomly. That is, select different set of choices for the parameters at random.
2. By evaluating its fitness to the data, select those who will mate and give rise to a new generation. Also allow for mutations.
3. Evaluate the fitness and select the first N individuals with higher fitness, where N is the maximum size allowed for the population.
4. Check for the fitness threshold. If met, end the algorithm and return the individual with the highest fitness. Else, return to step 1.

2.2 Implementation of the algorithm in C language

First off, we must code what will be an individual. Following the biological idea that an individual is represented by its genome, this is the way an individual is represented in our code

```
typedef struct Genome {
  unsigned long c1[GENES_C1];
  unsigned long c2[GENES_C2];
  double fitness;
} Genome;
```

The way to initialize this individual is the following

```
void generate_genome(Genome * genome) {
  //generating initial states for chromosome 1
  genome -> c1[0] = random_int(1000000000);
  genome -> c1[1] = random_int(1000000000);
  genome -> c1[2] = random_int(1000000000);
  //generating initial states for chromosome 2
  genome -> c2[0] = random_int(4096);
  genome -> c2[1] = random_int(512);
  genome -> c2[2] = random_int(64);
  genome -> c2[3] = random_int(64);
  genome -> c2[4] = random_int(512);
  genome -> c2[5] = random_int(512);
  genome -> c2[6] = random_int(512);
  genome -> c2[7] = random_int(64);
  genome -> c2[8] = random_int(512);
  genome -> c2[9] = random_int(4096);
  genome -> c2[10] = random_int(4096);

  genome->fitness = -1;
}
```

where `random_int` is a function generating a random integer up until the number specified inside the parenthesis. The way to translate the number assigned to each position of the chromosome from `unsigned long` to `double` is by using the following function

```
#define crom2IC(c) (((double) (c % 1000000000UL))/1000.0) // Initial conditions
#define crom2HSPar(c) (((double) (c % 1099511627776UL))/1099511627776.0) // High sensitivity
#define crom2Par(c) (((double) (c % 1048576U))/1048576.0) // Medium sensitivity
#define crom2LSPar(c) (((double) (c % 1024U))/1024.0) // Low sensitivity
//...
void genotype_to_phenotype(Genome * genome, double * c1, Parameters * c2) {
  c1[1] = crom2IC(genome -> c1[0]);
  c1[2] = crom2IC(genome -> c1[1]);
  c1[3] = crom2IC(genome -> c1[2]);
  c1[4] = DATA[0][0];
  c1[5] = DATA[0][1];
  c1[6] = DATA[0][2];
  c1[7] = DATA[0][3];
  c1[0] = POP_SIZE - (c1[1] + c1[2] + c1[3] + c1[4] + c1[5] + c1[6]);

  c2 -> beta = crom2HSPar(genome -> c2[0]);
  c2 -> phi = crom2Par(genome -> c2[1]);
  c2 -> e1 = crom2LSPar(genome -> c2[2]);
  c2 -> eY = crom2LSPar(genome -> c2[3]);
  c2 -> sigma = crom2Par(genome -> c2[4]);
  c2 -> gamma1 = crom2Par(genome -> c2[5]);
  c2 -> gamma2 = crom2Par(genome -> c2[6]);
  c2 -> kappa = crom2LSPar(genome -> c2[7]);
  c2 -> p = crom2Par(genome -> c2[8]);
  c2 -> alpha = crom2HSPar(genome -> c2[9]);
  c2 -> delta = crom2HSPar(genome -> c2[10]);
}
```

Then, given an initial population of individuals generated randomly, that is, their parameters are initialized randomly, the following loop is executed until the threshold for the fitness is met or either until a maximum number of iterations is reached

```
while (iter < maxiter) {
    // fitness calculates the fitness of every guy in the population
    #pragma omp parallel for
        for (i = 0; i < individuals; i++) {// exclude those simulation of repeated genes to speed up simulation!
            if (population[i].fitness < 0) compute_fitness(population + i, ff); // TODO: parallel
    }

    if (iter % 200 == 0) {
      #pragma omp parallel for
      for (i = 0; i < individuals; i ++)
        if (population[i].fitness > 0) optimise_parameters(population + i, ff);
      // copy_genome(population + best_individual, temp_population);
    }

    mutation_bit = 1 + (int) (UL_SIZE - 1) * ((1.0 - ((float)iter) / ((float)maxiter)));

    if (recovery < cooldown) {
        best_individual = next_generation(population, temp_population,
                                          number_survivors, extinc_selection,
    extinc_cross, extinc_migration,
                      0.25, mutation_bit);
        recovery++;
        //if(iter % (maxiter/100) == 0)printf("Entering cooldown if\n");
        fitness_temp=population[best_individual].fitness;

      } else {
            int rdn=random_int(extinction_period);
            if (rdn < ek) {
              recovery=0;
          change_seed();
              ek = extinction( ek, population, temp_population, individuals,
    number_survivors);
```

```
31              printf("An extinction has occurred\n");
32            } else {
33          best_individual = next_generation(population, temp_population,
34                                                  number_elitism, number_selection,
    number_crossover, number_migration,
35                          0.49, mutation_bit);
36          //if(iter % (maxiter/100) == 0)printf("normal behaviour, values %.8f,%.8f\n",
    population[best_individual].fitness,fitness_temp);
37
38          if ((abs(population[best_individual].fitness -fitness_temp) < epsilon )
39            ||((population[best_individual].fitness -fitness_temp)==0)) ek++;
40          fitness_temp=population[best_individual].fitness;
41        }
42      }
43
44 /*
45     best_individual = next_generation(population, temp_population,
46                     number_elitism, number_selection, number_crossover, number_migration,
47                     0.1, mutation_bit);
48 */
49     // mutation_bit = UL_SIZE;
50     if (iter % (maxiter/100) == 0)
51       printf("Generation %d with fitness %.8f\n", iter, population[best_individual].fitness)
    ;
52
53         // exchange pointers of parents and children populations
54         Genome * tmp = population;
55         population = temp_population;
56         temp_population = tmp;
57     ++iter;
58   }
```

