Rapport EI DEMAILLE COURNIL RABEUX REBOLA CAHITTE

Mathéo Cahitte, Clément Cournil-Rabeux, Melkior Demaille, Clément Rebola

January 2026

1 Introduction

This project focuses on optimizing spatial trajectories of DNA sequences using genetic algorithms, with a specific application to plasmids. While the exact geometry of these plasmids is unknown, they are topologically constrained to form closed loops. The trajectory is modeled using rigid-body transformations for each of the 16 possible dinucleotide steps—defined by three rotational angles per pair with a fixed translation. Starting with a reference table derived from experimental data, our objective is to iteratively refine these parameters to find a solution that ensures loop closure while remaining biologically consistent with the reference model.

Research into plasmid biology indicates that circularity is not solely a product of intrinsic geometry (least-energy conformation) but is enforced by biological mechanisms—either through direct replication of cyclic strands or enzymatic ligation of compatible ends. Consequently, a static 'universal table' of dinucleotide parameters, derived from linear DNA models, is insufficient to predict loop closure for specific plasmids.

Furthermore, developing a universal predictive model similar to AlphaFold is unfeasible due to a lack of training data; we lack long reference sequences with known 3D atomic coordinates. Since the only guaranteed topological constraint is that the plasmid must form a closed loop, we redefine the objective: rather than seeking a universally valid parameter set, we utilize the genetic algorithm to find a sequence-dependent table that minimizes the gap between the strand's ends, effectively enforcing closure without assuming a specific spatial shape.

In this report, we have chosen to begin by presenting and justifying our theoretical and methodological choices, then explain our algorithm in more formal terms, and finally—with the aid of diagrams—explain how we tested it and the conclusions we drew from our tests.

2 Methodology

2.1 Why use a genetic algorithm ?

Optimizing the first two angles for each of the 16 dinucleotides results in a high-dimensional search space of 32 parameters. Given the cumulative nature of these transformations over thousands of base pairs, the computational complexity precludes an analytical solution. Furthermore, the objective function is likely non-convex, containing numerous local minima that would trap standard gradient descent methods. However, since the spatial trajectory is a continuous—albeit highly sensitive—function of these angular parameters, a genetic algorithm is the optimal choice. It effectively navigates the complex landscape, avoiding local traps to iteratively converge upon the most promising global solutions.

2.2 Which genesis ?

We have no preconceived notions about the optimal table, and the model only provides bounds to be respected for each of the parameters in the rotation table. To avoid selection bias, we wanted to start with the most "random" population possible. We therefore want to maximize Shannon entropy:

$$H(X) = -\int_{\mathcal{D}} f(x)\ln(f(x))\,dx \tag{1}$$

where $f(x)$ represents the probability density of the random variable X on $\mathcal{D}$.

That is why we decided to follow a uniform law (respecting the given uncertainty limits) for the initial draw (genesis). The uniform law maximizes entropy on a segment, without additional knowledge.

2.3 Which fitness ?

The subject is based on plasmids. Our understanding is that for each plasmid, we need to arrive at a table that gives the trajectory closest to a looping chain, within the standard deviations given by the model. From a pragmatic point of view, at plasmid size scales, the effects of intermolecular forces or temperature must curve identical dinucleotides differently, an effect that cannot be taken into account here (see introduction).

For the fitness function, the first idea that comes to mind is to measure the distance between the first and last bases in space. The closer the ends are, the more convincing it is that the table best models the path of the plasmid. Thinking a little further, since the plasmid must loop, what interests us is the difference between the position of the first base of the sequence (the origin) and the position where it would end up if it traveled through the loop to the end. We can even add the first k bases to the end of our sequence and minimize the distance between the position of each of them at the beginning and end of the chain. Measuring the distance over at least two bases also allows us to take the table into account when matching the ends. It is more desirable to loop in a way that is compatible with the rotation table than to find our ends superimposed but with some angle of incidence—this can be likened to a condition of continuity C^k.

We had an idea to further verify the compatibility of a table with a plasmid, but it pointed to the limitations of the model. The table allows us to visualize the circularization of the plasmid from a given linear representation; however, we currently have no guarantee that it will remain valid if the customer chooses another gene as the start of the plasmid. In other words, we would need to test the quality of the looping of the path obtained with a different "start/end" of the plasmid.

So we added to our function's parameters the number of cuts other than the one provided that will need to be tested (distributed evenly). Obviously, this multiplies the execution time of the fitness function and makes it much more demanding: we are not even sure that there is an acceptable solution in general with the model provided.

2.3.1 Definition of the fitness

The quality of an individual (a rotation table) is defined by its ability to properly close the plasmid nucleotide sequence onto itself. To ensure continuity of a higher order than simple topological closure, we impose an overlap of k bases.

