U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Otimização

Report: Implementation of classic and evolutionary optimization methods in engineering problems

Author:
Tomás Schuller de Almeida e Graça Barbosa

Supervisor:
Prof. Carlos Conceição António

Doctoral Program in Mechanical Engineering

February 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization can be seen as the act of obtaining the best result under given circumstances. For example, in maintenance of an engineering system, engineers have to take many technological and managerial decisions at several stages. The ultimate goal of all that decisions is usually either to minimize the effort required or to maximize the desired benefit. So, optimization can be defined as the process of finding the conditions that give the maximum or minimum value of a function [1, 4].

Numerical optimization is one tool that we have at our disposal, to design new products and to provide the desired results in a timely and economical fashion [1, 5].

There are two important things to distinguish when we are studying optimization: analysis and design. Analysis is the process of determining the response of a specified system to its environment, and design is used to mean the actual process of defining the system. This way, analysis is a part of the design process because is the way we evaluate the adequacy of the design [5]. So, numerical optimization help us to search for the best design according to our needs. Design can also be defined as the process of finding the minimum or maximum of some parameter. That parameter can be called objective function. And for the design to be acceptable, it must also satisfy a certain set of specified requirements called constraints [5].

There are many methods that can be applied to provide the design. Therefore, many different optimization methods have been developed for solving different types of optimization problems. The optimum seeking methods are also known as mathematical programming techniques, and are useful in finding the minimum of a function of several variables under a prescribed set of constraints [4].

In recent years, modern optimization methods have emerged as powerful and popular methods for solving complex engineering optimization problems. These methods include genetic algorithms, simulated annealing, particle swarm optimization, ant colony optimization, neural network-based optimization, and fuzzy optimization [1, 4].

In this work we intend to apply classical optimization algorithms and also evolutionary algorithms. The classical methods that we are going to apply are the Powell's method and the Golden section method. The evolutionary algorithms that we will apply are the Genetic algorithms. For that, these methods will be implemented and a numerical validation will be performed using a spring and weight system.

The work is organized in five chapters. In Chapter 1, a short introduction and the layout of this work are presented. The classical algorithms and the evolutionary algorithms used in this work are presented in Chapter 2 and in Chapter 3, respectively. In Chapter 4 the problem that we are going to solve with the numerical implementation of the classical and the evolutionary algorithms is presented. And, finally in Chapter 5 the main conclusions of this work are presented.

Chapter 2

Classical methods for unconstrained optimization problems

The classical methods of optimization are useful in finding the optimum solution of continuous and differentiable functions. Classical methods are analytical and make use of the techniques of differential calculus in locating the optimum points [4]. In this chapter we will present two classical methods for unconstrained optimization problems: The Golden section method, that consists of estimating the maximum, minimum or zero of a one-variable unconstrained function; and the Powell's method, that is a zero-order method to find the minimum of a n-variable unconstrained function. Firstly, we will present in a general way how a optimization problem is stated.

2.1 Initial concepts

The non-linear constrained optimization problem mathematically is written as follows:

$$\text{Minimize:} \qquad F(\mathbf{X}) \qquad \text{Objective function} \tag{2.1.1}$$

Subject to:

$$g_j(\mathbf{X}) \leq 0 \quad j = 1, ..., m \quad \text{inequality constraints} \tag{2.1.2}$$

$$h_k(\mathbf{X}) = 0 \quad k = 1, ..., l \quad \text{equality constraints} \tag{2.1.3}$$

$$X_i^l \leq X_i \leq X_i^u \quad i = 1, ..., n \quad \text{side constraints} \tag{2.1.4}$$

where $\mathbf{X} = (X_1, X_2, X_3, ..., X_n)$ are the design variables.

The statement presented before can be written in many different ways. For example, we may wish to state the problem as a maximization problem, where we want to maximize the objective function $F(\mathbf{X})$. In a similar way, the inequality sign in Eq. 2.1.2 can be reversed, so that $g(\mathbf{X})$ must be greater than or equal to zero. But, using the above notation, for an optimization problem that needs to be maximized, we simply minimize $-F(\mathbf{X})$. In a general way, the optimization strategy is an iterative procedure, that requires that an initial set of design variables, $\mathbf{X}^0$, to be specified. And, having this starting point, the design is updated iteratively. The iterative procedure is given by:

$$X^q = X^{q-1} + \alpha_q^* \mathbf{S}^q \tag{2.1.5}$$

where q is the iteration number and $\mathbf{S}$ is a vector search direction in the design space. The scalar quantity α^* defines the distance that we wish to move in direction $\mathbf{S}$.

Nonlinear optimization algorithms based on Eq. 2.1.5 can be separated in two parts. One part that corresponds to the determination of a direction search $\mathbf{S}$, and that will improve the objective function, and other part that is the determination of the scalar quantity α^*. These two steps have a very important role in the efficiency and reliability of a given optimization algorithm.

In the application of optimization techniques to design problems, we can define necessary conditions for an optimum, and show that under certain situations these conditions are also sufficient to ensure that the solution is a global optimum.

In *unconstrained* minimization problems, we want to minimize F($\mathbf{X}^*$) with no constraints imposed. To find the minimum for F($\mathbf{X}^*$), the gradient of F($\mathbf{X}^*$) must vanish:

$$\nabla F(\mathbf{X}^*) = 0 \tag{2.1.6}$$

where:

$$\nabla F(\mathbf{X}^*) = \begin{Bmatrix} \frac{\partial F(\mathbf{X})}{\partial X_1} \\ \\ \frac{\partial F(\mathbf{X})}{\partial X_2} \\ \\ \vdots \\ \\ \frac{\partial F(\mathbf{X})}{\partial X_n} \end{Bmatrix} \tag{2.1.7}$$

This is only a necessary condition and is not sufficient to ensure optimality, it is also required that the Hessian matrix should be positive definite. The Hessian matrix is a matrix of second partial derivatives of the objective function with respect to the design variables and is given by:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 F(\mathbf{X})}{\partial X_1^2} & \frac{\partial^2 F(\mathbf{X})}{\partial X_1 \partial X_2} & \cdots & \frac{\partial^2 F(\mathbf{X})}{\partial X_1 \partial X_n} \\ \\ \frac{\partial^2 F(\mathbf{X})}{\partial X_2 \partial X_1} & \frac{\partial^2 F(\mathbf{X})}{\partial X_2^2} & \cdots & \frac{\partial^2 F(\mathbf{X})}{\partial X_2 \partial X_n} \\ \\ \vdots & \vdots & \ddots & \vdots \\ \\ \frac{\partial^2 F(\mathbf{X})}{\partial X_n \partial X_1} & \frac{\partial^2 F(\mathbf{X})}{\partial X_n \partial X_2} & \cdots & \frac{\partial^2 F(\mathbf{X})}{\partial X_n^2} \end{bmatrix} \tag{2.1.8}$$

The Hessian matrix $\mathbf{H}$ is positive definite when all eigenvalues are positive. So, if the gradient is equal to zero and the Hessian matrix is positive definite for a given $\mathbf{X}$, this ensures that the design is at least a relative minimum, but does not guarantee that is a global minimum. To be a global minimum the Hessian matrix must be positive definite for all possible values of the design variables $\mathbf{X}$.

In *constrained* minimization problems, we want to minimize $F(\mathbf{X})$ with constraints. There are a set of necessary conditions for constrained optimality. These are referred to as *Karush-Kuhn-Tucker* (KKT) optimality conditions, and they can be stated as follows:

if $\mathbf{X}^*$ is the optimum design, the following three conditions must be satisfied:

1. $\mathbf{X}^*$ is feasible;
2. $\lambda_j g_j(\mathbf{X}^*) = 0,$ with $j = 1, ..., m$ and $\lambda_j \geq 0$;
3. $\nabla F(\mathbf{X}^*) + \sum_{j=1}^{m} \lambda_j \nabla g_j(\mathbf{X}^*) + \sum_{k=1}^{l} \lambda_{k+m} \nabla h_k(\mathbf{X}^*) = 0,$ with $\lambda_j \geq 0$ and λ_{k+m} unrestricted in sign.

Condition 1. is a statement of the requirement that the optimum design must satisfy all constraints.

Condition 2. imposes the requirement that if the constraint $g_j(\mathbf{X}^*)$ is not precisely satisfied (meaning that $g_j(\mathbf{X}^*) < 0$) then the corresponding Lagrange multiplier, λ_j must be zero.

Condition 3. defines the necessary condition for a design to be a constrained optimum.

More can be seen in [4, 5] about these conditions.

2.2 Golden section method

In this section we are going to talk about one-variable minimization methods, more properly, the Golden section method. The one-variable minimization methods are the ones used for finding the scalar quantity α^* , that as we saw before, defines the distance that we wish to move in direction $\mathbf{S}$.

The Golden section method is one of those methods and is a popular technique because, although it is assumed that the function is unimodal, which means that the function only has a minimum in the search region, it does not need to have continuous derivatives, and in some special cases we can deal with functions which are not even unimodal or continuous. Other reasons why this method is so popular are known the rate of convergence is and the ease to program.

Now, we will present the Golden section method for determining the minimum of function F, where F is a function of the independent variable X. In order to determine the minimum, we will assume that the lower bound, X_l , and the upper bound, X_u , that bracket the minimum are known. We also assume that function F has been evaluated at those points, so we know F_l and F_u.

