Università degli Studi di Trento
Dottorato di ricerca in Modellazione, Conservazione e Controllo dei Materiali e delle Strutture

Structural optimization: an approach based on genetic algorithms and parallel computing

PhD candidate: Massimiliano Petrucci

Tutor: Prof. Enzo Siviero - Università IUAV di Venezia
Co-Tutor: Prof. Massimo Majowiecki - Università IUAV di Venezia

March, 20 2009

The present text has been typeset with LaTeX 2_ε

Massimiliano Petrucci
http://www.massimilianopetrucci.com
email: max@petrucci.it

0.1 Introduction

Everybody optimizes! Every aspect of life and technology are often deeply involved in optimization. Companies optimize in order to reduce costs. Finance investors optimize their portfolio in order to minimize risks and obtain high rates of return. Even nature optimizes. Physical systems spontaneously tend to a state of minimum energy. In every branch of engineering the designers are always called to optimize different aspects of their projects.

The importance of structural optimization (seen as minimum design of weight of structures) was first recognized by the aerospace industry where aircraft structural designs are often controlled more by weight than by cost considerations. In other industries dealing with civil, mechanical and automotive engineering systems, minimization of cost is the most important objective although the weight of the system affect its cost and performance [67]. Nowadays both people and industries are more and more sensitive to the environmental problems related to the scarcity of raw material and conventional energy sources. This fact leads to a growing demand for lightweight, efficient and low cost structures. As a consequence the interest in design optimization is increasing.

This work focus on optimization techniques with special emphasis on topics regarding structural optimization. The subject is treated both from a theoretical and practical point of view. A software has also been developed in order to implement some of the methods that have been presented. Some numerical simulations have been performed and the final results are critically discussed.

0.2 The research activity

The original idea of my research activity was the study of optimization methods dealing with truss structures.

The work has been divided in several phases. First of all I tried to evaluate the *state of art* in the different fields of optimization as appear from scientific literature and books on this topic.

Secondly I compared the different methodologies in order to understand which could be adapted to for optimization of discrete structures at real scale. In particular I have found worthy of interest the methods based on evolving strategies. So I concentrated on this approach, as it appeared to me very promising, examining in particular several implementation of genetic algorithms. In this phase I realized that a great improvement in performance could be achieved adopting parallel computing techniques, being genetic algorithms very adaptable to this kind of computation strategies.

So I spent a part of my time in studying topics regarding parallel programming and the creation of a computer cluster. After few consideration

I decided for a *Beowulf* architecture and chose C++ as programming language.

In the final part of my work I developed an optimization tool that is able to work with 3D truss-like structure and that benefits of the advantages of parallel computing. Several numerical simulations have been performed and some results are discussed at the end of this thesis. The software implementation has been a very important part of this work since it required the solving of several problems that at a first sight, with only a theorycal approach, I realized I have missed. In fact programming requires a complete comprehension of all the involved processes in order to have working application. The simulations have also been interesting because they have shown the importance of the calibration of the parameters required by the optimization procedure.

0.3 Structure of the thesis

This work focus on optimization techniques with special emphasis on topics regarding structural optimization.

The first Chapter is an introduction to concepts, terminology and problems related to optimization procedures.

Chapter Two covers different optimization methods, with specific emphasis on the field of structural optimization. It starts by explaining the basis of optimization and derivative based classical methodologies, then follows by explaining heuristic methods of optimization. For the sake of completeness only a little introduction is given about genetic algorithms since they are deeply discussed in another chapter. Finally a description of ESO, seen as a sort of introduction to evolving algorithms, is given.

Chapter Three focuses on genetic algorithms. Their main features, advantages and drawbacks are discussed.

Chapter Four deals with various topics on parallel computing that will be useful for the design of the GA's parallelization.

Chapter Five gives a description of the software and explains how it works. It includes also the numerical simulations with comments on the results. At the end, some conclusions are given.

Chapter 1

Optimization of structures

1.1 Introduction

In general, the optimization is a set of actions we can do in order to maximize one ore more objectives we have previously defined. To apply this tool we must have a comprehensive understanding of the real system we want to study, in other terms we have a *model* of it and we know the *unknown variables* that describe it. But this is not sufficient. We also need to know the *objective* we are looking for. When each of these requirements are well-defined, we can apply optimization.

Nowadays a lot of techniques have been defined, but none of them can be blindly applied to solve every problem. Each class of problems has a set of techniques which perform better than others.

Strictly speaking the optimization is the minimization (or the maximization[1]) of an *objective* function $f(x)$ subject to constraints in its own *variables* x (also called *parameters*) and could be expressed, from a mathematical perspective, in the following way[48]:

$$\min f(x)_{x\in\Re^n} : \text{subject to } \begin{cases} c_i(x) = 0, & i \in \Upsilon \\ c_j(x) \geq 0, & j \in \Xi \end{cases} \tag{1.1}$$

where c_i and c_j are functions in x while Υ and Ξ are set of indices. Those function represents the *constraints*.

The system (1.1) describe a generic optimization problem. In structural optimization the variables are often related to stresses, displacements, vibration frequencies or others. From a *structural* the point of view, many authors agree on the following classification about the different levels of optimization [8]:

size optimization deals with minimization (or maximization) of one or more response variable (such as tension, deformation, frequency of vibration) acting on one or more design variables (for example, thickness

[1] It is sufficient substitute f with $-f$ in (1.1).

of a plate or cross section of a bar) while respecting some conditions (equilibrium, restraints, and so on). In such a case the domain of design variables is known *a priori* and is fixed during the whole optimization process;

shape optimization aims to find the optimal shape of domain, which is no more fixed and become a design variable itself;

topology optimization for continuum structures deals with the number, position, shape of holes and topology of the domain.

1.1.1 Design variables

The parameters that can change, in order to optimize the structures, are called *design variables* and represent the degrees of freedom of the structures[2].

These variables can be *discrete* or *continuous*. Typically discrete variables can assume only isolated values, taken from a list of possible values. The can be expressed with an integer index which reference the value assumed with the position inside the list of possible values. This choice is quite common in steel structures where, for manufacturing reasons, the cross sectional dimension are usually mapped to a finite set of commercial available cross sections. Continuous variable instead can freely change inside a defined range of variation.

Usually solving an optimization problem with discrete design variables is much more difficult than solving an analogous problem with continuous variables. For this, when is possible, the discrete form of the problem is disregarded towards a continuous approach. Only once the (continuous) optimum is found, the design variable are translated to the nearest discrete value. However this *rounding off* techniques works well only if the available values of the integer variables are spaced very (or *enough*) close each to another[3].

Instead when the discrete solutions are spaced too far apart, the problem must be solved with discrete variables. The branch of mathematical programming which deals with this case is called *integer programming*.

The choice of each design variable should be done carefully, since it can deeply interfere with the success of the optimization procedure. In particular the design variables should be consistent with the model: for continuous structures this means that the distribution of design variables should be

[2]The could be cross sectional dimensions, member sizes, geometrical parameters, material properties...

[3]In other words we can say that the variables are spaced reasonably close when changing the continuous variable to the nearest integer does not change the response of the structure substantially.

coarser than the nodes of the discretized model. With frame structure usually this can't be done since usually each finite element is mapped one-to-one to a member of the structure.

In the following design variables are expressed in the vector $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$.

1.1.2 Objective function

The measure of the effectiveness of the optimization process is expressed by one function $f(\mathbf{x})$ or by a set of functions $\mathbf{f}(\mathbf{x}) = \{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_p(\mathbf{x})\}$ that are called *objective functions*. In the field of structural optimization the most common are weight, displacements, stresses, vibration frequencies, buckling loads, cost or any combination of them. In the latter case, i.e. when more than a single objective are taken in account, the optimization is defined as *multicriteria*.

Dealing with more objectives could be really difficult and if it is possible it is avoided. In many cases this can be done trying to reduce the number of objective functions to one. This can be done mainly in two ways:

- A new composite objective function is introduced, replacing all the objectives.[4]
- The most important objective function is chose and taken as the only objective function while limits are imposed on the remaining objectives.

Edgeworth-Pareto optimization

When it is difficult to chose (or to weight) between the objectives an help could come from the so-called *Edgeworth-Pareto optimization*: a branch of mathematical programming which gives a systematical approach to this kind of problems. It was firstly applied in the field of multicriteria structural optimization by [71] and [72]. For details see [67].

A vector of design variables $\mathbf{x}^*$ is Edgeworth-Pareto optimal if, for any other vector of design variables $\mathbf{x}$, either the values assumed by all the objective functions remain the same or at least one of the worsens compared to its value at $\mathbf{x}^*$. In order to estimate the values of weights an iterative procedure based on a series of Edgeworth-Pareto optimal designs can be successfully adopted, after that, constraints can be imposed.

A possible approach could be based on the minimization of the deviation of the individual objective functions from their individual minimal values. The expression

$$d_i(\mathbf{x}) = \frac{f_i(\mathbf{x}) - f_i^*}{f_i^*} \quad \text{i=1,...,p} \tag{1.2}$$

[4] For example, given p objectives o_i, p weight w_i are introduced to model the different importance among the different objectives, and the final objective is expressed in the form $f(x) = w_1 f_1 + w_2 f_2 + \ldots + w_p f_p$.

where

- f_i^* is the value assumed by f_i at $\mathbf{x}_i^*$ as a result of the independent minimization of the single i-th objective function.

represents the normalized distance of each of the objective functions (at a certain $\mathbf{x}$) from its individual optimum. Hence the Edgeworth-Pareto can be usually posed in two forms:

- as a problem of minimization of the largest deviation of the objective functions from their individual minima (l_∞ norm)

$$\text{minimize} \quad \max_{i=1,\ldots,p} [d_i(\mathbf{x})] \tag{1.3}$$

- as a problem of minimization of the distance (i.e. the l_2 or *euclidean* norm) from the reference point $\mathbf{f}^* = (f_1^*, f_2^*, \ldots, f_p^*)$ to $\mathbf{f} = (f_1, f_2, \ldots, f_p)$

$$\text{minimize} \quad \sum_{i=1}^{p} d_i^2 \tag{1.4}$$

  where, in (1.4), weighting coefficients could eventually be used in order to differentiate the importance in contribution of the single objective function f_i.

1.1.3 Constraints

In the formulation of an optimization problem, the values of design variables cant't vary freely but are subjected to constraints. These can be present in two forms:

- *inequality constraints* which impose upper or lower limits to a design variables.

- *equality constraints* which impose that a design variable must assume a known value.[5]

1.2 The solution process

The problem expressed in the form (1.1) can be solved in many different ways. For example the so-called *search methods* are numerical search techniques that start from an initial design and proceed in small steps to improve

[5]Most techniques developed for non linear optimization problems can usually handle only inequality constraints. When an equality constraint must be introduced, a common strategy is to impose two inequality constraints that form an upper and lower bound constraint that coincide with the equality constraint value we want to replace.

the value of the objective function or the degree of compliance with the constraints, or both. The search is terminated when no progress can be made in improving the objective function without violating some of the constraints, or when that progress become very slow. Others methods employ the necessary conditions that must be satisfied at minimum but many others are available [49]. Most of them will be discussed in the following chapters.

Roughly speaking, in the optimization of continua it's necessary to understand if, in each point of the domain, the material should be present or not. Typically a finite element approach is adopted, discretizing the domain a mesh and conducting an analysis to understand if each finite element must exist or not. Operating in this way, initially the structural topology is undefined, being itself a design variable.

For truss-like structures a common methodology is the *ground structure* approach (GS) which is based on the research of all the possible connection between a fixed number of points. The optimization procedure, using a finite element analysis, evaluates the connections and states between which nodes a member must exist, eliminating unnecessary bars.

In any case, i.e. for both continuum and discrete structures, the above optimization is a discrete problem that is transformed into a continuous problem by introducing a continuous function that, for continua, is a density parameter while for discrete structures represents the area of cross sections (admitting the existence of members with area equals to zero).

A drawback of those approach (for both continuum and discrete structures) is that, since the nodal points are not taken as design variables, the choice of the number of nodes and nodal positions is fundamental for the success of the procedure and to obtain an efficient topology design, a dense mesh (meaning an high computational cost) is required. For this reason the original GS method has been extended with the introduction of the optimization of both the position and connectivity of a certain number of nodes, leading to a sort of two-level combined optimization [8]. Unfortunately this approach is not satisfactory because it's hard to handle mathematically. Moreover,it has been adopted only for specific problems at little scale and, until now, there is no evidence that could also be used for more general problems or practical applications at larger scale.

In the past *Optimality Criteria* (OC) and *Mathematical Programming* (MP) methods gained a lot of popularity among researchers . Among them Vanderplaats [70] used a MP approach employing with success several numerical techniques such as *gradient method*, *sequential quadratic programming* and *approximation methods*. Most of them consider approximated objective and constraints functions based on Taylor series. Therefore, this approach allows, for example, to linearize the nonlinear problem before employing the algorithm for searching the optimal solution. On other occasions, techniques based on *Lagrange multipliers* or *linear programming methods* are used. Unfortunately neither of these methods is so robust and efficient to

be applied in general (or almost in a large class of problems). Also the OC methods [60], which are based on the Kuhn-Tucker theorem and that roughly speaking can be seen as a sort of generalization of the method of the lagrangian multipliers for the search of global minimum for a regular and convex function, made a success in the past but at, the end, they are characterized by the same limitation.

In fact, both OC and MP, are mathematical strategies (in literature are defined as *hard computing* techniques) that today appear less interesting because of the intrinsic difficulties that can be encountered for their application in structural problems. This is mainly due to the fact that they need to compute derivatives of functions which are, in great part, not regular. In addition their deterministic approach has been criticized since it can't take into account the uncertainty and approximations that are commonly present in real applications. Recently Adeli [2] tried to address this question with probability and fuzzy strategies in conjunction with evolutive algorithms.

From this study many other techniques have come to light that are worthy to remember such as the *Evolutionary Structural Optimization* (ESO) that was developed by Xie and Steven. This is a very flexible methodology since it can deal with different conditions of constraints (such as on tensions, stiffness, displacements or their difference, vibration frequencies) both on continuum and discrete problems. It is based on the definition of a *rejection criterion* which depends strictly from the adopted constraints and that is used to find the locations where the material is unnecessary and can be removed or eventually migrated to more stresses places if needed. It relies on the simple concept that by slowly removing inefficient material from a structure, the residual shape evolves in the direction of making the structure better. Despite its simplicity, it can produce result comparable to more complicated techniques and it can be used for a wide range of structural problems including statics, dinamics and buckling [74].

More sophisticates approaches are the so-called *solid isotropic material with penalization* (SIMP), or the Homogenization Method, pionereed by Bendsøe [8]. They have been successfully adopted for topology, shape and material optimization of both continuum and discrete linear elastic structures. The development of the Homogenization method of optimization led to an important step forward in the area of structural optimization. This is because this method can do two things at once. It can determine both the topology of the structure, and the microstructure of the material that constitutes the structure. This was achieved by minimizing the compliance of the structure using classical quadratic programming techniques, where material density in each element was a design variable. It has also been applied to discrete linear elastic structures [7].

Finally, in the opinion of the author of the present work, today the most promising approaches seem just those based on evolutive algorithms that, taking their inspiration from the observation of natural events, paved the

way for a new vision in optimization techniques. From a certain point of view, even ESO can be seen, in some way, a first step towards new strategies of resolution, but genetic algorithms and genetic programming [47] are the most important demonstrations of totally a different approach compared to the hard computing schemes seen above.

Genetic algorithms (GA) takes their inspiration from the evolutionist theory due to Darwin and its natural law of the survival of the fittest. This methodology perform a global search of the optimal solution by the succeeding of populations, made of random created individuals, that evolves and reproduce according to the laws of nature. The aim is to select a population of individuals that at each new generation improve respect to the previous. At the end of the selection, the individual with highest fitness represent the optimal solution. The algorithm use only discrete variables (since also real design variables are coded in binary string). So these algorithm works always on a finite set of points that belong to the so called research space (thus it must be defined before the optimization begins). In the form they have been described, GAs work only for unconstrained optimization problems but constrained optimization problems can be reduced to the unconstrained case simply by matching with a penalty or augmented lagrangian method. As will be explained in detail in the next chapters, genetic algorithms can be seen, from different points of view, as a really valuable alternative to the more traditional hill climbing methods.

1.3 Trends in structural optimization

To optimize, it is necessary to have clear in mind the objective of the optimization because only from a good choice of the objective could stem useful results. In structural optimization common objective are:

- Minimization of weight.
- Maximization of stiffness.
- Minimization of cost design.
- Some control on free vibrations.
- ...
- Any combination of the previous items.

Typically older papers on optimization of structures focused the attention on the maximization of stiffness combined with reduction of weight. Later were introduced techniques dealing with other kinds of objectives and restraints (for example on absolute and relative displacements or on frequencies). Finally nonlinear analysis has been used as well [68].

During the years a really large number of research papers have been published but most them has in common the fact that they consider, as objective, the minimization of the weight as it were an implicit assumption that weight of structure is the best measure for evaluating the cost of the structure. Even if, roughly speaking, this is partly true, there is the risk to neglect other important terms, besides material cost, that contribute to define the final cost. Recently [2] some critics raised against this approach by some authors who correctly stated that actually the *minimum weight cost* not necessarily coincides with the *minimum cost design*.

In their opinion, a modern point of view should evaluate the cost in terms of the life-cycle cost of the structure itself, taking in account more terms, such as the costs of

- materials;
- fabrication;
- erection;
- maintenance;
- disassembling at the end of structure life.

So it's clear that this approach in cost optimization introduce more realism but, in this way, additional difficulties are encountered. Among them there is the definition of the cost function and the uncertainties and fuzziness involved in determining the cost parameters.

Unfortunately, for real problems, constraint evaluation very often involves many sources of imprecision and approximation. If the algorithm is forced to satisfy these condition exactly it could fail, in the sense of missing the true optimal solution.

To address this issue, some authors [2] integrated techniques based on probability and fuzzy theories to evolutive algorithms. The first approach is known as reliability-based optimization and relies on the theory of probability[6]. The second is known as fuzzy optimization and is established around the concept of possibility[7].

1.4 Optimization tools

During the last decades many computational methods have gained success in many fields of engineering. Among them the Finite Element Method has proven to be of common use not only in academic contexts but also between professional engineers which, in the recent years, often need to execute more and more sophisticated calculations.

[6] Under the assumption that natural events can be considered as statistical variables.
[7] Expressed by membership functions according to fuzzy set theory of Zadeh [1].

For this reason Finite Element Applications (FEA) are now essential tools for modern design. Furthermore, such a diffusion is due partly to the great advances in technology which bring to us computers much more powerful respect to only few years ago; partly to the graphic user interfaces that are more intuitive and user friendly so that computers can now be used also by non specialists.

The same things can not be said about the application of the numerical optimization methods which, despite their advance, are actually limited to academic research or very specialized companies in industry. In structural engineering the practical applications are actually really rare. Much wider is their diffusion in the fields of mechanics, aeronautics and electronics.

Maybe, several reasons can be found to explain a so slow penetration of this methods in the field of structures (specially if compared to other branches of technology). Firstly the subject is really complex, requiring also a deep knowledge and solid background in many topics of structural mechanics (such as the finite element methods). Secondly (and this is an important difference with the FEA case) a unique formulation that can be successfully applied to a really large class of problems is still missing, and so the creation of "multipurpose" software tools that could deal with a sufficiently wide set of structural problems appear to be a really difficult task. This explains the lack of interest in practical optimization methods by professional engineers.

Formerly, further difficulties arose from the high computational cost deriving from the optimization algorithms. Specially from those that required a large number of finite element analysis, leading to optimization procedures that could be very time-consuming activities. In such a case the growing computation power of modern processors, the large availability of commodity hardware and the techniques of parallel computing can be, today, a valid response since they allow to obtain, cheaply, high computing power with relatively low cost using only *off the shelf* hardware and free open source software.

Operating in this way become possible to achieve so high performance that, only a few years ago, could not even be imagined for a simple user, being accessible only to research centers. Unfortunately all those benefits doesn't come at no cost: parallel computing code is much more difficult to implement when compared to traditional programming. This is the reason because parallel application are not so popular among users yet, even if today most computers and laptop are provided with multicore processors.

Chapter 2

Optimization methods

2.1 Introduction

Before the advent of high speed computation most of the solutions of structural analysis problems were mainly based on formulations employing differential equations. These equations were solved analytically (for example by using infinite series) and the unknowns were, typically, displacements or stresses, defined on continuum domain. Occasionally some numerical techniques could be employed, but usually only towards the end of the solution process.

In the early beginning of structural optimization a very similar approach was used because the unknowns were functions representing the structural properties (to be optimized). In such case the solution was based on *calculus of variations*. And the class of structural problems concerning the seek of the optimum of function (which model the structural properties to minimize or maximize) is known as *function* or *distributed parameter* structural optimization.

Starting from late $50s$ and $60s$ the introduction of electronic calculator in conjunction with the use numerical methods produced a deep transformation in the solution procedure. Among these, the *Finite Element Method (FEM)* gained a wide success, being extremely well suited to computer implementations[1].

This approach makes possible a strong simplification of the structural problems because, discretizing the domain, does not assume as unknown functions, but discrete values of displacements or stresses in a finite set of points of the model, called *nodes*. As a consequence, the differential

[1]FEM is based on *displacements* method of solution of structural problems. This is undoubtedly one of the main reason of its success between engineers. The displacement methods , in fact, is easy to translate in algorithms. In contrast the *force* method should require a much more complex implementation that only recently has been clearly formalized.

equations of the problem are translated into algebraic equations, much more easier to solve.

The same transition happened to structural optimization. With the natural adoption of the Finite Element Method, the searching of an optimum function has been substituted by the searching of the optimum values for a finite set of parameters; this is known as *parameter optimization*. The associated mathematic discipline is called *mathematical programming*.

2.2 General formulation

An optimization problem can therefore be expressed in the form

$$\begin{aligned} &\textbf{minimize} && f(\mathbf{x}) && && (2.1)\\ &\textbf{such that} && g_j(\mathbf{x}) \geq 0 && j = 1, \ldots, n_g && (2.2)\\ &&& h_k(\mathbf{x}) = 0 && k = 1, \ldots, n_e && (2.3) \end{aligned}$$

where $\mathbf{x}$ is the n-dimensional vector of design variables while $h_j(\mathbf{x})$ and $g_j(\mathbf{x})$ are respectively the equality and inequality constraints.

The choice of *minimizing* the objective function f in (2.1) is not restrictive in any way because it can be easily transformed into a *maximizing* problem considering the negative of f. For the same reason the sign adopted of inequalities constraints is purely conventional since if there is an inequality of opposite sign, such as $g_j(\mathbf{x}) \leq 0$, it can be transformed into a $\geq$ type by multiplying the expression by -1.

If both the objective function and the constraints are linear functions of the design variables $\mathbf{x}$ the optimization problem is said to be *linear*; otherwise *not linear*. Therefore in a linear problem the objective function f can be expressed in the form

$$f(\mathbf{x}) = c_1 x_1 + \ldots + c_n x_n = \mathbf{c}^T \mathbf{x} \tag{2.4}$$

and can be solved with a mathematical branch called *linear programming*.

2.3 Classical methods

These methods are based on the classical tools adopted to find the minima and maxima of functions and functionals and are based on the techniques of the ordinary differential calculus and the calculus of variations. Even this approach can be successfully adopted to find exact solutions in closed form only for very simple cases, because of the lack of realism due to the necessary simplifying assumptions that are implicitly required, it can be usefully employed to validate solutions based on numerical approximated methods. Unfortunately most realistic optimization problems cannot be

simplified to the point where the can be solved by techniques based on classical tools.

2.3.1 Differential calculus

The most common method for unconstrained optimization is differential calculus. This approach is based on the necessary and sufficient condition for the existence of a stationary point in a function, as explained below.

It is know that a continuously differentiable objective function $f(x_1, x_2, \ldots, x_n)$ of n independent design variables attains its minimum or maximum value inside its domain[2] when the n partial derivatives of the design variable $\mathbf{x}$ vanish simultaneously[3], or

$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial x_2} = \ldots = \frac{\partial f}{\partial x_n} = 0 \tag{2.5}$$

The (2.5) is a necessary condition for the existence of a stationary point. A stationary point $\mathbf{x}^*$ satisfies the condition $\frac{\partial f}{\partial x_i^*} \ \forall i = 1, n$. If the function is a *scalar-valued* type, the vector of first derivatives is the gradient vector ∇f and is commonly used to find search directions in optimization algorithms.

The sufficient condition for a stationary point $\mathbf{x}^*$ to be an extreme point can be expressed by the Hessian matrix[4] $\mathbf{H}$ of the objective function f.

$$\mathbf{H}(\mathbf{x}^*) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x}^*)}{\partial x_n^2} \end{bmatrix} \tag{2.6}$$

In fact, it can be proven that if $\mathbf{H}$ evaluated at $\mathbf{x}^*$ is positive-defined then the stationary point $\mathbf{x}^*$ is a minimum, otherwise if $\mathbf{H}(\mathbf{x}^*)$ is negative-defined than $\mathbf{x}^*$ is a maximum.

definition 1 *Given a symmetric matrix $\boldsymbol{A}$ of order n, and an n-dimensional vector $\boldsymbol{x}$ if $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} > 0 \ \forall \ \boldsymbol{x} \in \Re^n$ and $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = 0 \ \Leftrightarrow \ \boldsymbol{x} = \boldsymbol{0}$ then $\boldsymbol{A}$ is said to be positive-defined. Otherwise, if $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} < 0 \ \forall \ \boldsymbol{x} \in \Re^n$ and $\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x} = 0 \ \Leftrightarrow \ \boldsymbol{x} = \boldsymbol{0}$ then $\boldsymbol{A}$ is said to be negative-defined.*

Well known properties of matrix analysis can be adopted to further investigate the semi or full positive or negative definition of $\mathbf{H}$ to see if $\mathbf{x}^*$ could be a minimum, maximum or a saddle point.

This seems a very elegant method, however, the conditions for $\mathbf{H}$ to be either positive or negative definite may not always be satisfied, requiring

[2] the design space $\Re^n$.

[3] without considering, at this stage, constraints of any kind.

[4] i.e. the matrix of the second derivatives

higher derivatives of the objective function to satisfy these conditions. Although it can be used to find the optimum of simple structural systems, the OF must be twice differentiable, something that may not always be achievable, especially when dealing with discrete finite elements structural domains.

2.3.2 Variational calculus

Sometimes the objective function is expressed in the form of a *functional*. In this case the aim of the optimization is to find a function (or a set of functions) that minimize the numerical value assumed by the functional[5]. The problem can then be solved adopting the methods of the branch of mathematics known as *calculus of variations* that deals with the minima and maxima of functionals. Certain aspects of these procedures are similar to techniques shown in 2.3.1. More details can be found in [67].

2.3.3 Constrained optimization

Equally constrained problems can be expressed in the form

$$\text{minimize} \quad f(\mathbf{x}) \tag{2.7}$$

$$\text{subject to} \quad h_j(\mathbf{x}) = 0 \qquad (j = 1, 2, \ldots, n_e) \tag{2.8}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ and the number of equality constraints $n_e \leq n$[6].

Variable elimination (or direct substitution)

If the equality constraints can be solved explicitly for n_e design variables then the objective function can be written in a new form[7]. This approach is known as *variable elimination* or *direct substitution*. It should be noted that the new objective function won't subjected to any constraints and it can be minimized using the procedures described in 2.3.1 and 2.3.2.

Method of Lagrange multipliers

When the constraints can't be solved explicitily[8] is commonly used the so-called *method of Lagrange multipliers*.

For an objective function $f(\mathbf{x})$ of n design variables to be a minimum, the differential change df in the objective function must still vanish.

$$df = \frac{\partial f}{\partial x_1} x_1 + \frac{\partial f}{\partial x_2} x_2 + \ldots + \frac{\partial f}{\partial x_n} x_n = 0 \tag{2.9}$$

[5] A functional J can be expressed as $J = \int_a^b F(x, y, y')dx$ where $y' = \frac{dy}{dx}$. In a more general case F can be a function of more than one function $(y_1, y_1, \ldots, y_p)$ and each of these functions can depend on n independent variables $(x_1, x_2, \ldots, x_n)$.

[6] In fact, if $n_e > n$ the problem is *over-constrained* and in general it admits no solution.

[7] in terms of $n - n_e$ independent variables

[8] For example when they are defined in terms of integrals.

The derivative terms can not be set to zero individually because the differential changes in the design variables $(dx_1, dx_2, \ldots, dx_n)$ are strictly dependent each on another through the constraint equations (2.8). If, for the sake of simplicity, we assume the existence of only a single constraint equation $h(\mathbf{x}) = 0$, the differential changes in the design variables are related through

$$dh = \frac{\partial h}{\partial x_1} x_1 + \frac{\partial h}{\partial x_2} x_2 + \ldots + \frac{\partial h}{\partial x_n} x_n = 0 \tag{2.10}$$

Multiplying (2.10) by an arbitrary constant λ, and adding the result to the (2.9) we obtain

$$\left(\frac{\partial f}{\partial x_1} + \lambda \frac{\partial h}{\partial x_1}\right) + \left(\frac{\partial f}{\partial x_2} + \lambda \frac{\partial h}{\partial x_2}\right) + \ldots + \left(\frac{\partial f}{\partial x_n} + \lambda \frac{\partial h}{\partial x_n}\right) = 0 \tag{2.11}$$

λ is determined so that the quantities inside each of the parenthesis vanish in order to satisfy (2.11). This leads to a system of n equations for $n+1$ unknowns[9]. The constraints equation $h(\mathbf{x}) = 0$ provides the $(n+1)th$ required relation which is necessary to solve the system.

These considerations can be easily extended to the case of multiple constraints functions introducing a *Lagrange multiplier* λ for each of the constraints functions.

In other words an optimization problem with n design variables and n_e equality constraints (as seen in (2.7)) is equivalent to an *unconstrained* problem with an auxiliary function

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{j=1}^{n_e} \lambda_j h_j \tag{2.12}$$

where $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_{n_e})$. The optimum value of the design variables $\mathbf{x} = (x_1, x_2, \ldots, x_{n_e})$ can be obtained by solving, for the $n + n_e$ unknowns, a system of $n + n_e$ equations

$$\frac{\partial \mathcal{L}}{\partial x_i} = 0, \qquad i = 1, \ldots, n; \tag{2.13}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = 0, \qquad j = 1, \ldots, n_e. \tag{2.14}$$

In general we are interested to extremize functionals of several functions and their derivatives with respect to more than one independent variable like the following J

$$J = \int_a^b F(x, y_1, y_2, y_1', y_2', y_2'') dx \tag{2.15}$$

[9]The n design variables and the unknown multiplier λ.