Let us consider a trajectory composed of points $\mathbf{P}_i \in \mathbb{R}^3$. The closure distance for a given configuration is defined by:

$$d(\mathcal{P}, k) = \sqrt{\sum_{i=0}^{k-1} \|\mathbf{P}_i - \mathbf{P}_{N+i}\|^2}. \tag{2}$$

To overcome dependence on the initial cut-off point of the plasmid, we evaluate the quadratic sum of the score on m different starting points, distributed uniformly along the sequence:

$$\text{Fitness} = \sqrt{\sum_{j=0}^{m} d_j^2}, \tag{3}$$

where d_j is the score obtained for the j-th cut. This metric penalizes not only the spatial distance between the endpoints, but also slope breaks at the junction, thus acting as a condition of geometric regularity.

Compromise between minimizing distance and invariance

The choice of the number of cuts m (nbcuts) defines the nature of the optimization:

- **Local optimization ($m = 0$)**: We seek to minimize $\mathcal{F} = d_0$. Convergence is rapid because the space of admissible solutions is vast.

- **Structural optimization** ($m \geq 1$): We seek to minimize $\mathcal{F} = \sqrt{\sum_{j=0}^{m} d_j^2}$.

2.4 Which selection ?

To move from one generation to the next, we tried several selection methods, comparing them to determine which one converged best and fastest. We wanted to leave the proportion of individuals retained from one generation to the next as an argument.

Elite selection—selecting the k individuals with the best scores among the n in a generation—is a good start. Admittedly, we are giving up some of the randomness that makes genetic algorithms so effective, but until a good way of managing this randomness is found—one that is well suited to the sensitivity of trajectories and does not lose its value when considering mutations—this selection remains valid. It is a good reference point for determining whether another method is worthwhile or not.

To adapt tournament selection to any proportion of survivors, we have chosen to allow an individual to be drawn several times. This means that those with a much better score will have a better chance of being represented. The motivation behind this is that an "outsider" will always have a chance of being selected, and a very good candidate will be less likely to be overlooked. As this option is often ineffective when combined with the rest of the algorithm, we have gradually modified it so that it always retains a percentage of the best candidates.

We also introduced roulette selection. The problem with this method is that it depends not only on the order of the solutions, but also on the numerical value of their score. Thus, a priori, composing this score with any increasing function gives us a valid way to spin the roulette wheel. To keep things simple, we initially used the result of the fitness function (Euclidean distance) directly.

To refine roulette selection, we drew inspiration from simulated annealing. Indeed, evolutionary pressure can be seen as a "spring" that pushes the ends of the plasmid to come together. In this sense, a potential in x^2 (where x happens to be the fitness function) seems appropriate. For the Boltzmann factor, a temperature must be set to favor good solutions without directly overwhelming those that are close to them: this is the "exponential roulette" selection. To ensure that weights are not interpreted as zero by Python, we have chosen to divide them by the minimum weight (a kind of normalization, as the random.choose function simply requires positive weights, not necessarily a unit sum). The weight function is given by:

$$W_i = exp\left(\frac{S_{min}^2 - S_i^2}{T_0}\right).$$

Taking further inspiration from annealing, we may want to decrease the temperature with each generation to refine convergence. However, we must be careful not to be too abrupt, so as not to fall directly into a local extremum that is too poor. Conversely, not decreasing the temperature quickly enough renders this parameter useless. As the range of options was too broad, we reduced it to a minor affine decrease in temperature, choosing parameters that were suitable for the experiment. The weight function at generation k is given by:

$$W_i(k) = exp\left(\frac{S_{min}^2 - S_i^2}{\max(T_0 - kC, T_{min})}\right).$$

A final version of the annealing roulette wheel is the "Normalized annealing" selection. Here, to adjust the temperature decrease, we use the best score for a given generation: the lower the score, the colder the temperature, and therefore the greater the cuts will be. Indeed, if for an individual i we have $S_i = S_{min} + \Delta S_i$, then $S_i^2 \sim_0 S_{min}^2 + 2S_{min}\Delta S_i$. We then have the following for the weight function:

$$W_i = exp\left(\frac{S_{min}^2 - S_i^2}{2 \times S_{min}}\right) \sim_0 exp(-\Delta S_i),$$

et :

$$W_i = exp\left(\frac{S_{min}^2 - S_i^2}{2 \times S_{min}}\right) \sim_\infty exp\left(\frac{-S_i^2}{2S_{min}}\right).$$

We then make a selection that does not cut too quickly close to the best, but leaves little chance for the very bad cases to continue. This method is theoretically very elegant, but in practice we did not obtain very good results with our algorithm. It will therefore not be developed much further.

Next, we implemented roulette-ranking. Here, we even forget the value of the fitness function and only retain the order it gives—however, we still have the same problem of possible composition with any increasing function. We start by drawing with a probability proportional to the complement of the rank ($W_i = n - rank(i)$).

