Optimization - Genetic Algorithm

Eline Michiels (1624509)

January 9th 2022

1 Introduction

At La Banya, there are Audouin's gulls and their main competitor yellow-legged gulls. Over the years they have seen the population of Audouins's gulls evoluate like depicted in figure 1.

year	pop.	year	pop.	year	pop.	year	pop.
1981	36	**1990**	4300	**1999**	10189	**2008**	13031
1982	200	**1991**	3950	**2000**	10537	**2009**	9762
1983	546	**1992**	6174	**2001**	11666	**2010**	11271
1984	1200	**1993**	9373	**2002**	10122	**2011**	8688
1985	1200	**1994**	10143	**2003**	10355	**2012**	7571
1986	2200	**1995**	10327	**2004**	9168	**2013**	6983
1987	1850	**1996**	11328	**2005**	13988	**2014**	4778
1988	2861	**1997**	11725	**2006**	15329	**2015**	2067
1989	4266	**1998**	11691	**2007**	14177	**2016**	1586
						2017	793

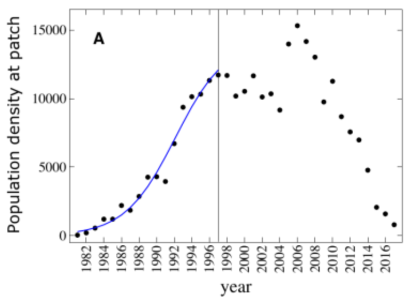


FIGURE 1 – A table with the population data of the Audouin's gulls from 1981 to 2017

We know that predators appeared in 1997, that's why the period between 1981-1997 is called the *First Epoch*, it is characterisized by a logistic growth and the fact that the population did not exhaust the food carrying capacity.

In 2005 there was an (at the moment) unexplicable and unexpected change in the population of Audouin's gulls. This is why the *Second Epoch* reaches from 2006-2017, this period has a regular behaviour with migration.

There are several biotic and abiotic factors that have an influence on the Audouins's gulls population. It is shown in previous studies that local biotic factors are the most important. Examples of biotic factors are :

- ❖ competition with yellow-legged gulls
- ❖ carnivore density
- ❖ food availability

However, these variables cannot explain the decline in population of the Audouin's gulls at La Banya in the *Second Epoch.*

Section 1,2 & 3 discuss the problem. These sections are summaries of the given assignment. Section 4 explains in detail the genetic algorithm used to find a solution. Section 5 describes what needs to be done after running the genetic algorithm function and section 6 discusses the obtained solution.

2 Mathematical model

The hypothesis is that the birds start dispersing in an independent, density-dependent manner due to the presence of predators. This would be the social response to the predators. The following model describes the population dynamics of birds in the patch of the study. The model considers immigration of individuals, intra-specific competition for resources and density-independent death rates. The following model is proposed :

$$\frac{dx}{dt} = \gamma x(1 - \frac{x}{K}) - \epsilon x - \lambda\Psi(x, \mu, \sigma, \delta) \tag{1}$$

With initial population $x(0)$. We can also write this considering the ecological processes :

$$\frac{dx}{dt} = \alpha x - \gamma\frac{x^2}{K} - \lambda\Psi(x, \mu, \sigma, \delta) \tag{2}$$

The first term in equation (2) stands for the immigration, growth and dead, the second is the nonlinear competition term and the third is the dispersal by social copying. We consider the following parameters and their units.

- ❖ $\alpha = \gamma - \epsilon$, units : $birds/year$
- ❖ γ : reproduction of birds and arrival of other birds to the patches (intrinsic growth rate) , units : $birds/year$
- ❖ K : carrying capacity, units : $birds$
- ❖ ϵ : density-independent dead rate, units : $birds/year$
- ❖ $\beta = \frac{\gamma}{K}$: intrinsic growthrate/carrying capacity, units : $years^-1$

The nonlinear dispersal function is given by :

$$\Psi(x, \mu, \sigma, \delta) := \begin{cases} \frac{1-\epsilon_{dir}(x,\mu,\sigma,\delta)}{1-\epsilon_{dir}(0,\mu,\sigma,\delta)} when 0<\text{x}<\delta \\ \frac{1-\epsilon(x,\sigma,\delta)}{1-\epsilon_{dir}(0,\mu,\sigma,\delta)} when \text{x}>\delta \end{cases} . \tag{3}$$

$$\epsilon_{dir}(x, \mu, \sigma, \delta) := (\mu\frac{\Theta + \sigma\delta}{2\Theta + \sigma\delta}(1 - \frac{x}{\delta}) + \frac{x}{\delta})\epsilon(x, \sigma, \delta) \tag{4}$$

where

$$\epsilon(x, \sigma, \delta) := \frac{\sigma(x - \delta)}{\Theta + \sigma|x - \delta|} \tag{5}$$

Equation (5) is an *Elliot sigmoid* : Θ-scaled, σ-strengthened, and δ-displaced.