Now, we choose two points X_1 and X_2 , such that $X_1 < X_2$, and evaluate function F in those points, getting F_1 and F_2. Since function F is unimodal, it comes that either X_1 or X_2 will form a new bound on the minimum.

If $F_1 > F_2$, then X_1 will be the new lower bound, and this way we will have a new set of bounds, X_1 and X_u , but if $F_2 > F_1$, then X_2 would form the new upper bound, being X_l and X_2 , as the points which now bracket the minimum.

If we consider the situation where X_1 is the new lower bound, we choose now another point X_3 , and evaluate function F in that point, being F_3 . Comparing F_2 and F_3 , if $F_3 > F_2$, then

X_3 will replace X u as the upper bound. This process is repeated, and we will limit the bounds to whatever tolerance we want.

To choose the interior points $X_1, X_2, X_3, ...$ a method should be determined. Initially we choose X_l , X_u and X_1 , and we need to evaluate function F in these points for each iteration, so the most efficient algorithm is the one that will reduce the bounds by the same fraction on each iteration. So, considering the golden section number will provide the ideal sequence for dividing the interval to find the minimum value of F, applying a few function evaluations. This, way having the interval bounded by X_l and X_u , the internal points can be written in terms of X_l , X_u and τ, where:

$$\tau = \frac{3-\sqrt{5}}{2} \tag{2.2.1}$$

and X_1 and X_2 are given by:

$$X_1 = (1-\tau)X_l + \tau X_u \tag{2.2.2}$$

$$X_2 = \tau X_l + (1-\tau)X_u \tag{2.2.3}$$

where the golden section number is the ratio between X_2 and X_1.

Now, we just need a computational procedure to iteratively apply the Golden section method. So, we must define a criterion, to identify when the process has converged to an acceptable solution. For that, assuming a initial interval of uncertainty, $X_u - X_l$, we may want to reduce the interval to some fraction of the initial interval. That fraction we are going to represent by ϵ and call it relative tolerance. Therefore, if we desire a specified tolerance ϵ we have:

$$\epsilon = (1-\tau)^{N-3} \tag{2.2.4}$$

where N is the total number of function evaluations, including the first three. So, for N we have:

$$N = \frac{\ln \epsilon}{\ln(1-\tau)} + 3 \tag{2.2.5}$$

This way, we can use N as the convergence criterion for terminating the iterative process when a total number of N function evaluations have been executed, and with this, we have a convergence criterion. The golden section algorithm can be seen in Fig. 2.2.1.

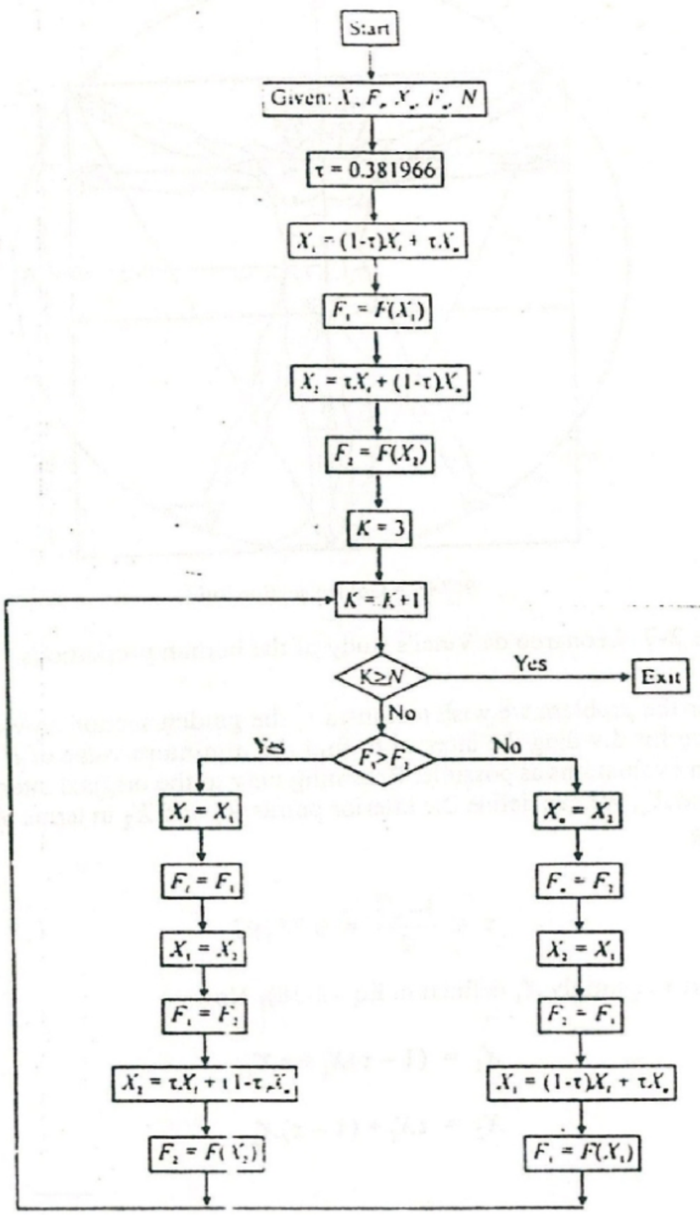

Figure 2.2.1: Golden section algorithm for unconstrained minimum, from [5].

To apply the algorithm, we start by specifying the tolerance, ϵ, as the fraction to which the interval of uncertainty must be reduced. Then using Eq. 2.2.5, we calculate the number of functions evaluations, N required to achieve the tolerance. The value of N should be rounded to the next higher integer number. The next step, will be specify the initial interval X_l and X_u and evaluate function F at X_l and X_u. For this we can use the method finding bounds on a minimum of F that is specified in [5].

2.3 Powell's method

Powell's method is a zero-order method, being an optimization algorithm that only needs function values. Zero-order methods are usually reliable and easy to program, and they deal with non-convex and discontinuous functions, and in many cases they work with discrete values of the design variables. The only problem is that these methods require thousands of function evaluations in order to obtain the optimum [5].

Powell's method is one of the most efficient and reliable zero-order methods and is based on the concept of conjugate directions.

Two directions $\mathbf{S}^i$ and $\mathbf{S}^j$ are *conjugate* if:

$$(\mathbf{S}^i)^T \mathbf{H} \mathbf{S}^j = 0 \tag{2.3.1}$$

where $\mathbf{H}$ is the Hessian matrix defined by Eq. 2.1.8.

The Powell's method has, as basic concept, to first search in n orthogonal directions, $\mathbf{S}^i$, $i = 1, ..., n$, that are the coordinate directions. Each search consists of updating vector X according to Eq. 2.1.5, after minimizing with respect to parameter α. These directions are not usually conjugate, but provide a starting point from which conjugate directions are built. After, completed the n unidirectional searches, a new search direction is created, connecting the first and the last design points, and with this, we have the *n+1* search direction. It is convenient to save the search direction in matrix $\mathbf{H}$ which plays a role similar to the Hessian matrix. First, matrix $\mathbf{H}$ is equal to the identity matrix:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.3.2}$$

where the columns of matrix $\mathbf{H}$ are the unidirectional search vectors $\mathbf{S}^i$, $i = 1, ..., n$.

After determining the minimum in each direction, $\mathbf{S}^i$ is replaced in matrix $\mathbf{H}$ by $\alpha_i \mathbf{S}^i$, to give after n iterations:

$$\mathbf{H} = \begin{bmatrix} \alpha_1^* \mathbf{S}^1 & \alpha_2^* \mathbf{S}^2 & \cdots & \alpha_n^* \mathbf{S}^n \end{bmatrix} \tag{2.3.3}$$

This way, we create a conjugate direction:

$$\mathbf{S}^{n+1} = \sum_{i=1}^{n} \alpha_i^* \mathbf{S}^i = \text{sum of columns of } \mathbf{H} \tag{2.3.4}$$

Next, we search in the direction previously obtained, to determine parameter α_{n+1}^* . And, now, we shift each column of matrix $\mathbf{H}$, one to the left, eliminating $\alpha_1^* \mathbf{S}^1$ entry and storing $\alpha_{n+1}^* \mathbf{S}^{n+1}$ in column n. With this, we obtain a new matrix $\mathbf{H}$ that contains n search directions to start the entire search process again. Now the first search direction is $\alpha_2^* \mathbf{S}^2$ from before and is actually a search in the x_2 coordinate direction. The Powell's method algorithm is given by Fig. 2.3.1.