In a variant of roulette-rank selection, we use a geometric distribution of probabilities. We take a selection pressure of $0 < q < 1$ as the ratio, and adjust the ratio of the probabilities of choosing the k-th over the k+1-th. This allows for a slightly more controlled distribution of "outsiders." However, it still seems a shame to "forget" some of the information provided by fitness, but less randomness can make it more powerful. Here:

$$W_i = q(1-q)^{n-rang(i)-1}.$$

For roulette selection, we have an algorithm like this:

```
SelectionRoulette(Population P, Rate τ, Mode, Temp T) :

        N_sel ← ⌊size(P) × τ⌋
        S_min ← min_{ind∈P}(ind.score)
        S_total ← Σ_{ind∈P}(ind.score)
        WeightList ← []

        For each individual ind in P :
        S ← ind.score

        If Mode == Classic :
        W ← 1 − (S/S_total)
        Elif Mode == Exponential :
        W ← exp((S²_min − S²)/T)
        Elif Mode == Normalized_Exp :
        W ← exp((S²_min − S²)/S_min)

        Add W to WeightList

        // Weighted Draw with replacement
        Geniteurs ← Draw(Population, Weights=WeightList, k=N_sel)

        Return Parents
```

2.5 Which reproduction ?

The question here is whether individuals with higher rankings should reproduce more than those with lower rankings, even after the selection stage. Indeed, testing several variations of the best candidate from one generation to the next can be interesting. We could play around with several parameters: the number of opportunities a survivor has to reproduce, and the weight and type of its impact on the genes of its descendants. The first aspect is partly taken into account by the selection stage, so we added it as an option (the "fish" parameter of the algorithm). As for the second aspect, here are the successive methods we chose.

Reproduction between two individuals I_1 and I_2, with respective scores f_1 and f_2, generates a descendant whose genes are a linear combination of those of the parents.

For each gene x_i, the new value x'_i is calculated by:

$$x'_i = \alpha \cdot x_{i,1} + (1-\alpha) \cdot x_{i,2}, \tag{4}$$

where the weighting coefficient α is determined by the relative importance of the scores:

$$\alpha = \frac{f_2}{f_1 + f_2}. \tag{5}$$

If $f_1 + f_2 = 0$, we define $\alpha = 0.5$ by default to obtain an equitable mixture. This method allows us to create a new individual located on the segment connecting the two parents in the solution space, with a bias in favor of the parent with the highest score (subject to the definition of the score).

Another method for creating new genes is to associate x'_i with $x_{i,1}$ with probability α or $x_{i,2}$ with probability $(1 - \alpha)$. This avoids mixing important genes and obtaining too many "malformed" offspring.

However, we realize that this method can be unstable. We therefore decide to combine it with the previous one, adding a coupling parameter β such that:

$$\mathbb{P}(x'_i = x_{i,1} * (1 - \beta) + (\alpha \cdot x_{i,1} + (1 - \alpha) \cdot x_{i,2}) * \beta) = \alpha \tag{6}$$

$$\mathbb{P}(x'_i = x_{i,2} * (1 - \beta) + (\alpha \cdot x_{i,1} + (1 - \alpha) \cdot x_{i,2}) * \beta) = 1 - \alpha. \tag{7}$$

In the next section, we will see which values for these parameters yielded the best results.

2.6 Which mutation?

Choosing the frequency and amplitude of mutations is essential. If the mutation frequency is too high, we lose the advantage of inheritance; if it is too low, we encounter the pitfalls of gradient descent—especially if the genomes are mixed with each new individual, with a tendency toward uniformity. We first chose to distribute the amplitude of mutations around the initial values with a normal distribution (truncated to avoid exceeding the model's limits), and to have an equal probability for each gene to mutate. In addition, we played with the width of the normal distribution and the probability of mutations over generations to refine the solutions once a good candidate was found. Later, we added a low probability of large mutations: the simple normal distribution allows us to move towards a local minimum, so we had to add a "strongly unexpected" aspect to ensure that we did not orbit indefinitely around a mediocre solution. Here is the formalization.

For each gene, if a random draw $u \sim \mathcal{U}(0, 1)$ is less than the mutation rate m, the new value v'_i is initially calculated as follows:

$$v'_i = \max\left(\mu_i - \sigma_i, \min\left(\mu_i + \sigma_i, v_i + \delta\right)\right) \tag{8}$$

Where:

- $\delta \sim \mathcal{N}(0, (\sigma_{mut} \cdot \sigma_i)^2)$ represents the mutation jump.
- μ_i and σ_i are the reference mean and standard deviation (from `Rot_data`).
- v_i is the value before mutation and v'_i is the value after mutation.

