Optimization - A Possible Migration Model

Marcell Veiner (1619878)

January 2022

1 Introduction and Prolem Statement

This report has been written to discuss finding a parametrisation of the mathematical model written for predicting the Andouin's population data at La Banya from 1981 to 2017 (Fig 1). The model states that given a starting population $x(0)$, the change in the population can be calculated as

$$\frac{dx}{dt} = \phi x - \beta x^2 - \lambda \Psi(x, \mu, \sigma, \delta) \tag{1}$$

where ϕ is the neat population growth term, βx^2 is the intrinsic growth, and $\lambda\Psi(x,\mu,\sigma,\delta)$ is the dispersal term caused by social copying for migration, and is defined by:

$$\Psi(x,\mu,\sigma,\delta) = \begin{cases} \frac{1-\varepsilon_{\text{dir}}(x,\mu,\sigma,\delta)}{1-\varepsilon_{\text{dir}}(0,\mu,\sigma,\delta)} & \text{when } 0 \le x \le \delta \\ \frac{1-\varepsilon(x,\sigma,\delta)}{1-\varepsilon_{\text{dir}}(0,\mu,\sigma,\delta)} & \text{when } x > \delta \end{cases} \tag{2}$$

where

$$\varepsilon_{\text{dir}}(x,\mu,\sigma,\delta) = \left(\mu \frac{\Theta + \sigma\delta}{\Theta + 2\sigma\delta}\left(1 - \frac{x}{\delta}\right) + \frac{x}{\delta}\right)\varepsilon(x,\sigma,\delta), \tag{3}$$

and

$$\varepsilon(x,\sigma,\delta) = \frac{\sigma(x-\delta)}{\Theta + \sigma|x-\delta|} \tag{4}$$

is an Elliot sigmoid Θ scaled (fixed to 1000), σ strengthened, and δ displaced. The task is to find the model parameters using a Genetic Algorithm (GA), by fitting the predictions to the observed data from 2006 to 2017. It can be seen from the data that no migration was present before 2006, and thus data previous to 2006 has been used to fit the parameter $\beta = 0.000024382635446$. The rest of the terms is to be found. For the theoretical and effective ranges of these parameters see Fig 2.

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

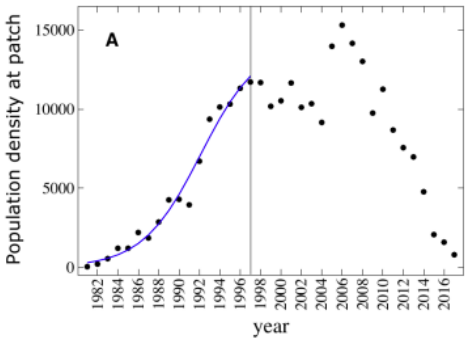

Figure 1: Audouin's gulls population from 1981 to 2017 (taken from the project description)

2 Methodology

Parameter	Theoretical Range	Phenotype: Effective Search Range	Phenotype: precision or discretization step	Genotype: Integer Search Range	Genotype: 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}	$[0, 2^{34}-1]$	$\mathbf{g}\cdot\frac{100.35}{2^{34}-1} - 100 \approx$ $\mathbf{g}\cdot 5.841138773\cdot 10^{-9} - 100$
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 2: Theoretical and Effective Ranges of Parameters, and their Representation (taken from the project description)

2.1 Prediction and Fitness

To calculate the Fitness of an individual, the ODE needs to be integrated, which is done using Runge-Kutta-Fehlberg method of order 7-8 with adaptive space, the code for which is separated in the RKF78-2.2.c folder, and is called from the Predit.c script. The Fitness can be then calculated in one of two ways:

$$\texttt{fitness}(x) = \max\left\{\left(x(t) - z(t+2006)\right)^2 \text{ for } t = 0, 1, ..., 11\right\} \tag{5}$$

where $z(t)$ is the actual population at time t. Similarly

$$\texttt{fitness}(x) = \sum_{t=0}^{11} W_t \left(x(t) - z(t+2006)\right)^2 \tag{6}$$

where W_t are non-negative weights, to be assigned to each datapoint. In the implementation all of these are set to zero, so it reduces to the L2 norm of the differences squared. These calculations are implemented in the `Genetic.c` script, and switching between them can be done by setting the `fitness_case` parameter.