To begin, we compute the fitness for each individual. This is done by first solving the ODE system with the Runge Kutta algorithm provided to us. Given the output trajectory fitness is computed by the `fitness_exp` function which is passed as an argument in line 4.

```
1 double fitness_exp(double ** solution) {
2   int i, j;
3   double f = 0.0;
4   for (i = 1; i < DAYS; i++)
5     for (j = 0; j < 5; j++)
6       f += exp(EXP_NU * i) * gsl_pow_uint(solution[i][j] - DATA[i][j], 2);
7
8   return f;
9 }
```

Where the `DATA` structure, contains the original data series of the different magnitudes. Once the fitness for each individual is computed (and stored inside the `Genome` structure) we are ready to create the next generation, which will involve selecting the best individuals for mating, mating, allow for mutations and finally, select the individuals who will become part of the new generation.
This is what the `next_generation` function looks like:

```
1 int next_generation(
2   Genome * parents, Genome * children,
3   int n_elitism, int n_select, int n_cross, int n_new, double p_mutation, int mutation_bit
4 ) {
5   int pop_size = n_elitism + n_select + n_cross + n_new;
6   int i;
7
8   int best_individual;
9   if (n_elitism > 0) // elitism takes the x bests and puts them to the new generation
10    best_individual = elitist_casting(parents, pop_size, n_select, n_elitism, children);
11  else // casting selects the best individuals randomly by fitness
12    best_individual = casting(parents, pop_size, n_select, children + n_elitism);
13
```

```
  // crossover crosses the survivals to get new better individuals
  for (i = 0; i < n_cross; i++) mating(children, n_elitism + n_select, children + n_elitism
    + n_select + i);

  // mutations mutate a bit some people so a bit of randomness is included
  // exclude elitist from mutation! this is rather artificial
  if (p_mutation > 0)
    for (i = n_elitism; i < pop_size - n_new; i++) mutate_genome(children + i, p_mutation);

  if (n_new > 0) migration(children + (pop_size - n_new), n_new);

  return best_individual;
}
```

As it can be seen we select the best candidates of the current generation to be part of the new generation via the `elitism` function, then we select randomly the best individuals by fitness with `casting` and finally they mate using the `mating` function. It is inside this function, that genome crossovers and bitwise crossovers are performed. In the end, we also allow for some mutations to happen using the `mutate_genome` function. Last but not least, we also add some new randomly generated individuals to the next generation using the function called `migration`.

The particular implementation of all "genetic" operators, including those bitwise is the following:

```
void mutate_genome(Genome * genome, double prob) {
  if (scaled_mutation(genome->c1, prob, 30)
    + scaled_mutation(genome->c1 + 1, prob, 30)
    + scaled_mutation(genome->c1 + 2, prob, 30)
    + scaled_mutation(genome->c2, prob, 40)
    + scaled_mutation(genome->c2 + 1, prob, 20)
    + scaled_mutation(genome->c2 + 2, prob, 10)
    + scaled_mutation(genome->c2 + 3, prob, 10)
    + scaled_mutation(genome->c2 + 4, prob, 20)
    + scaled_mutation(genome->c2 + 5, prob, 20)
    + scaled_mutation(genome->c2 + 6, prob, 20)
    + scaled_mutation(genome->c2 + 7, prob, 10)
    + scaled_mutation(genome->c2 + 8, prob, 20)
    + scaled_mutation(genome->c2 + 9, prob, 40)
    + scaled_mutation(genome->c2 + 10, prob, 40)) // gen `i` has been mutate, so reset
    fitness to default
      genome->fitness = -1;
}

void scaled_mutate_genome(Genome * genome, double prob, int max_bit) {
  if (scaled_mutation(genome->c1, prob, max_bit)
    + scaled_mutation(genome->c1 + 1, prob, max_bit)
    + scaled_mutation(genome->c1 + 2, prob, max_bit)
    + scaled_mutation(genome->c2, prob, max_bit)
    + scaled_mutation(genome->c2 + 1, prob, max_bit)
    + scaled_mutation(genome->c2 + 2, prob, max_bit)
    + scaled_mutation(genome->c2 + 3, prob, max_bit)
    + scaled_mutation(genome->c2 + 4, prob, max_bit)
    + scaled_mutation(genome->c2 + 5, prob, max_bit)
    + scaled_mutation(genome->c2 + 6, prob, max_bit)
    + scaled_mutation(genome->c2 + 7, prob, max_bit)
    + scaled_mutation(genome->c2 + 8, prob, max_bit)
    + scaled_mutation(genome->c2 + 9, prob, max_bit)
    + scaled_mutation(genome->c2 + 10, prob, max_bit)) // gen `i` has been mutate, so reset
    fitness to default
      genome->fitness = -1;
}


void bitwise_crossover_uniform(
  unsigned int p1, unsigned int p2,
  unsigned int *f1, unsigned int *f2,
```

```
  double prob
) {
  unsigned char len = 8*sizeof(*f1);
  unsigned int mask = 0U;
  register unsigned char i;
  for(i=0; i < len; i++) if(random_double() < prob) mask = mask | (1U << i);
  *f1 = (p1 & mask) | (p2 & ~mask);
  *f2 = (p2 & mask) | (p1 & ~mask);
}


void crossover_genomes(
  Genome * gen1in,
  Genome * gen2in,
  Genome * out
) {
  // cross initial conditions
  int val = random_int(GENES_C1 + GENES_C2 - 1) + 1;
  if (val < GENES_C1) {
    // crossover in the first chromosome
    memcpy(out->c1, gen1in->c1, val * UL_SIZE);
    memcpy(out->c1, gen2in->c1, (GENES_C1 - val) * UL_SIZE);

    memcpy(out->c2, gen2in->c2, GENES_C2 * UL_SIZE);
  } else {
    // crossover in the 2nd chromosome
    memcpy(out->c1, gen1in->c1, GENES_C1 * UL_SIZE);

    val -= GENES_C1;
    memcpy(out->c2, gen1in->c2, val * UL_SIZE);
    memcpy(out->c2, gen2in->c2, (GENES_C2 - val) * UL_SIZE);
  }

  out->fitness = -1;
}
```