3 The exercise

The goal is to fit the parameters of the following model :

$$\frac{dx}{dt} = \varphi x - \beta x^2 - \lambda\Psi(x, \mu, \sigma, \delta) \tag{6}$$

to the data from the *Second Epoch*, to check the hypothesis that Andouin's migration occurs with social copying with a genetic algorithm. Here we see the parameter φ, which is explained in figure 2.

Parameter	Range or value	Meaning or description
β	0.000024382635446	Intrinsic growth rate over the carrying capacity. Estimated with the Fisrt Epoch Data.
$\varphi = \alpha - \rho$	$\leq \alpha = 0.3489494085776018$	Neat population growth rate. It includes a linear migration term of the form ρx, where ρ is the linear dispersal rate.
$x_{\varphi,\lambda,\mu,\sigma,\delta}(0)$	$[0, K]$	ODE's Initial condition.
λ	$\mathbb{R}^+$	Non-linear Dispersal Rate.
μ	$\mathbb{R}^+$	Determines $\frac{d}{dx}\Psi(x,\mu,\sigma,\delta)\big\|_{x=0}$. It is $\begin{cases} 0 & \text{when } \mu = 1, \\ \text{negative} & \text{when } \mu > 1, \text{ and} \\ \text{positive} & \text{when } \mu < 1. \end{cases}$
σ	$\mathbb{R}^+$	Determines the "slopes" of the sigmoids. $\sigma \approx 600$ approximates a Heaviside function.
δ	$\mathbb{R}^+$	Point of change of concavity of $\Psi(x,\mu,\sigma,\delta)$.

FIGURE 2 – Parameters of equation (6) with their range and explanation

The solution of the model is $x(t) = x_{\varphi,\lambda,\mu,\sigma,\delta}$ and the solution depends on the initial condition $x(0)$, that must be considered a free parameter.

4 Genetic Algorithm

In this optimization problem, we use a genetic algorithm to find the possible solutions. A candidate solution (individual) has specific properties, which we call chromosomes. In our problem, the initial set of individuals (initial population) is chosen randomly in right range of the parameters. The algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce the offspring of the next generation.

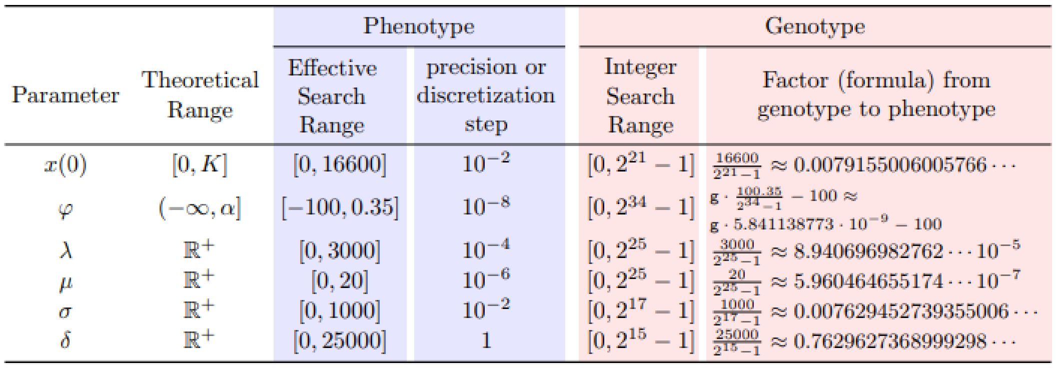
4.1 Individuals

We represent the individuals with six chromosomes, corresponding to the six free parameters. Because of Holland's convergence theorem, that works with unsigned integers expressed in binary, it is better to encode the real number phenotype into a discretized genotype consisting in unsigned integers. We also add the fitness parameter so we can later distinguish good from bad individuals, see paragraph 4.2.

```c
typedef struct{
    unsigned long int x0;
    unsigned long int delta;
    unsigned long int lambda;
    unsigned long int mu;
    unsigned long int phi;
    unsigned long int sigma;
    double fitness;
} Individual;
```

Listing 1 – Representation of an individual

The following table fixes the range, the precision and thus the discretization formula for obtaining the phenotype out of the genotype :