To model rarer and more significant mutations, we add a second mutation rate $m' << m$ and a standard deviation $\sigma' > \sigma_{mut}$. We then apply the same process, so the children's genes go through two different mutation stages.

3 The Genetic Algorithm

3.1 Algorithm Parameters

The behavior of the genetic algorithm is governed by a set of hyperparameters that define the evolution strategy.

- **Population size (`nb_individuals`)**:
- **Number of iterations ($nb_iterations$)**: The number of generations.
- **Fitness function** (`fitness()`): Evaluation function to be minimized.

- **Selection rate** ($selection_rate$): percentage of the population surviving from one generation to the next.
- **Selection method** (`select(`$selection_rate$`, fitness)`): Depends on the selection rate and the fitness function. In this example, the following strategies are implemented (see part 1):
  - *Elitist*
  - *Tournament*
  - *Roulette (with variants)*:
    - Classic Roulette (`selection_roulette`)
    - Exponential Roulette (`selection_roulette_exp`)
    - Normalized Exp. Roulette (`selection_roulette_exp_normal`)
- **Crossing method**: Defines how two parents generate a child. In our implementation, this is a **weighted barycentric crossing**: each parameter of the offspring is a weighted average of the parents, where the parent with the best score has more influence (stronger weight) on the final result (see 2.4).
- **Generation method**: Generates the initial population according to a uniform distribution.

3.2 General pseudo-code

```
P ← GeneratePopulation(number_of_individuals)

For k ranging from 1 to number_of_iterations:

      // 1. Evaluation and Selection
      scores ← fitness(P)
      Parents ← select(P, selection_rate, scores)

      // 2. Reproduction
      P_new ← Parents

      While size(P_new) < nb_individuals :
            p1, p2 ← random_choice(Parents)
            Child ← crossover(p1, p2)
            Add Child to P_new

      P ← P_new

Return BestIndividual(P)
```

3.3 Complexity calculation

Let n be the number of individuals in the population, G the number of generations, and τ the selection rate ($0 < \tau < 1$). Let C_{fit} be the time cost of evaluating the fitness function for an individual (calculation of the 3D trajectory).

3.3.1 Complexity of the Reproduction stage (Crossover and Mutation)

At each iteration, we have a population of n individuals. We therefore have a number of offspring to generate that is $n_{offspring} = \lfloor n(1-\tau) \rfloor$

The creation of a child (crossing of two vectors of fixed size + mutation) is an elementary operation with a constant cost $\mathcal{O}(1)$ relative to n. We therefore have the complexity for one generation:

$$C_{repro} = \mathcal{O}(n(1-\tau))$$

With τ set as a hyperparameter, we have

$$\boxed{C_{repro} = \mathcal{O}(n)}$$

3.3.2 Complexity of the Selection step

- **Tournament Selection**: We randomly draw a pair of individuals n times and compare them in time $2C_{fit} = \mathcal{O}(1)$.
  For each generation, we have the complexity:

$$C_{select} = \mathcal{O}(n)$$

- **Elitist Selection**:

  First, we must sort the individuals into $C_{fit}\mathcal{O}(n \log n)$, i.e., into $\mathcal{O}(n \log n)$. Then, we retrieve the first $\tau \cdot n$ individuals, in $\mathcal{O}(n)$.
  For each generation, we have the complexity:

$$C_{select} = \mathcal{O}(n \log(n))$$

- **Selection by Rank** (Real or Geometric):
  First, we sort the population to find the ranks: $\mathcal{O}(n \log n)$.
  Then, we assign a weight to each individual according to the sort ($\mathcal{O}(n)$) .
  Next, we must draw the parents according to these probabilities. Since weighted selection requires a dichotomous search ($\mathcal{O}(\log n)$ per individual), this phase costs $n \cdot \tau \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$.

  For each generation, the total complexity is therefore

$$C_{select} = \mathcal{O}(n \log n)$$

- **Roulette Selection** (Classic or Exponential):
  We scan the population to calculate the weights (linear or exponential sum) and the cumulative distribution in $\mathcal{O}(n)$. We then perform the weighted selection of parents in $n \cdot \tau \times \mathcal{O}(\log n)$.
  For each generation, we have the complexity:

$$C_{select} = \mathcal{O}(n \log n)$$

Only the Tournament method is in $\mathcal{O}(n)$ because its random selection is uniform (for a draw: $\mathcal{O}(1)$ if uniform vs $\mathcal{O}(\log(n))$ if weighted).
All other methods (Elitist, Rank, Roulette) include either a sorting step or a weighted draw, leading to a quasi-linear complexity $\mathcal{O}(n \log n)$.