2.2 Encoding

In order to solve the problem using a GA, the individuals needed to be encoded in binary, thus the restricted search space has been discretised. This yielded 6 integer search spaces of the form $[0, 2^n - 1]$, on which points are binary numbers. Individuals have the 6 genotypes listed in Fig 2, which need to be converted to phenotypes when predicting with the model (calculations in the last column). Due to the possible sizes of the genotypes the types of these chromosomes must be `unsigned long int`. Code fragments about the encoding can be seen on Listing 1.

```
typedef struct {
  unsigned long int x0;  unsigned long int phi;  unsigned long int lambda;
  unsigned long int mu;  unsigned long int sigma;  unsigned long int delta;
} Genotype;

typedef struct {
  double x0;  double phi;  double beta;  double lambda;
  double mu;  double sigma;  double delta;
} ODE_Parameters;

ODE_Parameters GenToPhen(Genotype gene){
  ODE_Parameters params;
  params.x0 = gene.x0 * X0_RES;
  params.phi = (gene.phi * PHI_RES * 0.000000001) - PHI_OFFSET;
  params.lambda = gene.lambda * LAMBDA_RES * 0.00001;
  params.mu = gene.mu * MU_RES * 0.0000001;
  params.sigma = gene.sigma * SIGMA_RES;
  params.delta = gene.delta * DELTA_RES;
```

```
  return params;
}

Genotype PhenToGen(ODE_Parameters params){
  Genotype gene;
  gene.x0 = params.x0 / X0_RES;
  gene.phi = (params.phi + PHI_OFFSET) / (PHI_RES * 0.000000001);
  gene.lambda = params.lambda / (LAMBDA_RES * 0.00001);
  gene.mu = params.mu / (MU_RES * 0.0000001);
  gene.sigma = params.sigma / SIGMA_RES;
  gene.delta = params.delta / DELTA_RES;
  return gene;
}
```

Listing 1: Encoding Listing (Snippets from Predict.c and Genetic.c)

2.3 Random Number Generation

The built in random number generator in C has been criticised for being linear congruential, and therefore the rand1 portable random number generator from [3] has been used to generate sufficiently stochastic values. The procedure is used via the API on Listing 2.

```
void randomize(void) {idum = time(NULL); if(idum > 0) idum = -idum;}
float uniform(void) {return ran1(&idum);}

unsigned char random_bit(void){ unsigned char f, s;
    do { f = 2*ran1(&idum); s = 2*ran1(&idum); } while (f == s);
    return f;
}

unsigned long int ULNGran(int size){ register unsigned char i;
    unsigned long int oneU = 1U, base = 0U;

    for(i=0; i < size; i++) if(random_bit()) { base = oneU; break; }
    for(i++; i < size; i++){ base = base << 1;
        if(random_bit()) base = base | oneU;
    }
    return base;
}
```

Listing 2: Random Number Generation Listing (Snippets from RandomBits.c)

2.4 Selection

5 selection procedures have been implemented in order to be used by the GA for selecting parents to produce offsprings for the next generation. Switching between these procedures can be done using the `select_case` parameter, and it is handled in the following wrapper function on Listing 3. The cases have been purposefully ordered from most exploitatory to exploratory. For a discussion of these procedures see [2] and [1].

```
void Selection(int i, Genotype * parents, unsigned short pop_size, Genotype * pop,
    double * fit, unsigned short k){
  switch (i) {
    case 1:
      RouletteWheelSelection(parents, pop_size, pop, fit);
      break;
    case 2:
      StochasticUniversalSampling(parents, pop_size, pop, fit);
      break;
    case 3:
      RankSelection(parents, pop_size, pop, fit);
      break;
    case 4:
      RandomSelection(parents, pop_size, pop);
      break;
    default:
      if(k > pop_size)
```

```
        k = pop_size;
      TournamentSelection(parents, pop_size, pop, fit, k);
  }
}
```

Listing 3: Selection Listing (Snippets from Selection.c)

2.5 Crossover

3 crossover types have been explored, all requiring two parents (One-Point, Two-Point and Uniform Crossover) implemented using bitwise operations. Switching between these can be done similarly to the parent selection with the `crossover_case` parameter. The three methods are modified versions of the ones given in the course that work with `unsigned long int`-s but do not generate invalid individuals. This is enforced by passing in the maximum number of bits for each of the parameter to the functions.