In addition there may be m *subsidiary constraints* in the form

$$h_i\left(x1,\ldots,x_n,y_1,\ldots,y_p,\frac{\partial y_1}{\partial x_1},\ldots,\frac{\partial y_p}{\partial y_n}\right)=0 \tag{2.16}$$

for $i=1,\ldots,m$. In this case the method of Lagrange multiplier will lead to an auxiliary functional in the form

$$\mathcal{L}=\int_v\left(f+\sum_{i=1}^{m}\lambda_i h_i\right)dv \tag{2.17}$$

where the Lagrange multipliers are no longer constants but functions of $x1,\ldots,x_n$.

example In the problem of constrained optimization described in 2.53 the equality constraints 2.54 state that $g_i(x)=b_i$ $(\forall i)$, so the problem remain unaffected if the objective function $f(x)$ is substituted, for any values assumed by the *Lagrangian multipliers* λ_i, by the function

$$\mathcal{L}(x,\lambda)=f(x)+\sum_{i=1}^{N}\lambda_i(g_i(x)-b_i) \tag{2.18}$$

which is the *Lagrangian function.* If the Lagrangian multipliers are found such that the point x where $L(x,\lambda)$ is minimized satisfies all the constraints $g_i(x)=b_i$ then the original constrained problem 2.53 has also been solved.

In order to find the values of λ_i that minimize (2.18), its derivative must be computed and the equated to the equality constraints:

$$\frac{\partial\mathcal{L}(x,\lambda)}{\partial x_j}=\frac{\partial f(x)}{\partial x_j}+\sum_{i=1}^{m}\lambda_i\frac{\partial g_i(x)}{\partial x_j}=0 \qquad (j=1,\ldots,N) \tag{2.19}$$

$$\frac{\partial\mathcal{L}(x,\lambda)}{\lambda_i}=g_i(x_1,\ldots,x_n)-b_i=0 \qquad (i=1,\ldots,M) \tag{2.20}$$

obtaining a system of $N+M$ equations in $N+M$ unknowns[10] that in most cases can be solved.

2.3.4 Local constraints

In many structural problems there are local constraints [11] that can be expressed in a form similar to the subsidiary constraints (2.16) except that the equalities are replaced by inequalities.

$$g_i\left(x1,\ldots,x_n,y_1,\ldots,y_p,\frac{\partial y_1}{\partial x_1},\ldots,\frac{\partial y_p}{\partial y_n}\right)\geq 0 \tag{2.21}$$

[10] $x_1,\ldots,x_N$ and $\lambda_1,\ldots,\lambda_M$.

[11] i.e. constraints in stress

for $i = 1, \ldots, m$. These inequalities can be easily transformed back to equalities by subtracting the *slack* functions t_i and s and rewriting the (2.21) as

$$g_i\left(x1, \frac{\partial y_p}{\partial y_n}\right) - t_i^2(x_1, \ldots, x_n) = 0 \tag{2.22}$$

again for $i = 1, \ldots, m$. The auxiliary functional previously expressed by (2.17), assumes now the form

$$L = \int_v \left[f + \sum_{i=1}^{m} \lambda_i (g_i - t_i^2) \right] dv \tag{2.23}$$

Now, computing the contribute of t_i in the variation of L we get

$$\frac{\partial L}{\partial t_i} = -2 \int_v \lambda_i t_i \delta t_i dv \tag{2.24}$$

and setting the coefficients of δt_i to zero we obtain that

$$t_i \lambda_i = 0 \tag{2.25}$$

for $i = 1, \ldots, m$ which implies that the Lagrangian multipliers λ_i are equal to zero when the slack variables t_i are not zero. In other words it means that the Lagrangian multipliers are zero at points in the design space where the corresponding constraint is not critical [67]. Moreover the (2.25) can be written in the form

$$g_i \lambda_i = 0 \tag{2.26}$$

again for $i = 1, \ldots, m$ because $t_i = 0 \Leftrightarrow g_i = 0$. The (2.26) is called *complementarity condition* equation. The advantage in using (2.26) is that the slack functions can be avoided in the auxiliary functional; we dispense with the variation of auxiliary functional respect to the Lagrange multiplier and instead add the inequality constraints to the optimality conditions.

2.4 Mathematical programming

The field of *mathematical programming* deals with the search of extremes of a function f defined over an n-dimensional space $\Re^n$ and bounded by a set S of constraints[12]. This kind of *mathematical programming* problems are defined by f and S and are common in the field of operations research, the mathematic branch which concern with decision making problems. According to the nature of design variables, objective and constraint functions are classified in several categories.

[12]that can be, in a general case, equalities or inequalities and can assume *linear* and/or *nonlinear* forms.

2.4.1 Linear programming (LP)

An optimization process is said to be linear when both the objective functions and the constraint relations are linear functions of the design variables $x_1, \ldots, x_n$. For example

$$f(x) = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \mathbf{c}^T\mathbf{x} \tag{2.27}$$

where f stands for a generic objective or constraint relation.

Since the necessary condition for an interior minimum is that the first derivative of the function with respect to the design variables to be equal to zero, this special feature can be adopted to develop algorithm for finding optimum solutions. In fact since in linear programming all the functions are linear, performing this differentiation would create functions with constant terms which may not necessarily be equal to zero. Applying this property to the objective function means that the optimum solution can not be located in the interior of the feasible design space and must lie on the boundaries[13] of the design space.

But since the boundaries are defined by the constraints relations which are also linear functions of the design variables, the optimum design must lie at the intersection of two or more constraints functions, unless the bounding constraint is parallel to the contours of the objective function.

A linear programming problem is said to be in a *standard* form if it is written as:

$$\begin{aligned} \text{minimize:} \quad & f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} && && (2.28)\\ \text{subject to:} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} && \text{Equality constraints} && (2.29)\\ & \mathbf{x} \geq \mathbf{0} && \text{Inequality constraints} && (2.30) \end{aligned}$$

where $\mathbf{A}$, $\mathbf{c}$ and $\mathbf{b}$ are, respectively a $m \times n$ matrix, a n-dimensional vector and a m-dimensional vector. It is important to say that *any* linear program including inequality constraints can be put into the standard form by using the so-called *slack* or *surplus* variables.

If the rank of $\mathbf{A}$ is m, we can select m independent columns and obtain a $m \times m$ matrix $\mathbf{D}$ that is not singular and admit the solution $\mathbf{x}_D = \mathbf{D}^{-1}\mathbf{b}_D$ where $\mathbf{b}_D$ is the corresponding right-hand vector.

Therefore the vector

$$\mathbf{x} = \begin{Bmatrix} \mathbf{x}_D \\ \ldots \\ 0 \end{Bmatrix} \tag{2.31}$$

such a solution is known as a *basic* solution for (2.29). A basic solution need not to satisfy the non-negativity constraints (2.30), but if that happen the solution is known as a *basic feasible* solution and can be shown to be an

[13] which are described by the constraints relations.

extreme point. The number t of basic solution can be estimated from the theory of permutations, by selecting m variables from a group of n and it is

$$t = \binom{n}{m} = \frac{n!}{m!(n-m)!} \tag{2.32}$$

but not all basic solutions are also feasible.

The most efficient and reliable method for solving linear programming problems (also involving a large number of design variables and constraints) is called the *simplex method* [6]. The basic idea of the simplex method is to continually decrease the value of the objective function by going from one basic feasible solution to another until the minimum value of the objective function is reached.

Unfortunately real structural problems rarely can be expressed in a so simplified form that can be solved using LP, since highly nonlinear objective function and constraints are common in this kind of problems. Nevertheless LP is of great interest since it is used in different contexts:

- as a procedure for solving *nonlinear programming problems* NLP by iterating inside smaller LP steps.[14];
- as a part of more complicated solving schemes.

2.4.2 Integer linear programming (ILP)

The solution vector $\mathbf{x}$ to linear programming and calculus[15] based problems is assumed to be all positive and continuous. Thus the optimum solution could have any value between the upper and lower bounds of the design variables. There are many design situations however, where some or all of the design variables are restricted to have discrete values. For example cross sectional areas, number of plies in a laminated composite, etc. This type of problem is called *integer linear programming* (ILP) [6] and its standard form is :

$$\text{minimize:} \quad f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} \tag{2.33}$$

$$\text{subject to:} \quad \mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.34}$$

$$x_i \in X_i = \{d_{i1}, d_{i2}, \ldots, d_{il}\} \qquad i \in I_d \tag{2.35}$$

where I_d and X_i are respectively the set of design variables[16] that can assume only discrete values and the corresponding set of allowable discrete values. This kind of problem is commonly known as *discrete programming* problem and it can be easily transformed into a problem where design variables can

[14] as in *sequential linear programmin* SLP.

[15] such as in differential and variational methods described in 2.3.1 and 2.3.2.

[16] for example the areas of cross sections.

assume only integer values by representing the index $j = 1, \ldots, l$ in the term d_{ij} that appears in (2.35) with the design variable x_i. This is the so called *integer linear programming* (ILP) problem.

Instead, when some variables are allowed to be continuous while the remaining can assume only integer values we have the form:

$$\begin{aligned} \text{minimize:} \quad & f(\mathbf{x}, \mathbf{y}) = \mathbf{c}_1^T \mathbf{x} + \mathbf{c}_2^T \mathbf{y} && (2.36)\\ \text{such that:} \quad & \mathbf{A}_1 \mathbf{x} + \mathbf{A}_2 \mathbf{y} = \mathbf{b} && (2.37)\\ & x_i \geq 0 \quad \text{integer} && (2.38)\\ & y_j \geq 0 && (2.39)\\ & && (2.40) \end{aligned}$$

that is known as *mixed integer linear programming* (MILP) problem.

Other problems are characterized by design variables used to model binary or $0-1$ type decision making solution[17]. This type of problem is termed *zero/one* binary ILP. In such a case, is possible to pose any ILP[18] as a *binary* ILP by replacing the design variable x_i of $2^K - 1$ with K binary variables $x_{i1}, \ldots, x_{iK}$ in the form

$$x_i = x_{i1} + 2x_{i2} + \ldots + 2^{K-1} x_{iK} \ . \tag{2.41}$$

It is also possible to convert the linear discrete programming problem to a binary ILP by using binary variables ($x_{ij} \in \{0, 1\}, j = 1, \ldots, l$) such that

$$x_i = d_{i1} x_{i1} + d_{i2} x_{i2} + \ldots + d_{il} x_{il} \tag{2.42}$$

and

$$x_{i1} + x_{i2} + \ldots + x_{il} = 1 \ . \tag{2.43}$$

It may appear to be logical to obtain an integer solution from a continuous problem by rounding-off the optimal values of the variables to the nearest integer value, assuming them to be continuous [62]. This approach scales badly, because for problems with n variables there are 2^n possible rounded-off designs: a very huge number for large values of n.

Moreover such a method can not guarantee that the integer solution is contained within the constraints: for some problems the optimum design may not even be one of these rounded-off solutions, and for others none of these rounded-off designs may be feasible. What may actually happen can best be shown graphically (see Figure 2.4.2).

Consider the point where two constraints meet at a vertex between two rows of lattice points of a plane. Suppose that the optimum gives the vertex

[17] For example in a truss design problem, the presence of a particular member or its absence of it can be represented by a binary value.

[18] that admits an upper bound.

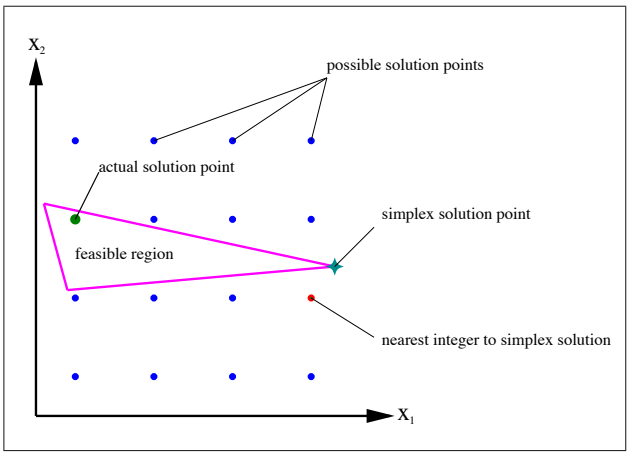


Figure 2.1: Nearest integer approximation and optimal solution

marked with the symbol ✦ as the solution. The nearest-integer approximation to this vertex lies outside the feasible region and hence is not a feasible solution. The integer solution to the problem may actually be far removed from the exact solution (as can be seen in Figure 2.4.2).

Brunch and bound

A common algorithm suitable for MILP and *nonlinear mixed integer linear programming* is the *brunch and bound* (BB) algorithm developed by A. H. Land and A. G. Doig [39] which relies on calculating the upper and lower bounds on the objective function so that points that result in designs with objective functions outside the bounds can be found and, therefore, the number of analysis can be reduced. BB is, in general, an algorithm suited for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded *in toto*, by using upper and lower estimated bounds of the quantity being optimized

If S denotes the set of the admissible solutions[19], a splitting procedure is firstly applied, with the result of returning two ore more subsets $S_1, S_2, \ldots, S_s$ so that $\bigcup_{i=1}^{s} S_i = S$. These subsets have the useful property that

$$\min f(x)|_{x \in S} = \min (v_1, v_2, \ldots, v_s) \tag{2.44}$$

where $v_i = \min f(x)|_{x \in S_i}$. This initial phase is called *branching*, since its recursive application defines a tree structure[20] with s nodes, each of them

[19] S is also known as the *search space* or *feasible region*
[20] also known as *the search tree*

representing the subsets S_i. The next phase is called *bounding* and consists in a procedure that computes upper and lower bounds for the minimum value of $f(x)$ within a given subset S_i.

The basic idea of the BB algorithm is: if the lower bound for some tree node[21] A is greater than the upper bound for some other node B, then A may be safely discarded from the search. This step is called *pruning*, and is usually implemented by maintaining a global variable m (shared among all nodes of the tree) that records the minimum upper bound seen among all subregions examined so far. Any node whose lower bound is greater than m can be discarded. The recursion stops when the current candidate set S is reduced to a single element; or also when the upper bound for set S matches the lower bound. Either way, any element of S will be a minimum of the function within S.

The method is applied, in a MILP context, as follows:

1. Solve the LP problem obtained from the starting MILP problem by assuming the varibles to be continuous valued.

2. If all the x variables variables of the solution in step 1 have integer values, the problem is solved and the execution stops here.

3. If several variables assume non integer values the corresponding value of the objective function is f_1. It can be shown that f_1 is a lower bound $f_L = f_1$[22]. This initial problem is called LP_1 and it is posed at the root of an enumeration tree.

4. Given l as the number of variables that don't satisfy the integer requirement, the algorithm splits (or *branches*) the node in two new LP problems by adding a new constrain to LP_1. This new constrain involves only one, let's say x_k, of the l non integers variables and makes possible the creation of two further problems: LP_2 and LP_3. As an example LP_2 will require that the value of the *branched* variable x_k to be less than or equal to the largest integer smaller than x_k, and LP_3 will have a constraint that x_k is larger than the smallest integer larger than x_k.

5. At this point several solution are possible:

   (a) no feasible solution: the corresponding node will be discarded.

   (b) an all integer feasible solution is reached: the corresponding node will be discarded but the value of the objective function will become an *upper* bound f_U for the MILP problem. In other words,

[21] each of them representing a set of candidates.

[22] This is due to the fact that imposing the continuous variables to take integer values will cause the objective function f to increase.

any node that has a LP solution with a larger value of the objective function will be neglected and only those solutions that produce an objective function $f^\star$ such that $f_L \leq f^\star \leq f_U$ will be pursued. If there are no solution with an objective function smaller than f_U then the node is an *optimum solution*, otherwise if there are other solutions with objective function smaller than f_U, they may still include valued variables and are classified as *live nodes*.

6. *Live nodes* are branched again by considering one of the remaining non integer variables (step 4) and resulting solutions are discussed (step 5) iterating until all the nodes are fathomed.

It has been proven that performance of the *Branch and bound* algorithm relies heavily on the choice of the non integer value to be used for branching and the selection of the node to be branched (see [67]).

2.5 Unconstrained optimization

Even if most structural optimization problems deal with constraints that bound the design space also unconstrained optimization can be useful in several situation: in a stage of the design when the constraints are theorically present but not active yet, as a tool for solving linear and nonlinear systems and because in some cases also constrained optimization problems can be casted in an equivalent unconstrained problem optimization.

Several techniques have been developed over the year (see [67]):

- in the field of minimization of function of one variable: *univariate search, bracketing, quadratic interpolation, Fibonacci, golden section search, bisection, Davidson's cubic interpolation,Newton, safeguarded polynomial interpolation.*
- in the field of minimization of function of one variable:
  - zeroth order methods: *univariate search, bracketing, quadratic interpolation, Fibonacci, golden section search.*
  - first order methods: *bisection, Davidson's cubic interpolation.*
  - second order methods: *Newton, safeguarded polynomial interpolation.*
- in the field of minimization of function of several variables:
- zeroth order methods: *sequential simplex, Powell's conjugate direction and modifications.*

- first order methods: *steepest descent, Fletcher-Reeves's conjugate gradient, Beale's restarted conjugate gradient.*
- second order methods: *Newton, Quasi newton (or variable metric).*

2.6 Heuristic algorithms

A bad disadvantage of the methods seen so far is, unfortunately, their inability to distinguish between local and global minima (or maxima). In many structural optimization problem there are more than one local minimum and, depending from the choice of the starting point, these algorithms may converge to one of these local extremes. An improvement to the solution could be made by restarting the optimization from different starting points but even this approach, besides not being practical for real problems involving a large number of variables, cannot guarantees the possibility of finding the global minimum.

If the variables are required to be integer dealing with the problem of local minima could be even worse. This happen mainly for two reasons. First of all, in this kind of problems the design space is disjointed and discontinuous, so derivative information is either useless or even not defined. Secondly, the discrete values of the variables introduce local minima due to various combinations of such variables even if the objective function has a single minimum (for continuous variables). A possible approach is the use of either random search techniques working inside the design space or enumerative algorithms.

An interesting feature of this class of methods is that the use a bottom up approach to model complex system: the derived algorithms stem from very simple rules, which when combined are able to provide accurate simulations of complex processes. This trend began in the last few decades, in the field of artificial intelligence, Langton [33 - 36] and in genetic algorithms, Holland [37, 38] and Goldberg [39].

This is in contrast with the common scientific approach: usually scientists tries to simplify what happens in nature. They look for smoothly changing functions that are able to explain the processes that occur in nature, in a way that can be calculated. This is the top down approach. In general equations that explain an event, an action or a structure, may be very complex to use. As a consequence they need to be simplified in some fashion, for example by linearizing or squaring them. In other terms, to be analyzed, a complex system is reduced to a simpler one [56].

2.6.1 Simulated annealing

The origin of this algorithm takes its roots back to studies in the field of statistical mechanics which deals with equilibrium of atoms at different temper-

atures. Studies demonstrates that the rate of cooling has great importance during the process of solidification of materials or formation of crystals. A too rapid cooling leads to a low level of stability because the atom tend to assume relative positions in the lattice structure to reach an energy state which is minimal, but only in local sense. More stable positions[23] for the atoms can be reached by *annealing* which consists in reheating to high temperature, and after cooling the material slowly, in order to let the atoms find positions which minimize a steady state potential energy. Before the steady state is reached it can be observed that the system can jump to higher level of energy for limited time. This feature is used in the algorithms based on *simulated annealing* such as *Metropolis algorithm.*

Metropolis algorithm

At a given temperature T the position of a randomly chosen atom is perturbed and the resulting change in the energy of the system ΔE is computed.

If $\Delta E < 0$, meaning that the new energy state is lower than the initial state, the new configuration of the atoms in the system is accepted. Otherwise, if $\Delta E \geq 0$ the new configuration can be accepted or rejected depending on a random probabilistic decision.

The probability of acceptance $P(\Delta E)$ of a higher energy state is computed as

$$P(\Delta E) = e^{\left(\frac{-\Delta E}{k_B T}\right)} \tag{2.45}$$

where k_B is the Boltzmann's constant. The more T is high, the more $P(\Delta E)$ is close to one. Otherwise when T is near to zero $P(\Delta E)$ becomes really small.

The acceptance-rejection criterium is based on a random chosen number $x \in (0, 1)$ that is compared with $P(\Delta E)$. If $x < P(\Delta E)$ then the perturbed state is accepted, otherwise if $x \geq P(\Delta E)$ the state is rejected.

The algorithm can be described in the following steps:

1. At a given temperature T a number of system configurations are generated by randomly perturbing atomic positions until a steady state is reached[24].

2. The temperature is reduced and the iteration of the previous step are started again.

3. The steps 1 and 2 are repeated iteratively while reducing the temperature to achieve the minimal energy state.

[23] in the sense of *global* minimum energy.

[24] or, in other words a *thermal equilibrium* state is found.

The algorithm has been also extended to optimization of functions with many variables by replacing the energy state with an objective function f, and by using variables $\mathbf{x}$ for the description of the position of the particles. After these substitutions the moves in the design space from one point $\mathbf{x}^i$ to another $\mathbf{x}^j$ cause a change in the objective function Δf^{ij}.

The initial temperature T_0 has great importance in the performance and convergence of the algorithm. If a too low value of T_0 is chosen the probability of finding a global minimum are low, so the initial value must be high enough to permit virtually all moves in the design space to be acceptable so that almost a random search is performed. In addition the number of moves must be decided before that T is reduced.

Some authors provide criteria to estimate good values for T_0, others suggest different temperature updating schemes and ideas for evaluating the number of moves that are needed[25], see [67].

2.6.2 Genetic algorithms

These algorithms (which will be described in details later, see Chapter 3) rely on Darwin's theory of survival of the fittest. It can be observed from biology that when a population of biological creatures is allowed to evolve over generations, the individual characteristics that are most useful for survival tend to be passed on to the future generations, because individuals carrying them have more chances to breed. Those features are encoded in chromosomes which are subjected to operations of genetic mechanics such as *reproduction*, *crossover*, *mutation*, *inversion* that result in randomly exchange of chromosomal information.

It will be shown that in this kind of approach it is necessary to represent the possible combinations of the variables in terms of bit strings and an objective function is used to evaluate the fitness of each solution. For a simple problem that can be expressed in the form

$$\text{minimize } f(\mathbf{x}), \qquad \mathbf{x} = \{x_1, x_2, x_3, x_4\} \tag{2.46}$$

a possible binary string representation could be

$$\underbrace{0\,1\,1\,0}_{x_1}\ \underbrace{1\,0\,1}_{x_2}\ \underbrace{1\,1}_{x_3}\ \underbrace{1\,0\,1\,1}_{x_4} \tag{2.47}$$

where base 10 values of the variables are $x_1 = 6$, $x_2 = 5$, $x_3 = 3$, $x_4 = 11$, and their ranges are $\{15 \geq x_1, x_4 \geq 0\}$, $\{7 \geq x_2 \geq 0\}$ and $\{3 \geq x_3 \geq 0\}$.

This kind of representation is easier when variable are discrete, while when they are continuous a large number of bits are usually required. In general, a continuous variable $x_i \in (x_i^U, x_i^L)$ to be approximated (with an

[25]The number of moves must be large enough to allow the solution to escape from a local minimum. For discrete problems it depends from the number of variables.

accuracy between two adjacent values x_{incr}) requires m binary digits according to the expression

$$2^m \geq \frac{x_i^U - x_i^L}{x_{incr}} + 1 \tag{2.48}$$

where m can be taken as the lowest integer that satisfies this expression.

Working with a population of solution, this kind of algorithms suffer a lower risk of getting stuck at a local minimum and give not a single design but a set of optimal designs (with eventually differen level of fitness) that could be very useful to engineers.

The algorithm can be roughly summed up in the following steps:

1. The size of population is chosen and the values of variables are assigned with randomly chosen values for the bits.

2. The individuals with best values of objective functions are chosen for *reproduction*.

3. The new generation is created by exchanging bit information (*crossover*) at randomly chosen locations between the individuals found at the previous step. Crossover can be single or multipoint. A crossover between 2 chromosome of length L can be at a location (or more locations) k included between 1 and $L-1$. As an example given two parent with $L = 9$

   parent 1 0 1 1 0 1‖ 0 1 1 1 (2.49)

   parent 2 0 1 0 0 1‖ 0 0 0 1 (2.50)

   and $k = 5$ we have

   offspring 1 0 1 1 0 1‖ 0 0 0 1 (2.51)

   offspring 2 0 1 0 0 1‖ 0 1 1 1 (2.52)

4. Occasionally, random alterations of strings are introduced (*mutation*).

5. The objective function is computed for all the individuals of the new generation and iterations from step 2 are performed until there are no more improvement between generations.

2.7 Constrained optimization

Most problems in the field of structural optimization can be formulated as *constrained optimization problems*. Typically the objective function is a

fairly simple function[26] but the design is usually subjected also to a certain number of constraints (about stresses, displacements, buckling loads, frequencies, etc.). Moreover the relations between these kind of constraints and the design variables are complex, and it is necessary to adopt a finite element analysis in order to evaluate them.

The methods described in this section can be usefully employed when, as said before, the objective function is not too hard to compute and the constraints can be evaluated with a moderate computational effort because they require the exact[27] evaluation of both the objective functions and the constraints whenever it is required by the optimization algorithm.

Even when the problem become large and complex the same approach can be followed, by substituting the objective function and the constraints with approximations and finally applying the optimization procedure on the *approximated* problem.

The general form of a constrained optimization problem with both equality and inequality constraints can be expressed in the form below:

$$\text{minimize:} \quad f(\mathbf{x}) \tag{2.53}$$

$$\text{subject to:} \quad h_i(\mathbf{x}) = b_i \quad (i = 1, 2, \ldots, M) \quad \text{Equality constraints} \tag{2.54}$$

$$g_j(\mathbf{x}) \geq 0 \quad (j = 1, 2, \ldots, N) \quad \text{Inequality constraints} \tag{2.55}$$

where:

$f(\mathbf{x})$ is the objective function;

$h_i(\mathbf{x})$ and $g_j(\mathbf{x})$ are the constraint functions[28];

$b_i(\mathbf{x})$ are constant values;

$\mathbf{x}_j$ are the design variables[29].

The constraints have the property of dividing the design space in two domains: the *feasible* space where all constraints are satisfied, and the *infeasible* domain where at least one of the constraints is violated. In real problems the minimum of $f(\mathbf{x})$ is commonly found on the boundary between the feasible and the infeasible domain or, in mathematical terms, where $g_j(\mathbf{x}) = 0$ for at least one j; otherwise the inequality constraints can be removed without altering the solution.

To reduce numerical issues and ill-conditioning all the design variables should have similar magnitude and, at the same time, the constraints should have similar values when they have to represent similar levels of criticality.

[26] Such as the computation of the total weight of the structure (expressed in function of the design variables).

[27] in the sense of a *finite element context*.

[28] They could be linear or nonlinear.

[29] It could be expressed in integer form.

Some techniques can not manage equality constraints. In such a situation it is possible to replace equality constraints of the form $h_i(\mathbf{x}) = 0$ with two inequality constraints $h_i(\mathbf{x}) \leq 0$ and $h_i(\mathbf{x}) \geq 0$. However, it is not a good idea to increase the number of constraints. When that number becomes large it is possible to replace an entire family of inequality constraints by an equivalent constraint by using the *Kreisselmeier-Stainhuaser* [KS] function defined as

$$KS[g_i(\mathbf{x})] = -\frac{1}{\rho} ln \left[\sum_i e^{-\rho} g_i(\mathbf{x}) \right] \tag{2.56}$$

where the term ρ is a parameter which measures the closeness of the KS-function to the smallest inequality min $[g_i(\mathbf{x})]$. For any positive value of ρ, the KS-function is always more negative than the most negative constraints and so the KS-function is a lower bound envelope to the inequalities[30]. In general

$$g_{min} \leq KS[g_i(\mathbf{x})] \leq g_{min} - \frac{\ln(m)}{\rho} \tag{2.57}$$

As shown before, an equality constraint $h_i(\mathbf{x}) = 0$ can be represented by a pair of inequality constraints $h_i(\mathbf{x}) \leq 0$ and $h_i(\mathbf{x}) \geq 0$ (or $-h_i(\mathbf{x}) \leq 0$) and in such a case the solution is at a point where both constraints are active simultaneously ($h_i(\mathbf{x}) = -h_i(\mathbf{x}) = 0$) and

- the value of the KS-function tends to zero as the value of ρ tends to infinity, since from (2.57) we have $0 \geq KS(h, -h) \geq -\frac{\ln(2)}{\rho}$;
- the gradient of the KS-function at the solution point $h_i(\mathbf{x}) = 0$ tends to zero independently of the value assumed by the parameter ρ.

In general an optimization problem

$$\begin{aligned} &\textbf{minimize} \quad f(\mathbf{x}) && (2.58)\\ &\textbf{such that} \quad h_k(\mathbf{x}) = 0, \quad k = 1, \ldots, n_e && (2.59) \end{aligned}$$

can be rewritten in the form

$$\begin{aligned} &\textbf{minimize} \quad f(\mathbf{x}) && (2.60)\\ &\textbf{such that} \quad KS(h_1, -h_1, h_2, -h_2, \ldots, h_{n_e}, -h_{n_e}) \geq -\epsilon && (2.61) \end{aligned}$$

where ϵ is a small tolerance.

[30] The larger is ρ the closer the KS-functions are to the minimum functions.

2.7.1 The *Kuhn-Tucker* conditions

For the problem (2.53), in the special case of equality constraints only, the necessary conditions for a minimum can be found with the Lagrange multiplier method, defining the *Lagrangian* function $\mathcal{L}$ as follows

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{j=1}^{n_e} \lambda_j h_j(\mathbf{x}) \tag{2.62}$$

where λ_i are the lagrangian multipliers. The necessary conditions for a stationary point at a *regular point*[31] are

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial f}{\partial x_i} - \sum_{j=1}^{n_e} \lambda_j \frac{\partial h_j}{\partial x_i} = 0 \qquad i = 1, \ldots, n \tag{2.63}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = h_j(\mathbf{x}) = 0 \qquad j = 1, \ldots, n_e \tag{2.64}$$

that represent $n + n_e$ equations for the n_e Lagrange multipliers and the n coordinates for a staionary point.