3.3.3 Overall Complexity

The total complexity over G generations sums the costs of evaluation, selection, and reproduction. There are two cases:

1. **With Tournament Selection**:
   Selection is linear ($\mathcal{O}(n)$). The total cost is:

$$C_{total} = G \times (\underbrace{n \cdot C_{fit}}_{\text{Evaluation}} + \underbrace{\mathcal{O}(n)}_{\text{Selection}} + \underbrace{\mathcal{O}(n)}_{Crossover+mutation})$$

   Here, the terms are of the same order. Since the unit cost $C_{fit} \geq 1$, the evaluation term dominates mathematically:

$$\boxed{C_{total} = \mathcal{O}(G \cdot n \cdot C_{fit})}$$

2. **With other selections** (Elitist, Rank, Roulette):
   The selection is done in ($\mathcal{O}(n \log n)$) and we therefore have

$$\boxed{C_{total} = G \cdot (n \cdot C_{fit} + (n \log n))}$$

   In our context, the physical computation cost C_{fit} can become very high. As long as the population n remains reasonable ($C_{fit} \gg \log n$), the computation time is dominated by physics rather than sorting. We then find the same practical order of magnitude:

$$\boxed{C_{total} \approx \mathcal{O}(G \cdot n \cdot C_{fit})}$$

   In practice, $C_{fit} \approx k \cdot n_{cut} \cdot Plasmid_{size}$ where $k \approx \frac{1}{120000}(seconds)$, so for a population of 1000 individuals, 25 generations, and a plasmid size of 8000, the computation time is approximately 0:30-1 hour.

4 Results and model selection

4.1 Fitness selection

We compared different versions of fitness:

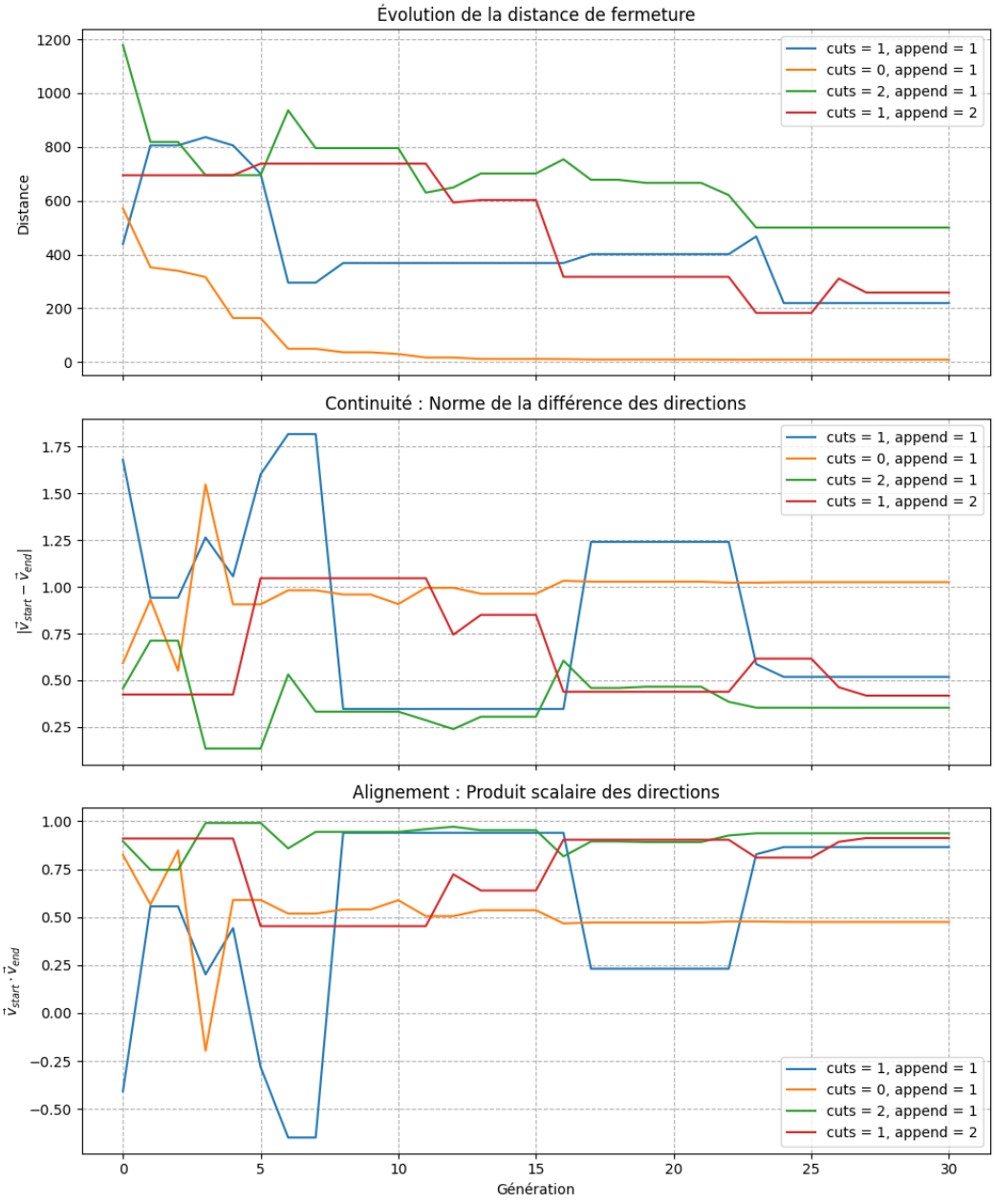

Figure 1: Comparison of fitness functions according to the number of cuts. Fixed parameters: `nb_individuals = 150`, `nb_generations = 24`, `selection_rate=0.3`, `selection_type = \elitist"`

Unsurprisingly, selection based solely on the overlap of the two ends of the plasmid yields the best results for this task. For alignment and continuity, the other fitness functions ($nb_cuts >= 1$) perform better. For $nb_cuts = 2$ and $append = 1$, performance is better in continuity and alignment than for all the others, although $nb_cuts = 1$ and $append = 2$ come close.
We can conclude that, in order to obtain a satisfactory result according to indicators other than just the distance between the ends, an advanced fitness function with several rejoining bases and several cuts is a better solution. However, more advanced fitness functions are computationally expensive, so the choice will depend on the goal and constraints:

If a quick result is desired, or if only the proximity of the endpoints is of concern, the basic fitness function is preferable.
Otherwise, additional cuts and rejoining points should be added.

4.2 Selection choice

We first preselected the four methods that seemed to perform best. Methods such as real rank or classic roulette gave us poor results early on in the first simulations. They were therefore not considered further.

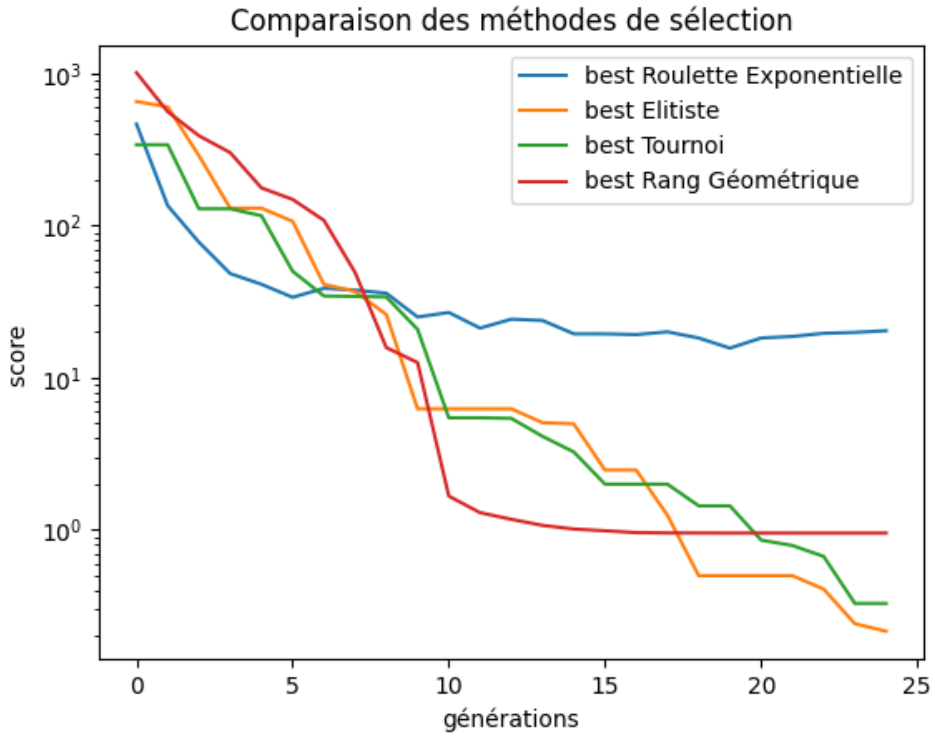

Figure 2: Comparison of performance according to different selection methods. Fixed parameters: $number_of_individuals = 150$, $number_of_generations = 24$, $selection_rate = 0.3$, and 'basic' fitness (0,1)

We note that although $roulette_exp$ converges more quickly than the others at the beginning, it then stagnates (hence the advantage of varying the temperature). `elitist` remains the method that ultimately gives the best results, so it is still our default choice. The `tournament` and *rang_geo* (geometric rank) methods also give satisfactory results—perhaps because they offer a choice of survivors close to that of `elitist`.
Since `tournament` is slightly less complex than other stochastic methods, it remains a relevant choice if you want to save computing time, get a good result, and retain some randomness. It is useful, for example, if you want to run a simulation with a particularly large population.