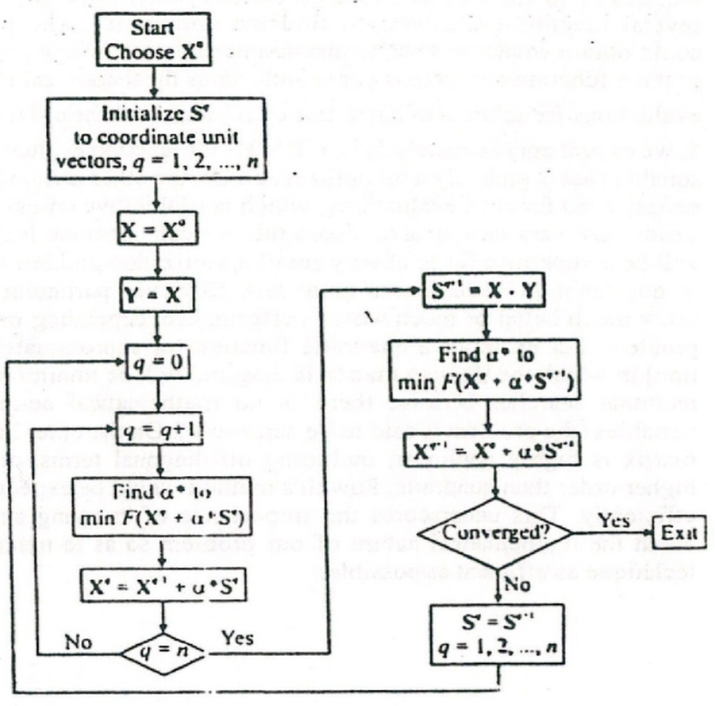
Start
Choose X^0
Initialize S^q to coordinate unit vectors, $q = 1, 2, ..., n$
$X = X^0$
$Y = X$
$q = 0$
$q = q+1$
Find α^* to min $F(X^q + \alpha^* S^q)$
$X^q = X^{q-1} + \alpha^* S^q$
$q = n$
No
Yes
$S^{n+1} = X - Y$
Find α^* to min $F(X^q + \alpha^* S^{n+1})$
$X^{n+1} = X^q + \alpha^* S^{n+1}$
Converged?
Yes
Exit
No
$S^q = S^{q+1}$
$q = 1, 2, ..., n$

Figure 2.3.1: Algorithm for Powell's method, from [5].

Powell's method breaks in two situations. The first situation is when some search direction gains no improvement at all ($\alpha_q^* = 0$), the subsequent search directions will not be conjugate. The second situation is that after a few iterations, the search directions tend to become parallel, because of numerical imprecision related to the nonquadratic nature of the actual function that is being minimized. Therefore, the simplest and most effective solution is to restart the process with unidirectional searches whenever the optimization process becomes slow.

Chapter 3

Evolutionary methods: genetic algorithms

In this chapter, we are going to talk about Genetic algorithms, that are evolutionary methods. Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics [3]. Genetic algorithms are stochastic methods that can find the global minimum with a high probability and are naturally applicable for the solution of discrete optimization problems [4]. The basic elements of natural genetics, that are reproduction, crossover, and mutation, are used in the genetic search procedure. Genetic algorithms are different from the classical methods of optimization, because:

- for starting the procedure, a population of points (trial design vectors) is used, instead of a single design point;
- these methods only use values of the objective function, so the derivatives are not used in the search procedure;
- the design variables are represented as strings of binary variables which correspond to the chromosomes in natural genetics. So, the search method can be applied for solving discrete and integer programming problems, and for continuous design variables, the string length can change to get any desired resolution;
- the value of the objective function that corresponds to a design vector, plays the role of the fitness in natural genetics;
- for every new generation, a new set of strings is created by using randomized parents selection and crossover from the old generation.

In a simple way, the basic ideia of the approach to the algorithm is to begin with a set of designs, that are randomly generated using the allowable values for each design variable. For each design, is assigned a fitness value, and for that, usually is used the objective function. From the current set of designs, is selected, randomly, a subset with a bias allocated to more fit members of the set. Then, is used random processes to generate new designs using the selected subset of designs. The size of the set of designs is kept fixed and since more fit members of the set are used to create new designs, the successive sets of designs have a higher probability of having designs with better fitness values. Then, the process continues until a stopping criterion is achieved [2].

Three genetic operators are used to perform this task: **reproduction**, **crossover** and **mutation**. **Reproduction** is the process of selecting a set of designs from the current population and carrying them into the next generation. To implement the reproduction operator we need to implement a selection procedure, that is aimed at the most capable members of the current design set (population). **Crossover** is the process of allowing selected members of the new population to exchange characteristics of their designs among themselves. Finally, **mutation** is the process that safeguards the process from a complete premature loss of valuable genetic material during reproduction and crossover [2]. These three tasks are repeated for successive generations of the population until there is no improvement in fitness. Then the member in that generations with the highest level of fitness is taken to be the optimum design. Genetic algorithms are not simple random search techniques, because in a efficiently way they explore the new combinations with the available knowledge to find a new generation with better fitness or objective function value [4].

3.1 Representation of the design variables

As we saw before, in Genetic algorithms the design variables are represented as strings of binary numbers, 0 and 1. So, we need a method to represent the design variables, so they can be manipulated by the algorithm. So, if each design variable X_i , $i = 1, 2, ..., n$ is coded in a string of length q, a design vector is represented using a string of total length nq. In a general way, if a binary number is given by $b_q b_{q-1} ... b_2 b_1 b_0$, where $b_k = 0$ or 1, $k = 0, 1, ...q$, then the equivalent decimal number y (integer) is given by:

$$Y = \sum_{k=0}^{q} 2^k b_k \tag{3.1.1}$$

So, a continuous design variable X can only be represented by a set of discrete values if a binary representation is used. This way, if a given variable X, whose bounds are given by, X_{min} and X_{max} , is represented by a string of q binary numbers, as we can see by Eq. 3.1.1, its decimal value can be computed as:

$$X = X_{min} + \frac{X_{max} - X_{min}}{2^q - 1} Y \tag{3.1.2}$$

Therefore if a continuous variable is to be represented with high accuracy, we need to use a large value of q in its binary representation. In fact, the number of binary digits, q, needed to represent a continuous variable in steps of ΔX can be computed from the relation:

$$2^q \geq \frac{X_{max} - X_{min}}{\Delta X} + 1 \tag{3.1.3}$$

3.2 Representation of the objective function

As we saw before, the value of the objective function for a given design vector is the value that is going to be used to measure the fitness of each individual. In fact, a higher value of the fitness implies a better design [2].

There are many strategies to obtain the fitness from the objective function, the one that we are going to use is based on the idea of Goldberg [3]. For that, we are going to consider $F(X_i)$ as the objective function, where X_i are the individuals in the population and $f(X_i)$ the fitness function for each individuals. The process is then to:

- evaluate $F(X_i)$ for each individual of the current population, in order to obtain the maximum value, F_{max} ;
- for each individual, calculate the fitness:

$$f(X_i) = F_{max} - F(X_i) + p(F_{max} - F_{min}) \tag{3.2.1}$$

where p is the amount of artificial increase relatively to the best individual value of the fitness for all individuals. With this, we avoid that the worst individuals will get null fitness, giving them the chance to reproduce.

3.3 Genetic operators

To optimize a problem using Genetic algorithms, as we saw before, we start with a population of random strings that denote several design vectors, and the size of the population is kept fixed. Each string is evaluated, so that we can find its fitness value and then use the three genetic operators already mentioned, reproduction, crossover and mutation. This way, we obtain a new population that is evaluated to find the fitness values and that is tested for the convergence of the process. When we apply one cycle of reproduction, crossover, mutation and evaluation of the fitness values, we have a generation in Genetic algorithms. If the convergence criterion is not satisfied, the population is iteratively operated by the three genetic operators and the new population that results from that iterative process is evaluated for the fitness values. The process continues over several generations until the convergence criterion is satisfied and the process is terminated [4].

3.3.1 Reproduction

The first operation applied to the population is reproduction, where we select strings (designs) of the population to form the parents. As we saw before, we need to implement a selection procedure in order to select good strings of the population. So, a string is selected in such a way that its probability is proportional to its fitness. Therefore, we apply the roulette wheel algorithm, where the strings are randomly selected providing higher chances of reproduction to the fittest strings of the population. For more details in the roulette wheel algorithm, see [2, 4].

3.3.2 Crossover

Once the parents are selected, crossover is applied to introduce variation into the population. Crossover is the process of combining or mixing two different strings (designs) in the population. The commonly used process for crossover is known as single-point crossover operator, where a position in the string is selected at random along the string length, and the binary digits lying on the right side of the position selected are swapped (exchanged) between the two strings. The strings generated by the crossover operator are known the child strings. In practice, a crossover probability, p_c , is used in selecting the parents for crossover.

3.3.3 Mutation

Mutation is the next operator to be applied on the new strings with a specific small mutation probability, p_m . This operator appears to safeguard the process from a complete premature loss of valuable genetic material during reproduction and crossover steps. In terms of a genetic string, this step corresponds to selecting a few members of the population, determining a location on each string randomly, and switch 0 to 1 or vice versa, with probability p_m.

3.4 Elitism

Elitism consists of forcibly selecting to the next generation the strings that represent the better solution until that generation. To apply, this process all strings should be sorted according to their fitness and the first strings are transferred to the next generation.

3.5 Stopping criteria

To stop the Genetic algorithm process we must define a maximum number of generations or the fittest strings do not change for a certain number of iterations.

3.6 Genetic algorithms process

According to [4], the computational procedure of Genetic algorithms can follow these steps:

1. Choose a string length $l = nq$ to represent the n design variables of the design vector X. Assume suitable values for the following parameters: population size, crossover probability p_c , mutation probability p_m, permissible value of standard deviation of fitness values of the population to use as a convergence criterion, and a maximum number of generations to be used as a second convergence criterion.

2. Generate a random population of size m, each consisting of a string of length $l = nq$. Evaluate the fitness values of the m strings.