		Phenotype		Genotype	
Parameter	Theoretical Range	Effective Search Range	precision or discretization step	Integer Search Range	Factor (formula) from genotype to phenotype
$x(0)$	$[0, K]$	$[0, 16600]$	10^{-2}	$[0, 2^{21} - 1]$	$\frac{16600}{2^{21}-1} \approx 0.0079155006005766\cdots$
φ	$(-\infty, \alpha]$	$[-100, 0.35]$	10^{-8}	$[0, 2^{34} - 1]$	$g \cdot \frac{100.35}{2^{34}-1} - 100 \approx$ $g \cdot 5.841138773 \cdot 10^{-9} - 100$
λ	$\mathbb{R}^+$	$[0, 3000]$	10^{-4}	$[0, 2^{25} - 1]$	$\frac{3000}{2^{25}-1} \approx 8.940696982762\cdots 10^{-5}$
μ	$\mathbb{R}^+$	$[0, 20]$	10^{-6}	$[0, 2^{25} - 1]$	$\frac{20}{2^{25}-1} \approx 5.960464655174\cdots 10^{-7}$
σ	$\mathbb{R}^+$	$[0, 1000]$	10^{-2}	$[0, 2^{17} - 1]$	$\frac{1000}{2^{17}-1} \approx 0.007629452739355006\cdots$
δ	$\mathbb{R}^+$	$[0, 25000]$	1	$[0, 2^{15} - 1]$	$\frac{25000}{2^{15}-1} \approx 0.7629627368999298\cdots$

FIGURE 3 – Phenotype and genotype

We store the parameters necessary to solve the differential equation and generate the predictions in the struct **ODE_Parameters** (struct given in the description of the assignment).

We can convert the genotype saved in the individual, to the phenotype saved as a struct of the **ODE_Parameters**, using the last table.

```
ODE_Parameters Phenotype(Individual individual)
{
    ODE_Parameters phenotype;
    phenotype.x0 = (double) individual.x0 * 16600/((1UL<<21)-1);
    phenotype.delta = (double) individual.delta * 25000/((1UL<<15)-1);
    phenotype.lambda = (double) individual.lambda * 3000/((1UL<<25)-1);
    phenotype.mu = (double) individual.mu * 20/((1UL<<25)-1);
    phenotype.phi = (double) individual.phi * 100.35/((1UL<<34)-1)-100;
    phenotype.sigma = (double) individual.sigma * 1000/((1UL<<17)-1);
    phenotype.beta = 0.000024382635446;
    return phenotype;
}
```

Listing 2 – Conversion from genotype to phenotype

4.2 Fitness

It is important that the genetic algorithm keeps track of the fitness of the individuals to select the individuals that create offsprings for the next generation. Eventually we take the individual with the best fitness and generate the prediction of $x(t)$ with the parameters of this individual. The fitness function we used is the sum of the squared difference between $x(t)$ and $z(t)$:

$$\sum_{t=0}^{11} W_t(x(t) - z(t + 2006))^2 \tag{7}$$

Where $x(t)$ denotes the prediction made with the parameters of a individual and $z(y)$ is the population size of Andouin's gulls at year y. Since we estimate the *Second Epoch*, this data is known. We set the weights W_t equal to 1. The code for this fitness is the following :

```c
double fitness(Individual individual)
{
    double data[] = {15329,14177,13031,9762,11271,8688,7571,6983,4778,2067,1586,793};
    int i;
    double sum = 0;
    unsigned years = 12;
    double x[years],max=0,tmp;
    ODE_Parameters pars = Phenotype(individual);
    int result = Generate_EDO_Prediction(x,pars.x0,years,&pars);
    for(i=0;i<years;i++)
    {
        tmp = pow(x[i]-data[i],2);
        sum=sum+tmp;
    }
    return sum;
}
```

Listing 3 – Fitness function

4.3 The genetic algorithm

The **GA**-function takes two arguments : the *popsize* argument and the *generations* argument. In the argument *popsize*, you specify the amount of individuals in every generation (and thus the size of the population). In the *generations* argument, you specify the amount of generations you want. In every iteration we have a parent array P and we generate their offsprings in the the C array and in the same iteration, we define the parents for the next generation, again in the P array. After a certain number of generations (we set the default to 60 generations), the function outputs the best individual. The **GA**-function can be found in the Appendix.

4.3.1 Initialization

We first start to initialize the algorithm. In our case, we start by introducing the initial population. We chose the chromosomes per individual randomly, but in the range specified in figure 3, with the function **ULONGran**. Obviously we create the number of individuals defined by *popsize*.