4.3 Choice of model parameters

4.3.1 Choice of selection rate

Figure 3: Comparison of performance according to different selection rates. Fixed parameters: $nb\_individuals = 150$, $nb\_generations = 20$,
`elitist` selection, and 'basic' fitness (0 cuts and 1 append)

4.4 Choosing the β parameter for mutation

β	**Fitness without cut-off**	**Fitness for a cut-off**
0 (stochastic selection)	3.11	35.48
0.3	0.03	31.75
0.7	0.15	4.32
1 (linear combination)	1.85	34.5

Table 1: Fitness values as a function of the parameter β (optima are boxed)

Based on these results, we decided to take $\beta = 0.3$ for basic fitness, and $\beta = 0.7$ for fitness with clipping.

5 Results obtained

5.1 Basic Fitness vs. With Cuts

We first ran a simulation with ($nb_individual = 500$) and ($nb_generations = 30$). As the calculation time was long, we decided to use the classic fitness function with ($nb_cuts = 0$ and $append = 1$). We used the best method tested previously, (`elitist`).

We compared it with a simulation using ($nb\_individual = 500$) and ($nb\_generations = 30$), also with the (`elitist`) selection method. For this second simulation, we used $nb\_cuts = 1$ and $nb\_append = 1$.

Here are the results

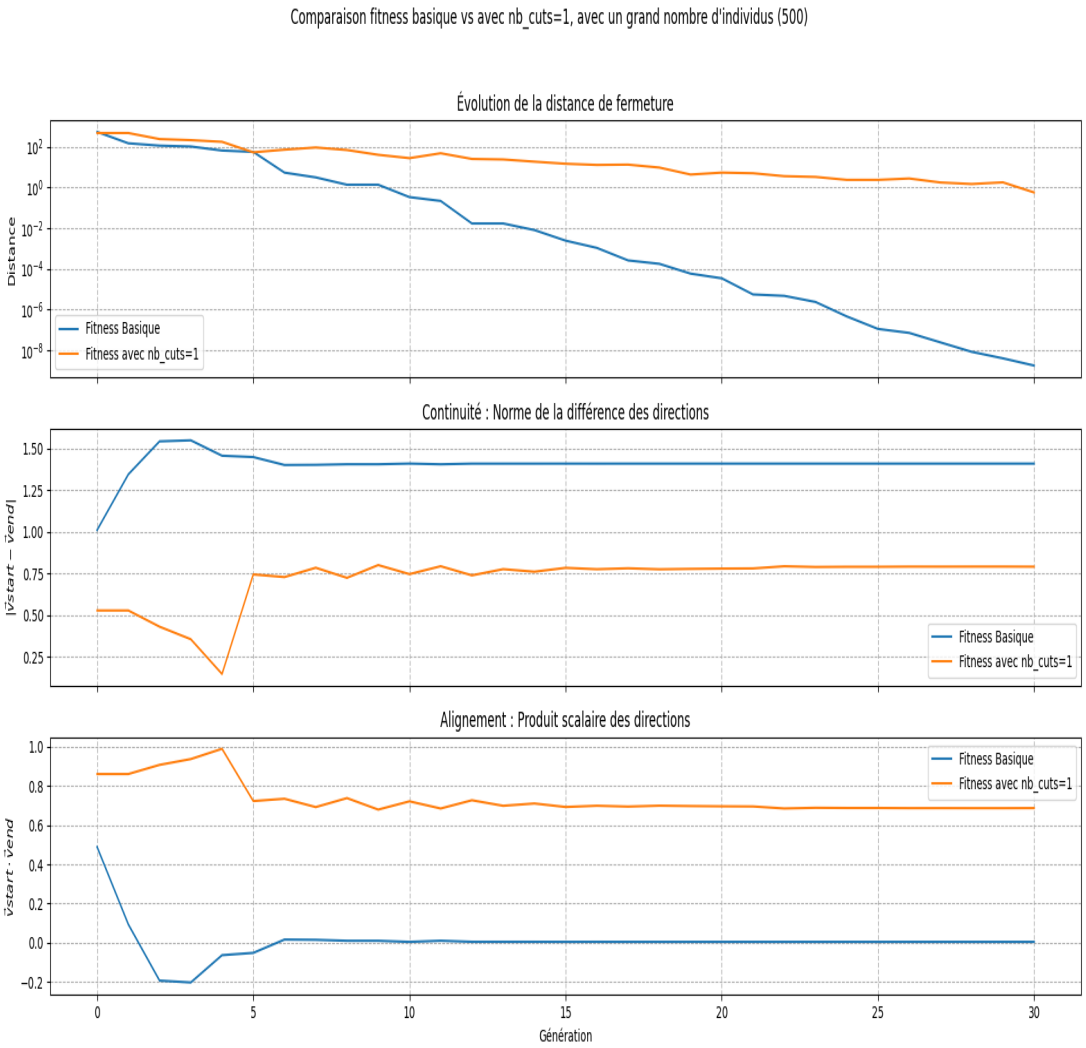

Figure 4: Results obtained from 500 individuals

We observe that basic fitness effectively minimizes the distance, but as expected, there is no alignment or continuity observed: the angles of the incident vectors are not close.