2.3 Different strategies used

A Genetic algorithm treats individuals with genomes and tries to simulate an artificial evolution in which the fitness is the goal we want to achieve, with this, the individuals will evolve gradually to the expected result. For this to happen, we need some strategies that help the individuals to evolve, the ones we used are explained above:

2.3.1 Mutation

In order to add some variability to the individuals and have a more exploratory algorithm the mutation is introduced in order to change some values of an individual randomly, this allows to introduce new individuals or slightly different individuals who in the case of having a better fitness will substitute the old individuals.

2.3.2 Whole gene Crossover

The crossover simulates the generation of “children” genomes out of good fitness “parents” genomes.
It is done in order to mix and obtain a new individual out of two old good individuals so the genomes from the best individuals are taken randomly and get crossed. In our case we made a whole gene crossover instead of a bit crossover, this was done in order to reduce the complexity of both code and performance assuming that the mutation will take the role of changing the gene itself and using the fact that whole genes lead to a parameter so changing and mixing different parameters can easily lead to a worse result.

2.3.3 Elitism

In evolution the best individual persist and determine the direction of the species evolution, the elitism method is a bit artificial but ensures that we keep the best genomes so we cannot loose them through mutation or crossover.

2.3.4 Migration

A new method was used in order to introduce more variability. At each step the worse individuals are thrown away and we simulated a migration of new individuals with a different random seed; this simulates a migration of a different population which if fit enough, it will cross with the old good population and maybe lead to a better population.
It is worth noticing that the migrated individuals are usually not very fit and can be easily thrown away again. This is somehow normal for real evolution and we don't expect this mechanism to solve our problem but rather to increase variability among the population which maybe help the algorithm escape from a relative minimum.

2.3.5 Massive extinction / colonization

When we saw the model was easily getting stuck in relative minimums we thought about implementing a new mechanism able to give a strong kick and put the population out of this minimum.
The extinction mechanism is, as its name says, a mechanism that deletes all the population but a small number of survivors (10), taking the best phenotypes. After this the colonization takes place, the colonization consists on putting a almost whole new population in the space we have, after this a cooldown epoch happens in where there is weak selection but while crossing randomly all the individuals. This allows the new population to get fixed better (It is an upgraded migration) and give a huge amount of variability in the population.
After this cooldown iterations the algorithm retakes the usual procedure but with a new and more diverse population.
The strength and cooldown of the extinction needs to be calibrated properly to happen only when the population got stuck in a certain fitness. A combination of normal evolution and weak evolution is needed in order to make this mechanism work and get a better final fitness.

2.3.6 Ecosystem change

We thought this mechanism could also work as a way to take a population out of a relative minimum.
It is based on changing the "ecosystem" of the population making that the individuals adapt to a new fitness rules.
This may seem complicated and closer to evolution theory than to the algorithm, but we used the fact that, in theory, all fitness lead to a similar good approach of the Covid model, so this "ecosystem change" was mainly a fitness change.
Similarly to the extinction when the population got stuck in a minimum the fitness was randomly changed, changing thus the shape of the fitness function and making the minimum to disappear.
Unfortunately we couldn't obtain good results with this mechanism. We think this could be due to the wrong combination of the fitness periods, should we change radically the fitness until we find a new minimum or we just need it some steps so we get out of the minimum?
Another issue was the non normalized fitness, despite normalizing the fitness so we could compare the final results. When changing the fitness function the fitness best value suffered big increases making the algorithm get lost jumping from minimum to minimum.
So we ruled out the mechanism for the high exploratory behaviour but we suggest some variation of this may be useful for further Genetic Algorithms.

2.3.7 Deterministic optimisation

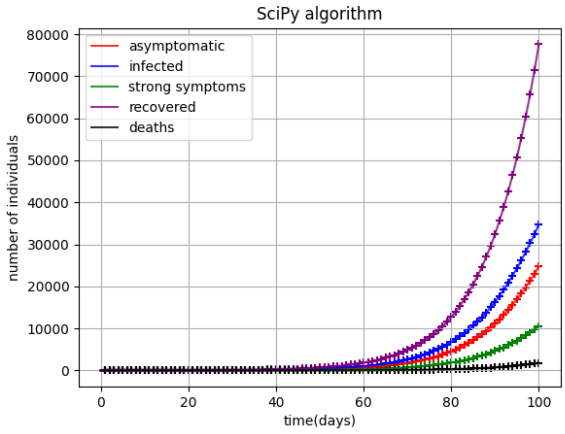
In order to help a bit the algorithm we added this "non genetic" part in where every N steps (being N a large number) a small deterministic optimisation was done, to the whole population. This optimization was about 10 or 15 steps of a pseudogradient method.
This may seem not enough to optimize a value, but the good part is that fit individuals will remain in the population and so we will get many small optimisations leading to a good big optimisation. With this we don't loose a lot of performance and we kind of apply "natural selection" to the individuals that will get optimised.
This method despite working pretty well and improving always the fitness (sometimes not substantially) was very time demanding and was decreasing the performance, but on the other hand it helped the genetic algorithm to work better.