2.6 Mutation

4 mutation operators have been implemented usign bitwise operations, 2 of which are modified versions given in the course. These can be seen on Listing 4, and the function names are quite telling. Note, however, that `BitFlipMutation` is a mutation where each bit has a probability to be flipped, and that `RandomResetMutation` and `BitSwapMutation`, do not require a probability check and will occur in all cases.

```
void SingleBitFlipMutation(unsigned char len, unsigned long int *f, double prob){
  if(uniform() < prob){
    unsigned char p = uniform() * (len - 1); unsigned long int mask = 1 << p;
    *f = (*f) ^ mask;
  }
}

void BitFlipMutation(unsigned char len, unsigned long int *f, double prob){
  unsigned long int mask = 1; register unsigned char i;
  for(i = 0; i < len; i++) if(uniform() < prob) *f = (*f) ^ (mask << i);
}

void RandomResetMutation(unsigned char len, unsigned long int *f){
  unsigned long int mask = 1; register unsigned char i;
  for(i = 0; i < len; i++) if(uniform() < 0.5) *f = (*f) ^ (mask << i);
}

void BitSwapMutation(unsigned char len, unsigned long int *f){
  unsigned char p = uniform() * (len - 2) + 1; /* p \in [1, len-2] */
  unsigned char q = uniform() * (len - 1 - p) + 1; /* q \in [1, len-1-p] */
  unsigned long int xor = ((*f >> p) ^ (*f >> q)) & ((1U << 2) - 1);
  *f = *f ^ ( (xor << p) | (xor << q));
}
```

Listing 4: Mutation Listing (Snippets from Crossover.c)

2.7 Main Code

Since all operations needed for the GA (Selection, Crossover, Mutation, Fitness) have been abstracted away, and have wrapper functions to switch between the cases, the main GA loop (Listing 5) is concise. All arrays are initialised before the main loop and are used throughout the whole run to save execution time by mallocs, and to decrease the chance for a memory leak. After each iteration, we keep note of the fittest individual seen so far, log, check for convergence, but also possibly alter the parameters.

```
  // ... Initialise Parameters, and first population
  while(iter < n_iter && unchanged_counter <= unchanged_max){
    // For Half the population do
    for (size_t i = 0; i < pop_size / 2; i++) {
      // select two individuals from old generation for mating
      Selection(select_case, pars, pop_size, pop, fit, k);

      // Combine the two individuals to give two offspring
      GetOffspings(crossover_case, pars, offsprings, crossover_prob);
```

```
      // Mutate
      MutateOffsprings(mutation_case, offsprings, mutation_prob);

      // insert offspring in new generation
      new_pop[i*2] = offsprings[0];
      new_pop[i*2 + 1] = offsprings[1];
    }

    // switch population and new population
    temp = pop;
    pop = new_pop;
    new_pop = temp;

    // update Fitness scores and report fittest
    for (size_t i = 0; i < pop_size; i++) {
      fit[i] = Fitness(fitness_case, xt, pop[i]);
      if (fit[i] < best){
        best = fit[i];
        best_gene = pop[i];
      }
    }
    // ... Log and Convergence Check
    iter++;
  }
  // ... Cleanup
```

Listing 5: Main GA Loop Listing (Snippet from Predict.c)

One of the parameters passed to the function is the `autopilot` option, which when turned on alters the selection, crossover, mutation cases and increases the probabilities. If autopilot is turned on, and the fittest individual is unchanged for some iterations, altering the cases, may get the algorithm unstuck, as they increase search range, as some cases are more exploratory than others, and some even random. On Table 1 we can see all the parameters and their functionality.

Parameter	Functionality	Range	Default
autopilot	Determines if the GA can change parameters if stuck.	0, 1	1
n_iter	Maximum number of generations.	1 - ∞	500
pop_size	Number of Individuals in each generation.	1 - ∞	1000
unchanged_max	Number of Generations with no better fittnes before program is stopped.	1 - ∞	40
fitness_case	Determines the fitness function.	0, 1	0
select_case	Determines the parent selection function.	0 - 4	0
crossover_case	Determines the crossover function.	0 - 2	0
mutation_case	Determines the mutation function.	0 - 3	0
crossover_prob	The probability in uniform crossover.	[0, 1]	0.1
mutation_prob	The probability of single and multiple bitflip mutations.	[0, 1]	0.1
k	The number of individuals selected at random for tournament (in selection).	1 - ∞	100
path	Path for logging.	-	"logs/fittness.csv"
fittest_path	Path for parameter and residual logging.	-	"logs/residues_fittest.csv"

Table 1: Parameters in main.c

3 Results

The algorithm has been tested in each case, and the outputs can be found in the `logs` folder. Here we report the two cases where autopilot has been turned on, once with the fitness defined in Eq 5, referred to as 'Max Norm', and with the fitness in Eq 6, referred to as "Weighted Norm". Both of these runs used a population size of 1000, starting with a single bit mutation, one point crossover, both with probability 0.1, and tournament selection with $k = 100$. Both cases used 500 as `max_iter`, but were stopped if the population did not change for 100 generations. At 50 unchanged generations, the autopilot was turned on. The predictions of the model have been compared to the actual values in both cases (see Figs 3, 4).