4.3.2 Selection

The driving idea behind the selection is that only relatively fit individuals are chosen to create offsprings to the next generation. The individuals are thus selected based on their fitness scores. In this algorithm we use the tournament selector to specify the individuals that create offsprings. The tournament selector randomly selects t individuals in the population and then outputs the individual with the best fitness as a parent of the next generation's offspring. We let the parameter t be linearly dependent of the generation. When the generation we chose the parent in, is low, we want to select more randomly to be more explorative and thus our t is lower. When the number of the generation is higher, we want to be more exploitative and thus our t is higher.

4.3.3 Genetic operators

In our algorithm we use the one-point crossover and the bitflipmutation as genetic operators.

The one-point crossover makes sure that the children contain genetic information of both parents. One point in the parent's chromosomes is chosen randomly in the correct range of that chromosome specified in figure 3 (if the crossover point is chosen at a bit that is too high out of the range of that chromosome, the children will be the same as the parents). Then the bits on the right of this point get interchanged.

The bitflipmutation changes a bit in the representation of the chromosome in the complementary bit. It does this in the specified range specified in figure 3 per chromosome. The bitflipmutation has a certain mutation probability with which it changes the bit. We chose here to linearly decrease the mutation probability with an incrementing number in generations. In this case, we are more exploratory in the beginning and more exploitatory in the end.

4.3.4 Parents next generation

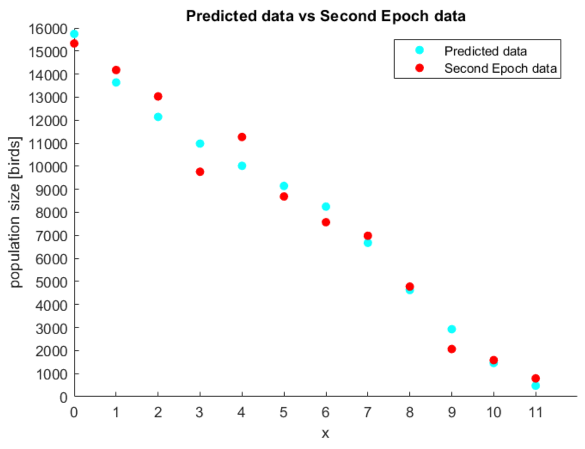
Once we generated the children in this generation in the C array, it is also necessary to define the parents of the next generation. We chose here a *elitist selection*, which means we pass the best individuals of the previous generation automatically on to the next generation. We chose here to pass the best 4%. We fill the rest of the parents with the 96% best children in the C array. Using the *elitist selection*, we make sure that the solution quality will not decrease from generation to generation.

5 Post GA-function

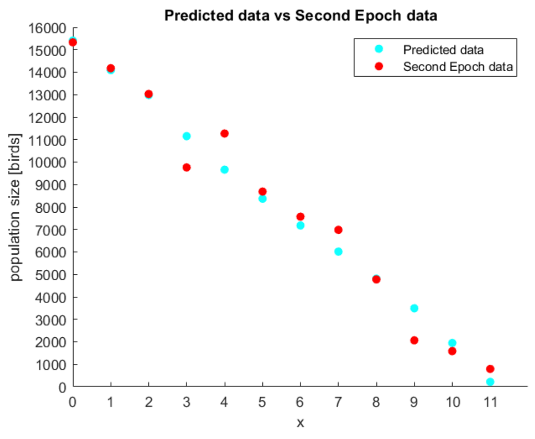
After the GA function outputs the best found individual, we want to see if in the environment of the found solution, there might be even better individuals that are not found by the GA function. We want to find those in a purely exploitative way,namely hill climbing. We adjust the parameters a little bit and see whether the fitness improves or not. This happens in the **adjust** function and in the **Hillclimbing** function. This function keeps on searching for better solutions in every iteration. But we set a stop criterium when either the function reaches 1000 iterations or when in 50 iterations the fitness has not improved. The function eventually outputs a the same or improved individual.
Next, we can convert the individual back to the phenotype, which present the values of the parameters in the ODE. With these parameters we run eventually the **Generate_EDO_Prediction**-function to obtain the predicted values for the population of Andouin's Gulls.

6 Obtained solution

We ran the algorithm with a population size of 10 000 individuals and we want 60 generations. Out of 10 runs we get the best fitness of 5957107 and a worst fitness of 8230749. We plot the predictions and the real data of the *Second Epoch*.



(a)



(b)

FIGURE 4 – Graph of observed population and predicted population based on a) a solution with fitness 5957107 b) a solution with fitness 8230749

In tabel 1, we find the parameters of equation 6 for both solutions described above.