2.3.8 Final mixture

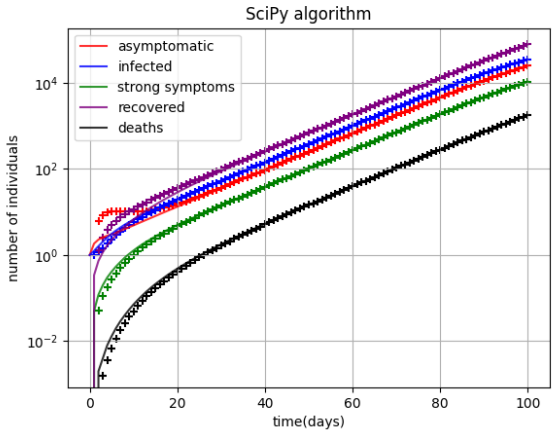
At the end we will mix all the methods presented before excepting the ecosystem change which was decreasing model's performance.
An idea we had in case we couldn't achieve a proper fitness would be to initialize different Genetic algorithms at the same time with the OpenMP library and every N steps taking the best individuals of all the parallel algorithms and putting them in the algorithm running on the master thread. With this we would generate fit individuals with different genotypes and mix them at the end in a central population with only the best individuals of each algorithm run.

3 Results

The issue giving the most difficulties in this problem is the fact that we don't know how good the model can adapt to this data obtained and as a consequence we don't know how good we can perform with a genetic algorithm. Maybe the bad results could come from inaccuracy of the model or either from the wrong setting of the genetic algorithm.
In order to solve this issue we decided to code a simpler code using Python and a pre-made library called SciPy, in this library there is a function which runs a Genetic algorithm in a very optimised way, so we took the results of this case as reference or goal to achieve, the fitness reached by the Scipy algorithm was $f = 455$ but in fact the value of the fitness is non-comparable between different fitness which may have different rescalings or fitness functions. So we used the comparison between the real data and the values obtained from the genetic algorithm, this results are presented in Figure 1.



(a)



(b)

Figure 1: SciPy simulation in + over real data (a) in linear y-scale (b) in logarithmic y-scale

Clearly the Scipy algorithm could fit the model to the data obtained reflecting the exponential growths of every process. We can conclude the model is good enough and the results will depend mainly on our genetic algorithm.

3.1 Different Fitnesses used

In our problem the fitness will be a function to minimize, we will be calculating in different ways the differences between the real data and the Runge Kutta results from the genotype. Each of the following fitnesses respect this main idea but have their pro's and con's

3.1.1 Uniform fitness

The uniform fitness can be defined as the most basic fitness, it consist on adding up all the absolute values of the differences between points and RK data. This fitness is maybe the better in terms of performance but treats equally a point close to the starting point than one further.

3.1.2 Linear fitness

We could expect that predicting well the initial conditions it's pretty easy and will only take in account the first chromosome, but when coming to the parameters of the second chromosome we can state that given a poor parameter it can be predicted pretty okay the first values but will mess up on the last ones.
So in order to reward the parameters that predict well the further points (where more error is accumulated) every difference between point and RK prediction will be multiplied by the day of the prediction.
This will cause that a determined difference in day 100 will be a hundred times larger than this same difference in day 1.

3.1.3 Exponential fitness

The previous fitness give more value to predicting the last elements good than the first ones, but knowing that the system we are predicting has an exponential behaviour we may ponderate the difference value in an exponential way, therefore the weight for every difference will be $e^{t\Delta x}$ which is an extreme case of the linear fitness.

3.1.4 Max fitness

All the last fitness have a larger computational time so we can define a good performing one that will reward the individuals that remain a lower difference threshold. This fitness will only take the largest difference and keep it as the total fitness.
We will try this fitness knowing that it might not be the best but should lead to acceptable results.

3.2 Rescaling factors

After running several experiments with the algorithm stated before and for all the combinations possible we achieved very poor results, we were obtaining values, in the case of the exponential fitness of $f \sim 4.5 \times 10^8$. After this we analyzed which proportion was coming out of every fitness weight. What we found are the following proportions:

$p_{A_d} = 0.149$ $p_{I_2} = 0.176$ $p_Y = 0.011$ $p_R = 0.67$ $p_D = 0.00023$

We can see how this fitness makes that the deaths will have a low impact on the final result while the recovered will impact the most. By rescaling them we achieve a model that will try to improve all of them in the same way.

3.3 Model accuracy

Despite this improvements the algorithm was still returning quite bad results only getting closer to the real value in the lasts values, here we have some examples for a run of 1000 generations and 100 individuals:

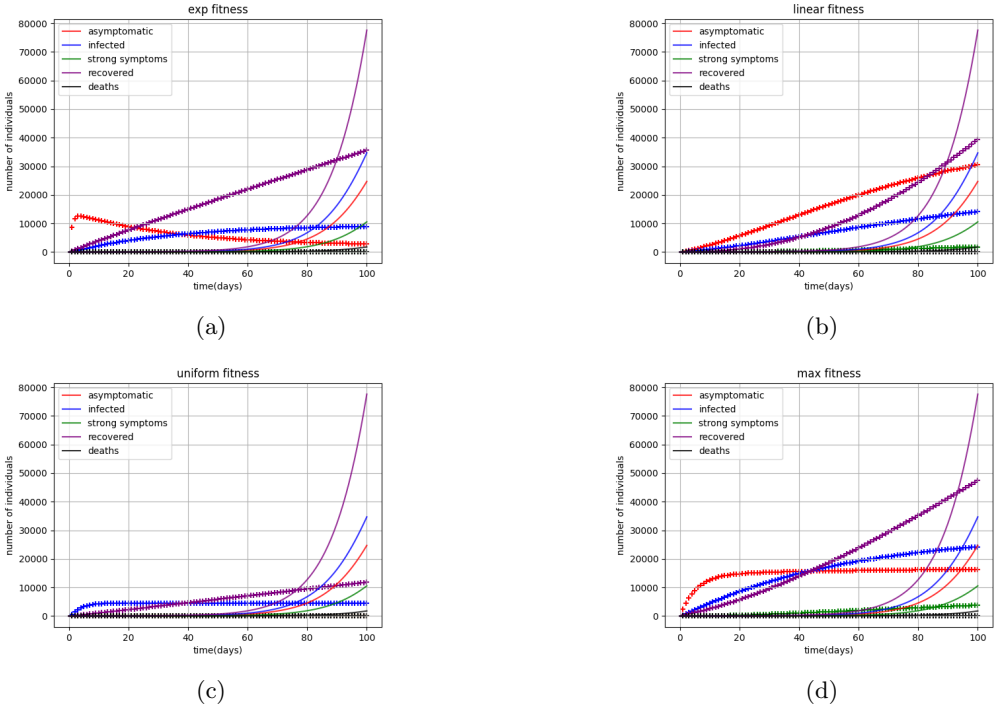


Figure 2: Comparison between real data and algorithm results of non improved accuracy for (a)exponential fitness (b) linear fitness (c) uniform fitness (d) max fitness