When the problem (2.53) includes also equality constraints they can be transformed into inequality constraints by adding slack variables and rewriting them in the form

$$g_j(\mathbf{x}) - t_j^2 = 0 \qquad j = 1, \ldots, n_g \tag{2.65}$$

where the slack variable t_j measures how far the j-th constraint is from being critical. The new *Lagrangian* function $\mathbf{L}$ can then be rewritten in the form

$$\mathcal{L}(\mathbf{x}, \mathbf{t}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{j=1}^{n_e} \lambda_j (g_j - t_j^2) \tag{2.66}$$

and differentiating with respect to $\mathbf{x}$,$\mathbf{t}$,$\boldsymbol{\lambda}$, follows that the necessary conditions for a stationary regular point are

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial f}{\partial x_i} - \sum_{j=1}^{n_g} \lambda_j \frac{\partial g_j}{\partial x_i} = 0 \qquad i = 1, \ldots, n \tag{2.67}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda_j} = -g_j + t_j^2 = 0 \qquad j = 1, \ldots, n_g \tag{2.68}$$

$$\frac{\partial \mathcal{L}}{\partial t_j} = 2\lambda_j t_j = 0 \qquad j = 1, \ldots, n_g \tag{2.69}$$

Those, slightly modified to yield the necessary conditions for a stationary point to be *minimum*, are known as *Kuhn-Tucker* conditions and can be summarized as follows:

[31] i.e. at a point where the gradients of the constraints are linearly *independent*. Otherwise one or more constraints can be removed without affecting the final solution.

Kuhn-Tucker condition 1 *Into an inequality constraint problem, a point* $\boldsymbol{x}$ *is a local minimum if and only if a set of* λ_j*'s may be found that for each* j *satisfies the following requirements:*

- $\lambda_j \geq 0$*;*
- *the equation* (2.67) *is satisfied;*
- *if a constraint is not active the corresponding* $\lambda_j = 0$*.*

In general it can be shown (see [67]) that the Kuhn-Tucker conditions are necessary but not sufficient for optimality. They become sufficient when the number of active constraints equals the number of design variables; if it doesn't happen it is necessary to evaluate second derivatives of the objective function and constraints. In particular, a sufficient condition is that the Hessian matrix of the Lagrangian function is positive definite in the subspace tangent to the active constraints.

The Kuhn-Tucker conditions are sufficient also in the case of convex problems. An optimization problem is convex when both the objective function and the feasible space are convex. An interesting property of these kind of problems, is that the minimum is unique and the Kuhn-Tucker conditions can be successfully adopted to find it.

convexity condition 1 *A set of points* S *is said to be* convex *whenever the line segment connecting two points that are in* S *is in* S *too.*

$$(x_1, x_2) \in S \Rightarrow \alpha x_1 + (1-\alpha)x_2 \in S \qquad 0 < \alpha < 1$$

convexity condition 2 *A function* f *is said to be* convex *if*

$$f[\alpha x_2 + (1-\alpha)x_1] \leq \alpha f(x_2) + (1-\alpha) f(x_1) \qquad 0 < \alpha < 1$$

For a function in n *variables it happens when the matrix of the second derivatives is positive semi-definite.*

It can be shown that the feasible space is convex if all the inequality constraints g_j are *concave*[32] and the equality constraints are linear.

2.7.2 Quadratic programming problems

When the objective function is quadratic and both the equality and inequality constraints are linear we have one of the simplest form of nonlinear constrained optimization problems, known as *quadratic programming* (QP)

[32] or all $-g_j$ are convex.

problem. If, for the sake of simplicity we consider only inequality constraints, it can be expressed in the form

$$\textbf{minimize} \quad f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \tag{2.70}$$

$$\textbf{such that} \quad \mathbf{A}\mathbf{x} \geq \mathbf{b}, \tag{2.71}$$

$$x_i \geq 0 \qquad i = 1, \ldots, n \tag{2.72}$$

Since the constraints are linear, if the matrix $\mathbf{Q}$ is semi-positive or positive defined then a global minimum solution for the problem exists. Moreover, if the quadratic form $\mathbf{x}^T\mathbf{Q}\mathbf{x}$ is positive (or semi-positive) defined then the problem can be solved with the Khun-Tucker conditions.

In such a case the Lagrangian function is

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}, \mathbf{t}, \mathbf{s}) = \mathbf{c}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} - \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \{t_j^2\} - \mathbf{b}) - \boldsymbol{\mu}^T(\mathbf{x} - \{s_i^2\}) \tag{2.73}$$

where $\boldsymbol{\lambda}$ and $\{t_j^2\}$ are, respectively, the vector of the Lagrangian multipliers and the vector of positive slack variables for the inequality constraints. $\boldsymbol{\mu}$ and $\{s_i^2\}$ are the same for the nonnegativity constraints. It is possible to obtain the necessary condition for a stationary point by differentiating the Lagrangian as follows

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{x}} &= \mathbf{c} - \mathbf{Q}\mathbf{x} - \mathbf{A}^T\boldsymbol{\lambda} - \boldsymbol{\mu} = 0 \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\lambda}} &= \mathbf{A}\mathbf{x} - \{t_j^2\} - \mathbf{b} = 0 \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}} &= \mathbf{x} - \{s_i^2\} = 0 \\
\frac{\partial \mathcal{L}}{\partial t_j} &= 2\lambda_j t_j = 0 \qquad j = 1, \ldots, n_g \\
\frac{\partial \mathcal{L}}{\partial s_i} &= 2\mu_i s_i = 0 \qquad i = 1, \ldots, n
\end{aligned}$$

where n_g is the number of inequality constraints, while n is the number of design variables. For additional details on achieving solution see [67]

2.7.3 On the practical computation of Lagrangian multipliers

The direct application of Kuhn-Tucker conditions to find a minimum is could be really complex in real problems. More often they are used to check wether a candidate minimum point satisfies the necessary conditions or to evaluate the sensitivity of the optimum solution to small changes in some parameters of the problem.

The equation (2.67), for reasons that will be clearer later, can be rewritten according to matrix notation in the form

$$\boldsymbol{\nabla} f - \mathbf{N}\boldsymbol{\lambda} = \mathbf{0} \tag{2.74}$$

Where, said n the number of the design variables and r the number of active constraints, $N_{ij} = \frac{\partial g_j}{\partial x_i}$, with $j = 1, \ldots, r$ and $i = 1, \ldots, n$.

Typically $r < n$, so that with n equations in terms of r unknowns the system is over-determined. Assuming the gradients linearly independent the rank of the system is r. For numerical reasons the system is not solved by a subset of r equations but is is used the following *least-squares* approach.

A residual vector $\mathbf{u}$ is introduced as $\mathbf{u} = \mathbf{N}\boldsymbol{\lambda} - \boldsymbol{\nabla} f$.

The Euclidean norm of the residual is

$$\|\mathbf{u}\|^2 = (\mathbf{N}\boldsymbol{\lambda} - \boldsymbol{\nabla} f)^T(\mathbf{N}\boldsymbol{\lambda} - \boldsymbol{\nabla} f) = \boldsymbol{\lambda}^T\mathbf{N}^T\mathbf{N}\boldsymbol{\lambda} - 2\boldsymbol{\lambda}^T\mathbf{N}^T\boldsymbol{\nabla} f + \boldsymbol{\nabla} f^T\boldsymbol{\nabla} f \tag{2.75}$$

A least-square solution of (2.74) will aim to the minimization of the square of the Euclidean norm of the residual respect to $\boldsymbol{\lambda}$; differentiating with respect to each Lagrangian multiplier we obtain

$$2\mathbf{N}^T\mathbf{N}\boldsymbol{\lambda} - 2\mathbf{N}^T\nabla f = 0 \tag{2.76}$$

or

$$\boldsymbol{\lambda} = (\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T\nabla f \tag{2.77}$$

This is the best solution from a least square point of view. Substituting the (2.77) into (2.74) we obtain

$$\mathbf{P}\nabla f = 0 \tag{2.78}$$

where $\mathbf{P}$ is defined as

$$\mathbf{P} = \mathbf{I} - \mathbf{N}(\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T \tag{2.79}$$

From a geometrical point of view, $\mathbf{P}$ projects a vector into the subspace tangent to the active constraints, and it is called projection matrix. Hence, (2.78) implies that the gradient of the objective function has to be orthogonal to that subspace.

The main problem with (2.77) is that it is an ill-conditioned and inefficient method. Better results can be achieved adopting a $\mathbf{QR}$-factorization for the matrix $\mathbf{M}$. It consists in finding an $r \times r$ upper triangular matrix $\mathbf{R}$ and an $n \times n$ orthogonal matrix $\mathbf{Q}$ such that

$$\mathbf{QN} = \begin{pmatrix} \mathbf{Q}_1 & \mathbf{N} \\ \mathbf{Q}_2 & \mathbf{N} \end{pmatrix} = \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \tag{2.80}$$

where $\mathbf{Q}$ has been further subdivided in $\mathbf{Q}_1$ (containing the first r rows of $\mathbf{Q}$) and $\mathbf{Q}_2$ (containing the last $n - r$ rows of $\mathbf{Q}$). Since $\mathbf{Q}$ is orthogonal

$\|\mathbf{u}\|^2 = \|\mathbf{Q}\mathbf{u}\|^2$ and

$$\|\mathbf{u}\|^2 = \|\mathbf{Q}\mathbf{u}\|^2 = \|\mathbf{Q}\mathbf{N}\boldsymbol{\lambda} - \mathbf{Q}\nabla f\|^2 = \left\| \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \boldsymbol{\lambda} - \mathbf{Q}\nabla f \right\|^2 = \left\| \begin{pmatrix} \mathbf{R}\boldsymbol{\lambda} - \mathbf{Q}_1\nabla f \\ -\mathbf{Q}_2\nabla f \end{pmatrix} \right\|^2 \tag{2.81}$$

From (2.81) it can be seen that $\|\mathbf{u}\|^2$ is minimized by choosing a value for $\boldsymbol{\lambda}$ such that

$$\mathbf{R}\boldsymbol{\lambda} = \mathbf{Q}_1\nabla f \tag{2.82}$$

2.7.4 Gradient Projection method

Introduced by Rosen, it is based on projecting the search direction into the subspace tangent to the active constraints. For the case of linear constraints the problem is defined as

$$\textbf{minimize} \quad f(\mathbf{x}) \tag{2.83}$$

$$\textbf{such that} \quad g_j(\mathbf{x}) = \mathbf{a}_j^T(\mathbf{x}) - b_j \geq 0, \quad j = 1, \ldots, n_g \tag{2.84}$$

The selection of the r active constraints only, can be done by restricting the values of j such that $j \in I_A$, where I_A is the set of the index of active constraints. In such a case we have

$$\mathbf{g}_a = \mathbf{N}^T\mathbf{x} - \mathbf{b} = \mathbf{0} \tag{2.85}$$

where $\mathbf{g}_a$ is the vector of active constraints and the columns of the matrix $\mathbf{N}$ are the corresponding gradients.

The basic idea is that $\mathbf{x}$ lies in the subspace tangent to active constraints. If

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha\mathbf{s} \tag{2.86}$$

and both $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ satisfies (2.85), then

$$\mathbf{N}^T\mathbf{s} = \mathbf{0} \tag{2.87}$$

Trying to find the direction along the most negative directional derivative (*steepest descent*) which satisfies (2.87), the problem can be rewritten in the form

$$\begin{aligned} \textbf{minimize} \quad & \mathbf{s}^T\nabla f \\ \textbf{such that} \quad & \mathbf{N}^T\mathbf{s} = \mathbf{0} \\ \textbf{and} \quad & \mathbf{s}^T\mathbf{s} = 1 \end{aligned} \tag{2.88}$$

The corresponding Lagrangian is

$$\mathbf{L}(\mathbf{s}, \boldsymbol{\lambda}, \mu) = \mathbf{s}^T\nabla f - \mathbf{s}^T\mathbf{N}\boldsymbol{\lambda} - \mu(\mathbf{s}^T\mathbf{s} - 1) \tag{2.89}$$

and the condition for $\mathbf{L}$ to be stationary can be obtained by differentiation

$$\frac{\partial \mathbf{L}}{\partial \mathbf{s}} = \nabla f - \mathbf{N}\boldsymbol{\lambda} - 2\mu\mathbf{s} = 0 \tag{2.90}$$

and premultiplying by $\mathbf{N}^T$ we obtain

$$\mathbf{N}^T\nabla f - \mathbf{N}^T\mathbf{N}\boldsymbol{\lambda} = 0 \tag{2.91}$$

and we can find $\boldsymbol{\lambda}$

$$\boldsymbol{\lambda} = (\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T\nabla f \tag{2.92}$$

and so, from (2.90)

$$\mathbf{s} = \frac{1}{2\mu}\left[\mathbf{I} - \mathbf{N}(\mathbf{N}^T\mathbf{N})^{-1}\mathbf{N}^T\right]\nabla f = \frac{1}{2\mu}\mathbf{P}\nabla f \tag{2.93}$$

where $\mathbf{P}$ is the projection matrix defined in 2.79. But, for numerical reasons, (2.79) is not an efficient way to compute $\mathbf{P}$. A better approach is based on a QR-factorization:

$$\mathbf{P} = \mathbf{Q}_2^T\mathbf{Q}_2 \tag{2.94}$$

where the matrix $\mathbf{Q}_2$ consists of the last $n - r$ rows of the $\mathbf{Q}$, as computed in the QR-factorization of the matrix $\mathbf{N}$.

2.7.5 Generalized reduced gradient methods

With this approach, the matrix $\mathbf{N}$ of the previous paragraph is partitioned as

$$\mathbf{N}^T = \begin{bmatrix}\mathbf{N}_1 & \mathbf{N}_2\end{bmatrix} \tag{2.95}$$

where $\mathbf{N}_1$ is the transpose of r linearly independent rows of $\mathbf{N}$.

Recalling (2.87) is possible to evaluate the components s_i of the direction vector. The r equations corresponding to the elements of $\mathbf{N}_1$ are used to eliminate the r components of $\mathbf{s}$ leading to a reduction of the order of the problem. Once computed $\mathbf{N}_1$, $\mathbf{Q}_2$ can be evaluated as follows

$$\mathbf{Q}_2^T = \begin{bmatrix}\text{-}\mathbf{N}_1^{-1}\mathbf{N}_2 \\ \mathbf{I}\end{bmatrix} \tag{2.96}$$

The optimization procedure can be summed up as follows:

1. The vector $\mathbf{s}$ is computed from (2.93). A search with a one dimensional optimization is done until $\mathbf{s} = \mathbf{0}$.

2. When $\mathbf{s} = \mathbf{0}$ from (2.78), it means that the Kuhn-Tucker conditions may be satisfied.

3. The Lagrangian multipliers are evaluated according to (2.77) or (2.82).

4. If all the Lagrangian multipliers are nonnegative the Kuhn-Tucker conditions are satisfied and the optimization can be terminated.

5. If some of the Lagrangian multipliers are negative, no progress can be done with the current set of active constraints. The constraints associated with the most negative multiplier is removed, and both $\mathbf{P}$ and $\mathbf{s}$ are recalculated. If $\mathbf{s} \neq \mathbf{0}$ a one dimensional search can be executed (go to step 1), otherwise if $\mathbf{s} = \mathbf{0}$ and there are still negative multiplier, another constraint is removed, until all Lagrange multipliers are nonnegative.

After a direction has been computated (1) a one dimensional search is carried to evaluate the parameter α which appear in (2.86) and if new constraints became active the set of the active constraints must be updated. For details see [67].

The same approach can be extended to non linear constraints by linearizing the constraints about x_i

$$\mathbf{N} = \begin{bmatrix} \nabla g_1(x_i) & \nabla g_2(x_i) \dots & \nabla g_r(x_i)\mathbf{I} \end{bmatrix} \tag{2.97}$$

and taking care of the following observations:

- Because of the nonlinearity of the constraints the one dimensional search moves[33] away the constraint boundary (see Figure 2.7.5). In order to move x back to the constraints boundaries a correction is necessary. As it happens in nonlinear finite element analysis with Riks-Wempner or Crisfield methods, a restoration procedure must be done. Using a linear approximation we have

$$g_j \approx g_j(x_i) + \nabla g_j^T(\overline{x}_i - x_i) \tag{2.98}$$

  and so the desired correction $\overline{x}_i - x_i$) in the tangent subspace which tends to reduce g_j to zero can be computed as

$$\overline{x}_i - x_i = -\mathbf{N}(\mathbf{N}^T\mathbf{N})^{-1}\mathbf{g}_a(x_i) \tag{2.99}$$

- $\mathbf{N}$ must be re-evaluated at each point.

For details see [5].

2.7.6 The feasible directions method

While in the gradient projection method we try to follow the constraint boundaries, in the feasible directions method we try to stay as far away as possible from them. Starting from a point $\mathbf{x}$ we look for a feasible direction.

[33] in tangent subspace

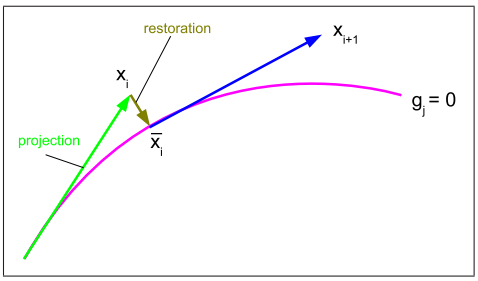


Figure 2.2: Projection and restoration moves

The vector **s**, which defines the direction, is said to be feasible if, applied to **x**, keeps **x** in the feasible domain. This concept can be expressed with the following the formula (if constraints are smooth)

$$\mathbf{s}^T \nabla g_j > 0 \qquad j \in I_A \tag{2.100}$$

In addition, if the application of **s** improve (in the sense that diminish) the objective function that direction is said to be *usable*. In mathematical terms:

$$\mathbf{s}^T \nabla f = \mathbf{s}^T \mathbf{g} < 0. \tag{2.101}$$

The direction which is - in some sense - "best" is the direction that, at the same time, reduce the objective function and keep away from the constraint boundaries as much as possible. That compromise can be obtained by the following problem definition:

$$\begin{aligned}
&\textbf{maximize} && \beta && && (2.102)\\
&\textbf{such that} && -\mathbf{s}^T \nabla g_j + \theta_j \beta \leq 0 && j \in I_A && (2.103)\\
& && \mathbf{s}^T \nabla f + \beta \leq 0 && \theta_j \geq 0 && (2.104)\\
& && |s_i| \leq 1 && && (2.105)
\end{aligned}$$

where the θ_j are positive numbers[34] which measure how far **x** will move from the constraint boundaries. The solution can be obtained via the simplex method. If $\beta_{max} > 0$ then we have found a feasible solution. Otherwise if $\beta_{max} = 0$ it can be shown that the Kuhn-Tucker equations are satisfied. For details see [25].

2.7.7 Projected Lagrangian methods (Sequential Quadratic Programming)

It's based on a theorem which states that the optimum is a minimum of the Lagrangian function in the subspace of vectors orthogonal to the gradients

[34] Also called *push-off* factors.

of the active constraints (the tangent subspace). In this subspace the Lagrangian is approximated in a quadratic form leading to a more complex (but also more efficient[35]) direction seeking algorithm which requires the solution of a quadratic programming problem where the objective function is quadratic, while the constraints are linear.

For the sake of simplicity, in this context will be discussed a simplified version of Powell's projected Lagrangian algorithm (see [14]) containing only inequality constraints.

$$\textbf{minimize} \qquad f(\mathbf{x}) \tag{2.106}$$

$$\textbf{such that} \qquad g_j(\mathbf{x}) \geq 0 \qquad j = 1, \ldots, n_g \tag{2.107}$$

Let's suppose that at the i-th iteration the design is at $\mathbf{x}_i$. We are looking for a search direction $\mathbf{s}$. This can be computed as solution of the following quadratic problem

$$\textbf{minimize} \qquad \phi(\mathbf{s}) = f(\mathbf{x}_i) + \mathbf{s}^T g(\mathbf{x}_i) + \frac{1}{2}\mathbf{s}^T \mathbf{A}(\mathbf{x}_i, \boldsymbol{\lambda}_i)\mathbf{s} \tag{2.108}$$

$$\textbf{such that} \qquad g_j(\mathbf{x}_i) + \mathbf{s}^T \nabla g_j(\mathbf{x}_i) \geq 0 \qquad j = 1, \ldots, n_g \tag{2.109}$$

where $\mathbf{g}$ is the gradient of f and $\mathbf{A}$ is positive definite approximation to the Hessian of the Lagrangian function. After the solution both $\mathbf{s}$ and $\boldsymbol{\lambda}_{i+1}$ are known. The update procedure is $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha\mathbf{s}$ where α is evaluated by minimizing the function

$$\psi(\alpha) = f(\mathbf{x}) + \sum_{j=1}^{n_g} \mu_j |\min(0, g_j(\mathbf{x}))| \tag{2.110}$$

where μ_j coincide with the absolute values of the Lagrange multipliers for the first iteration, i.e.

$$\mu_j = \max\left[\left|\lambda_j^{(i)}\right|, \frac{1}{2}\left(\mu_j^{(i-1)} + \left|\lambda_j^{(i-1)}\right|\right)\right] \tag{2.111}$$

where $^{(i)}$ means i-th iteration.

Going more into details, matrix $\mathbf{A}$ is at first initialized as identity matrix, and the updated using a BFGS (Broyden [22], Fletcher [57], Goldfarb [13], Shanno [20]) procedure

$$\mathbf{A}_{new} = \mathbf{A} - \frac{\mathbf{A}\Delta\mathbf{x}\Delta\mathbf{x}^T\mathbf{A}}{\Delta\mathbf{x}^T\mathbf{A}\Delta\mathbf{x}} + \frac{\Delta\mathbf{l}\Delta\mathbf{l}^T}{\Delta\mathbf{x}^T\Delta\mathbf{x}} \tag{2.112}$$

where $\Delta\mathbf{x} = \mathbf{x}_{i+1} - \mathbf{x}_i$ and $\Delta\mathbf{l} = \nabla_x L(\mathbf{x}_{i+1}, \boldsymbol{\nabla}_i) - \nabla_x L(\mathbf{x}_i, \boldsymbol{\lambda}_i)$. L is the Lagrangian function and ∇_x is its gradient with respect to $\mathbf{x}$.

[35] in the sense of a faster convergence of the algorithm

To be sure that $\mathbf{A}$ is positive definite, if $\Delta\mathbf{x}^T\delta\mathbf{l} \leq 0.2\Delta\mathbf{x}^T\mathbf{A}\Delta\mathbf{x}$ then $\Delta\mathbf{l}$ is modified in

$$\Delta\mathbf{l}_{new} = \theta\Delta\mathbf{l} + (1-\theta)\mathbf{A}\Delta\mathbf{x} \tag{2.113}$$

with

$$\theta = \frac{0.8\Delta\mathbf{x}^T\mathbf{A}\Delta\mathbf{x}}{\Delta\mathbf{x}^T\mathbf{A}\Delta\mathbf{x} - \Delta\mathbf{x}^T\Delta\mathbf{l}} \tag{2.114}$$

2.8 Approximated methods

2.8.1 Strategies for computational cost reduction

A structural analyst need a design software that includes the calculation of the structural response as well as the implementation of a constrained optimization algorithm [67]. This approach requires an high computational cost since the optimization procedure may require evaluating objective function and constraints a really large number of times. This is commonly achieved by repeated finite element analysis that are needed to evaluate the structural response (stresses, displacements, vibration periods, and so on...).

To address the problem of the computational cost, several techniques have been introduced. Among them, the *sequential approximate optimization* by Schmit and Farshi [64], employs a sequence of approximate analysis during portions of the optimization process. Typically the structure is analyzed only at a limited number of points in the design space. The response at those points is then used to construct a polynomial approximation. Finally the optimization procedure is applied to the structural problem described by the polynomial approximation.

In general the approximations can be classified in:

explicit that is generated by repeated use of finite element analysis;

implicit due to the dependence of the response on the structural design variables via finite element analysis;

global such as the polynomial approximation described so far (if n is the number of design variables then for a quadratic approximations $n(n+1)/2$ design points are needed);

local when the objective function and constraints are replaced by approximations (linear, in their simplest form) based on derivatives. These approximations are acceptable only in the neighborhood of the design space, and limits (known as *move limits*) to the magnitude in changes on the variables must be introduced.

An exact analysis is computed only when a design point is obtained from the approximated optimization. After that new derivatives are calculated in order to evaluate the new approximations for the objective function and

constraints. The process is repeated until convergence. Those are called *cycles* in order to be distinguished from the *iterations* computed inside the approximate optimizations.

The approximations used for constraints and objective unctions are commonly based on the value of the function to approximate and its derivatives at one or several points. They can be subdivided in:

local such as those based on the Taylor series. They are sufficiently accurate only in a very limited region of the design space. Linear forms are often inaccurate even in the neighborhood of the center point of the expansion. More precise values can be obtained adding one or more terms in the series, requiring the calculation of higher-order derivatives. Moreover, some authors introduce *ad hoc* variables[36] leading to the so-called *reciprocal* or *conservative* approximations;

global approach attempts to approximate the function in the entire design space. It is the case of the *response surface approach* where the function is sampled at a number of points and a polynomial expression is fitted to the data.

midrange they are a compromise between the previous two types of approximations.

The reduction of computational cost can also be addressed with *fast reanlysis* techniques. Those, assuming that the structural response is known at the design point $\mathbf{x}_0$, aim to evaluate the effect of a small perturbation $\Delta\mathbf{x}$ on the response.

Said $\mathbf{u}_0 = \mathbf{u}(\mathbf{x}_0)$ the displacement field at $\mathbf{x}_0$ and $\mathbf{u}_0 + \Delta\mathbf{u} = \mathbf{u}(\mathbf{x}_0)$ at the perturbed state $\mathbf{x}_0 + \Delta\mathbf{x}$, the equations of equilibrium at the design point $\mathbf{x}_0$ can be written, from a *Finite Element Method* perspective, as

$$\mathbf{K}_0\mathbf{u}_0 = \mathbf{f}_0 \tag{2.115}$$

where $\mathbf{K}_0$ and $\mathbf{f}_0$ are, respectively, the stiffness matrix and the load vector at $\mathbf{x}_0$.

Instead, the equations of equilibrium at the perturbed state $\mathbf{x}_0 + \Delta\mathbf{x}$ are

$$(\mathbf{K}_0 + \Delta\mathbf{K})(\mathbf{u}_0 + \Delta\mathbf{u}) = (\mathbf{f}_0 + \Delta\mathbf{f}) \tag{2.116}$$

Subtracting (2.115) from (2.116) follows

$$(\mathbf{K}_0 + \Delta\mathbf{K})\Delta\mathbf{u} = \Delta\mathbf{f} - \Delta\mathbf{K}\mathbf{u}_0 \tag{2.117}$$

Neglecting the term $\Delta\mathbf{K}\Delta\mathbf{u}$ in (2.117) a first approximation $\Delta\mathbf{u}_1$ is obtained to $\Delta\mathbf{u}$:

$$\mathbf{K}_0\Delta\mathbf{u}_1 = \Delta\mathbf{f} - \Delta\mathbf{K}\mathbf{u}_0 \tag{2.118}$$

[36] also known as *intervening* variables.

Higher order approximations can then be achieved subtracting (2.118) from (2.117) as follows

$$(\mathbf{K}_0 + \Delta\mathbf{K})(\Delta\mathbf{u} - \Delta\mathbf{u}_1) = -\Delta\mathbf{K}\Delta\mathbf{u}_1 \tag{2.119}$$

As before, neglecting the term $\Delta\mathbf{K}(\Delta\mathbf{u} - \Delta\mathbf{u}_1)$ in (2.119) an approximation $\Delta\mathbf{u}_2$ to $\Delta\mathbf{u} - \Delta\mathbf{u}_1$ can be computed simply by solving

$$\mathbf{K}_0\Delta\mathbf{u}_2 = -\Delta\mathbf{K}\mathbf{u}_1 \tag{2.120}$$

At the end $\Delta\mathbf{u} = \sum \Delta\mathbf{u}_i$ where $\Delta\mathbf{u}_i$ is obtained from

$$\mathbf{K}_0\Delta\mathbf{u}_i = -\Delta\mathbf{K}\Delta\mathbf{u}_{i-1} \tag{2.121}$$

A similar approach can be extended to Eigenvalues[37] problems too. This can be written in the form:

$$\mathbf{K}_0\mathbf{u}_0 - \mu_0\mathbf{M}_0\mathbf{u}_0 = \mathbf{0} \tag{2.122}$$

where now $\mathbf{M}_0$, $\mathbf{u}_0$ and μ_0 are, respectively, the mass matrix, the eigenvector and the eigenvalue. All of them evaluated at $\mathbf{x}_0$. If μ_0 is a non-repeated eigenvalue at the point $\mathbf{x} + \Delta\mathbf{x}$ we have

$$(\mathbf{K}_0 + \Delta\mathbf{K})(\mathbf{u}_0 + \Delta\mathbf{u}) - (\mu_0 + \Delta\mu)(\mathbf{M}_0 + \Delta\mathbf{M})(\mathbf{u}_0 + \Delta\mathbf{u}) = \mathbf{0}. \tag{2.123}$$

Subtracting (2.122) in (2.123) and neglecting, in the perturbation, quadratic and cubic terms we have

$$(\mathbf{K}_0 - \mu_0\mathbf{M}_0)\Delta\mathbf{u} + (\Delta\mathbf{K}_0 - \mu_0\Delta\mathbf{M})\Delta\mathbf{u}_0 - \Delta\mu\mathbf{M}_0\mathbf{u}_0 \approx \mathbf{0} \tag{2.124}$$

Since both $\mathbf{M}_0$ and $\mathbf{K}_0$ are symmetric, pre-multiplying (2.122) by $\mathbf{u}_0^T$ gives

$$\Delta\mu \approx \frac{\mathbf{u}_0^T(\Delta\mathbf{K} - \mu_0\Delta\mathbf{M})\mathbf{u}_0}{\mathbf{u}_0^T\mathbf{M}_0\mathbf{u}_0}. \tag{2.125}$$

Alternatively multiplying (2.123) by $\mathbf{u}_0 + \Delta\mathbf{u}^T$ and neglecting higher terms gives

$$\mu_0 + \Delta\mu \approx \frac{\mathbf{u}_0^T(\Delta\mathbf{K}_0 + \Delta\mathbf{K})\mathbf{u}_0}{\mathbf{u}_0^T(\Delta\mathbf{M}_0 + \Delta\mathbf{M})\mathbf{u}_0}. \tag{2.126}$$

Other techniques with the same purpose, are available (see [55]). Obviously there is no guarantee that this method converges.