3. Carry out the reproduction process.

4. Carry out the crossover operation using the crossover probability, p_c.

5. Carry out the mutation operation using the mutation probability, p_m to find the new generation of m strings.

6. Evaluate the fitness values of the m strings of the new population.

7. Test for the convergence of the algorithm or process, and if the convergence criterion is satisfied the process may be stopped. Otherwise, go to step 8.

8. Test for the generation number. If the computations have been performed for the maximum permissible number of generations and hence the process may be stopped. Otherwise, set the generation number as $i = i + 1$ and go to step 3.

Chapter 4

Nonlinear analysis of a spring and weight system

In this chapter, the optimization methods introduced in Chapters 2 and 3 are applied to optimize a non-linear spring and weight system taken from [5]. Both algorithms were implemented in python and the code is shown in Appendix.

4.1 Problem description

The problem that we are going to analyze is presented in [5] as an example of application for several classical optimization methods. The problem consists of a simple system supporting weights at the connections between the springs. This system is going to be analyzed to determine the equilibrium position by minimizing the Potential Energy (PE). We will assume that the system is comprised of five weights and six springs, shown in the undeformed position in Fig. 4.1.1 and in the deformed position in Fig. 4.1.2

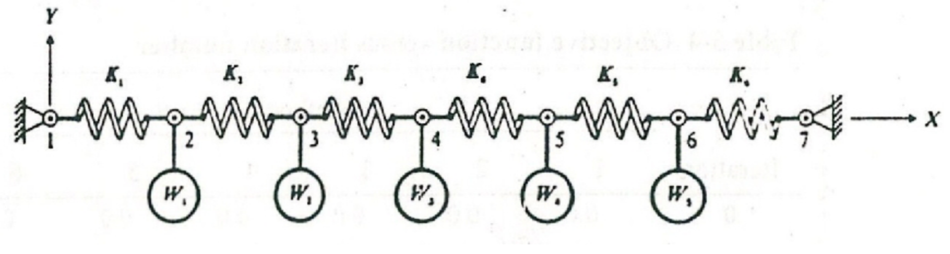

Figure 4.1.1: Undeformed position, from [5].

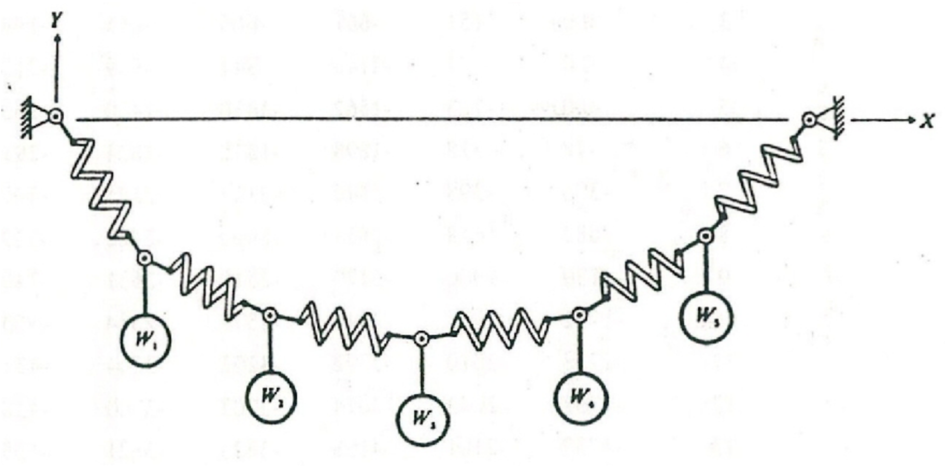

Figure 4.1.2: Deformed position, from [5].

The deformation of spring i is given by:

$$\Delta L_i = \left[(X_{i+1} - X_i)^2 + (Y_{i+1} - Y_i)^2\right]^{1/2} - L_i^0 \tag{4.1.1}$$

where L^0 is taken to be 10 m for each spring. There are a total of $N + 1$ springs, with N being the number of weights. Here $N = 5$.

The stiffness of spring I is given by:

$$K_i = 500 + 200\left(\frac{N}{3} - i\right)^2 \qquad \text{N/m} \tag{4.1.2}$$

Weight W_j is defined to be:

$$W_j = 50j \qquad \text{N} \tag{4.1.3}$$

where j corresponds to the joint where W_j is applied.

PE is given by:

$$PE = \sum_{i=1}^{N+1} \frac{1}{2} K_i \Delta L_i^2 + \sum_{j=1}^{N} W_j Y_j \qquad \text{Nm} \tag{4.1.4}$$

and the coordinates are positive as shown in Figures 4.1.1 and 4.1.2. We are considering a five-weight and six-spring system, so the design variables are X_i , $i = 1, ..., 5$ and Y_i , $i = 1, ..., 5$, so we have a total of 10 design variables. The objective is to minimize function *PE*.

In [5] the problem was solved using 6 different methods, and in each method the Golden section method was used in the one-dimensional search, followed by a cubic interpolation. Of all the 6 methods that were applied in [5], we are only interested in the results from the application of Powell's method, because is the one that we are going to implement. So, for Powell's method, the optimization result obtained in [5] is given by Table 4.1 and the objective function and the number of iterations, by Table 4.2.

Table 4.1: Optimization results from [5].

Design variables	**Initial**	**Powell's method**
X_1	10.0	10.3
X_2	20.0	21.1
X_3	30.0	31.7
X_4	40.0	42.1
X_5	50.0	51.8
Y_1	0.0	-4.24
Y_2	0.0	-7.90
Y_3	0.0	-9.86
Y_4	0.0	-9.40
Y_5	0.0	-6.01

Table 4.2: Objective function value and the iteration number from [5].

Optimum	**Powell's method**
Iterations	178
PE_{min}	-4416

4.2 Results

In this section we will present and discuss the results obtained with Powell's method and the Genetic algorithms.

4.2.1 Powell's method

For the numerical implementation of Powell's method, the flowchart from Fig. [**fig:powell**] was followed. The determination of the scalar quantity α^* , that defines the distance that we wish to move in direction **S**, is performed with the Golden section method (presented in Section 2.2), that it goes until the initial interval is reduced to 1% of its amplitude. The following limits for the design space are defined:

$$0 \leq X_i \leq 60, \tag{4.2.1}$$

$$-20 \leq Y_i \leq 20, \tag{4.2.2}$$

To avoid the breakdown of the algorithm due to numerical imprecision, the search directions are restarted every r-iterations (here is considered $r = 11$). The stopping criteria employed are:

- **Maximum number of iterations :**
  The number of iterations should not exceed a prescribed number of iterations. The value considered is: 200 iterations.

- **Relative change in the objective function:**
  This criterion checks the relative change between successive iterations. The convergence is indicated if:

$$\frac{|F(\mathbf{X^q}) - F(\mathbf{X^{q-1}})|}{max[|F(\mathbf{X^q})|, 10^{-10}]} \leq \epsilon_R \tag{4.2.3}$$

where ϵ_R represents a specified fractional change. The criterion verifies this condition after some specified number of consecutive iterations. The value of 10^{-10} prevents the division by zero in the event that $F(\mathbf{X})$ approaches zero. The value considered for ϵ_R is 10^{-7}, after 5 consecutive iterations.

The initial values are the same used in [5] and are shown in the second column of Table 4.1. As mentioned before all the python code is included as Appendix. The results obtained with Powell's algorithm are shown in Table 4.3.

Table 4.3: Results obtained with Powell's algorithm.

Design variables	**1**	**2**	**3**	**4**	**5**
X_i	10.3	21.0	31.6	42.0	51.7
Y_i	-4.31	-7.95	-9.89	-9.33	-5.89

The maximum number of iterations criterion was verified. So, the algorithm stopped after 200 iterations. In Fig. 4.2.3 the evolution of the potential energy of the system is represented and in Fig. 4.2.4 the final spatial configuration of the system is represented.

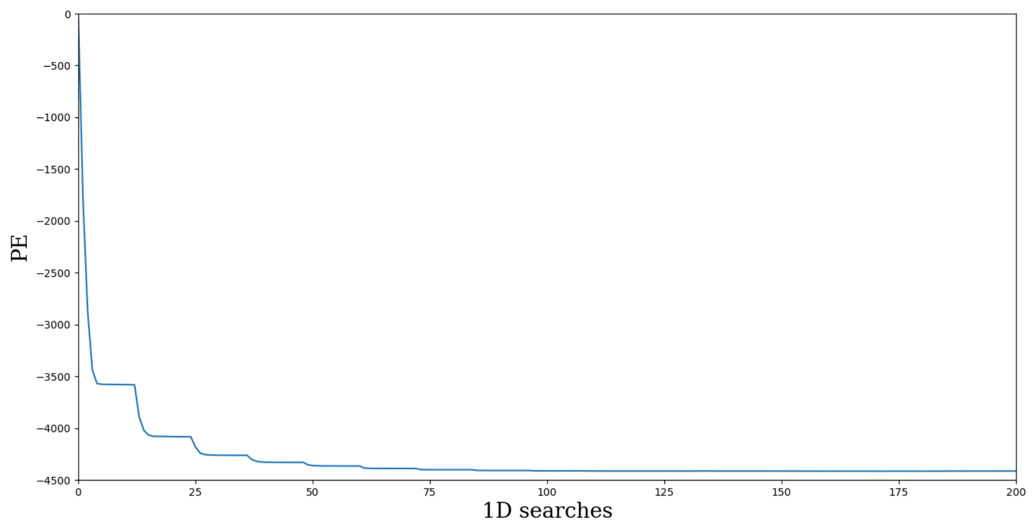

Figure 4.2.1: Evolution of the potential energy of the system.