Here we can observe how the model is not predicting very well the expected results for any of the fitness, in particular we have that the max fitness is approaching somehow the final values but has a wrong curve prediction, and on the other side we have the uniform distribution which performs really bad.
Our hypothesis is that giving full precision from the beginning could generate a "high definition" fitness with lots of relative extremes,so as a solution we might give a low initial precision so it can find the larger minimum zones and after we will increase gradually the precision turning the coarse fitness function to a more defined function. This gradual increase of the precision can be achieved by adding a high mutation index affecting, for longer iterations smaller values of the unsigned long values.

In order to contrast if this hypothesis was true we set a low precision of only 10 decimals for all the parameters, added the gradual mutation and run the same tests as before. The results we obtained are in Figure 6.
In particular the maximum and uniform fitnesses performed the best in this cases but at least we can observe this exponential behaviour which before was unachievable. In general it can be seen in this results that lowering the precision of the model bringed us to better results. From the beginning we were using unsigned long variables for all the parameters and taking the asked precision at the end.
A possible explanation for this results improvement is that by using unsigned long the number of combinations possible are `sizeof(unsigned long)^11` which is an extremely large number, this makes that the algorithm has difficulties on mapping and searching in all the space; apart from this the "high resolution" of the mapping generates a "high definition" fitness function with more relative extremes, and with this, the algorithm gets easier stuck in one of this minima and keeps the result as the final one. By using this low precision variables we reduced the search scope of the initial values or migrations being able to reach a wider area in the hyperplane of parameter combinations and we also have a more coarse fitness map in where the small minimums are

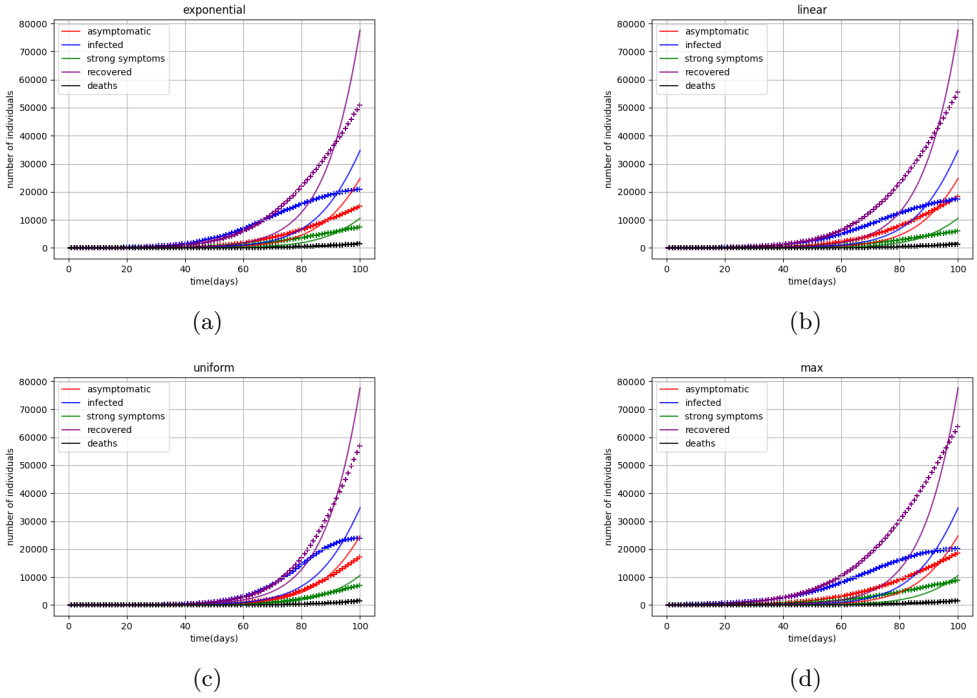


Figure 3: Comparison between real data and algorithm results with lower precision values for (a)exponential fitness (b) linear fitness (c) uniform fitness (d) max fitness

averaged and deleted; with this the algorithm will not get stuck as easier as before and will be easier in the attracting zone of the best minimum.
Using this hypothesis we tried to lower the initial conditions even more until the point of the precision asked in the pdf. With this values instead of having $4,294,967,295^{11} \approx 9 \times 10^{106}$ we only have 6×10^{29}

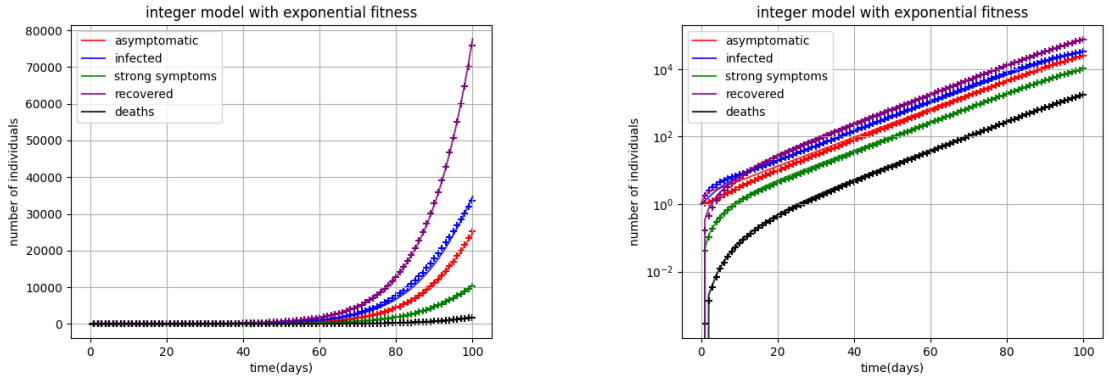
Parameter	β	ϕ	ϵ_I	ϵ_Y	σ	γ_1	γ_2	κ	p	α	δ
Initial precision	$\frac{1}{4096}$	$\frac{1}{512}$	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{512}$	$\frac{1}{512}$	$\frac{1}{512}$	$\frac{1}{64}$	$\frac{1}{512}$	$\frac{1}{4096}$	$\frac{1}{4096}$