[37] Such as optimizations involving vibrations or buckling.

2.8.2 Sequential linear programming (SLP)

There are problems where the computation of a single evaluation of the objective function, constraints and derivatives is very large, compared to the computational cost associated with the optimization operations. With these problems, it is beneficial to simplify them by redefining the original problem into an approximate one. One of the most popular methods is that of Sequential Linear Programming (SLP).

In order to see how this method works, it is necessary to look at a typical optimization problem of the form:

$$\textbf{minimize} \quad f(\mathbf{x}) \tag{2.127}$$

$$\textbf{subject to} \quad g_j(\mathbf{x}) \geq 0 \qquad j = 1, \ldots, n_g. \tag{2.128}$$

Starting from an initial trial solution $\mathbf{x}_0$, the objective function and constraints equations are approximated by linear equations using a Taylor series expansion about $\mathbf{x}_0$. The optimization problem can then be represented as follows:

$$\begin{aligned} \textbf{minimize} \quad & f(\mathbf{x}_0) + \sum_{i=1}^{n} (x_i - x_{0i}) \left(\frac{\partial f}{\partial x_i} \right)_{\mathbf{x}_0} \\ \textbf{subject to} \quad & g_j(\mathbf{x}_0) + \sum_{i=1}^{n} (x_i - x_{0i}) \left(\frac{\partial g}{\partial x_i} \right)_{\mathbf{x}_0} \geq 0 \quad j = 1, \ldots, n_g \\ \textbf{and} \quad & a_{li} \leq x_i - x_{0i} \leq a_{ui}. \end{aligned} \tag{2.129}$$

The last set of constraints represent the upper (a_{ui}) and lower (a_{li}) bounds of the allowed changes in x_i and are called *move limits*. If these limits are small, a good approximation can be guaranteed. In such a case the final solution to the linearized problem $\mathbf{x}_L$, can then be treated as an optimum[38], and if the original solution $\mathbf{x}_0$ is replaced with $\mathbf{x}_L$ the optimization process can then be started with this new solution. The process is then repeated, thus replacing the original optimization problem with a sequence of linear programming (LP) problems. Each linear optimization is called *cycle*. This approach has some drawbacks:

1. It is efficient only if the computational cost of the LP cycles is small (compared to the cost of the analysis);

2. if the move limits are not chosen in a proper way, the method may never converge (the limits should be shrinked as the solution become closer to the optimum);

[38] Even if, in general, $\mathbf{x}_L$ is far from the optimum solution for the involved approximations.

3. if the starting point is infeasible then the solution may be infeasible too. In such a case the constraints should be relaxed. This can be done, by replacing the problem described by the system (2.129) with

$$\begin{aligned} \textbf{minimize} \quad & f(\mathbf{x}_0) + \sum_{i=1}^{n} (x_i - x_{0i}) \left(\frac{\partial f}{\partial x_i} \right)_{\mathbf{x}_0} + k\beta \\ \textbf{subject to} \quad & g_j(\mathbf{x}_0) + \sum_{i=1}^{n} (x_i - x_{0i}) \left(\frac{\partial g}{\partial x_i} \right)_{\mathbf{x}_0} + \beta \geq 0 \quad j = 1, \ldots, n_g \\ \textbf{and} \quad & a_{li} \leq x_i - x_{0i} \leq a_{ui}, \quad \beta \geq 0. \end{aligned} \tag{2.130}$$

where β is an additional design variable representing the allowed margin of constraint violation while k is a number chosen to make the contribution of β in f large enough so that LP cycles will emphasize reducing β over reducing f (see [67]).

4. Finally, in some cases the solution could cycle between two points.

2.8.3 Sequential nonlinear programming (SNLP)

The SLP method can be generalized by using nonlinear approximations for some of the constraints and objective function. But, differently from SLP, with this more general approach, only the functions which are computationally expensive to calculate are approximated by using either linear[39] or nonlinear[40] approximations, while inexpensive constraints do not need to be approximated at all (see [67]). The solution process to these type of problem is similar to SLP. An initial trial solution $\mathbf{x}_0$ to the problem is required. The functions[41] which requires large computational resource for evaluation are then approximated about $\mathbf{x}_0$. As with SLP, appropriate move limits must be specified to safeguard against large changes in the design variables, resulting in poor approximations. Said $\mathbf{x}_1$ the solution of the approximate problem, a new exact structural analysis is performed[42]. The results are then used to construct the new approximation and to perform a new optimization, repeating the process until the minimum value of the objective function is reached. So the original problem (2.127) is replaced by the system

$$\begin{aligned} \textbf{minimize} \quad & f_{approx}(\mathbf{x}, \mathbf{x}_0^i) \\ \textbf{subject to} \quad & g_{approx,j}(\mathbf{x}, \mathbf{x}_0^i) \geq 0 \quad j = 1, \ldots, n_g \\ \textbf{and} \quad & \|\mathbf{x} - \mathbf{x}_0^i\|, \leq a_i \quad \text{for} i = 0, 1, 2, \ldots, \end{aligned} \tag{2.131}$$

[39] i.e. SLP
[40] i.e. quadratic or cubic
[41] Objective and/or constraints.
[42] At $\mathbf{x}_1$.

where f_{approx} and $g_{approx,j}$ are, respectively, the approximate objective function and constraints about $\mathbf{x}_0$. $\mathbf{x}_0^i$ is the solution of the i-th minimization and a_i is a properly chosen move limits.

2.9 Other methods

2.9.1 Homogenization Method

The homogenization method (see [56] [9], [7]), allows for the simultaneous optimization of a structure's topology, shape and size. It can do this, because the structure is described by a global density function that assigns material to all parts of the structure. The structure is represented using finite elements, each of which consists of a composite material with microvoids with a density variation covering all values in the interval $\{0, 1\}$. In its most general form, the homogenization optimization method is concerned with minimizing the compliance of a structure with a fixed given volume of material, where the density of the material is used as the design variable. The given volume is discretized using finite elements, and the density of the each of these elements is controlled with geometric variables that govern the material and its microstructure. This is done to correctly relate the material density with the effective material property. The mathematical formulation of the homogenization optimization problem is:

$$\begin{aligned} \textbf{minimize} \quad & l(\mathbf{u}), \quad \mathbf{u} \in U, E && (2.132) \\ \textbf{subject to} \quad & a_E(\mathbf{u}, \mathbf{v}) = l(\mathbf{v}) \quad \forall \mathbf{v} \in U \\ & E \in \mathsf{E}_{ad} && (2.133) \end{aligned}$$

with

$$a_e(\mathbf{u}, \mathbf{v}) = \int_\Omega E_{ijkl}(x)\varepsilon_{ij}(\mathbf{u})\varepsilon_{kl}(\mathbf{v})d\Omega \tag{2.134}$$

$$l(\mathbf{u}) = \int_\Omega f\mathbf{u}d\Omega + \int_\Gamma t\mathbf{u}d\Gamma \tag{2.135}$$

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.136}$$

where

$l(\mathbf{u})$ is the load linear form;

$a(\mathbf{u}, \mathbf{v})$ is the energy bilinear form and can be seen as the internal virtual work of an elastic body at the equilibrium $\mathbf{u}$ for an arbitrary virtual displacement $\mathbf{v}$;

$\mathbf{f}$ are the body loads;

t are the surface tractions;

ε_{ij} are the linearized strains;

U is the space of kinematically admissible displacement fields;

E_{ad} is the set of the admissible stiffness tensors [8];

E_{ijkl} is the rigidity tensor dependent on the design variables, which could be the density, layering and orientation of the layering of each element.

The solution to this problem can be found using SLP, NLP or any other of the techniques mentioned previously. For example, discretizing the problem (2.132) with a finite element mesh

$$\begin{aligned} \min_{\boldsymbol{u},E_e} \quad & \boldsymbol{f}^T\boldsymbol{u} \\ \text{con:} \quad & K(E_e)\boldsymbol{u} = \boldsymbol{f}, \\ \text{e} \quad & E_e \in E_{adm}. \end{aligned}$$

where $\boldsymbol{f}$ and $\boldsymbol{u}$ are, respectively the load and displacement vectors. The tangent stiffness matrix $\boldsymbol{K}$ depends on the stiffness E_e of in element e with $e = 1, \ldots, N$ and can be written in the form

$$\boldsymbol{K} = \sum_{e=1}^{N} \boldsymbol{K}_e(E_e)$$

where $\boldsymbol{K}_e$ is the element stiffness matrix.

It's important to note that there are two field of interest: the displacement **u** and the stiffness E.

2.9.2 Optimal layout theory

According to Omquerin [56], the basic assumptions for this theory come from the work of Mitchell on truss optimization [43]. These were further developed by Hemp [73] and, later, by Prager and Rozvany [54]. The *optimal layout theory* is based on four fundamental concepts (taken from [56]).

1. The idea of *structural universe*, consisting of the union of all potential members in all possible directions at all points of the available space.

2. The necessity of a *Continuum-type Optimality Criterion* (COC), a condition of cost (or weight) minimization which can be based on the Kuhn-Tucker conditions.

3. The concept of an *adjoint structure*, which is a fictitious structure used as a convenient mechanical analogy for interpreting certain quantities in optimality criteria.

4. The *layout criterion function* ϕ^e which is defined from the optimality criteria and which can be written in the form $\phi^e = 1$ for $A^e \neq 0$ and $\phi^e \leq 1$ for $A^e = 0$ where A^e is the cross-sectional area of the e-th element.

The expression for the layout criterion function ϕ usually contains the strain ε^e in e-th element of the real structure and the strain in e-*th* of the adjoint structure. However the optimal structure found by this method may be restricted to a discrete system consisting of a finite number of bars. The real and adjoint strain fields are defined on the entire available space in a continuum type fashion and both of these strain fields must be kinematically admissible. More in detail, the exact layout optimization method consists in the following operations:

1. Find the adjoint strain field such that:
   (a) The kinematic boundary (supports) conditions and the kinematic continuity conditions (compatibility) are fulfilled.
   (b) The layout criterion function ϕ^e must take a unit value in at least one direction at all points of the available space.

2. Adopt member cross sections along lines with $\phi = 1$ such that the resulting system can transmit the external loads in a stable fashion, and the resulting real and adjoint strains in the non-vanishing members agree with those in the strain fields adopted previously.

It can be noted that in order to simultaneously fulfill the conditions mentioned above (especially the last one), the method requires considerable intuitive insight and ingenuity from the operator who is carrying out the optimization (see [61]).

2.9.3 Shape optimization

In the shape optimization method, a structural domain is described by its contours, which are geometrically represented by splines. These splines are defined by the coordinates of the control nodes (the design variables). Shape optimization, therefore consists in varying the boundaries of the analysis models (see [44]).

Let's consider a structural domain (see Figure 2.9.3 on page 51). A section of its boundary may be represented by a shape function f_s, and an optimal shape for this region is desired based on a chosen criterion. This problem can be described by the function

$$J_s : f_s \longrightarrow x \in \Re \tag{2.137}$$

where the value of c can represent, for example, the Von Mises stress or the compliance of the structure. To minimize J_s, f_s is represented as a function of N unknown parameters $\overline{\mathbf{x}} = (x_1, x_2, \ldots, x_N)$.

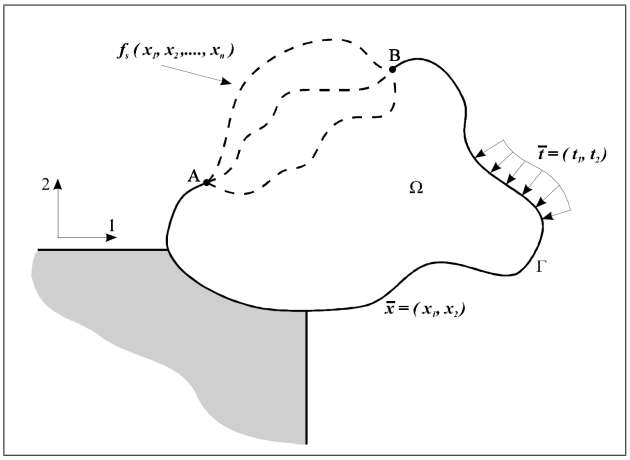


Figure 2.3: Body of structure: the section AB is to be optimized.

According to Krzesinski (see [23]) the formulation of the shape optimization problem can then be written as a nonlinear programming problem for a function $F : \overline{\mathbf{x}} \longrightarrow c$. The optimization problem then becomes:

$$\text{Minimize} \quad F(\overline{\mathbf{x}}) \tag{2.138}$$

$$\text{subjected to} \quad h_j = 0 \text{ for } j = 1, 2, \ldots, k \tag{2.139}$$

$$g_i \geq 0 \text{ for } i = 1, 2, \ldots, m \tag{2.140}$$

Some researches do a distinction between this type of optimization. In particular, *shape optimization* can be intended in a *narrow* or *broad* sense. From a *Finite Element Method* point of view it can be considered as shape optimization an optimization which involves the positions of the nodes of the finite element mesh, and their *connectivity*[43]. In contrast *sizing* optimization takes in account the properties of the elements which characterizes their stiffness, such as cross-sectional area for bars, or thickness for plates. In a narrow sense, shape optimization regards the optimum design of the $2D$ or $3D$ shape of structural elements. In broad sense, it includes also the *geometrical* optimization for skeleton structures (the search of the optimum locations of the joints of the structure)[44], and topological optimization which regards the connectivity of the structure (for example, which nodes

[43] i.e. the existance or the removal of finite elements between nodes.

[44] When combined with *sizing optimization*, a two-level optimization approach is recommended where geometry variables are treated in a different way than the sizing variables (see [67]).

are connected by elements). Unfortunately, in any case, several problems must be considered in shape optimization:

- As the shape of the domain changes the *mesh* must be changed. In fact simple re-meshing rules that are limited to translate boundary nodes can introduce distortions and, consequently, loss of accuracy. To solve this problem, manual re-meshing, or sophisticated mesh generators are then required (see [63], [76]).
- Some difficulties may arise for the existence or creation of internal boundaries (holes)

2.9.4 Computer aided shape optimization (CAO)

This method takes inspiration from nature by observing the growth of trees (see citemattheck, [56]). In trees, only the outermost growth ring adapts to external loading. This adaptation is done in such a way that a state of constant Von Mises stress is present at the surface of the tree. In so doing the structure generated will try to avoid localized stress peaks, and will produce a minimum weight design. This growth mechanism has been studied to develop an optimization methodology that follows these biological growth rules and is implemented with the aid of software for finite element analysis. This methods is based on the following steps (see [56]):

1. A finite element model is generated, based on a structure with an initial reasonable design shape.
2. The finite element mesh is then covered with a thin layer of finite elements which have a much smaller modulus of elasticity than the rest of the structural material. This layer is introduced to model the growth ring in trees.
3. The application of loads will cause some regions to be heavily stressed and others to be lowly stressed. The surface layer will be treated as a movable layer that can ba wither swelled or shrunken by using the equation
$$\dot{\varepsilon}_v = k\left(\sigma_{VonMises} - \sigma_{ref}\right) \tag{2.141}$$
where $\sigma_{VonMises}$ and σ_{ref} are, respectively, the Von Mises stress distribution calculated using FEA and a reference stress value. The swelling (or growth) of the surface layer will produce a distribution of swelling displacements on the surface according to (2.141). It is also possible to allow shrinking ($\dot{\varepsilon}_v < 0$) or to prohibit this as is the natural case in real tree growth.

   If this swelling option is unavailable in the FE software, the swelling procedure may be replaced by a thermal expansion of the soft surface

layer. The temperature is then set equal to the Von Mises stresses in the surface layer. Only this layer has a non-zero thermal expansion coefficient but the material below the surface layer has a vanishing coefficient of thermal expansion. This means that only the surface layer is allowed to increase its volume.

The incremental displacements defined by either of the above mentioned techniques are now multiplied by reasonably defined factors, in order to obtain visible shape corrections. The multiplied displacements are added to the nodal point coordinates of the surface nodes.

4. 4 The inner border of the soft layer will be shifted outwards in such a way that the thickness of the soft layer will now be kept constant all the time.

5. Starting with the new structure modified at step 4, steps 3-5 are then repeated.

6. 6 Finite element analysis is then carried out with the same modulus of elasticity everywhere to determine if there are still stress peaks in the structure.

7. Steps from 3 to 6 are repeated until all notch/peak stresses are completely reduced or until design limitations prohibit any further increase in the overall dimensions of the design.

In other terms, this method attaches material in regions which are highly loaded and removes material from under stressed regions.

2.9.5 Evolutionary structural optimization

Among the different methods have been developed over the last two decades to produce structures with minimum weight there is the *Evolutionary structural optimization (ESO)* developed in 1992 by Steven and Xie[74] which worked removing understressed (and so unnecessary) material from a structural domain in an evolutionary fashion, aiming to obtain a fully strained structure (as possible).

This approach has been entirely developed heuristically without a real mathematical foundation, but taking inspiration from rules that can be observed from nature, and it has been applied essentially to 2D plane-stress finite elements. Moving from this roots Querin [11], extended the method also to the application for structures modeled with many different typologies of finite elements: truss , 2D plane stress, 2D plane strain, axisymmetric, plates, shell and 3D brick elements. The original ESO formulation was characterized for strictly binary decision method $\{0, 1\}$ also known as *hard skill*. This boolean value simply states if a finite element must exist or not. This works well in most cases (when a topology must be found), but for certain problems is not satisfactory (for example when plates of variable thickness, or beams with different cross sectional areas are more desirable). For this reason, the original binary feature has been extended to include a finite set of N decision variables $\{x_i\}_{i=1,\dots,N}$ or an entirely variable solution, leading to the so-called *morphing ESO*, sometimes also known as *soft skill* [4], [69]. The range of applications was also extended to non linear problems and for structures with moving boundaries and support.

The same author has finally developed the *Bidirectional Evolutionary Structural Optimization (BESO)* that extended the original ESO idea, allowing not only removal but also introduction for material, based on a new parameter (*Performance Index*) that provides an alternative means of measuring the efficiecy of the structure. Performance index comes to be useful when it's necessary to chose between two o more topologies that, at the same time, can satisfy the same domain and same objectives (like minimum compliance or weight). This parameter is a numerical value that can be used to compare the different solutions (under the same conditions of loading and constraints) in order to make a proper choice.

With ESO the analysis become an integral part of the design process (in the sense of creation) and not an activity that is done after the design has been prescribed.

The general case should take into account:

- Size, shape and topology optimization at different parts of structure.
- Different optimization criteria at different parts of structure.
- Multiple load environment;

- Multiple support environment;
- Multiple material environment;
- General 2D and 3D shapes;
- Optimization with material and geometric non-linearities.

Eso uses *Finite Element Analysis (FEA)* as computational engine. FEA is done over and over until a final solution has been reached. At the end of each FEA, the computer evaluate the results of analysis and applies some ESO rules in order to remove inefficient material. The procedure is repeated over and over, until all redundant material is removed.

The simple concept of *Evolutionary structural optimization* is that slowly removing inefficient material from a structure, the shape of the structure evolves towards an optimum. There are different material rejection criteria, depending of the kind of constraints has been adopted (stiffness, frequency, buckling load,...).

stress level rejection criterium

The stress (or strain) is an important indicator in structural analysis. Low level of stress (or strain) denotes the presence of inefficient material. Excessively high levels are indicators of possible imminent structural failure. In ideal condition, the optimum could be represented by a *fully stressed design* where all the material is at the same stress level.

Stress level rejection criterium is based on the idea that lowly stressed material is under-utilized and then can be removed. In the meanwhile, during the ESO process, the more material is removed, the more uniform become the stress level on the structure.

The application of ESO can be summed up in the following steps:

1. Definition of a removal criterion
2. Creation of a *fine*[45] mesh of finite elements.
3. Application of loads and boundary conditions.
4. Evaluation of stress analysis;
5. Removal of under-utilized material[46];

Steps 2,3,4 are a typical of *finite element analysis*. The procedure iterates from 2 to 5 until convergence.

[45] With many small elements in order to make possible step 5.
[46] It is sufficient to remove the element from the mesh

In this case the *stress level* can be measured by some sort of average of all stress components[47]. For plane stress and isotropic material the von Mises stress σ^{vm} can be computed as follows:

$$\sigma^{vm} = \sqrt{\sigma_x^2 + \sigma_y^2 - \sigma_x \sigma_y + 3\tau_{xy}^2} \tag{2.142}$$

where, as usual, σ_x and σ_y are normal stresses in x and y directions, while τ_{xy} is the shear stress.

At the end of every cycle of finite element analysis for each element e the *stress level* is obtained as ratio between the computed von Mises stress of the element $\sigma_e^v m$ and the maximum von Mises stress $\max(\sigma_e^v m)$ of the whole structure. If this value is less than the current *rejection ratio* RR_i than the element e is removed from the mesh.

$$\frac{\sigma_e^{vm}}{\max(\sigma_e^{vm})} \leq RR_i \tag{2.143}$$

The combined cycle of finite element analysis and element removal is repeated using the same value of RR_i until a *steady state*[48] is reached. At this stage the *rejection ratio* is updated according to the following expression:

$$RR_{i+1} = RR_i + ER \tag{2.144}$$

where ER is the *evolutionary rate* and the iterations can proceed until another steady state is reached.

The ESO process continue when a condition is satisfied, meaning the reaching of an optimum. It could be when there is no material in the final structure with a stress level less than a certain ratio (for example 25%) of the maximum. Ideally the structure at the final stage could become a *fully stressed design* if the material ad each point is at maximum stress level, but this is possible only under particular conditions.

This procedure requires to parameters: the *initial rejection ratio* RR_0 and the *evolutionary rate* ER. Typical values are $RR_0 = 1\%$ and $ER = 1\%$, but often lower values are need to be used. The correct values can be found with a try and test procedure: if too much material is removed in a single iteration or steady state then smaller values of RR_0 and ER are required.

Multiple load cases and support environments

During their life, real structures are subjected to multiple load cases and they must be designed taking into account that. Moreover some kind of structure can be supported in different ways at different time[49].

[47] For isotropic material is common to use Von Mises stress.

[48] It happens when no more element are deleted at the current iteration.

[49] This is - for example - the case of airplanes that are supported by air pressure acting on the wings during the flight and by undercarriage in discrete points when on ground.

The method seen in 2.9.5 can be easily extended to multiple load conditions maintaining the same rejection criterion and the same objective of achieving a more uniform stress distribution as possible in the new designs. This can be done in the same way seen before but repeating the analysis for the different load conditions. The only difference is that for each element e the rejection criterion expressed in 2.143 must be checked against any load case[50]. Finally the element e can be removed only if the condition of rejection is satisfied for every load condition; it means that each remaining element plays a structurally significant role in almost one load condition. Moving loads can be simulated creating a set of load conditions that follows the path of the moving load.

With the presence of multiple support environments, things are almost the same, since the stress analysis must be repeated for each support condition but this is more computationally onerous than the case of the multiple loads case since each analysis requires the creation of a different stiffness matrix, and so for n support environments is necessary to create n different matrix and to solve the algebraic system there is the need of computing n matrix inversion. Finally for each element is necessary to evaluate the rejection criterion of 2.143 as done before, with the only difference that an element e can be removed only if that criterion is satisfied in every support condition.

Structures with stiffness or displacements constrains

One of the most important constraints in structural optimization is based on stiffness, often in order to reduce maximum displacements at certain points within a certain limit.

stiffness constrains According to finite element method the state of a structure can be written as

$$[K]\{u\} = \{P\} \tag{2.145}$$

where $[K]$ is the global stiffness matrix, $\{u\}$ and $\{P\}$ are the nodal displacement and nodal load vector.

The strain energy of the structure can be expressed as

$$C = \frac{1}{2}\{P\}^T\{u\} \tag{2.146}$$

and it is a sort of inverse measure of stiffness[51].

[50] From a FEA point of view this is not onerous computationally since the global stiffness matrix must be inverted only once, like in the case of one load condition.

[51] So maximizing stiffness is analogous to minimizing the strain energy.

After the removal of the generic element i from a structure with n finite elements, the stiffness matrix will change by the quantity[52].

$$\Delta[K] = [K^*] - [K] = -[K^i] \tag{2.147}$$

where $[K^*]$ is the stiffness matrix of the resulting structure after the element removal and $[K^i]$ is the stiffness matrix of the ith element.

Ignoring higher order terms from (2.145)

$$\{\Delta u\} = -[K]^{-1}\Delta[K]\{u\} \tag{2.148}$$

and from (2.146) and (2.147)

$$\Delta C = \frac{1}{2}\{P\}^T\{\Delta u\} = -\frac{1}{2}\{P\}^T - [K]^{-1}[\Delta K]\{u\} = \frac{1}{2}\{u^i\}^T[K^i]\{u^i\} \tag{2.149}$$

where $\{u^i\}$ is the displacement vector of the ith element.

The expression

$$\alpha_i = \frac{1}{2}\{u^i\}^T[K^i]\{u^i\} \tag{2.150}$$

defines the so-called *sensitivity number* that represent the change in stiffness after the removing of the ith element.

The optimization procedure aims to find the lightest structure that satisfies the limit

$$C \leq C^* \tag{2.151}$$

, where C^* is the prescribed limit for C. Each time an element is removed the overall stiffness decreases and, consequently, the strain energy C increases. So the most effective way to remove elements is to choose the elements with the minimum α_i in order to minimize the grow of C. The procedure of removals is repeated until is no more possible to remove elements without violating (2.151). At each iteration the number of elements to remove is defined by the *element removal ratio* which is the ratio between the number of elements to remove at each iteration and the number of elements of the original model, typical values are between 1% and 2%.

displacements constrains In this case the displacement at a specific location (for example u_j) must be within a prescribed limit (u_j^*). This constrain can be expressed in the form:

$$|u_j| \leq u_j^* \tag{2.152}$$

In order to find the change in u_j due to element removal, it's necessary to introduce the unit load vector $\{F_j\}$ in which only the jth component is equal

[52] It's implicit that the element removal won't affect the vector $\{P\}$

to unity and the others are equal to zero. Multiplying (2.148) by $\{F_j\}^T$ we obtain:

$$\Delta u_j = \{F^j\}^T - [K]^{-1}[K^i]\{u\} = \{u^j\}^T[K^i]\{u\} = \{u^{ij}\}^T[K^i]\{u^i\} \quad (2.153)$$

where $\{u^j\}$ is the displacement due to the unit load $\{F^j\}$, $\{u^i\}$ and $\{u^{ij}\}$ are the element displacement vectors containing respectively the entries of $\{u\}$ and $\{u^j\}$ which are related to the ith element. The value

$$\alpha_{ij} = \{u^{ij}\}^T[K^i]\{u^i\} \quad (2.154)$$

indicates the change in the displacement component $\{u_j\}$ due to the removal of the ith element. Differently from α_u in (2.150) that is always positive, α_{ij} in (2.154) can be either positive or negative, meaning that u_j may change in opposite directions. To minimize the change of $|u_j|$ in the most effective way, only the elements with the minimum α_{ij} should be removed according to the *element removal ratio*. The expression

$$\alpha_i = |\alpha_{ij}| \quad (2.155)$$

defines the so-called *sensitivity number* in the case of displacements constrains.

The optimization procedure can be summed up as follows:

1. Discretize of structure using a *fine* finite element mesh.
2. Solve the (2.145) for the given load $\{P\}$ and the virtual unit load $\{F^j\}$.
3. Compute the *sensitivity number* α_{ij} for each element.
4. Remove elements with lowest α_{ij}.
5. Iterate from 2 to 4 until 2.152 can no longer be satisfied.

constrains on the difference of two displacements This type of constrains can be useful in the design of buildings[53]. It can be expressed in the form

$$|u_j - u_k| \leq \delta^*_{jk} \quad (2.156)$$

where δ^*_{jk} is the prescribed limit for the difference between the displacements u_j and u_k. The effect of removing the ith element can be obtained in the form

$$\Delta|u_j - u_k| = |\{u^{ij}\}^T[K^i]\{u^i\} - \{u^{ik}\}^T[K^i]\{u^i\}| = |\{\delta^i_{jk}\}^T[K^i]\{u^i\}| \quad (2.157)$$

[53]For example, constraining the relative displacement between two adjacent floors within certain limits.

where $\{u^{ij}\}$ and $\{u^{ik}\}$ are the element displacement vectors of the ith element due to a unit load applied at the location and direction of u_j and u_k and $\delta^i_{jk} = \{u^{ij}\} - \{u^{ik}\}$ and the *sensitivity number* is defined as:

$$\alpha_i = \{\delta^i_{jk}\}^T [K^i] \{u^i\} \tag{2.158}$$

To minimize the change in displacement difference, it is most effective to remove the element whose sensitivity number α_i from (2.158) is closest to zero.

The optimization procedure can be summed up as follows:

1. Discretize of structure using a *fine* finite element mesh.
2. Solve the (2.145) for the given load $\{P\}$ and the two virtual unit load corresponding to $\{u_j\}$ and $\{u_k\}$.
3. Compute the *sensitivity number* α_i for each element according to (2.158).
4. Remove elements with lowest α_i.
5. Iterate from 1 to 3 until 2.156 can no longer be satisfied.

Multiple displacements constrains When it is required that displacements at several different locations or directions be within prescribed limits there is a so-called *multiple displacements constrain* and can be written in the form:

$$|u_j| \leq u^*_j \qquad \text{(j=1,...,m)} \tag{2.159}$$

where m is the total number of constrained displacements.