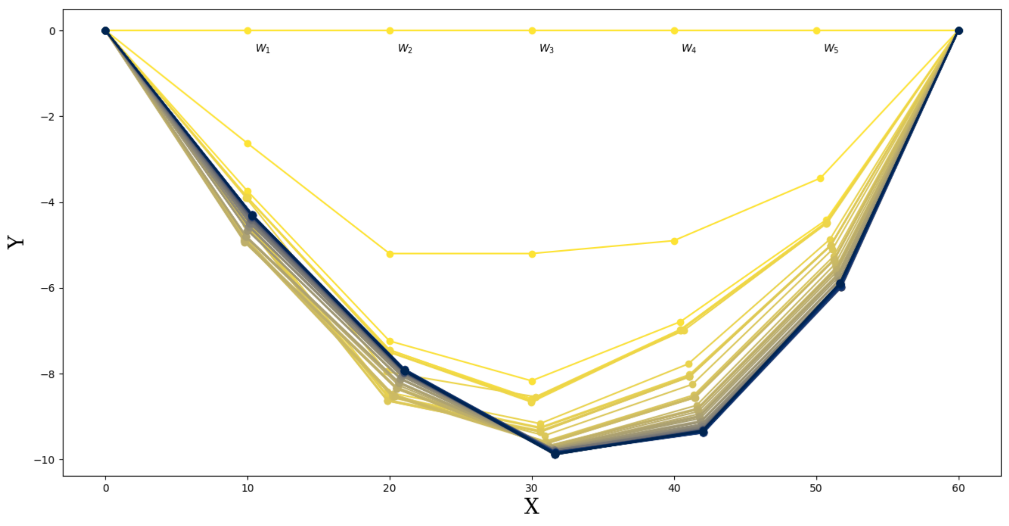

Figure 4.2.2: Final spatial configuration of the system.

The results obtained are in agreement with the results obtained by [5], and the final PE is 4414.5. In Fig. 4.2.4 the evolution of the system between the initial and the final configuration is shown. We can see that the differences between successive positions are increasingly smaller over iterations.

4.2.2 Genetic algorithms

To run the Genetic algorithms described in Chapter 3, we performed a parametric study com- paring some parameters. For all runs of the algorithm, the parameters that were kept constant are:

- number of individuals in the population: $N_{ind} = 500$;
- number of properties(or chromosomes): $N_{prop} = 10$;
- number of bits: $N_{bits} = 16$;
- maximum number of generations: $N_{germax} = 2000$;
- equal number of generations: $N_{eqger} = 100$;

The parameters that were changed in order to do the parametric study are:

- crossover probability: $p_c = 0.75$ or $p_c = 1$;
- mutation probability: $p_m = 0.01$ or $p_m = 0.001$;
- number of individuals that pass to the next generation: $N_{elit} = 2$ or $N_{elit} = 4$.

The minimal number of individuals to pass to the next generation is set to 2 ($N_{elit} = 2$), because the new generation of the population is done two by two, i.e. each two parents generates two new children, and due to elitism, if an odd number is passed, then we couldn't obtain a pair number of individuals in the new generation.

The design space is defined by:

$$5 \leq X_1 \leq 15, \tag{4.2.4}$$

$$15 \leq X_2 \leq 25, \tag{4.2.5}$$

$$25 \leq X_3 \leq 35, \tag{4.2.6}$$

$$35 \leq X_4 \leq 45, \tag{4.2.7}$$

$$45 \leq X_5 \leq 55, \tag{4.2.8}$$

$$-20 \leq Y_i \leq 20, \tag{4.2.9}$$

Changing the parameters presented before, we are going to show 2 different tables, that present the minimum *PE* achieved for each combination of parameters. After that, the final results of the best combination are presented.

For the best combination we present:

- a table with the design variables X_i and Y_i obtained;
- a figure with the evolution of the *PE* for 3 individuals over the generations: the individual with the minimum *PE* (in blue), the individual which *PE* is at Q_1 (in green) and the individual which *PE* is at Q_2 (in red);
- a figure with the spatial configuration of the design variables for each individual that is represented in the evolution of *PE*, and with the design variables obtained by [5] for the Powell's method.

Q_1 and Q_2 stand for first quartile and second quartile, respectively. The choice of the individuals which *PE* is at Q_1 and at Q_2 allow us to evaluate the convergence of the algorithm. Since mutation occurs along the generations, we can obtain individuals with bad fitness, so analysing amplitude or average is a bad indicator of convergence.

Table 4.4: Results obtained with the Genetic algorithms considering $N_{elit} = 2$.

		p_c				
		0.1	0.25	0.5	0.75	1
p_m	0	877.2*	-2242.6*	-3509.0*	-4313.2	-3169.4*
	$1E^{-4}$	-1022.6	-2751.0	-3994.3	-2395.8	-2256.8
	$1E^{-3}$	-1029.8	-4202.7	3501.3	-4398.6	-4358.8
	$1E^{-2}$	-4119.5	-2262.0	-4289.8	-4020.0	-4280.7
	$1E^{-1}$	1432.8	-2691.6	1092.4	-1713.2	1029.9

* Results with an asterisk (*) are due to a zero-division error

Table 4.5: Results obtained with the Genetic algorithms considering $N_{elit} = 4$.

		p_c				
		0.1	0.25	0.5	0.75	1
p_m	0	2701.5*	-643.0*	-3979.5*	-1944.3*	-4108.5*
	$1E^{-4}$	-778.6	-1889.6	-4409.8	-2481.9	-3757.9
	$1E^{-3}$	-4139.4	-4139.7	-4394.3	-4398.3	-2436.9
	$1E^{-2}$	-3277.4	-347.0	-4405.0	-4390.3	-627.8
	$1E^{-1}$	-2374.4	-2492.0	655.4	-1538.0	989.5

* Results with an asterisk (*) are due to a zero-division error

Table 4.6: Results obtained with Genetic algorithm - best combination.

Design variables	**1**	**2**	**3**	**4**	**5**
X_i	10.4	21.0	31.6	42.0	51.6
Y_i	-4.25	-7.45	-9.82	-9.35	-5.87

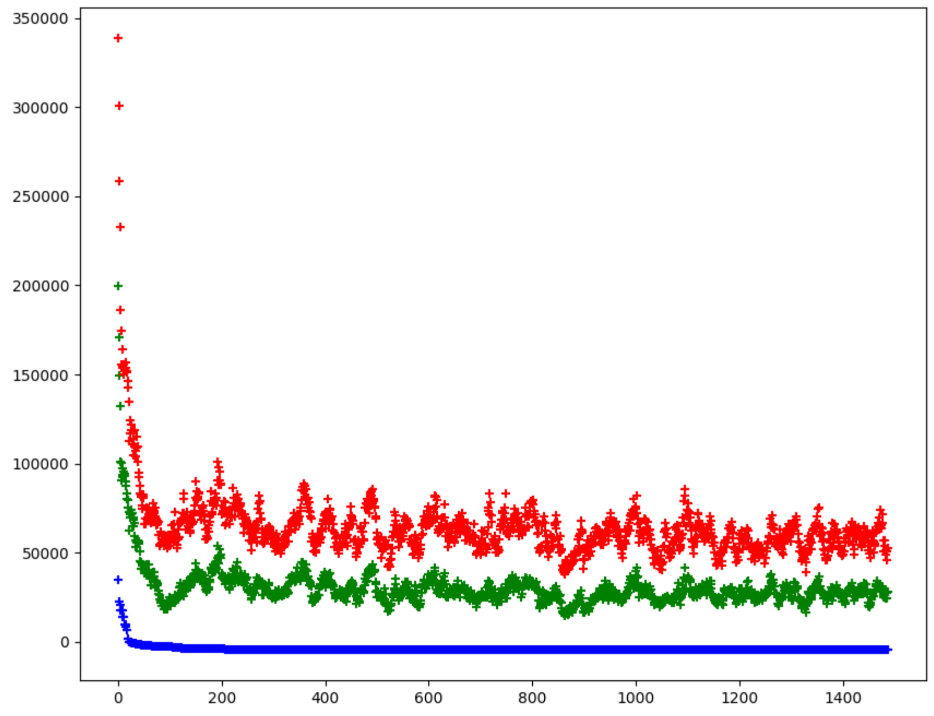

Figure 4.2.3: Evolution of the potential energy of the system - best combination.