Parameter	fitness = 5957107	fitness = 8230749
$x(0)$	15742.838355	15407.031158
λ	1312.605450	2249.822594
φ	0.141727	0.276247
μ	0.001952	1.822646
σ	1.991287	15.815856
δ	7584.612568	13283.181249

TABLE 1 – Parameter values of two obtained solutions

7 Appendix

```
Individual GA(int popsize, int generations)
{
    int generation = 0, static_generations=0;
    int Index;
    int half_popsize = popsize/2;
    float percentage1 = 0.04;
    Individual *P;
    Individual *C;
    Individual Parent1;
    Individual Parent2;
    Individual overall_best;
    Individual best_of_gen;
    if((P=(Individual*)malloc(popsize*sizeof(Individual)))==NULL)
        ExitError("Couldn't allocate memory for individuals",32);
    if((C=(Individual*)malloc(popsize*sizeof(Individual)))==NULL)
        ExitError("Couldn't allocate memory for individuals",32);

    randomize();
    for(int i=0; i<popsize; i++)
    {
        P[i].x0 = ULONGran(21);
        P[i].delta = ULONGran(15);
        P[i].lambda = ULONGran(25);
        P[i].mu = ULONGran(25);
        P[i].phi = ULONGran(34);
        P[i].sigma = ULONGran(17);
        P[i].fitness = fitness(P[i]);
    }

    overall_best = P[0];
    overall_best.fitness = fitness(overall_best);

    do
    {
        unsigned int tournament_selector_param = (unsigned int) (((0.1*popsize*generation)/
    generations) +2);
        float mutation_prob = 0.09-(0.085/generations)*generation;
        best_of_gen = P[0];
        best_of_gen.fitness = fitness(best_of_gen);
        for(int i=0; i<popsize;i++)
        {
            if(P[i].fitness<overall_best.fitness)
            {
                overall_best = P[i];
                static_generations = 0;
            }

            if(P[i].fitness<best_of_gen.fitness)
            {
                best_of_gen = P[i];
            }
        }
    printf("Generation= %d Best_fitness= %f\n",generation, best_of_gen.fitness);
```

```c

        for(int i=0;i<half_popsize;i++)
        {
            Index = half_popsize +i;
            Parent1=tournament_selection(P,tournament_selector_param,popsize);
            Parent2 = tournament_selection(P,tournament_selector_param,popsize);
            OnePointCrossover(Parent1.delta,Parent2.delta,&C[i].delta,&C[Index].delta,15);
            OnePointCrossover(Parent1.x0,Parent2.x0,&C[i].x0,&C[Index].x0,21);
            OnePointCrossover(Parent1.mu,Parent2.mu,&C[i].mu,&C[Index].mu,25);
            OnePointCrossover(Parent1.sigma,Parent2.sigma,&C[i].sigma,&C[Index].sigma,17);
            OnePointCrossover(Parent1.lambda,Parent2.lambda,&C[i].lambda,&C[Index].lambda,25);
            OnePointCrossover(Parent1.phi,Parent2.phi,&C[i].phi,&C[Index].phi,34);
            BitFlipMutation(&C[i].delta,mutation_prob,15);
            BitFlipMutation(&C[i].x0,mutation_prob,21);
            BitFlipMutation(&C[i].mu,mutation_prob,25);
            BitFlipMutation(&C[i].sigma,mutation_prob,17);
            BitFlipMutation(&C[i].lambda,mutation_prob,25);
            BitFlipMutation(&C[i].phi,mutation_prob,34);
            BitFlipMutation(&C[Index].delta,mutation_prob,15);
            BitFlipMutation(&C[Index].x0,mutation_prob,21);
            BitFlipMutation(&C[Index].mu,mutation_prob,25);
            BitFlipMutation(&C[Index].sigma,mutation_prob,17);
            BitFlipMutation(&C[Index].lambda,mutation_prob,25);
            BitFlipMutation(&C[Index].phi,mutation_prob,34);
        }

        for(int i=0; i<popsize; i++)
        {
            C[i].fitness = fitness(C[i]);
        }

        //sort populations here
        qsort(P, popsize, sizeof(Individual), Ind_comparator);
        qsort(C, popsize, sizeof(Individual), Ind_comparator);

        //change population here
        for(int i=0;i<popsize;i++){
            if(i<percentage1*popsize){
                P[i]=P[i];
            }
            else{
                P[i]=C[i-(int)(percentage1*popsize)];
            }

        }
        generation++;
        static_generations++;
    } while (generation<generations && static_generations<25);
    return overall_best;
}
```

Listing 4 – GA function