Table 1: Initial precision of parameters

With this we did several runs and with the exponential fitness we achieved the following results:

individuals	1000
generations	40000
exponential fitness	14792.89
uniform fitness	4943.81
linear fitness	8676.04
maximum fitness	52956.94

Table 2: Main parameters of the Genetic Algorithm result



(a) (b)

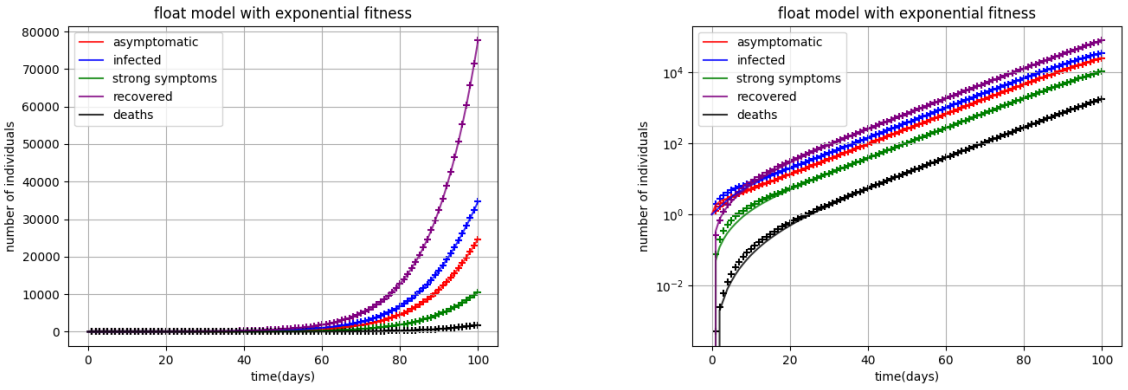
Figure 4: Results for a fitness $f = 14792.89$ compared to the real data (b) in logarithmic y-axis

This was quite shocking for us because we could obtain a substantially better results not by adding individuals or generations neither adding strategies but by changing the size of the parameters. We will keep this result as one of our final results on the Conclusion.
Which is a pretty good result if we compare it with the expected values in figure 5.
After going on the conclusions we will try one last change in our program.

3.4 Changing Unsigned long for Floats and Integers

From the beginning the unsigned longs have been the base for our algorithm. We used the 32 bits of information contained in one unsigned long, to describe each parameter value. We then converted this `unsigned long` to a `float` number using our function `genotype_to_phenotype`. This way we were able to establish a mapping from a random `unsigned long` to a `float` between 0 and 1 for most of the parameters. However, we found that at some point, the procedure of generating a random integer, and then mapping it to a float, is somehow biased. That is, when generating new individuals randomly as the algorithm goes on, the values of the parameters were quite similar to the ones discarded in the iteration before. We couldn't figure out what was happening, so we instead tried to use `floats (doubles)` directly, using a random `double` generator of the same `gsl` library. The results improved quite a lot when this change was made, being able to obtain an individuals with a fitness of 50.
A different explanation for this behaviour is the greatly reduced search space when using `double's`. A `double` has a precision up to 16 decimal places. This means that for our parameters constrained to be between 0 and 1 there are only 10^{16} numbers, whereas if we use the full range of an `unsigned long` there are $\sim 10^{106}$ possibilities. This immensely reduced search space may account for better performance of the genetic algorithm, making it less probable to get lost while trying to minimize the function.

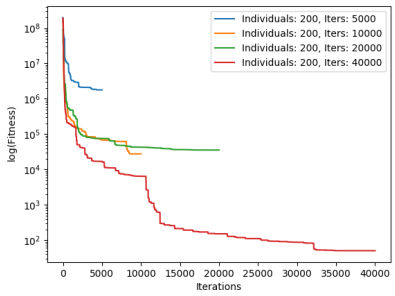
individuals	200
generations	40000
exponential fitness	50.26
uniform fitness	43.07
linear fitness	86.04
maximum fitness	1152.64



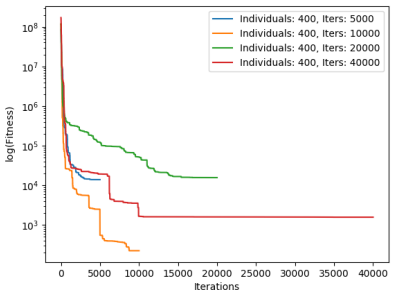
(a) (b)

Figure 5: Results for a fitness $f = 50.26$ in the float variable model compared to the real data (b) in logarithmic y-axis

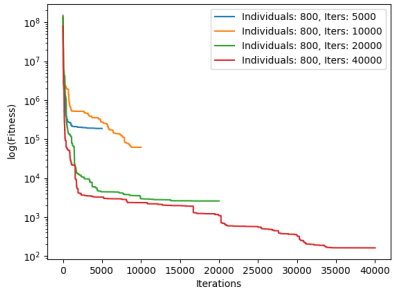
With this successful results we tried to do a small analysis of the fitness in terms of generations or number of individuals, and the results are in the following figures:



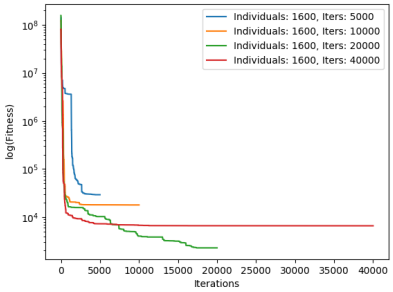
(a)



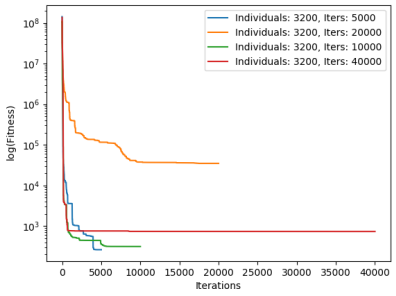
(b)



(c)



(d)



(e)

Figure 6: Evolution of the logarithm of fitness for different number of individuals and iterations and algorithms. As it can be seen there doesn't seem to be much of a difference among the final results when changing the number of individuals or iterations. It looks like the path followed by the algorithm is really irregular, and depending on it we may get better or worse results.

We can clearly see how this genetic algorithm set up has a roof top or bottom in this case from where the values will never go below, we reached a quite low fitness with this set up but anyway we can see that after a number of steps almost always a relative minimum is found so it won't go below that values.
Also it can be seen that increasing infinitely the number of individuals or generations improve until a certain point, and at some point a large population slows down the program a lot.

3.5 Final model results

So if we take the final results of the int genetic algorithm and the float genetic algorithm for two of our best runs we can try to extract conclusions of the parameters obtained for each algorithm. In table 3 we have the two sets of results and their fitnesses From the table we can observe how the float algorithm reached a

Parameter	Int algorithm	Float algorithm
E	32.75	0.53
I_1	32.73	34.81
A	5.40	0.25
β	0.281250	0.522926
ϕ	0.0000000	0.359148
ϵ_I	0.999023	0.548952
ϵ_Y	0.299804	0.054899
σ	0.236328	0.125097
γ_1	0.233398	0.058164
γ_2	0.014648	0.085190
κ	0.006835	0.760644
p	0.114257	0.608317
α	0.030273	0.053266
δ	0.014648	0.014500
Fitness	14792.89	50.26

Table 3: Results obtained with the two genetic algorithms

substantially better fitness so we can ensure the values were way closer to the real data and in consequence may fit better the real situation.

3.5.1 Integer algorithm results

For the integer algorithm we found that the initial exposed and infected with weak symptoms is about 32 people, also the asymptomatic are around 5 something that makes sense, low numbers that represent just the beginning of the exponential growth.
When coming to the parameters we observe an average infectivity of around 0.3 which means that per every infected we get in the next day 0.3 exposed people. For Covid this value seems a bit low but if we talk about the average infectivity it may be quite accurate because we need to take in account the non infected interacting and the different strategies used depending on the symptoms.

In this simulation we found a value of $\phi = 0$ which means that this approach states that there is no pre-symptomatic infectivity factor. From real world we got that this infectivity even being low is non-zero. This is then a contradiction between the values obtained in this simulation and what is stated by doctors in the news.

When coming to the isolation efectiveness this model found an isolation efectiveness of strong symptoms practically one which means that we could almost manage completely to isolate the strong symptoms individuals, but also shows a lower value $\epsilon_Y = 0.3$ for the severe symptoms, this reflects somehow the situation we lived with a good isolation of strong symptoms making the people staying at home but shows a breach on the isolation of the severe cases (hospitals) which fits with the health care collapse and the overcrowding of the hospitals during the pandemic making that lots of doctors got infected. Despite the radical isolation

efectiveness of strong symptoms these two values were according quite much to the real situation.

σ is the parameter responsible for the rate of appearance of infectiveness, or what is the same the average time the exposed turn to infected, if we want to get the average days we need to do the inverse of the parameter which in this case is $1/23 \approx 4.3$ days. Which are reasonable results knowing that the average time given in many studies is around 5 days. We can justify this small difference to the high viral charge exposure we got in this first wave, if the amount of virus in the environment is considerably higher the incubation time may get slightly reduced.

For the rate of symptoms to appear we have similar values that for σ so approximately four days after starting to be infectious you develop symptoms. For this model the time from being exposed to having symptoms is around a week, something reasonable knowing that is said that Covid takes between 2-14 days to show up symptoms.
Something very interesting is the rate of recovery which for the infection in general terms is said to be around a month but in this case we found a value $1/0.0146 \approx 68$ days which is an enormous value showing how long it may take to get recovered once you get infected. This big difference between the estimated general rate and the one in this case may be because the overexposure and the saturation of hospitals, if in the environment there is a huge viral charge it can be more difficult for the organism to fight this infection and so in consequence take longer to get recovered. Similarly can happen in closed rooms where people got confinated with big amounts of virus in the air.

For the κ value we get a quite descriptive value of how controlled was the situation and how many tests were done compared to the real infected people, it shows the detectability of the asymptomatic, which for our model we get a detectability of 0.68% which is really low, although this value is extremely low and we think the real one must be a bit larger because 0.7% is almost nobody.

Also we found a probability of strong symptoms of 10% which is a consistent result according to the observed cases where only a small amount of the infected show strong symptoms. And similarly the rate of having strong symptoms and developing severe symptoms was even lower, around 3% which leads to a 0.0033% of developing severe symptoms if having the infection.

To conclude we also measured the mortality of the disease which for this model resulted on a mortality of 1.4% in the case of our first wave of the Covid-19, from the experts it was stated that the mortality of the coronavirus could be between $0.5 - 1\%$ which is lower than our result. For our result we cannot attribute this high mortality to the asymptomatic because we took them in account too, so we can of make a simple hypothesis and assume that the aging population of Spain together with the hospital saturation leaded to an increase of $0.4 - 0.9\%$ of the mortality of this virus.