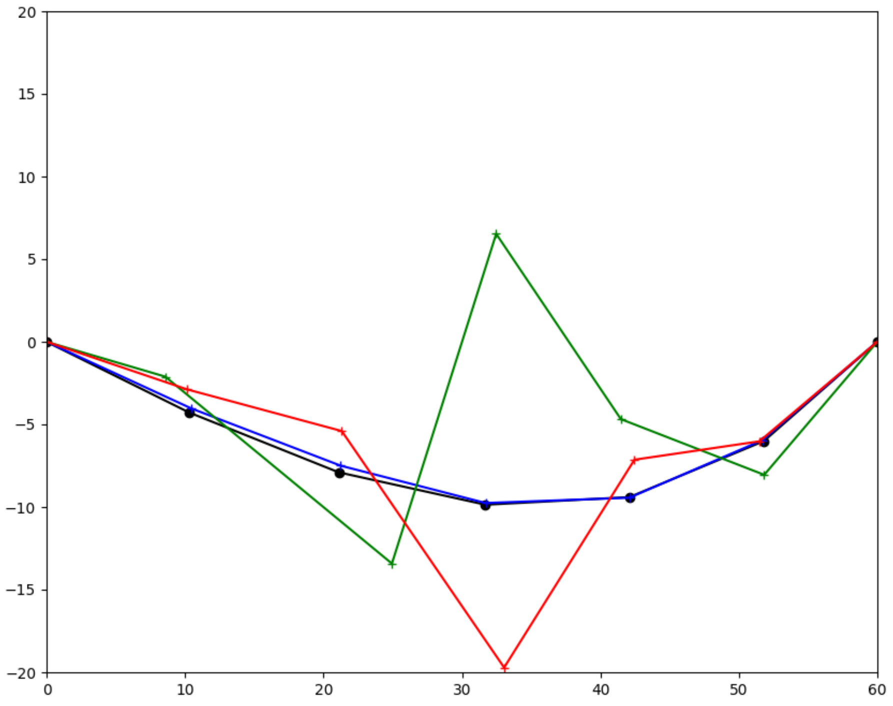

Figure 4.2.4: Final spatial configuration of the system - best combination.

Chapter 5

Conclusion

The main objective of this work was to make a review of classical optimization algorithms (Golden section method and Powell's method) and evolutionary methods (Genetic algorithms). For that, some fundamental concepts were introduced and a numerical implementation with python was performed. To numerically validate the implemented algorithms, a non-linear unimodal problem of a weight and spring system proposed in [5] was analysed. The implementation of the Powell's method was easy to perform and the numerical results were in agreement with the results from [5] obtaining a very small difference. For the implementation of the Genetic algorithms a parametric study was performed, and 50 different results were shown. To perform those results crossover probability, mutation probability and number of surviving individuals were changed, and we could see that if the number of surviving individuals increases, the algorithm generally gives better results, for the final value of PE and for the spatial configuration. But, if we increase mutation probability or crossover probability, we can see that necessarily doesn't happen.

For all the results presented in the Genetic algorithms, the initial configuration of the individuals were the same. It was necessary to run the algorithm many times in order to obtain the best combination of parameters.

Genetic algorithms are very dependent on the parameters of the algorithm. So, when it comes to choose these parameters, some care should be taken, because the computational effort becomes too expensive. We analysed a simple spring and weight system problem, where the computational cost can be taken as negligible. However, when we have real engineering problems, each evaluation of the objective function can be complex and the computational cost can become intensive.

Bibliography

[1] C. C. António. *Otimização de Sistemas em Engenharia - Fundamentos e algoritmos para o projeto ótimo.* ENGEBOOK, 2020.

[2] J. S. Arora. *Introduction to Optimum Design, Second edition.* Elsevier Academic Press, 2004.

[3] D. E. Goldberg. *Genetic algorithms in Search, Optimization Machine Learning.* Addison-Wesley Publishing Company, 1989.

[4] S. S. Rao. *Engineering Optimization, Theory and Practice.* John Wiley & Sons, Inc., 1996.

[5] G. N. Vanderplaats. *Numerical optimization techniques for engineering design: with applications.* McGraw-Hill College, 1984.

Appendices

Appendix A

Powell's method - Python Implementation

A.1 Main script

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
from f_obj import f_obj
from find_bounds import find_bounds
from g_sec import g_sec
from powell import powell
import numpy as np
import math

import seaborn as sns
from cycler import cycler
import matplotlib.patches as patches
# matplotlib and seaborn for plotting
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib
from mpltools import layout
from mpltools import color
from mpltools import style


N = 5
x_min = 0
```

```
x_max = 60
y_min = -20
y_max = 20

#Initial bounds
bounds = np.zeros((2*N,2))
bounds[0:N,0] = x_min
bounds[0:N,1] = x_max
bounds[N+1:2*N,0] = y_min
bounds[N+1:2*N,1] = y_max

x = np.arange(10,60,10)
y = np.zeros(len(x))
xy = np.append(x,y)

#Powell_method
xy,F_list,xy_new = powell(xy, bounds)
F = f_obj(xy)
np.append(F_list,F)
print(xy)
print(F)



fig1, ax1 = plt.subplots(figsize = (16, 8))
ax1.plot(F_list,  label='PE')

ax1.set_ylim([-4500, 0])
ax1.set_xlim([0, 200])
ax1.set_ylabel('PE' ,family='serif', size=20)
ax1.set_xlabel('1D searches' ,family='serif', size=20)
plt.show()
C=np.zeros(len(xy))
for i in np.arange(0,len(xy_new),20):
    C=np.vstack((C,xy_new[i, :]))
exes=C[:,:5]
ys=C[:,5:]

exes = np.column_stack((exes,60*np.ones(len(exes))))
exes = np.insert(exes,0,np.zeros(len(exes)),axis=1)
ys = np.column_stack((ys,np.zeros(len(ys))))
ys = np.insert(ys,0,np.zeros(len(ys)),axis=1)
```

```
fig2,ax2=plt.subplots(figsize = (16, 8))
cmap = plt.get_cmap('cividis_r')

for i in range(len(exes)):
    plt.plot(exes[i],ys[i],'o-', color=cmap(float(i)/len(exes)))

plt.annotate(r'$W_1$', xy=(10, 0),xytext=(10.5, -0.5),family='serif')
plt.annotate(r'$W_2$', xy=(20, 0),xytext=(20.5, -0.5),family='serif')
plt.annotate(r'$W_3$', xy=(30, 0),xytext=(30.5, -0.5),family='serif')
plt.annotate(r'$W_4$', xy=(40, 0),xytext=(40.5, -0.5),family='serif')
plt.annotate(r'$W_5$', xy=(50, 0),xytext=(50.5, -0.5),family='serif')
ax2.set_ylabel('Y', family='serif', size=20)
ax2.set_xlabel('X' , family='serif', size=20)

plt.show()
```

A.2 Objective function

@author: Tomás Schuller

schuller@fe.up.pt

Otimizacao 20/21

```
def f_obj (xy):

    import numpy as np

    N=len(xy)
    if N % 2 != 0:
        print("N is an odd number")

    #N - number of weights
    N = N//2
    #Li - length for each spring
    Li = 10
    L0 = Li*np.ones(N+1)
    X = np.transpose(xy[0:N])
    Y = np.transpose(xy[N:2*N+1])
    #K - stiffness of spring i
    K=np.ones(N+1)
    for i in range(N+1):
        K[i] = 500 + 200*(abs(N/3-(i+1))**2)

    #W - weight
    W =np.ones(N)
    for i in range(N):
        W[i] = 50*(i+1)
    Xt = np.insert(X,0,0)
    Xt = np.append(Xt,Li*(N+1))
    Yt = np.insert(Y,0,0)
    Yt = np.append(Yt,0)

    #Delta_L - deformation of spring i
    diff1=np.diff(Xt,1)
    diff2=np.diff(Yt,1)
    Delta_L = (diff1**2 + diff2**2)**(0.5)-L0
    #F - Potential energy
    F = 0.5*np.sum(K.conj()*Delta_L**2, axis=0)+ ...
    np.sum(W.conj()*Y, axis=0)

    return F
```

A.3 Find bounds function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
from f_obj import f_obj
from g_sec import g_sec

def find_bounds (bounds, xy, S):
    L_min = min(bounds[:,0])
    U_max = max(bounds[:,1])
    a = (1+(5)**0.5)/2
    Lr = 1e-1
    alpha_max = U_max
    alpha_L = L_min*Lr
    alpha_U = U_max*Lr
    FL = f_obj(xy + alpha_L*S)
    FU = f_obj(xy + alpha_U*S)
    if FU > FL:
        print("alpha_U upper limit")
    else:
        while FU <= FL:
            alpha_1 = alpha_U
            F1 = FU
            alpha_U = (1+a)*alpha_1-a*alpha_L
            if alpha_U > alpha_max:
                print("Exit: no limit found")
                break
            else:
                FU = f_obj(xy + alpha_U*S)
                if FU > F1:
                    print ("alpha_U upper limit")
                    break
                else:
                    alpha_L = alpha_1
                    FL = F1

    return alpha_L, alpha_U
```

A.4 Golden section function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
def g_sec (alpha_L, alpha_U, xy, S):


    import numpy as np
    import math
    from f_obj import f_obj
    #from find_bounds import find_bounds



    tol_r = 1e-2
    tau = (3-(5)**0.5)/2
```

```
N_iter = np.ceil((math.log(tol_r)/(math.log(1-tau)))+3)