A classical way of dealing with this kind of constrain is the *Lagrangian* multipliers method. A simpler approach is the use of the weighted average of the expected changes in the displacements with constraints due to element removal. The weighted average can be expressed as:

$$\alpha_i = \sum_{j=1}^{m} \lambda_j |\alpha_{ij}| \tag{2.160}$$

and is defined as the *sensitivity number* for the ith element. In (2.160) α_{ij} is computed with (2.154) for each displacement constraint, and the weighting parameter λ_j is given by

$$\lambda_j = |u_j| / u^*_j \tag{2.161}$$

With this approach when a displacement is far below the prescribed limit, the corresponding weighting parameter will be very small, and thus the corresponding constraint will be a little significance in (2.160), but that constraint can become dominant when, after some iterations, the corresponding displacement is approaching to the limit. The optimization procedure can be summed up as follows:

1. Discretize of structure using a *fine* finite element mesh.
2. Solve the (2.145) for the given load $\{P\}$ and the virtual unit loads corresponding to all displacement constraints.
3. Compute the *sensitivity number* α_i for each element according to (2.160).
4. Remove elements with lowest α_i.
5. Iterate from 2 to 4 until conditions in 2.159 can no longer be satisfied.

Frequency optimization

The dynamic behaviour of a structure depends largely on the first few natural frequencies. When the frequency of the external dynamic load is "near" one of them, sever vibrations can occur. In order to avoid that it is often necessary to shift the fundamental frequency or several of the lower frequencies away from the frequency range of the external dynamic loading.

Sensitivity number for frequency optimization Sensitivity derivatives are often needed to found the best locations for structural modifications. But usually sensitivity analysis is usually very complicated, especially in dynamics problems. For this reason it can be used a simpler sensitivity number which measure the effect on a natural frequency due to the removal of a generic element.

It is known from FEA that the dynamic behaviour of a structure can be expressed, as eigenvalue problem, in the form

$$([K] - \omega_n^2[M])\{u_n\} = \{0\} \tag{2.162}$$

where $[K]$ and $[M]$ are the global stiffness and global mass matrix. ω_n is the nth natural frequency and $\{u_n\}$ is the eigenvector corresponding to ω_n. ω_n and $\{u_n\}$ are strictly related each other by the Rayleigh quotient

$$\omega_n^2 = \frac{k_n}{m_n} \tag{2.163}$$

where the modal stiffness k_n and the modal mass m_n are defined as

$$k_n = \{u_n\}^T[K]\{u_n\} \tag{2.164}$$

$$m_n = \{u_n\}^T[M]\{u_n\} \tag{2.165}$$

Removing the i-th element from the current structure the change in frequency will be (from (2.163))

$$\Delta(\omega_n^2) = \frac{\Delta k_n}{m_n} - \frac{k_n \Delta m_n}{m_n^2} = \frac{1}{m_n}(\Delta k_n - \omega_n^2 \Delta m_n). \tag{2.166}$$

To obtain the value of $\Delta(\omega_n^2)$ we assume that the eigenvector $\{u_n\}$ is approximately the same before and after the removal of the element[54]. Therefore we have

$$\Delta k_n \approx \{u_n\}^T[K]\{u_n\} = -\{u_n^i\}^T[K^i]\{u_n^i\} \tag{2.167}$$

$$\Delta m_n \approx \{u_n\}^T[M]\{u_n\} = -\{u_n^i\}^T[M^i]\{u_n^i\} \tag{2.168}$$

where $[K^i]$ and $[M^i]$ are the stiffness and mass matrices of the i-th element, and $\{u_n^i\}$ is the eigenvector of the i-th element.

After substitution of approximations (2.167) and (2.168) into equation (2.166) the approximation of the change of the frequency due to the removal of i-th element is

$$\Delta(\omega_n^2) \approx \frac{1}{m_n}\{u_n^i\}^T(\omega_n^2[M^i] - [K^i])\{u_n^i\} \tag{2.169}$$

To decide which element should be removed from the structure so that the frequency will be shifted towards a desired value it is necessary to calculate, for each element, the sensitivity number for frequency as follows:

$$\alpha_n^i = \frac{1}{m_n}\{u_n^i\}^T(\omega_n^2[M^i] - [K^i])\{u_n^i\}. \tag{2.170}$$

This value is an indicator of the change in ω_n^2 as a result of the removal of the i-th element.

Evolutionary procedures for frequency optimization With evolutionary structural optimization the frequency of a structure can be shifted towards a desired value by gradually removing material from the structure. The procedure can be summed up as follow:

1. discretize the structure with a fine mesh of finite elements;
2. solve the eigenvalue problem (2.162);
3. calculate sensitivity number α_n^i using (2.170)
    - To increase a chosen frequency: remove a number of elements which have the highest α_n^i.
    - To reduce a chosen frequency: remove a number of elements which have the lowest α_n^i.
    - To keep a chosen frequency ω_n constant: remove a number of elements which have the smallest $|\alpha_n^i|$.

[54]The assumption that the mode shape does not change significantly in between design cycles is commonly used in frequency optimization [58].

- To increase the gap between two natural frequencies ω_p and ω_q: remove a number of elements which have the highest α^i_{pq} where

$$\alpha^i_{pq} \approx \Delta(\omega_p^2 - \omega_q^2) \approx \alpha^i_p - \alpha^i_q \quad \text{where } p > q \tag{2.171}$$

4. Repeat steps from 2 to 3 until optimum is reached.

Truss-like structures

Evolutionary procedures can be applied also for pin jointed frames using very small area steps:

1. An initial structure is defined including loads and support conditions.
2. Conduct FEA.
3. Calculate stresses.
4. Compare member with target value.
5. If absolute stress is above target increase area by a small increment. Instead if absolute stress is below target decrease area by a small increment.
6. If area is diminished to zero, remove element from structure. If area has reached a lower or an upper bound then freeze area.
7. Check to see if previously frozen areas need unfrozen.
8. If volume change per iteration is within a small convergence tolerance or a prescribed iteration limit has been reached, then stop, otherwise go to step 2.

A similar approach could be adopted also for rigid jointed frame (see [74]).

Chapter 3

Genetic algorithms

3.1 Introduction

Most of the methods described in the previous chapters of the present thesis are based on deterministic algorithms. Many problems can be solved with those approaches, but many others can't. Computer are usually seen as "static" machines: it seems obvious that a computer program begins from an initial point to a final point, mindlessly following a fixed specific path. This is true for the vast majority of applications we are accustomed to, but it is not in general true. Over the years, some techniques have been developed to introduce the concept of *adaptabilty* in computer programs, leading to the creation of softwares which are able to adapt their own behaviour when conditions, previously unforeseen by the same programmers, happens.

The original idea takes its inspiration simply by looking at the nature around us. In biology adaptability and flexibility are the foundations of life. Implementing biological concepts creates software that evolves solutions. In particular, genetic algorithms are based on the observations due to Charles Darwin (in the mid of nineteen century) and in his evolutionary theory based on the process of *natural selection*.

According to Darwin and later scientists, while the survival of individuals of a population determines the characteristics of the next generation, it is its reproductive success of population of a whole that determines, over the years, while generations succeed each other, the evolution of a species.

The process of natural selection, commonly known as "the survival of the fittest", actually operates through the survival of the best organisms available at the time. The same concept of organism's fitness is changing over the years. What is best today, can't be that tomorrow and it couldn't even been good yesterday. While ignoring how the characteristics were passed from parents to offsprings Darwin noticed that all those was happening.

At his time DNA was totally unknown but later it was discovered that those molecule store a really large amount of information, moreover in a

very tiny space. Offspring inherit characteristics through genes (included in DNA) received by parents. Reproduction can happen in different forms:

- simplest organisms (fungi, bacteria...) reproduce asexually by duplicating themselves; a single-celled amoeba, for example, creates offspring by splitting into two new organisms containing the same DNA.
- complex organisms reproduce sexually by combining genes from two parents in their offspring.

The last approach, by mixing DNA from two different organisms, guarantees a large variation within a species. Anyway, with this kind of reproduction, genes are mixed within a population but there isn't creation of new genes.

The genetic information in a population is defined as *genetic pool*. The more the genetic pool is large, the more the population will be healthier because this will allow a grater number of genetic combinations. It also will give greater adaptabilty and less chanche to develop recessive genetic disorders.

The variability among individuals is also increased by *mutation* which is a random change in genes. Often mutations has no effect or they disappear as a result of natural selection. But rare mutations are useful because they can introduce new genes within the population. Reproduction and mutations evolve population. The evolution can happen in a straight way (carrying a species from a form to another) or it can lead to the creation of a new species.

A simplified version of those natural phenomena are simulated in genetic algorithms (GA). More detail can be found in with Goldberg's book [21]. In the opinion of the author of this thesis, GAs are a powerful tool for structural optimization. A computer program in C++ has been written to test the potentiality of this approach. More details on GAs and some topics on their implementation will be discussed in this chapter.

3.1.1 On the definition of genetic algorithm

According to Ladd see [38], the quest to apply evolution to computer software is not new. Historically, the first papers on this topic date back to 50s and are due a Norwegian-Italian mathematician: Nils Aall Barricelli[1]. Other studies were published during 60*s* and 70s but only starting from 1975, with Holland's work [27] (and later with his phD students such as Goldberg [21])

[1]Nils Aall Barricelli (b.1912 - d. 1993) was one of the pioneers in evolutionary computation. His publication of "Esempi Numerici di processi di evoluzione", a study in what would now be called artificial life, in the journal Methodos, in 1954 is perhaps the earliest published record of an evolutionary simulation. The paper was republished in 1957 in English, and detailed the results of programs that were run at the Institute for Advanced Study in Princeton, NJ, in 1953.

genetic algorithms become popular among scientists. The first industrial applications were developed during 80s and 90s.

While computer scientists still discuss about a rigorous and commonly accepted definition for genetic algorithms, it can be said that, in a broad sense, it is an adaptation of a biological process, whereby organisms evolve by rearranging genetic material to survive in environments confronting them. A genetic algorithm creates a set of solutions that reproduce based in their fitness in a given environment. It can summed up in the following basic steps:

1. An initial population of random solutions is created.
2. A value of fitness is first evaluated and then assigned to each member of the population.
3. Solution with higher fitness values are most likely to parent new solutions during reproduction.
4. When the new solution set replaces the old (i.e. a generation is complete) the process continues at step 2 until an "acceptable" solution is found.

The outcome of a genetic algorithm is strictly based on probability. The least-fit individual has a small chance at are reproduction, will the most-fit solution could not reproduce at all. In analogy to DNA the data are stored in chromosomes containing genes. Chromosomes have been introduced to group the optimization parameters in homogenous group.

3.2 Parameters of solution

To implement a genetic algorithm, several parameters must be precisely defined such as

- the population size;
- the selection criterion;
- the *genetic operators* (crossover and mutation).

3.2.1 Population

The population consist in the set of solutions which must be evaluated at each generation. In general each organism can be seen as a vector of data. In general, each element of the vector can contain any kind of information: integer and real numbers, boolean expressions, characters, bits and so on. According to standard C++ data types, the software that has been developed support the following types of data:

- `int`s;
- `float`s;
- `bool`s;
- `char`s;

3.2.2 Selection

Genetic algorithm use a selection mechanism to select individuals from the population to insert into a mating pool. Individuals from mating pool are used to generate new offspring, with resulting offspring forming the basis of the next generation. As the individuals in the mating pool are the ones whose genes are inherited by the next generation, it is desiderable that the mating pool contains good individuals. For this reason selection is one of the most important aspects of the GA process.

Many techniques have been introduced to find the individual that will be chosen as parents [3]. The most popular are:

tournament ;

roulette wheel ;

stochastic universal sampling ;

ranking ;

steady-state ;

uniform ;

and others.

The most important selection algorithms will be discussed later but before it is necessary to give some definitions. A selection mechanism is simply a process that favors selection of better individuals in the population for the mating pool. Instead the *selection pressure* is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. It can also be seen as the probability of the best individual being selected compared to the average probability of selection of all individuals. The selection pressure drives the GA to improve the population fitness over succeeding generations. The convergence rate of a GA is largely determined by the selection pressure: higher selection pressure results in higher convergence rates.

Genetic algorithms are able to identify optimal or near-optimal solutions under a wide range of selection pressure (see [26]). However, if the selection pressure is too low, the convergence rate will be slow, and the GA will take

longer to find the optimal solution. If the selection pressure is too high, there is an increased chance of the GA prematurely converging to an incorrect (i.e. non optimal) solution.

It is known that in order to have an effective search there must be a search criteria (which is given by the fitness function) and a selection pressure that gives individuals with higher fitness a higher chance of being selected. A good selection algorithm should

- provide selective pressure (and avoid premature convergence[2]);
- maintain high levels of population diversity;
- allow a wide exploration of the whole research space.

According to Whitley [16], without selection pressure, the search process becomes random and promising regions of the search space would not be favored over non-promising regions. On the other hand, population diversity is crucial to let the genetic algorithm continue a fruitful exploration of the search space. If the lack of population diversity takes place too early, a premature stagnation of the search is caused. Under these circumstances, the search is likely to be trapped in a region not containing the global optimum. This problem, called premature convergence, has long been recognized as a serious failure mode for GAs (see [40] and [21]).

Moreover, selective pressure and population diversity are inversely related [16]: increasing selective pressure results in a faster loss of population diversity, while maintaining population diversity offsets the effect of increasing selective pressure. The problem of finding the best balance between these two factors in order that they can bring their beneficial advantages simultaneously have been addressed in [42], where a method for finding a *useful* compromise is proposed. To accomplish this, most selection methods include stochastic functions and are designed so that a small proportion of less fit solutions are selected. This aspect is really important because it helps to keep the diversity of the population large, preventing premature convergence on poor solutions. The space of all feasible solutions is called *search space*. Each point in the search space represent one feasible solution.

Tournament selection

Tournament selection provides selection pressure by holding a tournament among s competitors, chosen at random from the population. The parameter s represents the tournament size. The winner of the tournament is

[2]Premature convergence can be detected in different ways: for example evaluating takeover time (i.e. till best individual replaces all others). If the best individual is good enough it means that convergence was not premature. Obviously it also depends on what kind of solution we are expecting (the optimal solution, a good solution, any solution, a solution within some specific time, a good (on average) set of solutions, and so on.)

the individual with the highest fitness of the s tournament competitors[3], and the winner is then inserted into the mating pool. The mating pool, being comprised of tournament winners, has a higher average fitness than the average population fitness. This fitness difference provides the selection pressure, which drives the GA to improve the fitness of each succeeding generation. Increased selection pressure can be provided by simply increasing the tournament size s, as the winner from a larger tournament will, on average, have a higher fitness than the winner of a smaller tournament. The chosen individual can be removed from the population that the selection is made from if desired, otherwise individuals can be selected more than once for the next generation. The whole process is repeated as often as many individuals must be chosen.

Moreover, with little modifications, a more general procedure can be introduced, which include stochastic approach by a probability p:

- fix the tournament size s;
- chose, randomly, s individuals from the population;
- select the best individual from pool or tournament with probability p;
- select the second best individual with probability $p(1-p)$;
- select the third best individual with probability $p(1-p)^2$, and so on. In general the i-th individual has probability $p(1-p)^{(i-1)}$.

where p is in the range $0.5 < p \leq 1$. This values are the probability of the fittest candidate being selected in any given tournament. The restriction that the probability must be greater than 0.5 has been added, since any lower value would favour weaker candidates over strong ones, negating the "survival of the fittest" aspect of the evolutionary algorithm. When $p = 1$ the tournament selection is deterministic and simply selects the best individual in any tournament.

Tournament selection has several advantages: it is simple to code and efficient for both non-parallel and parallel architectures. Besides, since larger values of s decrease the chance of least-fit individuals to be chosen, selection pressure can be easily adjusted by changing the tournament size. Details and advanced applications can be found in [45]. More recently some modifications have been suggested in [75]. Compared to other algorithms is less computationally expensive since it does not require the fitness evaluation of all the individuals.

[3] A 1-way tournament ($s = 1$) selection is equivalent to random selection.

Roulette wheel selection

The value of fitness f_i is used to associate a selection probability p_i of selection for the i-th individual with the expression

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{3.1}$$

where N is the number of individuals in the population.

It is a stochastic algorithm and involves the following technique:

The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. It also known as *fitness proportionate selection* or *stochastic sampling with replacement*. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness, see Figure . If a marble is thrown there then an individual is selected.

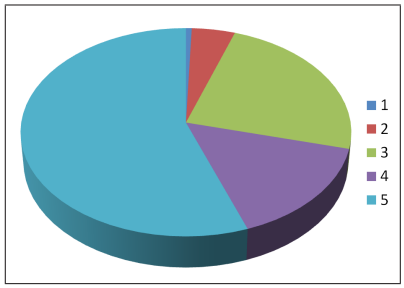


Figure 3.1: Roulette wheel for a population of 5 individuals: each slice is proportional in size to the fitness.

Individuals with bigger fitness will be selected more times.

While candidate solutions with a higher fitness will be less likely to be eliminated, there is still a chance that they may be. Moreover, there is a chance some weaker solutions may survive the selection process[4]. This technique has high selection pressure but can easily lead to premature convergence.

In particular some considerations should be taken in account when this algorithm is chosen. The first is that the results are influenced by the population size. The larger is that size, the more diminish the influence of high fitness individuals on reproductive success [38]. Secondly *fitness scaling* should be take into account.

[4]It doesn't happen in *truncation selection*, a simpler and deterministic algorithm, which eliminates a fixed percentage of the weakest individuals.

Fitness scaling is an enhancement to the original procedure. As a population converges on a definitive solution, the differences between fitness values may become very small. In other words, that produces a roulette wheel with all the "slices" really similar in size, preventing the best solutions from having a significant advantage in selection. Fitness scaling solve this problem by adjusting the fitness values to the advantage of the most fit individuals.

Windowing is the simplest form of fitness scaling [38]. After having computed the fitness value for each of all individuals, the smallest value is found. Then this minimum value is subtracted to the fitness of each element leading to a fitness array with the smallest value equal to zero. As an alternative, to ensure that all the individuals have a chance at reproduction, it is also possible to subtract a number which is slightly smaller than the minimum fitness.

example Table 3.1 shows a population of 11 individuals with their fitness values f_i and probability p_i.

individual	fitness f_i	probability p_i
1	2.0	0.18
2	1.8	0.16
3	1.6	0.15
4	1.4	0.13
5	1.2	0.11
6	1.0	0.09
7	0.8	0.07
8	0.6	0.06
9	0.4	0.03
10	0.2	0.02
11	0.0	0.00

Table 3.1: roulette wheel selection example: population

The process od selection of 6 individuals consists in the generation of 6 random numbers such as:

0.81 0.32 0.96 0.01 0.65 0.42

As can be seen from Figure 3.2.2 the selection results in the following individuals:

1 2 3 5 6 9

Stochastic universal sampling

Introduced by Baker [32] it can be seen as a development of the roulette wheel selection. Individuals are mapped to segments of a line. The segments

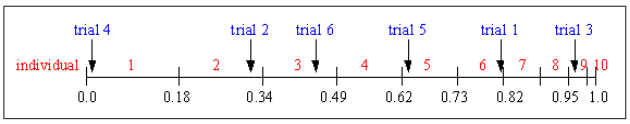


Figure 3.2: roulette wheel selection example: graphic representation of selection mechanism.

are continuous. Each of them represents the fitness of an individual as seen in Figure 3.2.2 but, in this case, the trials (or pointers) are equally spaced. The total number of pointers n equals the number of individuals to select. So, differently from roulette wheel selection where n repeated random sampling is required, in this case only a single random value is necessary to sample all of the solution.

Example Using the same population of previous example, the distance d between pointers is computed:

$$d = 1/6 = 0.167.$$

In this case only one random number (in the range [0,0.167]) is required. Let's suppose is

$$0.1$$

The other pointers can then be computed from the random point

$$0.267 \qquad 0.367 \qquad 0.434 \qquad 0.601 \qquad 0.768 \qquad 0.935$$

The procedure can be seen in Figure 3.2.2 where the final selection is shown:

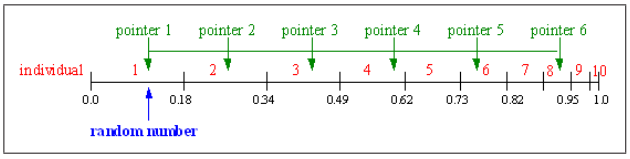


Figure 3.3: Stochastic universal sampling selection

$$1 \qquad 2 \qquad 3 \qquad 4 \qquad 6 \qquad 8$$

Rank-based selection

In rank-based selection (see [66]) population is sorted according to the fitness values of the single individuals. The fitness assigned to each individual depends only on its position in the individual rank and not on the actual

objective value. The worst (least fit) individual will have fitness 1, second worst 2 and so on, and the best will have fitness n, where n is the population size. In this way all the individuals have a chance to be selected, but this can lead to slower convergence, because the best individuals do not differ so much from other ones.

The effects of rank-based selection can be seen in Figure 3.2.2 that shows a population with $n = 5$ with different values of fitness. Figure 3.2.2) shows the same population after rank selection.

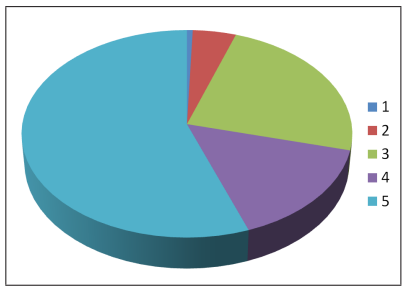


Figure 3.4: Graph of fitness (before rank selection)

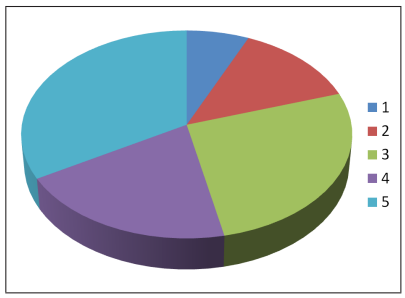


Figure 3.5: Graph of order numbers (after rank selection)

This approach overcomes the scaling problems of proportional fitness assignment, it tends to preserve diversity, and reduce selection pressure.

Rank based selection includes several variants: it can be presented with linear or nonlinear scaling. In the linear form the fitness is written as

$$\text{fitness(position)} = \text{minValue} + (\text{maxValue} - \text{minValue})(\text{position}) - 1)/(n-1) \tag{3.2}$$

where minValue and maxValue are chosen by operators, such in the following expression

$$\text{fitness(position)} = 2 - sp + 2(sp-1)\frac{\text{position} - 1}{n-1} \tag{3.3}$$

where sp is the selective pressure ($sp \in [1, 2]$) and n is the population size. In the nonlinear form the fitness could be computed as

$$\text{fitness(position)} = \frac{nX^{(\text{position}-1)}}{\sum_{i=1}^{n} X^{(i-1)}} \tag{3.4}$$

where X is computed as the root of the polynomial

$$0 = (sp - n)X^{(n-1)} + sp \cdot X^{(n-2)} + \ldots + sp \cdot X + sp \tag{3.5}$$

and $sp \in [1, n-2]$.

A qualitative comparison can be seen in Figure 3.2.2 in Figure 3.2.2

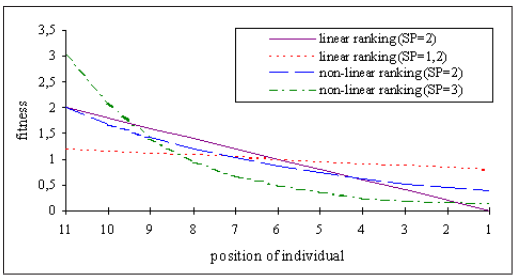


Figure 3.6: Comparison between different Rank-based selection algorithms (linear and nonlinear with different values of selective pressure)

Ranked roulette wheel selection

Al Jadaan, Rajamani, and Rao [3] applied a rank based approach to the roulette wheel selection, by substituting (3.1) with

$$p_i = \frac{2 \cdot \text{position}}{n(n+1)}. \tag{3.6}$$

This is known as *ranked roulette wheel selection*. The authors applied this technique to eight test functions taken from the GA literature. The results obtained from the comparison between applying the roulette wheel selection and ranked based roulette wheel selection, shows a faster performance of the latter. The main problem with the standard roulette wheel selection is that if good solution is discovered early, its fitness value dominates other fitness values. Then it will occupy majority portions of the mating pool. This will reduce the diversity in the mating pool and cause the GAs to converge to wrong solutions. Ranked roulette wheel selection tries to overcome this problem and by increasing the diversity.

Steady state selection

In every generation a few[5] individuals with high values of fitness are selected for creating a new offspring. Then a corresponding number of individuals are removed and the new offspring is placed in their place. The rest of population survives to new generation. The *replacement strategy* can be addressed in many ways (i.e. removing the worst, the oldest, or randomly chosen individual). This selection criterion leads to an *overlapping* system, where both parents and offspring compete for survival [65]. Its simpler form can be summed up in the following steps:

1. Select two parents from the population.
2. Create an offspring using crossover and mutation.
3. Evaluate the offspring with the fitness function.
4. Select an individual in the population, which may be replaced by the offspring.
5. Decide if this individual will be replaced.

More sophisticated variants can be introduced by studying more deeply steps 4 and 5: some researchers propose a replacement strategy that takes into account two features of the element to be included into the population: a measure of the contribution of diversity to the population and the fitness function. The idea is to replace an element in the population with worse values for these two features. With this strategy is possible to protect those individuals that allow the highest levels of diversity to be maintained. More details can be seen in [42].

Fitness uniform selection

Said m and M the lowest and highest fitness in current generation, a fitness f is randomly (uniformly) selected in $[m, M]$ and the individual with fitness closest to f is chosen. Then, to avoid memory problems, individuals from the most occupied fitness levels must be deleted. This technique maintains genetic diversity since only one solution is required to have maximal fitness. It aims to favor those solutions that are different from all others and encourages search [41].

Elitism

Elitism copies the best individual (or a few best individuals) to new population. Since it prevents losing the best found solution (until better solutions

[5] Usually only one or two (see [42]).

are found), elitism can very rapidly increase performance of genetic algorithms. Elitism size s_e denotes the number of best individuals that must survive at each generation.

3.3 Data structure

Each individual represent a possible solution of the optimization problem. The information it contains is stored, from analogy with biology, in genes which are in turn grouped in chromosomes. Differently from most approaches, here an individual contains one or more chromosome. Actually the chromosome structure is not strictly necessary, because the data are inside genes but they have been introduced in order to get better organized data. In fact, even if unneeded, they can be used to create homogeneous group of data. For example, in a structural optimization problem, the first chromosome could contain design variables dealing with structural geometry (such as nodal positions), the second dealing with external loads, the third with elements' cross section and so on. This could make the data files much more readable. Besides that, during an optimization process a chromosome could be temporary "frozen" and so applied only to a subset of the design variables. This approach could be useful when dealing with complex problems, leading to a sort of multilevel optimization.

3.3.1 Data encoding

On of the most important aspects of genetic algorithms is the data encoding, since it brings substantial consequences on the implementation and on the final results of the optimization process.

Commonly in structural optimization we deal with several types of data:

- integer variables (such as the number of nodes in the finite element mesh, the number of finite elements, the index representing a particular cross section inside a section data base, and so on).
- floating point values for representing the structural response (i.e stresses, displacements and frequencies), geometric properties (for example areas, moments of inertia), properties of materials, and so on.
- bool values for on-off choices.
- bit values for setting configurations (such as restrain)[6]

[6]Bit values can also be substituted by boolean variables, but when dealing with many parameters, bit values are more efficient in terms of memory and computation cost, because of the different way they are represented (and corresponding operations are implemented) in calculators.

In computer implementation we must take in account that each of them is characterized by different internal representation, depending on language (or compiler) choice and computer architecture.

Data are encoded in genes. In the most common approach data are encoded as bit. In such a case chromosomes (and individuals) are consequently represented by a string of bit-values. This has been shown in 2.6.2 at page 30. In that paragraph we saw that the expression (2.48) computes the number of binary digits that are required to approximate a continuous variable with the desired accuracy. The main disadvantage is that this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

In general, both integer and real values can be represented as bit strings in various ways. A discretization of real values can be mapped to integers, and integers can be finally mapped to bits by using the standard binary coded decimal representation. For example, the integers 15 and 16 are encoded as 01111 and 10000. These values are neighbors in integer space, but they are not neighbors in terms of the bit-space or *Hamming* neighborhood associated with the standard binary representation. This happens because the two mentioned bit string form a so-called Hamming cliff (i.e. adjacent integers are represented by complementary bit strings and thus share no bits in common). It may be desirable to use a bit encoding where adjacent integers are represented by bit strings that are neighbors in Hamming space and thus differ by a single bit. *Gray codes* benefit of this property: for any integer i the binary representation of i is Hamming distance 1 away from bit representation of the integers $i+1$ and $i-1$. Moreover, 0 and the maximum represented integer are adjacent as well (see [17], [19]). The actual number gray codes is not known, but one of the most commonly used is the *standard binary reflected gray code (SBRGC)*. One of the most important advantages in grey codes in place of the standard binary code is that the former are always guaranteed to produce a representation in Hamming space where the number of local optima is less than or equal to the number of optima in the original real-valued or integer function representation [18]. Some researchers argue that grey codes must be used if the natural representation of the problem is integer or real valued but the question is still open. Under certain circumstances binary seems to perform better than grey codes.

Particular attention should be given to precision as well. Numerical tests has demonstrated that comparing real valued and bit representation is much more complex than most of literature suggest: genetic algorithms at 20 bits of precision can be 10 to 100 times slower to converge using 20 versus 10 bits of precision. On one side low precision could lead to miss good solution (but allows a wider exploration); on the other side high precision can result in a narrower and slower exploration [15].

We have may other types of encoding. The most appropriate choice depend mainly on the features of the problem to solve. For ordering problem

can be used *permutation encoding*: each chromosome is a string of integers, which represents number in a sequence. For example

$$8 \quad 4 \quad 1 \quad 2 \quad 9 \quad 5 \quad 6 \quad 7 \quad 3$$

Many problem in the field of structural optimization deals mainly with integer and float values. In such a case (and in the writer's opinion) the *value encoding* seems to be the most proper approach, by encoding directly integer and real numbers. Eventually also chars, or complex object can be included. A chromosome is a string of some values[7] and could have the form:

$$1 \quad 4.5312 \quad 31.434 \quad \text{(up)} \quad 6 \quad 1.4383 \times 10^{-12} \quad r \quad \text{(down)}$$

This is the encoding adopted in the optimization software that has been developed.

Tree encoding is widespread in genetic programming, where data are expressions or evolving programs. Every chromosome is a tree of some objects, such as functions (see Figure 3.3.1) or commands in programming language (see Figure 3.3.1). It's mainly used with programming language LISP, because programs in it are represented in this form and can be easily parsed as a tree [30].

