Project – Discovering genetic algorithms
ENSEA/FAME Computer Science

We want to find the minimum of the function $f\colon [\![0,100]\!]^{10} \to \mathbb{R}$ defined by

$$f(n_1,\ldots,n_{10}) = \sum_{i=1}^{9} \sin(n_i n_{i+1}) + \sin(n_{10} n_1).$$

Question 1. What is the size of the configurations set? How many configurations do we have to evaluate in order to find the global minimum? Is there a unique solution?

Question 2. Apply the Monte Carlo method (described in Algorithm 1) to our problem.

Algorithm 1 Monte Carlo Algorithm

```
procedure MonteCarlo(n)
    x* ← random_configuration()
    min ← f(x*)
    for i = 1 to n do
        x ← random_configuration()
        fitness ← f(x)
        if fitness < min then
            x* ← x
            min ← fitness
        end if
    end for
    return (x*, min)
end procedure
```

Now, we would like to set up an evolutionary algorithm based upon genetic evolutionary theory and try to find a solution to our problem faster than the algorithm described above.

First of all, we have to define the **genotype** and the **phenotype*** of a configuration (see figure 1).

Question 3. Define a class `Individual` that contain its `genome` (here $(n_1,\ldots,n_{10})$) and its `fitness` (here $f(n_1,\ldots,n_{10})$). Code the evaluation function.

```python
import random as rand
import numpy as np

class Individual:
        def __init__(self, genome):
                self.genome = genome
                self.fitness = np.inf    #represent the infinity
                #evaluation of the configuration at his creation
                self.evaluate_fitness()
        def evaluate_fitness(self):
                self.fitness = ... #to be completed
```

*The genotype–phenotype distinction is drawn in genetics. "Genotype" is an organism's full hereditary information. "Phenotype" is an organism's actual observed properties, such as morphology, development, or behavior. *Wikipedia*

```
    #the following allows you to print a individual
    def __repr__(self):
        return '('+','.join(['{0}'.format(self.genome[i])
                        for i in range(10)])+')'
                        +' fitness:{0}'.format(self.fitness)
```

Question 4. Define a class `Population` that contain `list_individuals`. Implement a method `initialize_population(` that fills the list with `n` individuals taken randomly. Use the command `rand.randint(0,100)` to generate uniformly integers between 0 and 100 included.

```
class Population:
    def __init__(self, list_individuals):
        self.list_individuals = list_individuals
    def initialize_population(self, n):
        ... #to be completed
```

Question 5. In the class `Population`, define a method `best` that returns the position in `list_individuals` of the individual with the best fitness.

Question 6. Using the method of the class Population, explain how to perform the Monte Carlo method. Why is this idea very bad?

Question 7. In the class `Population`, define a method `worst` that returns the position in `list_individuals` of the individual with the worst fitness.

Question 8. In the class `Population`, define a method `random_individual` that returns an individual taken randomly in `list_individuals`.

Question 9. Define a function `crossover` that takes two individuals `father` and `mother` and returns a new individual `child` as follow:

- we take randomly an interval, named *locus*, $[a, b]$ in $[\![1, 10]\!]$.
- we copy the genome of `father`, $(n_1, \dots, n_{10})$, into the genome of `child` and we replace the sequence of genes $(n_a, \dots, n_b)$ by $(n'_a, \dots, n'_b)$ from the genome of `mother`.

See figure 2 for an example.

locus	$[2, 7]$									
father	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
$+$										
mother	n'_1	n'_2	n'_3	n'_4	n'_5	n'_6	n'_7	n'_8	n'_9	n'_{10}
$=$										
child	n_1	n'_2	n'_3	n'_4	n'_5	n'_6	n'_7	n_8	n_9	n_{10}

Figure 2: Crossover

Question 10. In the class `Population`, define a method `crossover_population(n)` that takes an integer and returns a new population of `n` individuals obtained by applying the function `crossover` to two individuals taken randomly in `list_individuals`.

Question 11. In the class `Individual`, define a method `mutate` that takes a real number `mutation_probability` in $[0, 1]$ and mutate its genome randomly as described in the algorithm 2. Don't forget to add the command `self.evaluate_fitness()` at the end of the method in order to update its fitness.

Algorithm 2 Mutation algorithm

```
procedure MUTATE(mutation_probability)
    for i = 1 to 10 do
        if np.random() < mutation_probability then
            n_i ←np.randint(0,100)
        end if
    end for
end procedure
```

Question 12. In the class `Population`, define a method `mutate_population` that takes a real number `mutation_probability` in $[0, 1]$ and mutate all the individuals in `list_individuals`.

Question 13. In the class `Population`, define a method `select` that takes another population `children` and replace its worst individual by the best individual in `children`.

We now have all the materials we need to implement a genetic algorithm to our problem.