FL = f_obj(xy+alpha_L*S*np.ones(len(xy)))
FU = f_obj(xy+alpha_U*S*np.ones(len(xy)))

alpha_1 = (1-tau)*alpha_L+tau*alpha_U
F1 = f_obj(xy+alpha_1*S*np.ones(len(xy)))
alpha_2 = tau*alpha_L+(1-tau)*alpha_U
F2 = f_obj(xy+alpha_2*S*np.ones(len(xy)))

alp = np.zeros((int(N_iter+2),2))
Falp = np.zeros((int(N_iter+2),2))
alp[0,:] = [alpha_L, alpha_U]
alp[1,:] = [alpha_1, alpha_2]
Falp[0,:] = [FL, FU]
Falp[1,:] = [F1, F2]

k=3
while k < N_iter:
    k = k+1
    if F1 > F2:
        alpha_L = alpha_1
        FL = F1
        alpha_1 = alpha_2
        F1 = F2
        alpha_2 = tau*alpha_L+(1-tau)*alpha_U
        F2 = f_obj(xy+alpha_2*S*np.ones(len(xy)))
    else:
        alpha_U = alpha_2
        FU = F2
        alpha_2 = alpha_1
        F2 = F1
        alpha_1 = (1-tau)*alpha_L+tau*alpha_U
    F1 = f_obj(xy+alpha_1*S*np.ones(len(xy)))

    alp[int(N_iter+1),:] = [alpha_L, alpha_U]
    Falp[int(N_iter+1),:] = [f_obj(xy+alpha_L*S*np.ones ...
    (len(xy))), f_obj(xy+alpha_U*S*np.ones(len(xy)))]


index_min = np.argmin([FL,F1,F2,FU])
if index_min == 0:
    alpha=alpha_L
elif index_min == 1:
    alpha=alpha_1
elif index_min == 2:
    alpha=alpha_2
elif index_min == 3:
    alpha=alpha_U


return alpha
```

A.5 Powell's method function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
from find_bounds import find_bounds
from f_obj import f_obj
from g_sec import g_sec
import numpy as np
import math




# Powell's method
# This function calculates the S-direction and ...
calculates the minimum

def powell (xy_0, bounds):
    #Number of variables
    N = len(xy_0)
    if N % 2 != 0:
        print("N is an odd number")
    N = int(N/2)
    Li = 10
    #Maximum number of iterations
    Ni = 200
    #Relative change in f_obj
    Tol = 1E-7
```

```
#Number of iterations in a row that check the stop criterion
m = 5
#Number of iterations to restart matrix H
r = 11
#Iteration counter
iter = 0
#Counter to restart matrix H
count = 0
#Stop criterion counter
j = 0
#Matrix H
H = np.eye(int(2*N))

#xy initial
xy = xy_0
F_list=f_obj(xy_0)
f_obj_new = np.zeros((2*N+1)*N+1)
xy_new= xy

while True:
    #xy initial
    for q in np.arange(0,2*N): #cycle through all columns
        S = H[:,q]
        #find_bounds - function
        [alpha_L, alpha_U] = find_bounds(bounds, xy, S)
        #g_sec - function
        alpha = g_sec(alpha_L, alpha_U, xy, S)
        #updates the solution
        xy = xy + alpha*S*np.ones(len(xy))
        #Results
        xy_new=np.vstack((xy_new,xy))

    #Look for the new direction
    S = xy-xy_0
    #S normalization
    S_max=max(abs(S))
    for l in range(len(S)):
        S[l]= S[l]/S_max
    #Find_bound - function
    [alpha_L, alpha_U] = find_bounds(bounds, xy, S)
    #g_sec - função
    alpha = g_sec(alpha_L, alpha_U, xy, S)
    #updates the solution
```

```
        xy = xy + alpha*S*np.ones(len(xy))
        #Results

        F_list=np.append(F_list,f_obj(xy))
        xy_new=np.vstack((xy_new,xy))

        #Stopping criteria
        #1st Criterion: maximum number of iterations


        if iter == Ni:
            break
        iter = iter+1


        #2nd Criterion: relative change in objective function
        if (iter > 1 and (abs(f_obj(xy)-f_obj(xy_0))/ max ...
        (abs(f_obj(xy)),1E-10))<=Tol):
            j = j+1
            if j >= m:
                break
        else:
            j = 0


        #Update matrix H
        count = count + 1
        if count > r:
            H = np.eye(2*N)
            count = 0
        else:
            H = np.roll(H, -1, axis=1)
            H[:, -1] = alpha*S/max(abs(S))


    return xy,F_list,xy_new
```

Appendix B

Genetic algorithm - Python Implementation

B.1 Main script

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import numpy as np
import math
import random
from scipy import sparse
import matplotlib
from matplotlib import rc
from matplotlib import pyplot as plt
import seaborn as sns
import os
import subprocess

from population import population
from select_o import select_o
from fitness import fitness
from bin2dec import bin2dec
from f_obj import f_obj
from roulette import roulette
from crossover import crossover
from mutation import mutation
from new_gen import new_gen
random.seed('default')
```

```
N = 5
x_min = 0
x_max = 60
y_min = -20
y_max = 20

#Initial bounds
bounds = np.zeros((2*N,2))
bounds[0:N,0] = x_min
bounds[0:N,1] = x_max
bounds[N:2*N,0] = y_min
bounds[N:2*N,1] = y_max

N_ind = 500
N_prop = 10
N_bit = 16
Boolean_M = -1
ProbMut = 1E-2
ProbCross = 0.5
N_ger_max = 2000
N_elit = 4
N_ger_eq = 100

bounds[0,0]=5
bounds[0,1]=15
bounds[1,0]=15
bounds[1,1]=25
bounds[2,0]=25
bounds[2,1]=35
bounds[3,0]=35
bounds[3,1]=45
bounds[4,0]=45
bounds[4,1]=55


# i generation
i_ger = 0

phi_fitness_sort=[]

#generate initial population (random)
pop_s=population(N_ind,N_prop,N_bit)
```

```
#evaluate population
#phi - property (xy)
#phi_fitness - FObj value for each property
phi, phi_fitness = fitness(N_ind, N_prop, N_bit, bounds, pop_s)
print(len(phi_fitness))
print(len(phi))
print(len(phi[0]))

#FIGURE INITIAL

I=np.zeros(len(phi_fitness))

for i in range(len(phi_fitness)):
    I[i]=Boolean_M*phi_fitness[i]

i_ind= list(I).index(max(I))

x_init=np.zeros(1)
x_fin=60*np.ones(1)
x_init= np.append(x_init,phi[0:N,i_ind])
x_init= np.append(x_init,x_fin)
y_init=np.zeros(1)
y_fin=np.zeros(1)
y_init= np.append(y_init,phi[N:,i_ind])
y_init= np.append(y_init,y_fin)
fig2, ax2 = plt.subplots(figsize = (10, 8))
plt.axis([0, 60, -20, 20])


ax2.plot([0, 10.3, 21.1, 31.7, 42.1,51.8, 60],[0,-4.28,-7.9,-9.86,-9.4,-6.01, 0],'o-', co
ax2.plot(x_init,y_init,marker='+',color='b')
plt.show(block=False)
plt.pause(0.5)
plt.close()
##END FIGURE INITIAL

fit_min=[]
fit_q1=[]
fit_q2=[]
n_iter=[]

count = 0
for i_ger in range(1,N_ger_max+1):
    phi_P = select_o(phi_fitness, Boolean_M)

    I=np.zeros(len(phi_fitness))

    for i in range(len(phi_fitness)):
        I[i]=Boolean_M*phi_fitness[i]

    i_ind= list(I).index(max(I))

    s_par=pop_s[(i_ind)*N_prop*N_bit:(i_ind+1)*N_prop*N_bit]

    pop_sons = new_gen(N_ind, N_prop, N_bit, pop_s, phi_P, ...
    ProbMut, N_elit, Boolean_M, phi_fitness, ProbCross)
    pop_s = pop_sons

    phi, phi_fitness = fitness(N_ind, N_prop, N_bit, bounds, pop_s)

    I=np.zeros(len(phi_fitness))

    for i in range(len(phi_fitness)):
        I[i]=Boolean_M*phi_fitness[i]

    i_ind= list(I).index(max(I))

    s_son = pop_s[(i_ind)*N_prop*N_bit:(i_ind+1)*N_prop*N_bit]

    phi = np.asarray(phi)
    phi_array = np.asarray(phi)
    #F_objective = f_obj(phi[:,(i_ind)*2*N+1:(i_ind+1)*2*N])
    testing=np.ravel(phi,'F')
    testing_2=testing[(i_ind)*2*N : (i_ind+1)*2*N]
    #print("Valor da funcao objetivo: ",F_objective)
    print(f_obj(testing_2))

#FIGURES


    I=np.zeros(len(phi_fitness))

    for i in range(len(phi_fitness)):
        I[i]=Boolean_M*phi_fitness[i]
```

```
index_I=I.argsort(axis=0)

i_ind_min= list(I).index(max(I))
i_ind_q1=index_I[math.floor(len(I)*(1/4))]
i_ind_q2=index_I[len(I)//2]

x_init_min=np.zeros(1)
x_fin_min=60*np.ones(1)
x_init_min= np.append(x_init_min,phi[0:N,i_ind_min])
x_init_min= np.append(x_init_min,x_fin_min)
y_init_min=np.zeros(1)
y_fin_min=np.zeros(1)
y_init_min= np.append(y_init_min,phi[N:,i_ind_min])
y_init_min= np.append(y_init_min,y_fin_min)

x_init_q1=np.zeros(1)
x_fin_q1=60*np.ones(1)
x_init_q1= np.append(x_init_q1,phi[0:N,i_ind_q1])
x_init_q1= np.append(x_init_q1,x_fin_q1)
y_init_q1=np.zeros(1)
y_fin_q1=np.zeros(1)
y_init_q1= np.append(y_init_q1,phi[N:,i_ind_q1])
y_init_q1= np.append(y_init_q1,y_fin_q1)

x_init_q2=np.zeros(1)
x_fin_q2=60*np.ones(1)
x_init_q2= np.append(x_init_q2,phi[0:N,i_ind_q2])
x_init_q2= np.append(x_init_q2,x_fin_q2)
y_init_q2=np.zeros(1)
y_fin_q2=np.zeros(1)
y_init_q2= np.append(y_init_q2,phi[N:,i_ind_q2])
y_init_q2= np.append(y_init_q2,y_fin_q2)







phi_fitness_sort=np.sort(phi_fitness)
fit_min=np.append(fit_min,min(phi_fitness))
fit_q1=np.append(fit_q1,phi_fitness_sort[math.floor(len(I)*(1/4))])
fit_q2=np.append(fit_q2,phi_fitness_sort[(len(I)//2)])
n_iter=np.append(n_iter,i_ger)
```