5.2 Simulated Annealing

Here is an example given for exponential roulette with affine annealing, with an additional cut and a reattachment test on two bases.

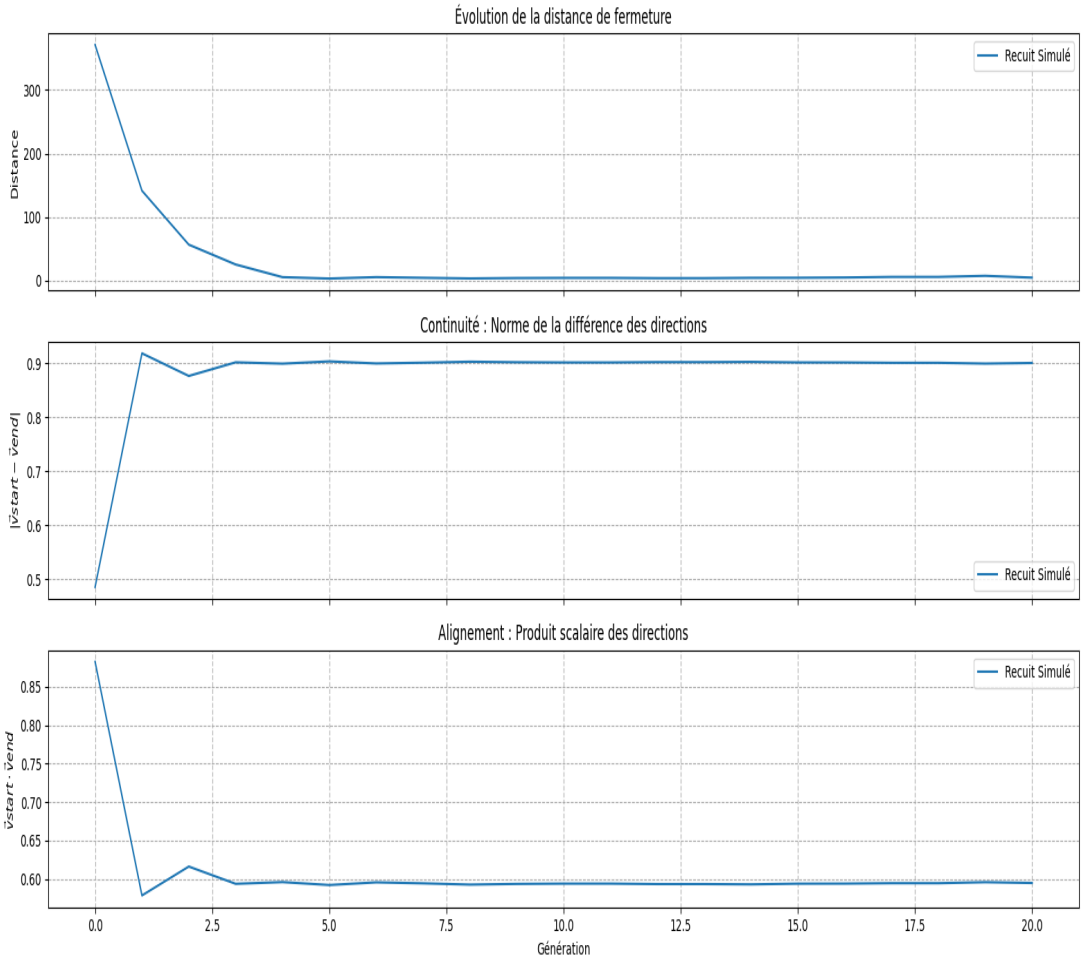

Figure 5: Results obtained using the simulated annealing method

We note that while the distance between the ends is correct, the angle is incorrect. Furthermore, the method converges very quickly to a fixed point, indicating that the temperature drops too quickly. More work needs to be done on this selection function. As the benchmarks sometimes take hours to run, we have not had the opportunity to do so.

6 Conclusion

In conclusion, the genetic algorithm provides very good solutions—provided that the population size and other hyperparameters are adapted to the problem at hand. The uncertainty surrounding what makes a solution good has led us to modularize our approach and design multiple benchmarks and varied representations and analyses of our results. The file (whose theoretical contribution is minimal) `executeur_comparaison_algos.py` is designed to make the majority of options that could be of interest accessible and user-friendly, both for an examiner and for a

potential user of the program. The file `README.md` presents in more detail the tools we built and used for this project.

Thank you for your attention.