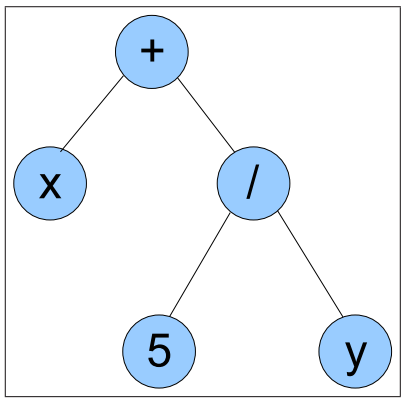


Figure 3.7: Tree encoding of expression $(+x/(5y))$

Depending on the adopted encoding scheme, proper crossover and mutation operators must be developed.

On value encoding for real numbers

Most implementation of genetic algorithms using floating values design variables work with bit strings. Others use a different encoding, such as trans-

[7] Values can be anything connected to problem: integer numbers, real numbers, chars or more complicated objects.

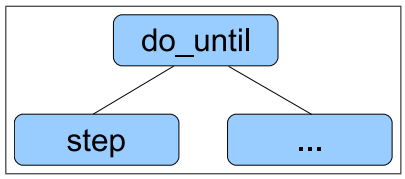


Figure 3.8: Tree encoding of a LISP expression

forming a 44 bit string into two floating point values via a series of operations [35] or similar approaches. In the author opinion the most interesting technique is due to Ladd [38] and has been adopted in the optimization software whose result are discussed at the end of the present thesis. Since the software has been developed in C++ it have been appeared natural to implement Ladd's enconding since it takes in account the way the floating point variables are stored in calculators.

The most common C/C++ compilers for PCs (like GNU gcc or Microsoft Visual C++) implement 32-bit floats and 64-bit long double according to *IEEE standard 754-1985*[8]. This standard, developed by the Institute of Electrical and Electronic Engineers (IEEE) defines those single precision and double precision floating points formats. More precisely a `float` is a 32-bit value, while a `double` is a 64-bit values. In IEEE 754 floating-point representation, these bits are divided in three basic components:

- a sign bit s;
- an exponent exp;
- a mantissa m.

The sign bit is the highest order bit and it simply defines the polarity of the number: a value of zero means the number is positive, while a 1 denotes a negative number.

The mantissa is divided into a fraction and leading digit. It mantissa represents the number to be multiplied by 2 raised to the power of the exponent. Numbers are always normalized, represented with one non-zero leading digit in front of the radix point. In binary math, 1 is the only non-zero number. Thus, the leading digit is always 1, allowing us to leave it out and use all the mantissa bits to represent the fraction (the decimals). This trick maximize the range of possible numbers as well.

But since the mantissa is a binary fraction, it can't always store *exactly* a decimal value. For example there is no binary fraction capable to represent

[8]Recently, a new revision has been published by IEEE. The current version is the IEEE 754-2008 that has been published in August 2008. It introduces some modifications such as 3 new formats. So now there are three binary floating-point formats (which can be encoded using 32, 64, or 128 bits) and two decimal floating-point formats.

without errors the values 0.6 or 1/3. Floating point must be seen as an approximation of a decimal value; this is where rounding errors come from.

The exponent represents a range of numbers, positive and negative. More precisely it is a binary number representing the number of binary digits is shifted left (for a positive actual exponent) or shifted right (for a negative actual exponent). It is also a biased value, thus a bias value must be subtracted from the stored exponent to yield the actual exponent. The single-precision bias is 127, while the double-precision bias is 1023. This means that for a `float` a stored value of 100 indicates a single-precision exponent of -27. The exponent base is always 2; this implicit value is not stored.

In `float`s the exponent occupies 8 bits, while the mantissa uses the remaining 23 bits. It can be seen in the following diagram:

```
 1     8               23               bit lenght
+-+--------+-----------------------+
|s|  exp   |  mantissa             |
+-+--------+-----------------------+
31 30      22                      0    bit index
```

Instead, `double`s have a 52-bit mantissa and a 11-bit exponent. A graphic representation follows.

```
 1     11                                52
+-+-----------+-------------------------------------------------------+
|s|  exp      |  mantissa                                             |
+-+-----------+-------------------------------------------------------+
64 63        52                                                      0
```

Moreover IEEE 754 admit some unusual values. For both representations, exponent representations of all 0s and all 1s are reserved and used to indicate the following special numbers:

- 0 when all digits (both in mantissa and exponent) are set to 0, while sign bit can be either 0 or 1;
- $\pm\infty$ when exponent is all 1s and fraction is all zeros;
- *NaN*[9] when exponent is all 1s and mantissa contains any set of bits that is not all zeros. In this case the sign bit is irrelevant.

These special cases are really important. Any routine with operates with bits in floating point representation must avoid generating these special values. Most of C++ compilers are not able to process them and generate

[9] *NaN* stands for "Not a number". Two versions of *NaN*s are available: they are used to represent the result of invalid operations such as dividing by zero, or indeterminate results such as operations with non-initialized operands.

exceptions, so the involved algorithm must be able to manage those situations.

Crossover and mutation operators are deeply influenced by the encoding. When Ladd's encoding is adopted, special techniques must be introducted in those operator in order to manage safely bits in floating point representation, avoiding the generation of the special number I've just described.

3.3.2 Crossover operators

Crossover operators depend on data encoding.

binary encoding

In the case of binary encoding several types of crossover are possible (for the sake of clearness only the case of one-chromosome individual is considered, but the following considerations are valid in general):

single point : one crossover point is selected, binary string from the beginning of the individual's chromosome to the crossover point is copied from the parent A, the rest is copied from the parent B. For example: **11001**011 + 11011**111** = **11001111** (see Figure 3.3.2);

multi point : several crossover points are selected. For simplicity let's consider a two point crossover: two crossover points are selected, binary string from the beginning of the chromosome to the first crossover point is copied from the first parent, the part from the first to the second crossover point is copied from the other parent and the rest is copied from the first parent again. For example: **11**001**011** + 11**011**111 = **11011111** (see Figure 3.3.2);

uniform : bit values are randomly copied from the first or from the second parent. For example: **11**001**011** + 11**011**1**0**1 = **11011111** (see Figure 3.3.2);

arithmetic : an arithmetic operator is chosen and applied to the parents to generate the offspring. For example: 11001011 AND 11011111 = 11001001 (see Figure 3.3.2).

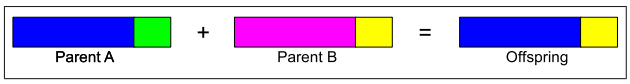


Figure 3.9: Single point crossover for binary encoding

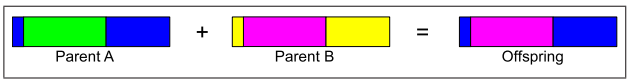


Figure 3.10: Two points crossover for binary encoding

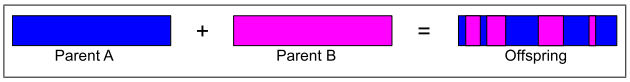


Figure 3.11: Uniform crossover for binary encoding

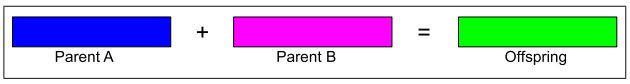


Figure 3.12: Arithmetic crossover for binary encoding

permutation encoding

For permutation encoding even a single point crossover can be computed in different ways. As an example, after having selected one crossover points, by copying the permutation is copied from the first parent till the crossover point, then the other parent is scanned and if the number is not yet in the offspring, it is added.

$$(\mathbf{1\ 2\ 3\ 4\ 5}\ 6\ 7\ 8\ 9) + (\mathbf{4\ 5\ 3\ 6\ 8}\ 9\ 7\ 2\ 1) = (\mathbf{1\ 2\ 3\ 4\ 5}\ 6\ 8\ 9\ 7)$$

value encoding

Theorically, for value encoding all crossovers operators from binary encoding can be used. But if Ladd's encoding is adopted, a more sophisticated technique must be used. Differently from mutation (see section 3.3.3) the roulette wheel mechanism is no more required, but it remains necessary to check the bits of the exponent part. This check is required to ensure that any output value is not one of the special numbers (NaN or $\pm\infty$) seen so far. If it happens, the crossover is repeated (discarding offsprings) until this condition is satisfied.

tree encoding

One crossover point is selected in both parents and the parents are divided in that point. A new offspring is finally generated by exchanging the parts below crossover points (see Figure 3.3.2).

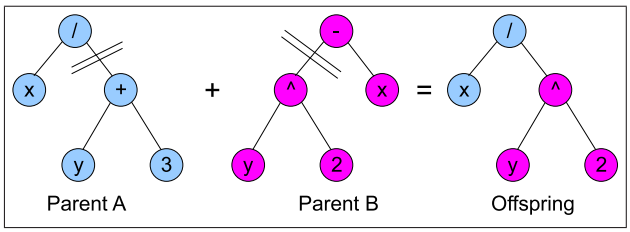


Figure 3.13: One point crossover for tree encoding

3.3.3 Mutation operators

Mutation is done after crossover and depends on data encoding as well.

binary encoding

When data are binary encoded mutation is computed with a simple bit inversion: one or more bits are selected and then inverted (see Figure 3.3.3).

Figure 3.14: Mutation for binary encoding

permutation encoding

In this case mutation consists in a simple order changing: two numbers are selected and exchanged.

$$(1\ \mathbf{7}\ 4\ 3\ 5\ \mathbf{6}\ 8\ 9\ 2) \rightsquigarrow (1\ \mathbf{6}\ 4\ 3\ 5\ \mathbf{7}\ 8\ 9\ 2)$$

value encoding

Several approaches are available. For real value encoding a simple strategy is adding (or subtracting) a small random number to the selected values.

$$(8.65\ \mathbf{1.22}\ 4.55\ 8.21\ \mathbf{2.01}\ 9.98\ 0.21) \rightsquigarrow (8.65\ \mathbf{1.27}\ 4.55\ 8.21\ \mathbf{1.98}\ 9.98\ 0.21)$$

If Ladd's encoding is adopted, a more sophisticated technique must be used.

mutation with Ladd's encoding requires special attention in managing the bits of IEEE 754 standard representation in order to avoid generation of special numbers (such as NaNs or ∞). Besides that the probability of crossover must take in account the different effects of exchanging bits in exponent, mantissa or sign. Each of these three groups has a different weight on the change of the floating value represented. For this reason, the adopted algorithm considers three values for crossover probability, representing the relative chances of changing the parts of floating point number:

- P_{exp} for the exponent;
- P_m for the mantissa;
- P_{sign} for the bit sign.

In this way it is possible to calibrate the influence of these three components. It's clear that changing the value of a bit collocated in the exponent or sign part of representation can produce dramatic effect, since that change may dramatically alter the magnitude of a floating point number. For the same reason changes in mantissa are usually less influent.

Ladd computed the probability that a random bit change will affect a specific component, assuming that all bits have equal chance of mutation. The result can be seen in table 3.2

	float	double
sign bit	3.1%	1.6%
exponent	25.0%	17.1%
mantissa	71.9%	81.3%

Table 3.2: Probability of bit locations for a random bit change (from [38])

The exponent, in particular, can introduce large fluctuation inside population. This behaviour can be corrected by adopting different probabilities and using a technique similar to roulette wheel selection that allow to weight mutation in favor of changing the mantissa. This can be done as follow:

The source code is largely taken from [38]. Two different version of mutation are necessary because of the different internal representation of `float`s and `double`s.

The procedure has been wrapped in the C++ class CGene which implements the necessary data and functions required for managing data stored inside genes. The algorithm requires the introduction of three `float`s `m_sign_weight`, `m_mantissa_weight`, `m_exp_weight`, which represents, respectively, the weight of bit change in sign bit, mantissa and exponent. A pointer `pRandomGenerator` to an instance of the class "CRandomGenerator" is given as well. This class wraps several random number generators and that pointer is used to obtain a random number in the interval $[0, 1]$ with the function `getRandomDouble()`.

Despite the differences in the different bit lengths of the two data types, the algorithm can be explained as follow:

- a copy of the value to mutate is done and stored for bit manipulations.
- a section (sign, exponent or mantissa) of IEEE 754 representation is randomly selected for bit mutation.
- Through bit mask a check is done to see if all exponent bits are set to 1. In such a case the value could be a NaN or an ∞ so the unmodified floating point value is returned (ensuring that that any output value is not one of these special numbers.)
- a roulette wheel mechanism is applied:
- a random number `random_pick` is generated inside the interval $[0, \texttt{m\_total\_weight}[$, where `m_total_weight` = `m_sign_weight` + `m_mantissa_weight` + `m_exp_weight`. Now, three situation can occurr:
  1. if `random_pick` < `m_sign_weight` then the sign bit is flipped;
  2. else if `m_sign_weight` < `random_pick` < `m_exp_weight` then a randomly chosen bit in the exponent part is flipped;
  3. else if `random_pick` $\geq$ `m_exp_weight` then a randomly chosen bit in the mantissa is flipped.
- The modified bit representation (now corresponding to a new floating value) is decoded and returned.

Common values for `m_exp_weight` and `m_sign_weight` are, respectively, 15% and $2 \div 3\%$. This operator has been implemented in the optimization software developed for this thesis.

tree encoding

A nodes is selected randomly and its content (operator or number) is changed (see Figure 3.3.3).

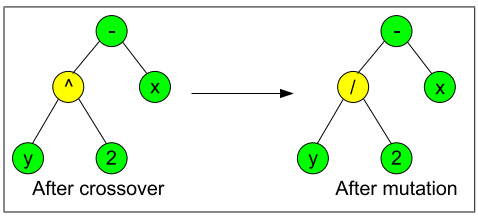


Figure 3.15: Mutation for tree encoding

3.4 On computer generated random numbers

Genetic algorithms largely depend on random numbers. At the same time computers are precise and deterministic machine, so it seems almost impossible that a computer could work as a random number generator. On this paradox researchers and modern philosophers have debated for a long (see [33]). More precisely, when we deal with computer generated sequences we should speak of *pseudo random* sequences, reserving the *random* term to the description of intrinsically random physical (natural) phenomena [28].

Even the human mind, which is surely unpredictable, is not good at generating a set of numbers which are completely unrelated each other.

In addition the random functions present in common languages are, unfortunately, totally inadequate, especially in circumstances where a really large number of random values must be generated. In fact, if those values are repetitive or reappear cyclically, the algorithm is unlikely to produce satisfactory results [38].

Computers perform calculation via algorithms: but truly random numbers cannot be generated in that way because an algorithm is a sequence of operations which, for a given set of parameters, produce an output which is predictable (computed in advance).

Even if, almost theorycally, a random number is a number that is unknown in advance because it can't be predicted, for practical applications *pseudo random* numbers can suffice. These numbers aren't really random, but their sequence is difficult to follow and, in a certain way, they *seem* to behave as they were *really* random.

Researchers have spent a lot of time trying to invent and analyze pseudo-random number generators that were able to produce the most unpredictable sequence of values. Typically, a pseudo-random number generators is initialized with a so-called *seed*. Some operations are performed on that seed and the generated result is a pseudo-random number. That number is then used as the next seed value.

A good pseudo-random generator should be characterized by:

- a long repetition cycle: sooner or later, as the algorithm is applied, the seed will eventually return to its starting value and the output values start repeating themselves. The later as possible it happens, the best it is;
- in order to be not predictabile, the produced numbers should have no particular features (such as patterns) in common;
- it should pass some statistical tests taken from literature.
- theorically, if two generators were coupled to a particular application program, they ought to produce statistically the same results;

Among the available algorithms, the *linear congruential* method is one the most commonly used and is also one of the simplest, since it involves only two operations. Introduced firstly by D. Lehmer in 1954 it can be written as

$$N_{i+1} = aN_i + c(\text{mod } m) \tag{3.7}$$

where N_i is the current seed, N_{i+1} is the produced pseudo-random number, a is a positive integer called *multiplier*, c is nonnegative integer called *increment*. The recurrence (3.7) must eventually repeat itself, with a period that can't be greater than the *modulus* m. All of those parameters define the efficiency and the "randomness" of the method and must be chosen carefully. In fact, it has been shown that the period can be of maximum length only for proper combinations of a, c and m. In the past, many early generator happened to make really bad choices of those parameters, leading to a widespread diffusion, among different systems, of low quality generators. Moreover, with some unfortunate parameter choices, successions like (3.7) can produce low order bits that are not random at all.

When $c = 0$ the recurrence (3.7) is also known as *multiplicative linear congruental* method.

This approach (with only little modifications) is used also in the ANSI-C implementation of `rand` and `srand` functions:

```
static unsigned long next = 1;

int rand(void) {
    next = next * 1103515245 + 12345;
    return ((unsigned int) (next / 65536UL) % 0x32767UL);
}

void srand(unsigned int seed) {
    next = seed;
}
```

which correspond to (3.7) with $a = 1103515245$, c=12345, and ,=2^{32}. In many implementations it provides value that lie only between 0 and 32767 (inclusive). So only few thousands values are available before repeating itself, an inadequate number for genetic algorithms. Other limitations arise from a software engineering point of view:

- it is necessary to explicitly call `srand`, otherwise in every execution of the program the same sequence is generated;
- both `rand` and `srand` share the same global seed, even if they are two different functions;

- since seed is unique only one sequence is possible in a program and it isn't possible to have several pseudo-random generator in the same program;
- the ANSI `rand` function returns only values between 0 and `RAND_MAX` [10]. This is a limitation because often the values are required between different intervals. For example $[1, 10[$ or $[0.0, 1.0[$.
- a templatized class could be useful since it could produce pseudo-random number for every type, not only for `long`s.
- some compilers use some byte-swapping tricks in function `rand`, in order to increase "randomness"[11]. but it has been proven that it ruins the generator and reduces the period of repetition.

Park & Miller [53] suggest parameters for (3.7) (see Table 3.3). In par-

	a	m	c
32 bit unsigned (long)	16807 or 42871 or 69621	2147483647	0
16 bit	171	30269	0

Table 3.3: suggested values for a, m, c in (3.7)

ticular they have proved that, if a and m are chosen carefully, the form

$$N_{i+1} = N_i \cdot c (\text{mod } m) \tag{3.8}$$

can be as good as the more general form (3.7) with $c \neq 0$.

A generator in the form (3.8) is said to be *multiplicative linear congruential*.

According to Park & Miller a good compromise is the so-called *Minimal Standard*[12] which is defined with the parameters

$$a = 7^5 = 16807 \tag{3.9}$$

$$m = 2^{31} - 1 = 2147483647 \tag{3.10}$$

One problem that can be encountered with the sequences seen in (3.7) or (3.8) is that they can suffer from sequences of repetitive values (for example, certain large values may always be followed by very small values). Such problems can be avoided by using the same generator to randomize itself. The basic idea is mixing up the generated values so that they dont appear in the usual sequence, thus avoiding any correlations or predictable sequences. This is the case of the technique called *shuffle*:

[10] According to ISO/IEC 9899:1999-**C99**, verb RAND_MAX must be at last 32767, but its actual value can vary depending on implementations.

[11] Linear congruential generators have low-order bits much less random than their high-order bits and, if possible, only the latter should be used.

[12] *Minimal Standard* was firstly introduced by Lewis, Goldman and Miller in 1969.

- an array is created and loaded with the first few generated values;
- subsequent invocations of the algorithm are used to generate a random index into that array;
- the corresponding indexed value is returned and it is replaced in the array with another random value.

Over the years several modifications have been introduced to overcome the arithmetic limitation of old computers (see *Schrage's method* in [37] and [50]) but neither of them has been proven to be efficient in general.

A more sophisticated approach is based on L'Ecuyer's algorithm. Better results could also been obtained by L'Ecuyer's approach, consisting in combining two different sequences with different periods [51]. This algorithm uses an approximate factorization and implement a procedure that aim to "shuffle" each result in order to reduce correlation in low-order bits. For a practical implementation see [38].

All the algorithms seen so far aim produce *Uniform deviates*. Uniform deviates are just random numbers that lie within a specified range, typically 0.0 to 1.0 for floating-point numbers, or 0 to 2^{32} or 2^{64} for integers. Within the range they can be thought as think "random numbers", in the sense that any one number is just as likely as any other. In other words, uniform deviates are different from other sorts of random numbers (such as numbers drawn from a normal[13] distribution with specified mean and standard deviation).

The state of the art for generating uniform deviates has advanced considerably in the last decade and now begins to resemble a mature field [29]. In particular the availability of unsigned 64-bit arithmetic in C and C++ and the standardization of programming languages and integer arithmetic have bring to this field big benefits. The main problem is that many techniques that now are inferior and outdated are currently employed, because the same were considered good only a few years ago. Recent studies (Press and others, 2007, see [29]) have shown that following generators shouldn't be used:

- generators based on a *linear congruential generator* (LCG) or a *multiplicative linear congruential generator* (MLCG);
- generators whose period is less than $2^64 \approx 2 \times 10^{19}$ or, even worse, unknown;
- generators that warns that their lower bits are completely random (they were good some time ago, but now are to be considered as obsolete);

[13] i.e. Gaussian.

- generators based on C and C++ standard functions `rand` and `srand` because of their poor implementation;
- overengineered generators (i.e with very high periods, or that require complex arithmetic and/or logical operations, and so on) because they are wasteful of resources.

The same authors quote:

> An acceptable random generator must combine at least two (ideally, unrelated) methods. The methods combined should evolve independently and share no state. The combination should be by simple operations that do not produce results less random than their operands [29].

and suggest several kind of high quality pseudo-random generator both for 64-bit unsigned integers and double precision floats.

Nowadays many good algorithm for pseudo-random generator exist. A list can be found in [24] and [52].

From this discussion it can be seen that the choice of a random generator is very complex and requires specialistic knowledge. In order to have the possibility of trying different generators it has been written a C++ class which implements all the functionality which are needed for a generator used in genetic algorithms. The advantage of this approach is that this class wraps all the implemented techniques maintaining the same public interface. In such a way it appears, to the optimization module, totally transparent and the human operator is free to change, modify or introduce new generators easily.

Nowadays many generators are available, but from a practical point of view, it is in the writer opinion that algorithm with few and fast operations (eventually using registers or first level cache) should be preferred to the others. If the generator requires to much resources or it is too slow it can reduce the efficiency and the speed of the genetic algorithm which uses it. In such a case an *hardware* random generator could be take into account as well.

Hardware random generators are independent random sources that are based on the observation of some physical process. As an example they could be built by measuring:

- a random noise on an electronic circuit, such as thermal noise;
- the nuclear decay radiation source detected by a Geiger counter;
- the atmospheric noise as detected by a radio receiver;

and so on.

Chapter 4

Parallel computing

Since genetic algorithm require a lot of repetitive (and independent) high-cost computation, it has been implemented with parallelism in the computer program that have been developed for the numerical simulations. This chapter illustrates some topics on parallel computing that must be taken in account in designing a parallel program.

4.1 Introduction

After few decades in which parallel computing has been considered a very important computer research subject, the long awaited breakthrough towards mainstream computing has delayed and has not yet completely materialized.

One of the most important reason of these delay is maybe due to the fact that parallel computing add a strong layer of complexity when compared to traditional sequential programming. For that even if parallel hardware has over the years shown continuous sign of progress and improvements (in terms of availability, increase of power, cost reduction and so on) parallel software has always lagged behind and fallen short of expectations, making it difficult to actually use the hardware in a profitable manner. Is is also know that programming parallel computers is in general more difficult, error prone and time-consuming than programming sequential computers.[11].

As will be clearer in the next sections the term *parallel* refers to both hardware (a set of computers or processor that can execute programs in parallel) and software (an algorithm formulated as concurrently running processes or tasks). At the hardware level, parallel processing can be implemented within the confines of a single machine or can be implemented within a wider system of connected machines such as a Wide Area Network (WAN), or indeed even the Internet. The most important purpose of introducing parallelism is the achievement of speedup[1]. Usually program-

[1] i.e. to decrease execution wall clock time, however in order to accomplish this, more CPU time is required: for example, a parallel code that runs in 1 hour on 8 processors

mers turn to parallel programming to create programs that are equivalent in functionality to a sequential version, but which take less time to run. But parallelism should not be taken as a sort of license to use less efficient algorithms. Morever for short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation because of he overhead costs associated with setting up the parallel environment (task creation, communications and task termination can comprise a significant portion of the total execution time for short runs).

In the ideal case, the speedup should be linear, in practice there are a number of pitfalls and problems associated with parallel programming, which very often make real speedup often much lower. In fact many other factors should be take into account such as the intrinsic algorithmic limits and the issues related to communication and synchronization.

Moreover the conceptual programming difficulties that make more difficult for a parallel program to be correctly written if compared to traditional (i.e. sequential) programming.

In traditional programming the approach that is used to solve a problem consists in the break of it in a discrete set of simple operations which are performed sequentially until the reaching of the solution. In this context the software is developed thinking to a single computer with a single *central processor unit (CPU)*. The problem is subdivided in a set of simple instructions which are performed sequentially[2]. In a single instant t_i only an instruction can be executed, since each of them requires, to be executed, that the previous one is finished (seeFigure 4.1).

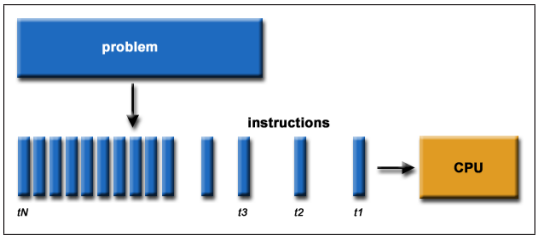


Figure 4.1: Schematic of a serial program

The way of thinking in parallel programming is very different. The problem we want to solve is subdivided in discrete parts that can be solved *concurrently*[3] in different CPUs. Each part is then subdivided in a set of instructions. At the same time t_i instructions from each part are executed on the CPUs (seeFigure 4.1). This kind of approach requires the availability of multiple CPUs. This requirement can be satisfied in different ways:

actually uses 8 hours of CPU time.

[2]in a precise order, from the first to the last, one after another

[3]At the same time

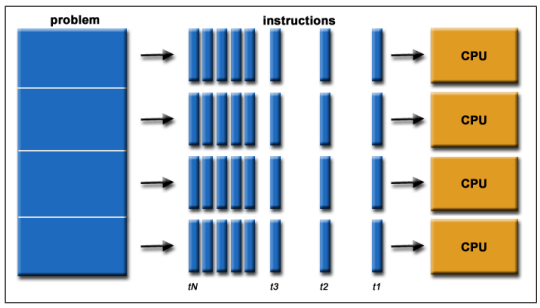


Figure 4.2: Schematic of a parallel program

- With a single computer running with multiple CPUs[4]
- With a set of computers connected in a network [5] (LAN, WAN)
- With a combination of both previous methods.

Anyway parallel programming can largely reduce computation time but it also requires that the problem to be solved must be efficiently *parallelizable*, so this kind of approach can't be employed in any case, but at the same time it can be often adapted to study complicated phenomena of the real world where there are many complex, interrelated events happening at the same time, yet within a sequence. For that it's commonly adopted for numerical simulations of the more complex systems (weather and climate evolution, chemical and nuclear reactions, biological systems as human genome, geological and seismic activity, mechanical devices, electronic circuits, manufacturing processes and so on) and where it's necessary to process a large amount of data (web engines, advanced graphics, virtual reality, ...). It has be stated that *parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time*.

The main advantages are:

- The reduction of computing time.
- Can be used for large problems.
- Can use remote resources (LANs, WANs or in the internet) if local are not sufficient.
- Can be economically advantageous since a cluster of cheap computers can be less expensive than a mainframe with the same computation power (cost savings).

[4] nowadays it is common even in cheap consumer computers: the actual Intel©and Amd©production for personal computers and laptop includes multicore (2-4-8 CPUs) tecnology.

[5] This is also known as *distributed computing*.

- From the point of view of hardware manifacturers there is a cost reduction due to the fact that it is really expensive to make a single processor faster if compared to the cost of using a larger number of moderately fast commodity processors to achieve the same performance.
- When there is the need of manage a large amount of data, can overcome the limits of memory of single computers.

4.2 Computer architectures

Before the introduction of parallel computing *virtually* all computers that appeared over the years were essentially based on the so-called *von Neumann*[6] computer model. This model is based on a so-called *stored-program* concept which means that the CPU executes a stored progran which specifies a sequence of read and write operations on the memory (see Figure 4.2). In more detail, its main features are the followings:

- *Program instructions* are coded data which tell the computer which operations to do.
- The single word *Data* is simply the information to be used by the program.
- Both *data* and *program instructions* are stored in memory.
- The CPU reads instructions and data from memory, decodes the instructions and sequentially perform them.

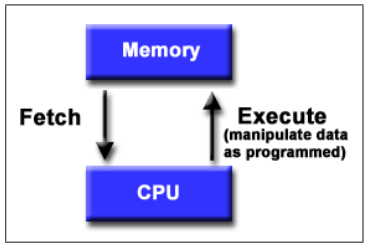


Figure 4.3: The von Neumann model

During the years with the introduction of parallel programming, new computer models have been introduced with multiple CPUs and all of them are schematically collected inside the *Flynn's Classical Taxonomy* . This classification is based upon the two possible states (*single* or *multiple*) which characterize *Data* and *Program* in multi-processor computer architectures (see Table 4.2).

[6]From the name of the Hungarian American mathematician and computer scientist John von Neumann (1903-1957) .

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Table 4.1: Flynn's Classical Taxonomy

4.2.1 Single Instruction, Single Data (SISD)

This is the oldest and until recently, the most common form of computers: includes most PCs, single CPU workstations and mainfranes (see Figure 4.2.1).

- Not parallel but serial computer.
- Single Instructions: only one instruction stream is being acted on by the CPU during any clock cicle.
- Single Data: only one data stream is being used as input by the CPU during any clock cicle.
- The execution is deterministic (no parallel operations).

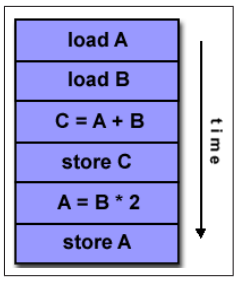


Figure 4.4: Single Instruction, Single Data (SISD)

4.2.2 Single Instruction, Multiple Data (SIMD)

This is a type of parallel computer which is best suited for problems charecterized by a high degree of regularity, such as image processing (see Figure 4.2.2). It comes in two variety: *Processor Arrays* (Connection Machine CM-2, Maspar MP-1, MP-2) and *Vector Pipelines*(IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820).