3.5.2 Float algorithm results

In the case of the float algorithm results we found that the initial exposed together with the asymptomatic was less than one and the main group in the beginning were the non isolated infectious with a value of 35, differently from the other algorithm giving a slightly wider and more spread initial situation.

In this case the average infectivity was higher reaching the 0.5 so in our more accurate model the infectivity was that every every two days an infected was infecting a new person in average.

In this case the ϕ is non zero and quite big compared to the zero value we obtained before, we have a pre symptomatic infectivity of 0.3 which means that every 3 days a presymptomatic individual will infect. This seems more accurate than the previous case being smaller than the infectivity of a symptomatic but different than zero.

Moreover now the isolation efectiveness of the strong symptoms is 0.5 now which means that one out of every

two strong simptoms is well isolated and the other is not. This also makes more sense because we had also infections of partners living together despite being isolated. Also in this model we got an even lower isolation for the severe symptoms remarking too the saturation of hospitals and the impossibility to keep controlled huge amounts of severe symptoms individuals which reflects also the hard situations in hospitals we had on the first wave, in this model the isolation efectiveness of severe symptoms was 0.05 which leads to a conclusion that the better the model the more evidence of a uncontrolled situation when trying to contain the infection spread.

In this case the σ was lower indicating that the individual takes around a week to be infectious, and similarly for the rate of appearance of symptoms we have an expected time of 15 days, definitively this two constants are maybe fitting the data we gave but do not fit the coronavirus rates, which should be more or less 2 days and 4 days respectively.

For the recovery rate we have now $1/0.085 = 11.8$ days which is closer to the real value and has more sense than the last two rates obtained.

The κ responsible for the detectability of the asymptomatic people now raised to 0.76 showing a better scanning of the population and more tests done than the previous model which shows maybe a non realistic case compared with the chaotic initial situation of the first wave.

Here the probability for strong symptoms is really high leading to a case in where if an individual has symptoms has 0.6 probability of developing strong symptoms every day, this would depend on what is meant by "strong symptoms" but in general we wouldn't say this parameter fits the coronavirus pandemic.

Also we also have the rate of appearance of severe symptoms which here is low again similar to the previous model $\alpha = 0.05$ and the combined probability gives that an infected will develop severe symptoms in the 3% of the cases which is 100 times more than the previous case. In general this combined result of severe symptoms out of being infected is not that unconscionable as the other two so we could say that despite being quite inaccurate on the individual rates in general performs good when predicting the severe symptoms individuals.

To finish we have the disease induced mortality which is pretty similar to the previous model so we could conclude that both models agree on the mortality and so this model corroborates the hypothesis of the previous model about the increase of average mortality of the disease.

4 Conclusions

For this report we can focus on two main aspects on the conclusions. The algorithm itself with the issues we found during the project and the conclusion for the model.

4.1 Code conclusions

We can conclude that a Genetic Algorithm is a very powerful tool to optimize very complicate systems but has some drawbacks, the main one is that you don't really know where is the rooftop or the best result you can achieve and due to it's random behaviour running it only once you can't know if a result is good, very good or incredibly bad. For this is good to rely on extra tools or well performing algorithms that can give you a clue of how good are your results.
In our case we used both a Scipy (Python) genetic algorithm to determine numerically a good fitness result and a plot comparing the runge kutta results achieved with the data given to fit the model. With this we could compare and give a qualitative (and somehow quantitative) conclusion of how good the results were.
We implemented different extra techniques on the classical genetic algorithm in order to improve the final results, for example the *extinction* technique that was renewing and giving randomness to the population, the *migration* that was adding gradually some individuals in the population replacing the bad ones, the *deterministic optimisation* which was slowly guiding in a deterministic way to the local minima the individuals, and the *ecosystem change* which was changing the fitness in order to allow the population to escape from the local minimum.
Furthermore we found that the initial population is really a key on the algorithm and having a widely and uniformly distributed population helps when trying to find a better minimum. That is why in the beginning when using a full `unsigned long` does not seem to work as good than when using a `double`. The vastly search space makes it easier for the difference in the parameters among individuals to fall near each other most of the time. Thus we conclude this may be one of the reasons for the poorer performance compared to when we use the `double` type for the parameters. Given its lower search space we are able to generate a more widely spaced parameter population. In addition, it is worth noting that we found no clear relationship between the number of individuals and iterations with respect to the fitness of the model. Perhaps iterations affect in greater measure to the final fitness. This makes sense since there is more time to perform pseudo-gradient methods and different techniques to make the algorithm pop out of the local minima it may be in. However, the number of individuals does not seem to make much of a difference for the achieved quality of the final fitness.

4.2 Model conclusions

When coming to the model the two results we obtained a better fitness for the float model than for the integer model, despite this when analyzing the parameters one by one and comparing with the standards with covid we can say the integer model achieved parameters closer to the coronavirus pandemic. The float model was representing a pandemic or a disease spread too but with slightly different parameters, for example it was representing a model with a higher infectivity, and with longer time between exposition and being infectious or time until symptoms are shown up. Also the disease has a substantially higher rate of leading to strong symptoms.
Also the situation is maybe not fitting the real case because the float model was stating that there were very low isolations practically zero isolation for severe symptoms and very good detectability of the asymptomatic individuals, in order to have a higher precision we would need to use a fitness function taking in account more data collected so the disease spread could be more constrained.

So in conclusion this model was good enough for fitting this case or this pandemic, but the huge amount of variables leads so many good combinations that achieve the same fitness. This causes that the conditions and parameters found may fit the data given and at the same time representing a different disease (case of the float model). In our case we've chosen the integer model result as the one fitting most the Covid first wave.

The model was reflecting the increased mortality in Spain compared to the average, it also reflects well the rate of appearance of infectious ability and symptoms, and returns a large recovery time.
We achieved a good set of parameters for this Covid model and it would be interesting to test this genetic algorithm in other periods of the pandemic in order to test if the model purposed also fits other stages of a disease spread.