```
    if count > 98 :
        fig3, ax3 = plt.subplots(figsize = (10, 8))
        plt.axis([0, 60, -20, 20])
        ax3.plot([0, 10.3, 21.1, 31.7, 42.1,51.8, 60], ...
        [0,-4.28,-7.9,-9.86,-9.4,-6.01, 0],'o-', color='k')
        ax3.plot(x_init_min,y_init_min,marker='+',color='b')
        ax3.plot(x_init_q1,y_init_q1,marker='+',color='g')
        ax3.plot(x_init_q2,y_init_q2,marker='+',color='r')


        fig4, ax4 = plt.subplots(figsize = (10, 8))
        ax4.scatter(n_iter,fit_min,marker='+',color='b')
        ax4.scatter(n_iter,fit_q1,marker='+',color='g')
        ax4.scatter(n_iter,fit_q2,marker='+',color='r')
        plt.show(block=False)
        plt.pause(0.1)



#END FIGURES

#STOPPING CRITERIA

    if s_par==s_son:
        count=count+1
    else:
        count=0

    if count == N_ger_eq:
        break

    print(count)


I=np.zeros(len(phi_fitness))

for i in range(len(phi_fitness)):
    I[i]=Boolean_M*phi_fitness[i]
```

```
index_I=I.argsort(axis=0)

i_ind=index_I[-1]

phi = np.asarray(phi)
print(min(fit_min))
print(phi[ (i_ind-1)*2*N+1 : i_ind*2*N ])


fig3, ax3 = plt.subplots(figsize = (10, 8))
plt.axis([0, 60, -20, 20])
ax3.plot([0, 10.3, 21.1, 31.7, 42.1,51.8, 60], ...
[0,-4.28,-7.9,-9.86,-9.4,-6.01, 0],'o-', color='k')
ax3.plot(x_init_min,y_init_min,marker='+',color='b')
ax3.plot(x_init_q1,y_init_q1,marker='+',color='g')
ax3.plot(x_init_q2,y_init_q2,marker='+',color='r')


fig4, ax4 = plt.subplots(figsize = (10, 8))
ax4.scatter(n_iter,fit_min,marker='+',color='b')
ax4.scatter(n_iter,fit_q1,marker='+',color='g')
ax4.scatter(n_iter,fit_q2,marker='+',color='r')
plt.show()
```

B.2 Objective function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import numpy as np

def f_obj (xy):



    N=len(xy)
    if N % 2 != 0:
        print("N is an odd number")
    #N - number of weights
    N = N//2
    #Li - length for each spring
    Li = 10
    L0 = Li*np.ones(N+1)
    X = np.transpose(xy[0:N])
    Y = np.transpose(xy[N:2*N+1])
    #K - stiffness of spring i
    K=np.ones(N+1)
    for i in range(N+1):
        K[i] = 500 + 200*(abs(N/3-(i+1))**2)

    #W - weight
    W =np.ones(N)
    for i in range(N):
        W[i] = 50*(i+1)
    Xt = np.insert(X,0,0)
    Xt = np.append(Xt,Li*(N+1))
    Yt = np.insert(Y,0,0)
    Yt = np.append(Yt,0)

    #Delta_L - deformation of spring i
    diff1=np.diff(Xt,1)
    diff2=np.diff(Yt,1)
    Delta_L = (diff1**2 + diff2**2)**(0.5)-L0
    #F - Potential energy
    F = 0.5*np.sum(K.conj()*Delta_L**2, axis=0)+ ...
    np.sum(W.conj()*Y, axis=0)

    return F
```

B.3 Population function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import random
import numpy as np
random.seed(0)
def population (N_ind,N_prop,N_bit):

    pop_array=np.zeros(N_ind*N_prop*N_bit)

    for i in range(N_ind*N_prop*N_bit):
        pop_array[i] = random.randint(0, 1)
    pop_s = ' '.join([str(elem) for elem in pop_array])

    pop_s=pop_s.replace(".0", "")
    pop_s=pop_s.replace(" ", "")
    print(len(pop_s))
    return pop_s
```

B.4 New generation function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import numpy as np
import math
from roulette import roulette
from crossover import crossover
from mutation import mutation

def new_gen (N_ind, N_prop, N_bit, pop_s, phi_P, ...
ProbMut, N_elit, Boolean_M, phi_fitness, ProbCross):

    pop_sons=list(pop_s) # we convert in list elements so ...
    that we can change each element in place

    I=np.zeros(len(phi_fitness))

    for i in range(len(phi_fitness)):
       I[i]=Boolean_M*phi_fitness[i]

    I=I.argsort(axis=0)
    for i_ind in range(N_elit):
        pop_sons[(i_ind)*N_prop*N_bit:(i_ind+1)*N_prop*N_bit] ...
        =pop_s[I[-1-i_ind]*N_prop*N_bit:(I[-1-i_ind]+1)*N_prop*N_bit]



    if (N_ind-N_elit) % 2 != 0:
        print('Erro: N_ind - N_elit ímpar!')

    for i_ind in range(N_elit,N_ind,2): # (4,500,2)
        par1_ind, par2_ind = roulette(phi_P)

        for i_prop in range(N_prop): # (0:10)
            #Parent 1
            s1=pop_s[(par1_ind[0])*N_prop*N_bit + (i_prop)*N_bit ...
            : (par1_ind[0])*N_prop*N_bit + (i_prop)*N_bit + N_bit]
            #Parent 2
            s2=pop_s[(par2_ind[0])*N_prop*N_bit + (i_prop)*N_bit ...
            : (par2_ind[0])*N_prop*N_bit + (i_prop)*N_bit + N_bit]

            son_1,son_2 = crossover(s1, s2, N_bit, ProbCross)

            son_1 = mutation(son_1, N_bit, ProbMut)
            son_2 = mutation(son_2, N_bit, ProbMut)

            pop_sons[(i_ind)*N_prop*N_bit+i_prop*N_bit:(i_ind)* ...
            N_prop*N_bit + i_prop*N_bit+N_bit] = son_1
            pop_sons[(i_ind+1)*N_prop*N_bit+i_prop*N_bit: ...
            (i_ind+1)*N_prop*N_bit + i_prop*N_bit+N_bit] = son_2

    # We convert back to string
    pop_sons_str =''.join(pop_sons)


    return pop_sons_str
```

B.5 Fitness function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
from f_obj import f_obj
from bin2dec import bin2dec
import numpy as np

def fitness (N_ind, N_prop, N_bit, bounds, pop_s):

    phi_min = bounds[:,0]
    phi_max = bounds[:,1]
    #matrix whose columns are the properties of each individual
    phi = np.zeros((N_prop, 0))
    #FObj vector for each individual
    phi_fitness = np.zeros((1, N_ind))
    #column with the precision (increment) for each property
    d_phi = np.divide((phi_max-phi_min),(2**N_bit-1))
    #column with number of increments in each property
    i_d_phi = np.zeros(N_prop)

    #fitness of each individual (i_ind)
    for i_ind in range(N_ind):
        for i_prop in range(N_prop):
            s = pop_s[(i_ind)*N_prop*N_bit + (i_prop)*N_bit ...
            : (i_ind)*N_prop*N_bit + (i_prop)*N_bit + N_bit]
            i_d_phi[i_prop] = bin2dec(s)

        #property value
        phi = np.column_stack((phi,phi_min + i_d_phi*d_phi))
        #value of FObj for the property
        phi_fitness[:,i_ind] = f_obj(phi[:,i_ind])
    phi_fitness=phi_fitness[0]



    return phi, phi_fitness
```

B.6 Select function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import numpy as np
import math

def select_o (phi_fitness, Boolean_M):



    phi_P = Boolean_M*phi_fitness
    #lowest value of FObj function
    phi_P_min = min(phi_P)
    #highest value of the FObj function
    phi_P_max = max(phi_P)

    phi_P = (phi_P-phi_P_min) + 0.01*(phi_P_max-phi_P_min)

    phi_P = np.cumsum(phi_P)

    temp = phi_P[-1]
    phi_P = phi_P/temp


    return phi_P
```

B.7 Roulette method function

@author: Tomás Schuller

`schuller@fe.up.pt`

Otimizacao 20/21

```
import random
import numpy as np


def roulette (phi_P):
    #random.seed('default')
    #index of Parent 1
    i_ind = np.argwhere(phi_P>random.random())
    #index of Parent 2
    j_ind = np.argwhere(phi_P>random.random())
    return i_ind[0], j_ind[0]
```