- Parallel computer.

- Single Instructions: all processing units execute the same instruction at any given clock cicle.
- Multiple Data: all processing units can operate on a different data element.
- The execution is deterministic and synchronous (with locksteps).

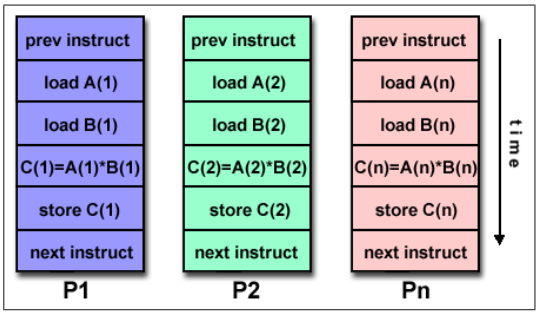


Figure 4.5: Single Instruction, Multiple Data (SIMD)

4.2.3 Multiple Instruction, Single Data (MISD)

Only few real examples exist of this parallel computer. A possible application is the implementation of multiple cryptographic algorithm which can be used to crack a single coded message, or for multiple frequency filters operating on a single signal stream (see Figure 4.2.3).

- Parallel computer.
- Single Data: a single data stream is fed into multiple processing units.
- Multiple Instructions: each processing units operates on the data indipendentely.
- The execution is deterministic and synchronous (with locksteps).

4.2.4 Multiple Instruction, Multiple Data (MIMD)

Most modern computers fall into this category. Currently it is the most common type of parallel computer it can be found in modern supercomputer, networked parallel computer "grids", multiprocessors *SMP* – including most recent PCs and laptops(see Figure 4.2.4).

- parallel computer

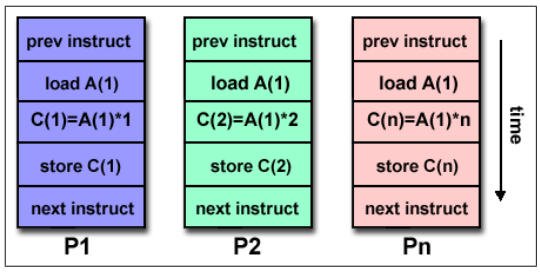


Figure 4.6: Multiple Instruction, Single Data (MISD)

- Multiple Instructions: each processing units may be exectuing a different instruction stream.
- Multiple Data: each processing units may be working with a different data stream.
- The execution can be synchronous or asynchronous, deterministic or non deterministic.

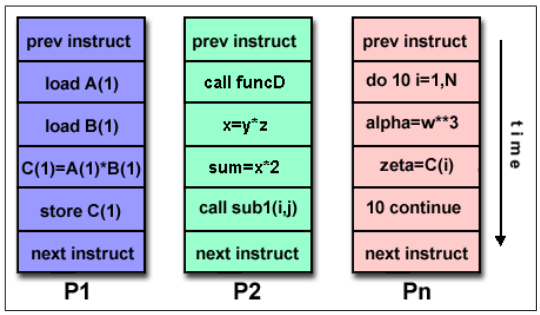


Figure 4.7: Multiple Instruction, Multiple Data (MIMD)

4.3 Parallel computer memory architecture

The most common architectures are *shared memory* and *distributed memory*.

4.3.1 Shared memory

All CPUs access all memory as a *whole* global address space. All processors can operate independently but all of them share the same memory resources and all changes in memory location done by one processors ar visible to all others CPUs (see Figure 4.3.1). A further subdivision could be done between the type of memory access:

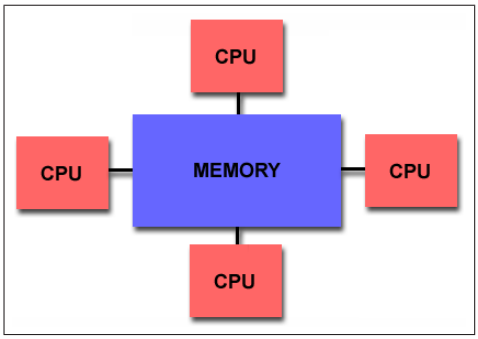


Figure 4.8: Shared memory architecture

Uniform Memory Access (UMA) It's common in Symmetric Multiprocessors (SMP) machines with identical processors, each of them has equal access and access time to memory. It's also known as *Cache coherent UMA*[7].

Non-Uniform Memory Access (NUMA) It's often made by physically linking two or more SMPs. One SMP can directly access to the memory of another SMP, but not all processors share the same access time to all memories[8]. The main disadvantage is the lack of scalability between memory and CPUs: adding more CPUs can geometrically increase the traffic between shared memory and CPUs and for cache coherent systems, can geometrically increase the traffic in the cache-memory management path.

4.3.2 Distributed memory

In this case each processor has its own local memory, so it operates independently. Memory addresses in one processor do not map to another processor, so there is the absence of the concept of global address space across all processors. At the same time is not possible to apply the concept of *cache coherence*[9]. Distributed memory systems require a communication network[10] to work in order to connect the memory of the single CPUs (see Figure 4.3.2). With this kind of approach memory scales well along with the number of processors[11]. Each processor access rapidly to its own memory

[7]When one processor update a location in shared memory, all the others processors know about the update. This is obtained at the hardware level.

[8]Memory access across link is slower.

[9]Because the changes that each professor makes to its local memory has no effect on the memory of the others.

[10]Various techniques can be adopted. Sometimes even a common Ethernet (Gigabit) can be enough.

[11]Increasing the number of CPUs let increase the size of memory proportionally.

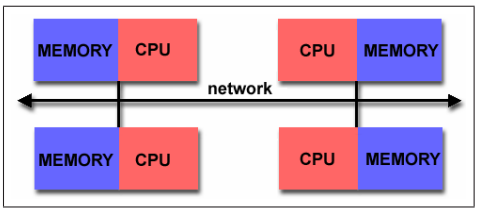


Figure 4.9: Distributed memory architecture

but it has *Non-uniform memory access (NUMA)* times. One of the most important advantages is the cost effectiveness since this model makes possible the use of commodity, off-the-shelf hardware (processors and networking). Against, it could be difficult to map existing data structures, based on global memory, to distributed memory architecture.

4.3.3 Hybrid shared-distributed memory

A few of the main features of the two previous approach are shown in Table 4.2.

The advantages of the two architecures seen until now can be combined in an *hibrid* architecture. The largest and fastest computers in the world are today based on this approach, which contains both shared and distributed components (see Figure 4.3.3):

The shared memory component usually consists in a *cache coherent* SMP machine. Processors on a given SMP machine can address that machine's memory as global.

The distributed memory component is made by the networking of multiple SMPs. Network data communications are required to move data from one SMP to another (as fast as possible) because each SMP knows only about its own memory, not the memory on another SMP.

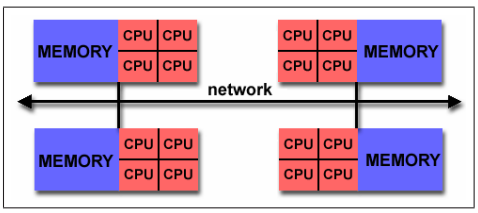


Figure 4.10: Hybrid distributed-shared memory architecture

Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3 Maspar IBM SP2
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	many 100s ISVs

Table 4.2: Comparison of Shared and Distributed Memory Architectures, taken from [10].

4.4 Parallel programming models

Parallel programming models exist as an abstraction above the hardware and memory architectures we have seen before. The more common models are:

- Shared Memory.
- Threads.
- Message Passing.
- Data Parallel.
- Hybrid.

It's important to note that this classification is totally independent from the hardware in the sense that each of these model are not specific to a particular type of machine or a specific memory models, but can be applied on any underlying hardware. As a consequence the choice of a *programming model* instead of another depends on personal taste and on what is actually available since there is not a model which is absolutely better than the others.

4.4.1 Shared memory

The *tasks*[12] share a common address space, which they can read and write asynchronously since the access to the shared memory is managed by mechanism such as *lock* or *semaphore*. Actually implementation of this model doesn't exist any more.

4.4.2 Threads

According to this model a single *process* is allowed to have multiple, concurrent execution path. A *thread*'s[13] work can be ideally seen as a subroutine inside a main program *a.out* (see Figure 4.4.2). This approach requires that while threads can be created and closed, the main program remain alive since it has to provide the shared resources and it needs *synchronization constructs*[14]. This kind of implementation is common in shared memory architectures and operating systems and usually consist in a library of subroutines that are called from the parallel code and a set of compiler directives embedded in the source code. The most famous are

[12] A Task is a logical discrete section of a computational work. Typically is a program or a set of instruction executed by one processor.

[13] The thread is a logical concept, it's a minimal executable portion of code.

[14] To ensure that one thread is not updating the same global address at any time.

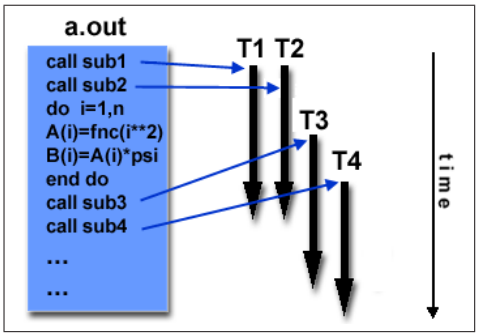


Figure 4.11: Threads model

POSIX Threads : commonly referred as *Pthreads* is library based, requires parallel code and available in C language only. It is a very explicit[15] implementation and so requires a lot of attention to details and care by the programmer.

OpenMP : can use serial code and is available in C/C++ and Fortran. Is portable and multi-platform, including Unix, Linux, Windows and Mac Os X. It can be easy and simple to use.

Also Microsoft© has its own implementation for threads, but is not related to either the UNIX POSIX standard or OpenMP.

A tipical *OpenMP* application apply the so-called *fork* and *join* approach (see Figure 4.4.2). The conceptual model of typical *OpenMP* application apply the so-called *fork* and *join* approach which is characterized by a set of alternate serial and parallel executions (see Figure 4.4.2).

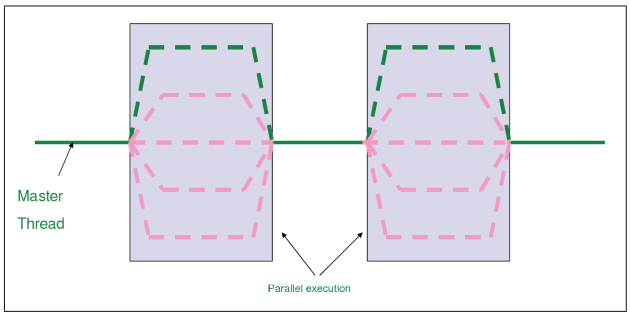


Figure 4.12: Threads model

1. At the beginning and the end the is a serial process *master thread*.

[15] In the sense of low level.

2. From time to time several parallel process *threads* begin and execute indipendently *fork*.

3. After the parallel part, the master thread waits for all the threads still executing and finally *join* them.

4. After joining, the serial process *master thread* continues.

4.4.3 Message Passing

The implementations of this model consist in a set of library which must be embedded in the source code, alle the parallelism must be defined by the programmer. Historically, starting from '80s several have been introduced several libraries of this kind, but each of them was very different from the others and all of them and unfortunatley all of them showed a total lack of portability. In the '90s, with the introduction of MPI and MPI-2 things radically changed. Nowadays MPI is the "de facto" industry standard for message passing.

In this model, during computation, a set of tasks use own local memory. Multiple task can reside in the same physical machine or could be divided across multiple machines. Tasks exchange data by sending and receiveing messages, this kind of data transfers requires collaboration between process[16](see Figure 4.4.3).

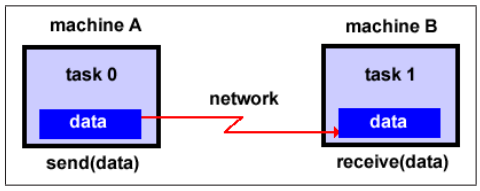


Figure 4.13: Message passing model

4.4.4 Data Parallel

In this model most of the parallel work is focused on performing operations on a data set. Usually the data is a common structure, such an array. A set of tasks works on the same data structure, but each of them on a different partition of the data. All the task perform "exactly" the same operations, but in different partitions (see Figure 4.4.4).There are implementations available in Fortran 90 and 95.

[16]For example, a *send* operation must have a matching *receive* operation.

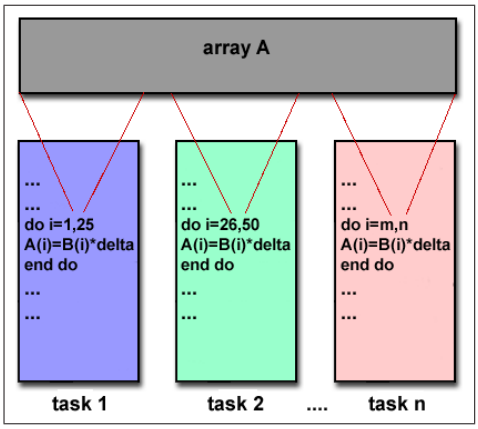


Figure 4.14: Data parallel model

4.4.5 Other models

Other models exist such as *hybrid*, *single program multiple data (SPMD)*, *multiple program multiple data (MPMD)*.

Hybrid

Two or more models are combined. For example:

- MPI + threads.
- MPI + OpenMP.
- Combination of data parallel with message passing[17]

Single Program Multiple Data (SPMD)

This is an high level programming model that can be built upon a combination of the parallel programming models shown before. A single program (a.out) is executed by all task simultaneously. At any moment in time, task can be executing the same or different instruction within the same program. Tasks may use different data. (see Figure 4.4.5).

Multiple Program Multiple Data (MPMD)

Like *SMPD* also *multiple program multiple data (MPMD)* is an high level programming model that can be built upon a combination of the parallel programming models shown before. Typically there are multiple programs

[17] For example, data parallel implementations on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

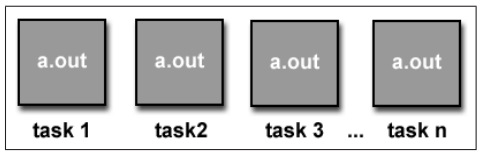


Figure 4.15: Single Program Multiple Data (SPMD)

(a.out, b.out, c.out, ...). Each task can be executing the same or different program as other tasks and all tasks may use different data (see Figure 4.4.5).

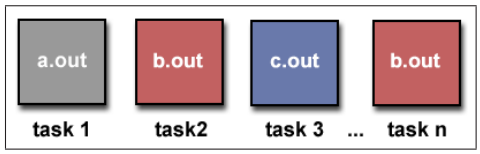


Figure 4.16: Multiple Program Multiple Data (MPMD)

4.5 The design of parallel programs

Designing a parallel program is undoubtedly a difficult task, which requires a lot of "manual" work by the programmer. It's time consuming and often an *error-prone* activity. For this reason have been developed some tools[18] which can help the programmer to parallelize existing serial code. Anyway this kind of tools fail for complex program and if they works, the performance are usually lower than a manual approach and the resulting program is not very flexible. For example, a common strategy used in these tools try to find *loops* inside the source code to parallelize and try to evaluate if a parallelization could really increase the performance of the software.

In fact not always *parallel* means *faster*. Several aspects should be taken in count that could rapidly degrade performance and so make not useful a parallelization of the code. Among the so-called *inhibitors*[19] there are:

- Possible bottlenecks due to intense I/O activity, which are usually slow if compared to the speed of processor and could bring the CPUs to wait for data.
- Internal inefficient algorithms that could be substituted by more efficient ones.

[18] Essentially for Fortran programmers.
[19] Everything that could slow down the code execution.

- If a program was born as a serial program, it could have an internal structure that is difficult to adapt to parallel computation, eventually it could be convenient that is entirely rewritten.

For these reasons it's very important to do a precise analysis of the problem that we want to parallelize evaluating, using *profiling* procedures:

- If the problem is parallelizable
- Possible bottlenecks.
- Which are the portions of code that could have benefit[20].
- If the most power-consuming parts can be parallelized.

4.5.1 Analysis of the problem: decomposition

The first think to do is to analyze the problem to solve, in order to see if it can be broken into discrete *chunks* of work that can be distributed to multiple task. This is called *decomposition* or *partitioning*. This can be done essentially in two ways: by *domain* or by *functions*. In one case the focus is on the data manipulated by the computations, in the other is on the computations themselves.

domain decomposition

The data of the problem is divided in different parts, in this way each parallel task works only on a portion of the whole data of the problem (see Figure 4.5.1). For both mono-dimensional and bi-dimensional arrays this

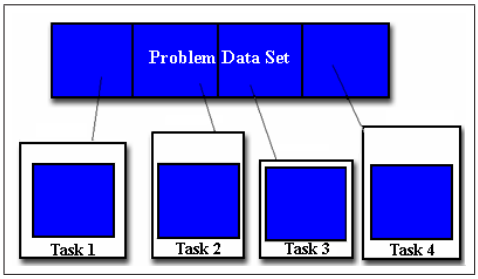


Figure 4.17: Domain decomposition

can be achieved with *block partition* or *cyclic partition* (see Figure 4.5.1).

[20]Majority of scientific and technical programs usually accomplish most of their work in a few places, so only them need to be parallelized. It also has no sense spend time in parallelizing parts which has only a little CPU usage.

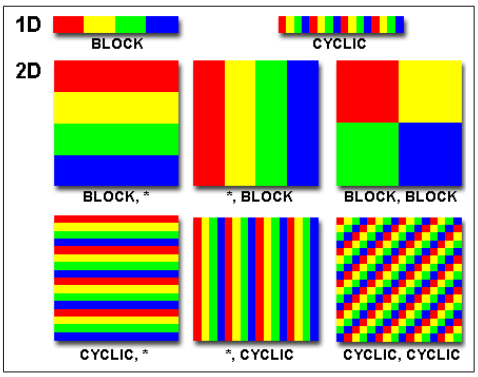


Figure 4.18: mono-dimensional and bi-dimensional domain decomposition

functional decomposition

In this case the attention is focused more on the computation that on the data. The decomposition is guided by the work that must be done. Each task performs a portion of that work (see Figure 4.5.1). This kind of de-

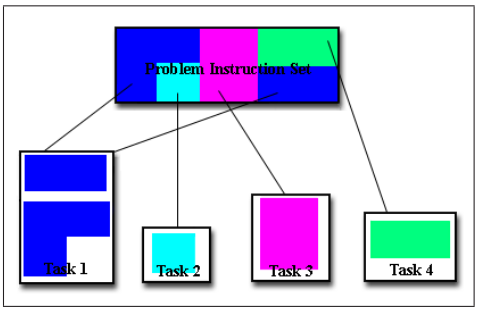


Figure 4.19: Functional decomposition

composition is well suited to problems that can be *naturally* splitted into different tasks. For example:

Ecosystem modeling

Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations Figure 4.5.1).

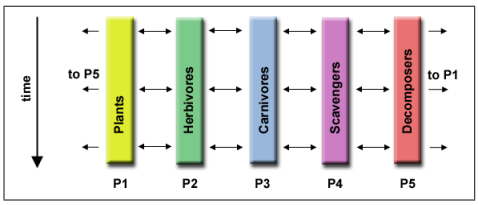


Figure 4.20: Ecosystem modeling

Signal processing

An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy Figure 4.5.1).

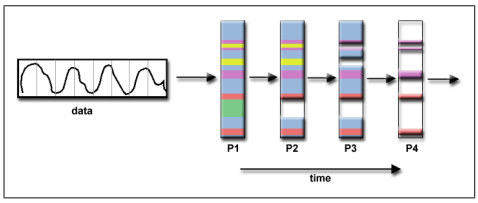


Figure 4.21: Signal processing

Climate modeling

Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation. Each component interacts with the others. Figure 4.5.1).

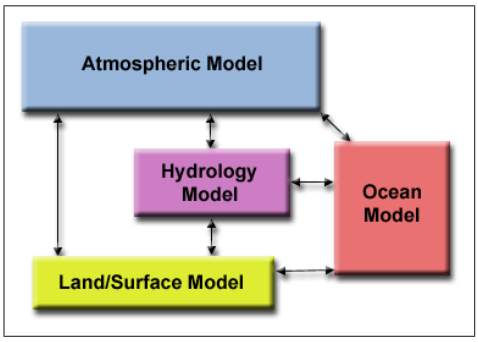
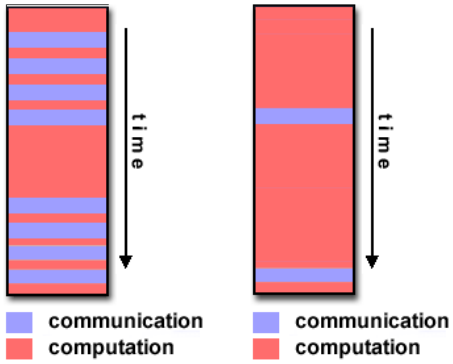


Figure 4.22: Climate modeling

4.6 Topics in parallel programming

A useful parameter that can be employed to classify parallel programs is the *granularity* which is the ratio between *computation* and *communication* time. This can be done because computations are always distinct from communications (see Figure 4.23).



(a) Fine (b) Coarse

Figure 4.23: Fine and coarse granularity

$$\text{granularity} = \frac{\text{computation}}{\text{communication}}$$

fine grain : a relatively small amounts of computational work are done between communications/synchronization ($\frac{computation}{communication}$) is low). This implies high communication overhead and less opportunity for performance enhancement and if granularity is very fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. Usually it is possible to apply load balancing techniques (see Figure 4.6).

coarse grain : a relatively large amounts of computational work are done between communication/synchronization events ($\frac{computation}{communication}$ is high). This usually implies more opportunity for performance increase, but it's harder to load balance efficiently (see Figure 4.6).

Which is the best granularity to pursue depends strongly on the algorithm and the hardware environment in which it runs. In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity, on the other side fine-grain parallelism can help reduce overheads due to load imbalance.

Besides that, in order to obtain an efficient program it's necessary to take in account several key topics:

- Reduction of bottlenecks. For example, a typical bottleneck are communications. In order to obtain the maximum efficency is necessary to consider the *latency*[21] of the communication system and the *bandwidth*[22] available to the system. Sending many small messages can cause latency to dominate communication overheads. Often it is better to package small messages into a larger message, increasing the bandwidth component.

- If possible, prefer asynchronous communications to synchronous, because are faster. This is due to the fact that all synchronous communications add an amount of overhead. Typical methods of synchronization are *barriers*[23], *locks or semaphores*[24] and *handshaking*[25].

- Reduction of *dependencies*. A *dependence* exists between program statements when the order of statement execution affects the results of the program. This inhibits parallelism. A *data dependence* results from multiple use of the same locations in storage by different tasks.

- Using tecniques of *load balancing*. The work among the task should be distributed in a manner that all tasks are kept busy all of the time as possible, in order to minimize the *idle* time. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance Figure 4.6. It's important to equally partition the work each task receives and if working with a /emphcluster of computers, the partition should take in account the different performance of each computer. Another useful tecniques is the use of dynamic work assignement to reduce, at /emphruntime, the load unbalancing as they occur during execution.

- In common situation prefer *coarse* to *fine* granularity.

A special attention should also be given to I/O. Usually I/O is a bottleneck for the program specially if it must be conducted over the network (over

[21] Usually expressed in μs represents the time that it takes to send a minimal packet (of 0 bytes) from one point to another.

[22] Usually expressed in Mb/s, represents the maximum amount of data that can be sent per unit of time.

[23] Each task perform its work until it reaches the *barries* and the stops (or "blocks"). When the last task reaches the barrier, all tasks are synchronized. Usually all tasks are involved.

[24] Only one task at a time may use (own) the /emphlock or/emphsemaphore or /emphflag. The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code. Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.

[25] It is used in communication operations where some form of coordination is required with all the tasks participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

a NFS/footnoteNetwork file system.), and parallel I/O systems are still immature today. In an environment where all tasks see the same filespace, write operations could result in file overwriting if for each tasks' input/output file(s) have not been created unique filenames. In the meanwhile read operations are affected by the fileserver's ability to handle multiple read requests at the same time.

A great role is also take by *load balancing* which is the practice of distributing work among tasks so that all tasks are kept busy all of the time. From an apposite point of view it can be considered as a minimization of task idle time. Load balancing is really important to parallel programs for performance reasons. As an example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance as can be seen from Figure 4.6).

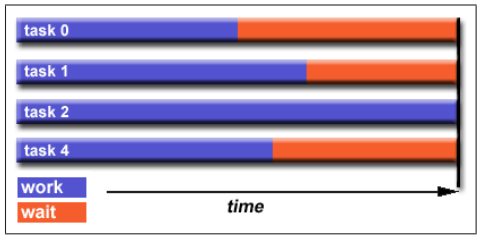


Figure 4.24: Load balancing

4.7 Limits of Parallel programming

The well known /emphAmdahl's law define the limits of parallel programming.

4.7.1 Amdahl's law

Amdahl's Law states that potential program speedup is defined by the fraction of code P that can be parallelized:

$$\mathsf{speedup} = \frac{1}{1-P} \tag{4.1}$$

If no part of the code can be parallelized, $P = 0$ and the $\mathsf{speedup} = 1$ (no speedup). If all of the code can be parallelized, P = 1 and, theorically, $\mathsf{speedup} = \infty$.

As an example if 50% of the code can be parallelized the code will run twice as fast, since $\mathsf{speedup} = 2$.

Introducing in 4.1 the number of processors N performing the parallel fraction of work we have:

N	$P = 50\%$	$P = 90\%$	$P = 99\%$
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90

Table 4.3: Speedup comparisons: N is the number of processors and P is the value of the parallel fraction

$$\mathsf{speedup} = \frac{1}{\frac{P}{N} + S} \tag{4.2}$$

where P and S are, in the given order, the parallel and the serial fraction of code.

From (4.2) appears clear that there are limits to the scalability of parallelism regarding to both the number of processors N and the value of the parallel fraction P. Some results have been summed up in Table 4.7.1 for different values of N and P, where is possible to see that for low values of parallel fraction the benefit that come from the addition of more processors is low and, obviusly, the more parallelizable is the code and more processors are used the more is the gain in term of speed that tends, asymptotically, to 100.

However, certain class of problems demonstrate increased performance by increasing the problem size. Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.

Moreover since parallel applications are much more complex than corresponding serial applications it should be taken in account the cost of this complexity as well. This can be measured, virtually, in programmer time, analyzing the most important parts of a generic software development cycle[26].

Another important aspect is the ability of a parallel program's performance to scale. *Scalabilty* is not easy to predict since it may depends from many factors. For example the algorithm (or the parallel libraries) may have intrinsic limits (adding more resource has no effects) or can be related to the hardware as well [10].

Besides all those difficulties the interest on parallel programming in these days is still very high. A new interesting parallel library is going to be released in this year. It is known as *OpenCL*[27] and is a framework for

[26] They are designing, coding, debugging, tuning and maintenance

[27] It is the acronim of *open computing language*.

writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL provides parallel programming using both task-based and data-based parallelism.

Chapter 5

Numerical simulations

5.1 Introduction

In order to evaluate the performance of the proposed approach a computer program have been developed.

It has been entirely written in C++, compiled with GNU gcc[1] (g++) and tested on a cluster of personal computers running Gentoo[2] Linux as operating system. The cluster is accessible via Internet using the `ssh` protocol.

The choice on Linux is due to many reasons:

- it is a small and fast operating system that can run even in old computers with low resources, allowing building cluster with commodity hardware;
- it is really stable and efficient in networking environments;
- it provides free compilers for almost programming languages;
- is well suited for distributed computing;
- it's free;

Moreover Gentoo distribution enhance some key features that are very important in high performance computing. Is highly modular and flexible. The installation can be personalized in every aspect. Only necessary tools have been installed, thus avoiding to waste precious and rare resources for superfluous tasks (such as graphical user interface overhead, unused services and so on). Since the sources are compiled directly by the user during installation (no binaries are given), it can can be deeply optimized for the user's machines. This feature is useful when the cluster deals with different architectures. Finally it is easy to maintain and upgrade.

[1] http://www.gcc.org
[2] http://www.gentoo.org

5.2 The software implementation

From the beginning, the software has been designed with some key points in mind. The first is that it could be essentially addressed to optimization of truss structures. Most optimization methods deal with continuum structures, and apply many techniques that can not directly applied to structures made with truss elements because of the different nature of problem they want to deal with.

For simplicity the attention is focused on linear elastic structure. Maybe an extension to non linear problem could be done in the future, but it is firstly necessary to test and study simpler problems in order to evaluate the effects of the variants that can be introduced in an approach based on genetic algorithms. Many parameters need to be calibrated and nonlinearity would increase the complexity of the problem.

The basic idea is the application of genetic algorithms to structural problem, for that it has been necessary to develop a module for the finite element analysis of 3D truss structures. This module has been written in ANSI C++, using data structures and algorithms from the standard library. During the development an XML support has been introduced for input and output files that has been very useful, also for debugging and data reading purpose.

XML is the acronym of *eXtensible Markup Language*[3]: it is an extensible language that allows the programmers to define the mark-up elements. It allows to collect data in well organized tree structure that can be written and parsed efficiently. Besides that, stylesheets can be associated so that XML files can be read inside web-browsers in very simple and clear way. This feature must not be underestimated, since it makes input/output files much easier to read and modify (specially in the case of complex structures). On the other hand this benefit doesn't come at no cost. XML files are usually much larger than corresponding binary files (especially if including tabular data), but at a final evaluation the advantages (for example they can be updated incrementally every time is necessary) are much more than the cost of size. Moreover, when a XML file is properly structured, the parsing can be done in a very fast and efficient way. For this purpose the `libxml2`[4] library has been used as a SAX and DOM[5] parser.

[3] http://www.w3.org

[4] See `http://xmlsoft.org/`

[5] SAX parsers have certain benefits over DOM-style parsers. The quantity of memory that a SAX parser must use in order to function is typically much smaller than that of a DOM parser. DOM parsers must have the entire tree in memory before any processing can begin, so the amount of memory used by a DOM parser depends entirely on the size of the input data. The memory footprint of a SAX parser, by contrast, is based only on the maximum depth of the XML file (the maximum depth of the XML tree) and the maximum data stored in XML attributes on a single XML element. Both of these are always smaller than the size of the parsed tree itself. Besides, processing documents can often be faster than DOM-style parsers. Memory allocation takes time, so the larger memory footprint

In order to optimize several input data files must be careful prepared.