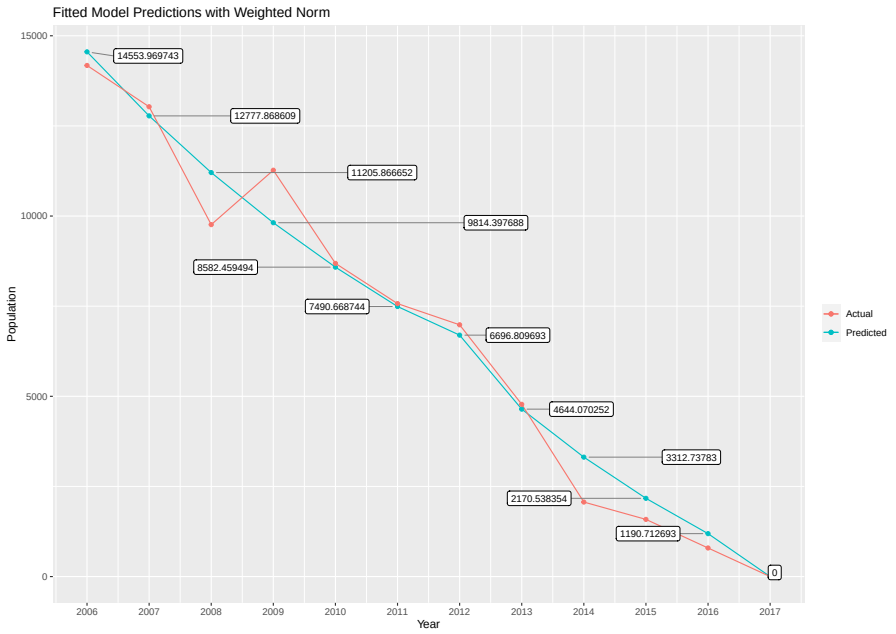

Figure 3: Actual vs. Predicted with Weighted Norm

We can see that both cases achieved good performance, and fit the observations quite well. For the Weighted Norm the best fitness was 6581780.082720, which was achieved in the 19th generation, and for the Max Norm the best fitness was 1927400.238402 reached in the 17th generation, although other versions of Max Norm reached lower fitness values (best with 1212009.836380), these versions gave a negative population prediction in year 2016.

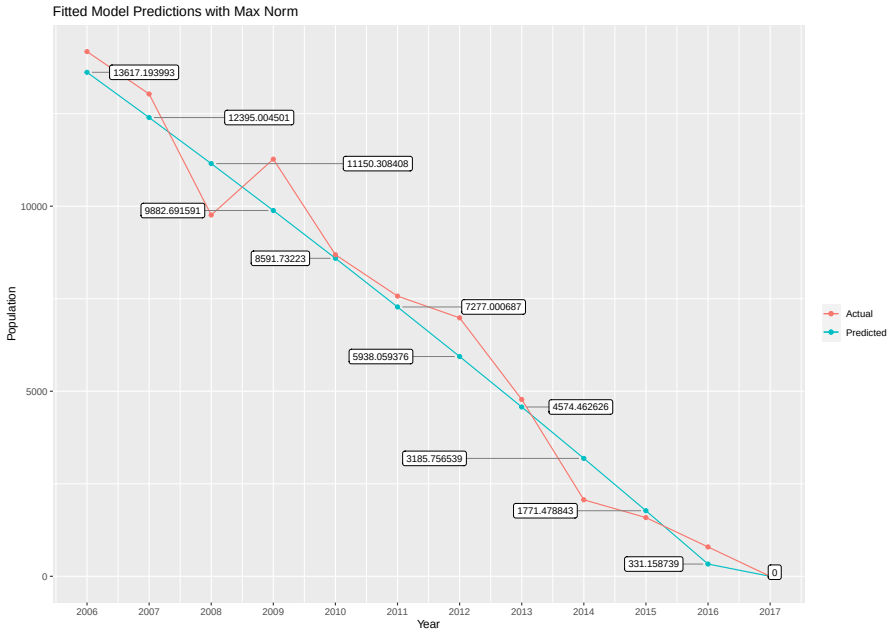

Figure 4: Actual vs. Predicted with Max Norm

We can also have a look at the residuals, but there are no signs of heteroscedasticity. However, it is more obvious that Weighted Norm fit better to the data.