Question 14. Define a function `genetic_algorithm` that takes (`population_size`, `mutation_probability`, `number_of_generations`) and returns an individual as follow:

1. **Initialization**: generate randomly a the population `parent` with `population_size` individuals
2. Do the following procedures `number_of_generations` times:
   (a) **Crossover**: build a new population `offstring` of `population_size` individuals by applying the method `crossover_population` to `parent`.
   (b) **Mutation**: mutate the population of children with `mutation_probability`.[†]
   (c) **Reduction/Selection**: replace the worst individual in `parent` by the best individual in `children`
3. Print the best individual in `parent`.

Question 15. Add the following method in the class `Population`:

```
def stat(self):
        #building of the list of fitness in the population
        list_fitness = np.array([self.list_individuals[i].fitness for i in
                range(len(self.list_individuals))])
        #return the min, the standard deviation, the mean and the max
        return [np.min(list_fitness), np.sqrt(np.var(list_fitness)),
                np.mean(list_fitness), np.max(list_fitness)]
```

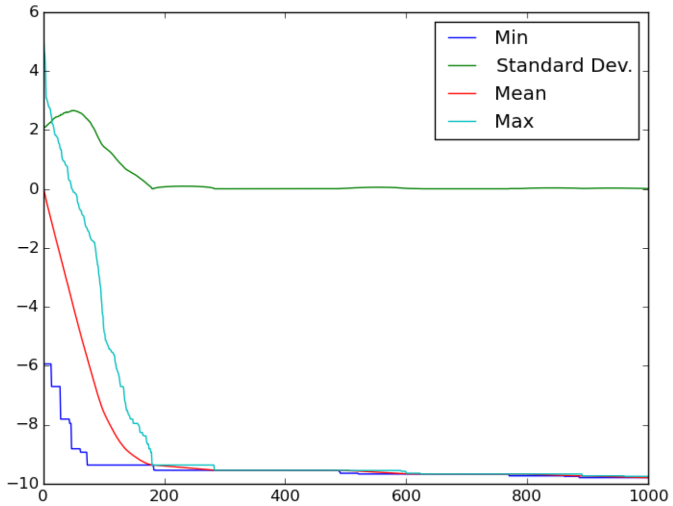
Moreover, adding the following code in your main program `genetic_algorithm` will allow you to keep track of the diversity of your population:

```
statistics = np.zeros([number_of_generation,4])
for i in range(number_of_generation):
        ...
        statistics[i] = parent.stat()
print(parent.list_individuals[parent.best()])
plt.plot(statistics[:,0], label='Min')
plt.plot(statistics[:,1], label='Standard Dev.')
plt.plot(statistics[:,2], label='Mean')
plt.plot(statistics[:,3], label='Max')
plt.legend()
plt.show()
```

[†]In general, we choose `mutation_probability` such that only one gene is mutated per generation per individual.

As an example, one can get:

```
>>> genetic_algorithm(100,.1,1000)
(36,4,42,23,50,31,49,51,100,15) fitness:-9.798599200193856
```



Comment this graphic.

Question 16. Can you find a solution with `fitness` smaller than -9.95 in less than 10 seconds (on a very cheap computer)?

Question 17. Demontrate that the optimal solutions have a fitness equal to $10\sin(6734) \approx -9.999925773$. Give the optimal solutions of our problem.

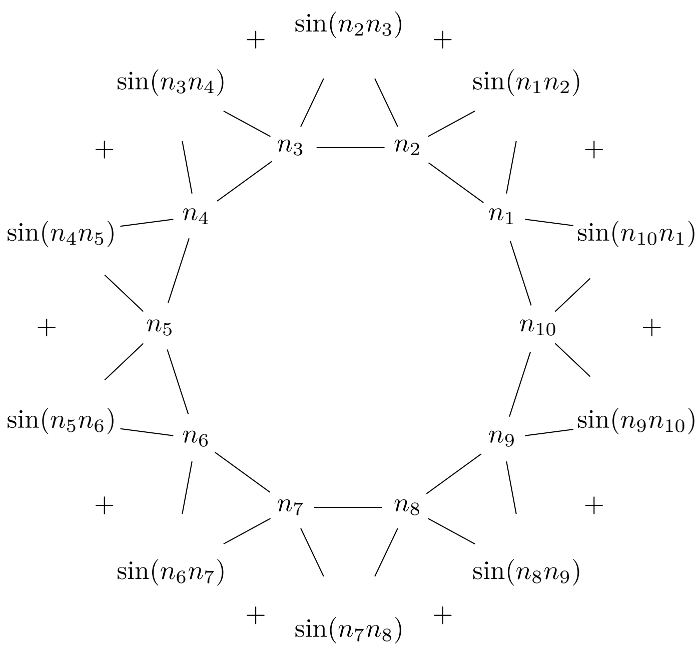
genotype	**phenotype**
$(n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10})$	

Figure 1: Definition of an individual