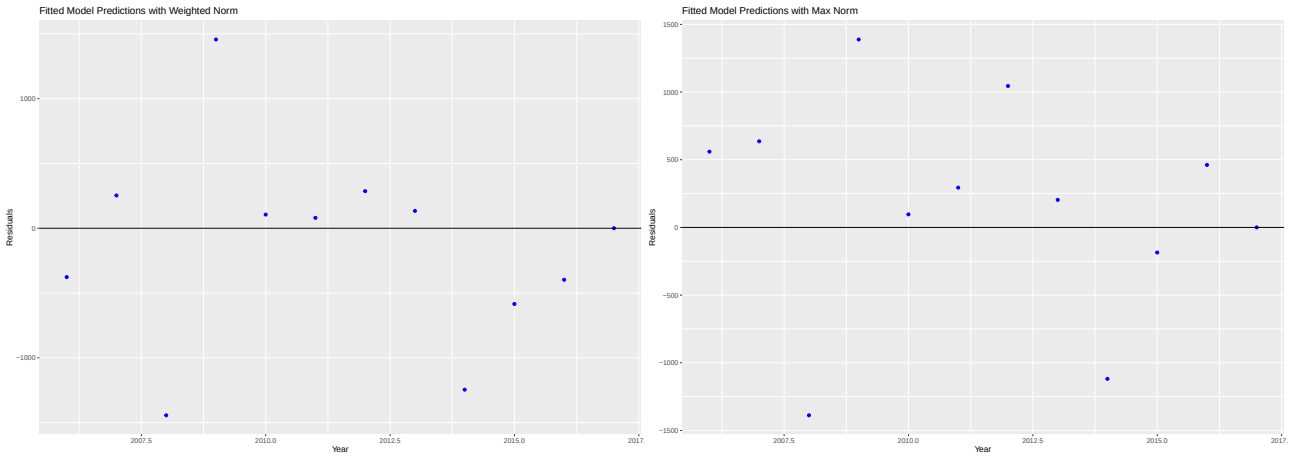

Figure 5: Residuals of Weighted (left) and Max (right) Norm

The found parameters can be seen in Table 2, and it is interesting to observe that even though $\lambda, \mu, \sigma, \delta$ are quite different in the two cases, the predictions of the two cases are similar.

Parameter	x_0	β	ϕ	λ	μ	σ	δ
Max Norm	13617.193993	0.000024	-0.033068	1459.542854	11.458837	627.995514	23878.444777
Weighted Norm	14553.969743	0.000024	-0.137817	723.953179	0.740590	138.581380	6917.783135

Table 2: Parameters by Fitness Function

4 Conclusions and Future Work

All in all the GA has been able to find 2 satisfying parametrisations of the migration model, and it achieved them in only a few generations. Finding the right hyperparameters to the GA is in itself another optimisation task as most runs tended to converge to a solution quickly and were unable to get unstuck in the specified number of iterations, which is why increasing the stochasticity mid-way seemed the most sensible option.

There are also a few mutation operators that would be worth adding as well, for example inversion and reordering, both working on continuous chunks of a binary number. Moreover, the selection operators could be extended as well, for example with elitism the idea is to carry over a part of the old population, and with boltzmann selection we could control the diversity of the search space by a hyperparameter decreasing over iterations.

Of course it is possible that better parametrisations exist (in terms of their fitness score), but from the limited data size, one can reasonably wonder how well should the model fit these few datapoints. That said with further analysis of the hyperparameters, the finetuning of the autopilot option, and through repeated running of the algorithm, one could find better fits.

References

[1] Genetic algorithms - parent selection. `https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm`. Accessed: 04/01/2022.

[2] Chetan Chudasama, SM Shah, and Mahesh Panchal. Comparison of parents selection methods of genetic algorithm for tsp. In *International Conference on Computer Communication and Networks CSI-COMNET-2011, Proceedings*, pages 85–87. Citeseer, 2011.

[3] Brian P Flannery, William H Press, Saul A Teukolsky, and William Vetterling. Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 24(78):36, 1992.