The file parameters.xml take in account the various setting of the genetic algorithms. In particular it allows to set the following parameters:

- Which pseudo-random number generator to use. Both linear congruential and multiplicative linear congruential implementation are available and the constants a, m, c must be introduced as well (for detail see section 3.4 at page 87).
- Population size s (see 3.2.1).
- Selection process. The choice can be done between
  - tournament: tournament size s is also required (see section 3.2.2);
  - ranking ;
  - roulette wheel: it should also be specified if *fitness scaling* must be done;

  (for details see section 3.2.2 at page 68).
- Elitism. The size of elitism s_e must be given, where s_e is the number of best individuals that must survive. For $s_e = 0$ no elitism is performed (see 3.2.2).
- Probability for crossover reproduction (in %).
- Probability for mutation (in %).
- Mutation for floating point values: the different weight for sign, mantissa or exponent mutation must be specified (see section 3.3.3 at page 85).
- Stop criterium. The optimization procedure can stop when
  - a fixed number gen_{max} of generation has been created;
  - said $delta_{fitness}$ the difference between maximum fitness among two generation, the optimization procedure can stop when after a number n of consecutive generations $delta_{fitness} < dfit$, where $dfit$ is a fixed value.
  - after n of consecutive generations the maximum fitness doesn't grow any more.

The description of the structure to optimize is included in another input file called structure.xml containing

---

of the DOM is also a performance issue. (taken from `wikipedia`)

- The list of materials. For each material must be specified the following data:
  - a unique ID (it is an integer value that let each element of the list be univocally identified by the program);
  - modulus of elasticity E;
  - the weight per unit of volume γ;
- The list of section. It contains
  - a unique ID (it is an integer value that let each element of the list be univocally identified by the program);
  - the cross section area A;
  - the moments of inertia in both principal directions J_1 and J_2 (not used);
  - a reference to the unique ID of the material;
- The list of nodes. It must include
  - a unique ID (integer value);
  - the position in global reference system:x, y and z;
- The list of node restraints. In general the nodes are considered free. If a node is not free, then its restraints are assigned with this list which contains (for each node that has almost one degree of freedom fixed)
  - the nodal unique ID (integer value);
  - the restraint for translation in x direction: it can be 0(free) or 1(fixed);
  - the restraint for translation in y direction: it can be 0 or 1;
  - the restraint for translation in z direction: it can be 0 or 1;
  - the restraint for rotation about the x axis: it can be 0 or 1 (not used);
  - the restraint for rotation about the y axis: it can be 0 or 1 (not used);
  - the restraint for rotation about the z axis: it can be 0 or 1 (not used);
- The list of elements. An element is generated from one node to another. A reference node is used to define the axis rotation in the three-dimensional space. For each element must be specified:
  - the element unique ID (integer value);

  - the section unique ID (integer value);
  - the unique ID of the FROM node;
  - the unique ID of the TO node;
  - the unique ID of the REFERENCE node (not used).

The loads acting on the structure are described in another input file called loads.xml that includes[6]:

- the list of the nodal loads. Each entry must declare the load acting on every loaded node:
  - the nodal unique ID (integer value);
  - the force acting in x direction;
  - the force acting in y direction;
  - the force acting in z direction;
  - the moment acting around the x axis (not used);
  - the moment acting around the y axis (not used);
  - the moment acting around the z axis (not used).

Please note that some of those input values are not necessary and are marked as *not used*. They should be seen as a sort of placeholders and have been included for the sake of clearness since they shows how the input file could be upgraded in future developments, being XML easy to extend.

The design variables must be chosen. The genetic algorithm ignore the nature of the problem. It works on the values of the design variables and evaluates the results at every iteration. A file must be given in order to specify which parameters must have to be considered as design variables. Among them we could have nodal coordinates, cross sections index (or cross section areas), and so on. Like before, these data are given within a file (designvariables.xml) that contains both the list of entities which must be considered as variables and then the values that each variable can assume. For each variable we must give minimum and maximum value and eventually, non admissible values (with $\neq$ operator).

For example if we want to chose as design variables the x coordinate of node number 3 and the section of truss number 2 (we can suppose that in the list of sections there are 10 elements with ID included in the range $1 \div 10$) the file designvariables.xml will contain

- nodal list
  - 3 (*node ID*);
  - x (*design variable*);

[6] for the sake of simplicity we will consider only one load condition

  - minimum value for x;
  - maximum value for x;

- element's list
  - 2 (*truss ID*);
  - section ID *(design variable)*;
  - 1 (minimum value for $sectionID$);
  - 10 (maximum value for $sectionID$);

The introduction of the file designvariables.xml gives the advantage of decoupling the data of the problem from the design variables. So the optimization procedure can be more easily extended to other kind of problems.

5.2.1 The finite element analysis

A program (FEsolver) for finite element analysis of three-dimensional truss structure has been developed implementing a standard 3D truss element formulation (see [12], [36], [46], [59]) which works with XML files. The syntax is

```
int FEsolver( FILE* structure, FILE* loads, FILE* results )
```

where solver, loads and results are three XML files. The first two are input files and correspond, respectively, to structure.xml and loads.xml as seen in section 5.2.

The file results is the output of the solver and contains

- The list of nodes. Each entry specifies:
  - the unique ID of the node (integer value);
  - the nodal displacement v_x in x-direction;
  - the nodal displacement v_y in y-direction;
  - the nodal displacement v_z in z-direction;
- The list of Truss element. Each entry specifies:
  - the unique elemental ID (integer value);
  - the value of axial force;

If for some reasons the analysis can not be done[7], a message description is written in the file and an error code is returned.

[7] For example stiffness matrix not invertible, numerical problems, errors in restraints and so on.

5.2.2 The fitness evaluation

The fitness evaluation depends on the objective function. In general it can be done through a function that reads an input file (the result produced by the FEsolver), extracts the data it is interested to (by using a SAX parser), does some computations and returns the value of fitness.

The syntax is

```
double Fitness( FILE* result )
```

5.3 How does it work ?

A flow chart of how the optimization procedure works can be found in Figure 5.3 and summarized as follow.

The main procedure read the following files:

- parameters.xml
- structure.xml
- loads.xml
- designvariables.xml

then produce s n-dimensional vectors

$$design[i] = \{v_1, v_2, \ldots, v_n\} \qquad i = 0, 1, 2, \ldots, s-1 \tag{5.1}$$

where v_j is the value of the j-th design variable, s is the population size and n is the number of design variables.

After its creation, the vector $design[i]$ is populated with random values v_js that have been produced with respect to the minimum and maximum admissible values for the corresponding design variables as defined in designvariables.xml. With the creation and population of the s organisms the first generation is completed.

Now the input files must be prepared for the finite element analysis and the following fitness evaluation. As can be seen in Figure 5.1, FEsolver requires two input files. So $s + s$ input files must be prepared at each generation. Those files are simply written by copying the files structure.xml and substituting inside the new i-th file, at the proper locations, the values of the variables of the array representing the i-th indivudual of the current generation (see Figure 5.2).

In other words, s copies of structure.xml are respectively merged with the values of the design variables included in $design[i]$ overwriting them in the proper position inside the new xml file.

After that, the file loads.xml must be updated in order to take in account the current values for the self weight nodal loads.

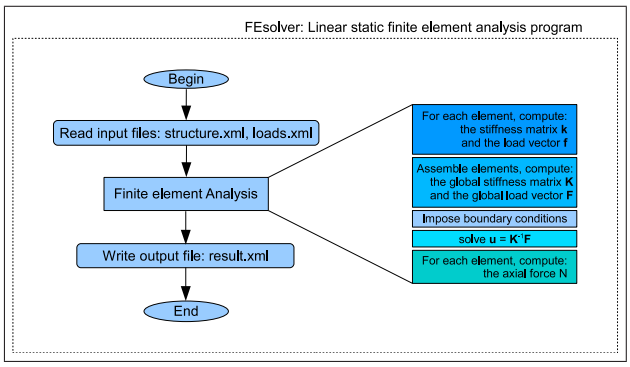


Figure 5.1: schematic for FEsolver program

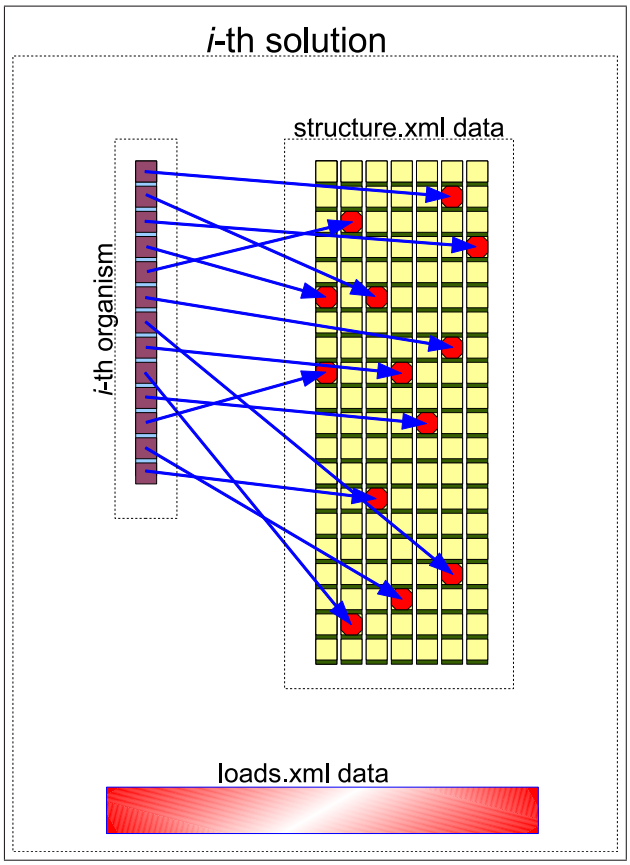


Figure 5.2: i-th solution: mapping of design variables into structure.xml

Typically computer programs tend to become slow when they are subjected to intense I/O activities. Moreover xml files tend to have larger size if compared to binary files which deal with same quantity of data. In order to gain performance (by reducing bottlenecks) thelibxml2 library has been adopted, coming with that a very efficient XML SAX parser.

Optimization cycles (see Figure 5.3) requires a very large number of finite element analysis, but the solution of the equilibrium equations $\mathbf{Ku} = \mathbf{F}$ (see Figure 5.1) are computationally very expensive. This is particularly true for large structures. For this reason, I thought as natural to employ paralleliza-

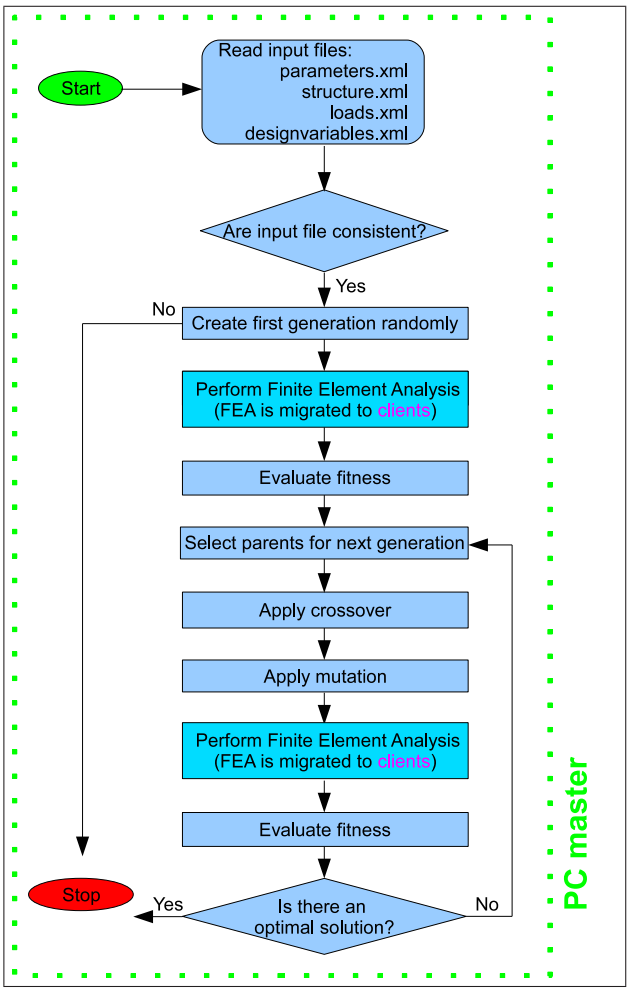


Figure 5.3: schematics of genetic algorithm on the *master* computer

tion in such a context, being all the finite element analysis independent each other.

To obtain a better efficiency the genetic algorithm has been parallelized

as can be seen in Figure 5.4.

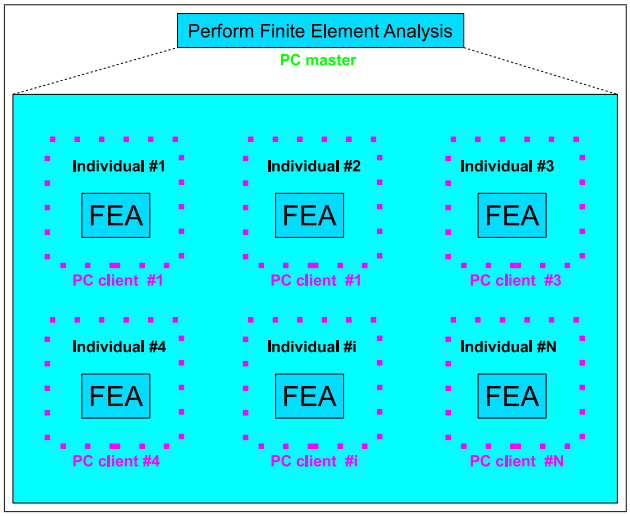


Figure 5.4: schematics of Finite element analysis parallelization on *clients*

This can be obtained building a cluster composed by a variable number of computers, running Linux as operating system and connected each other in a LAN environment. This approach reserve one *master* (or *head*) PC (which runs the main program and manages the whole optimization procedure) and a variable number of *slave* (or *client*) computers that parallel finite element analysis.

From recent literature [31] it can be seen that many solutions are today available to build a cluster cheaply: i.e. using only open source and free software and installing commodity hardware [34]. Moreover a lot of software packages and kernel modifications are available and even cluster installation kits have been recently introduced but, finally, the cluster has been built with a *from scratch* approach: adopting a typical *Beowulf* architecture and using standard kernels (provided by the Gentoo linux distribution) without any additional software. Only MPI[8] library has been installed as parallel library (to allow processing to be shared among clients). Finally some disk space has been shared via the standards Network File System protocol.

NFS has been chosen for its simplicity. Actually it is known that it is not the ideal solution in every situation, because it is not optimized for the types of I/O often needed with many high-performance cluster applications. For

[8]MPI is a communications library that enables parallel programs to be written in C, C++, FORTRAN and many other programming languages. It can be thought of as a standard that specifies a library. Users can write code in C, C++, or FORTRAN using a standard compiler and then link to the MPI library. The library implements a predefined set of function calls to send and receive messages among collaborating processes on the different machines in the cluster. (see http://www.lam-mpi.org/)

some tasks it could be better integrate NFS with other file systems which provides high-performance and are more adapt to a parallel computing, such as *Parallel Virtual File System* (PVFS)[9]. In addition, when dealing with large quantities of data, a *Database Management System* (DBMS) or an hardware hardware upgrade, such as a *network attached storage* (NAS) or a*storage area network* (SAN) could be taken into account as well.

At the end, this architecture behaves more like a single machine rather than many workstations. All the computers are networked into the same TCP/IP LAN. Client nodes do not have neither keyboards nor monitors, and are accessed only via secure shell. They can be thought of as a sort of CPU + memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard. A schematic can be seen in Figure 5.5.

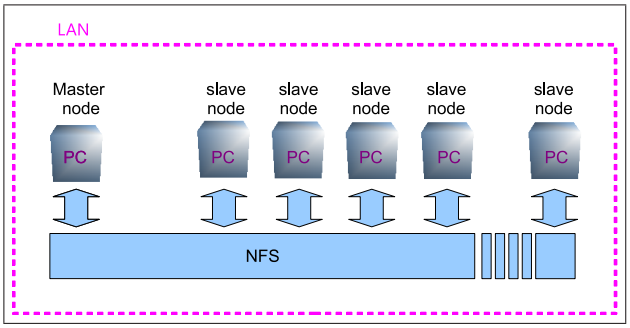


Figure 5.5: High performance Linux cluster: schematic of beowulf inside a LAN environment with Network File System

In such a way a so-called *high performance linux cluster* has been constructed (see [31]) using only commodity, off-the-shelf (COTS) inexpensive hardware and adopting only freely available open source software.

5.4 Numerical simulations

To evaluate the efficiency of the proposal approach several numerical simulations have been conducted on different truss-like structures. Some results are presented and discussed in the following sections.

5.5 2D truss cantilever

This example deals with both single and multiobjective structural optimizations. The procedures has been applied to a 2D steel structure, composed

[9] http://www.parl.clemson.edu/pvfs/

by 12 members and loaded by a nodal force P, as can be seen in Figure 5.6. This problem presents three different objectives:

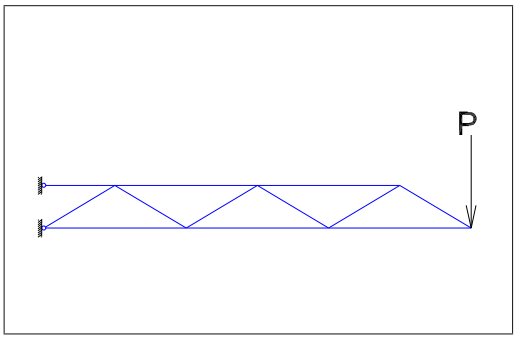


Figure 5.6: Optimization problem

- minimization of total weight;
- minimization of vertical displacement of the directly loaded node.
- a combination of both of the previous objectives.

The design variables are the cross area sections. Since the members are supposed to have a hollow circular cross section, there are two parameters to evaluate:

- the external diameter D_i;
- the thickness t_i.

resulting in 24 design variables for each solution. The constraints are:

- the member resistance;
- $0,1\ m < D < 0.7\ m$;
- $0,002\ m < t < 0.05\ m$;

For each member the resistance is computed according to allowable stress method ($\sigma \leq \sigma_{\text{allowable}}$), taking into account the resistance penalization for $t > 40mm$. The instability problem is neglected. Both the external load $P = 100\ kN$ and the varying self weight of structure are considered by the optimization procedure as well.

5.5.1 Material and structural data

The data for the material are:

- Steel type Fe430 (S275)
- Modulus of elasticity $E = 206000\ MPa$
- Shear modulus $G = 80000\ MPa$
- Weight for unit of volume $\gamma = 78.50\ kN/m^3$
- Poisson's coefficient $\nu = 0.3$
- $\sigma_{\text{allowable}} = 190\ MPa\ (t < 40mm)$
- $\sigma_{\text{allowable}} = 170\ MPa\ (t \geq 40mm)$

The structural geometry can be seen in Figure 5.7

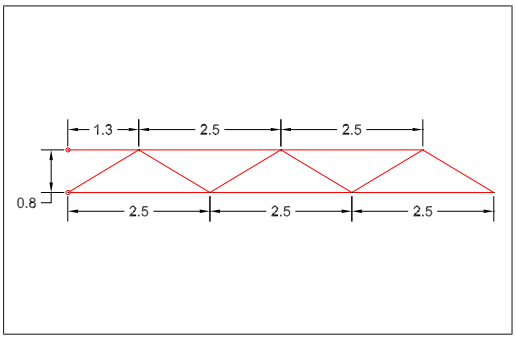


Figure 5.7: Structural geometry

5.5.2 GA parameters

The parameters for the genetic algorithms are:

- Population size $s = 100$
- Selection algorithm: roulette wheel;
- Selection probability $p = 0.05$;
- Crossover probability $p = 0.5$;
- Mutation probability $p = 0.1$;
- Elitism activated (elitism size=1).

5.5.3 The optimization process

The structure has been modeled with twelve truss elements, whose numbering scheme can be seen in Figure 5.8

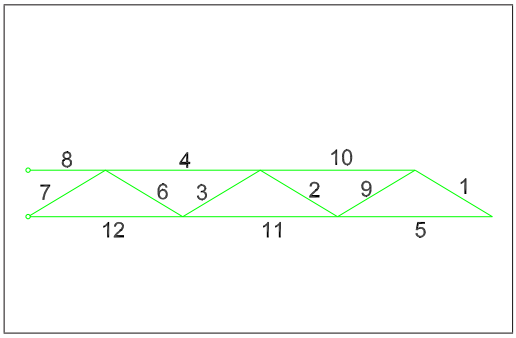


Figure 5.8: Numbering scheme

objective 1: weight minimization

The objective is the reduction of the total weight of the structure. The best result is 8368 N (solution id=57), the worst is 143009 N (solution id=19). The Figure 5.9 show the history of the objective function (including both feasible and infeasible solutions). It can be seen that the solution seem to behave randomly in the initial population and improve (on the average) going further the following generations. The optimal result is described in Table 5.1.

element ID	Diameter m	Thickness m
1	0,654	0,004
2	0,541	0,004
3	0,560	0,004
4	0,391	0,004
5	0,278	0,004
6	0,146	0,004
7	0,109	0,004
8	0,597	0,004
9	0,522	0,004
10	0,428	0,004
11	0,315	0,004
12	0,240	0,004

Table 5.1: Weight optimization

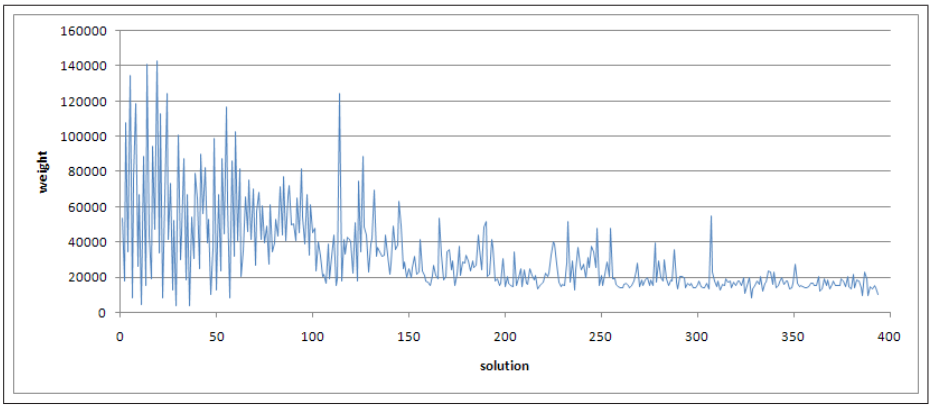


Figure 5.9: Weight's history (including both feasible and infeasible solutions)

objective 2: displacements minimization

The same structure has been studied in order to minimize the vertical displacements of node 2. The best result (solution id=1185) is described in Table 5.2. The corresponding displacement is 0,003894162 m.

element ID	Diameter m	Thickness m
1	0,248	0,045
2	0,382	0,05
3	0,671	0,049
4	0,7	0,05
5	0,337	0,037
6	0,639	0,047
7	0,658	0,049
8	0,685	0,05
9	0,646	0,037
10	0,517	0,044
11	0,7	0,045
12	0,7	0,05

Table 5.2: Displacement minimization

The Figure 5.10 show the history of the objective function (including both feasible and infeasible solutions). Usually the peaks in the value of the displacements represent solutions belonging to the infeasible domain.

objective 3: multiobjective optimization

The objective is made by the combination of the two objectives seen so far:

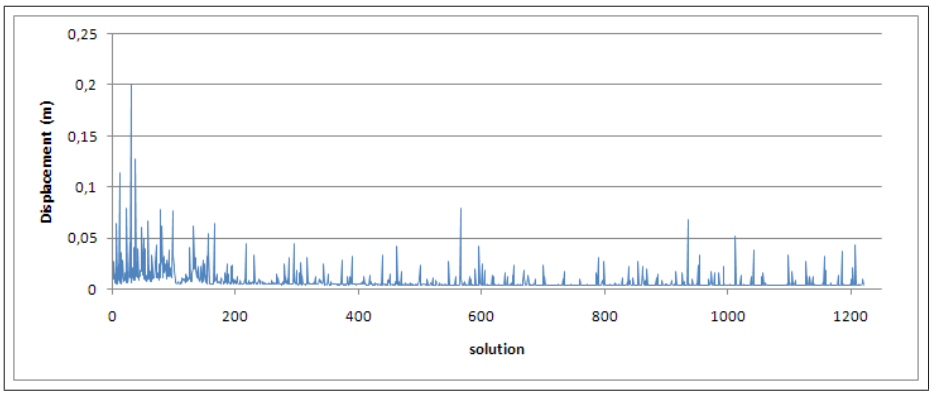


Figure 5.10: Displacements' history (The peaks usually represent solution belonging to the infeasible domain)

- minimization of total weight;
- minimization of vertical displacement of the directly loaded node.

In order to understand what happen during the optimization process we will focus our attention in member number 8. This is only for the sake of simplicity. The same consideration could then be extended also to the other members. Moreover, since the multi objective problem come from only two different objectives it is possible to use a plane graph load-displacement. This makes easier to understand graphically the performance of the solutions.

The first generation was randomly generated with 100 individuals. In Figure 5.11 is possible to see the distribution of diameters d_i within the

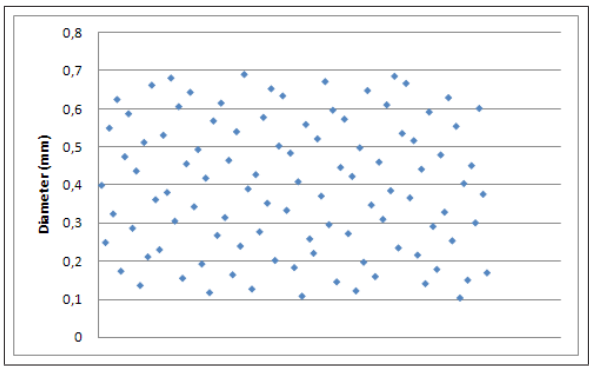


Figure 5.11: Distribution of bar diameters D in the first generation

initial population. Figure 5.12 and 5.13 show the distribution for thickness t_i and area of cross sections. Figure 5.14 show how the first population

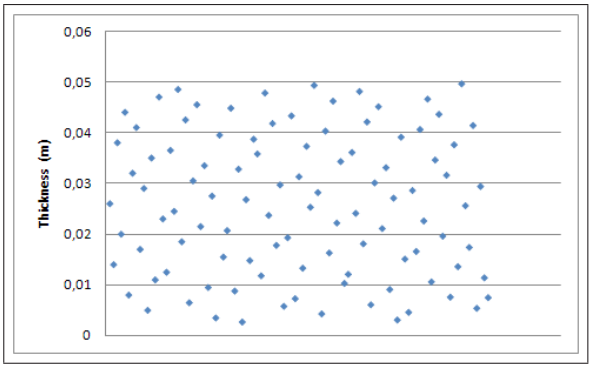


Figure 5.12: Distribution of bar thickness t in the first generation

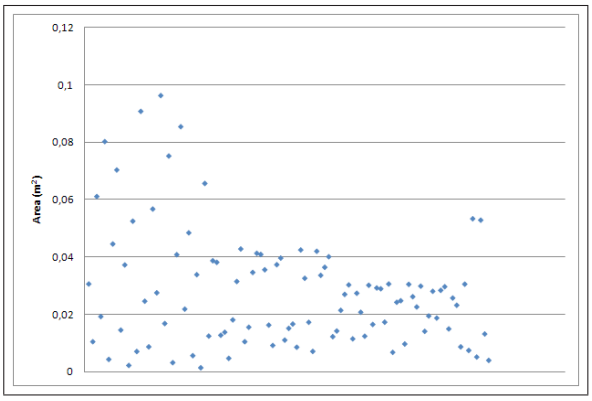


Figure 5.13: Distribution of bar area in the first generation

satisfies load and displacement objectives.

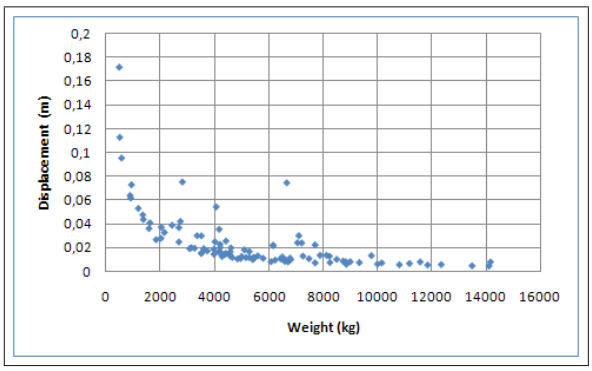


Figure 5.14: Weight-Displacement in the first generation

After many generation the optimal (or suboptimal) solutions are obtained. The corresponding results can be seen in Figure 5.15, 5.16 and

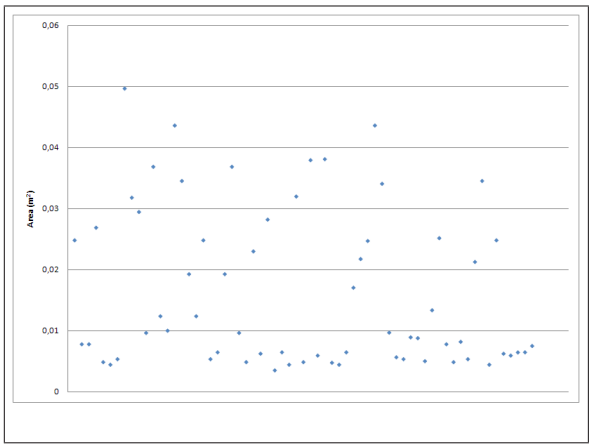


Figure 5.15: Distribution of bar area between optimal solutions

5.17 where are represented, respectively, the distribution of the area and the design variables (d_i and t_i). Figure 5.18 shows the position of the best solution in the graph load-displacement. It can be seen that the points are less scattered than in Figure 5.14. This is due to the fact that the present solutions are better than the initial population. They can be seen, in their own domain space in Figure 5.19. The execution stops after the creation of 1215 individuals, meaning that $1215 \cdot 24 = 29160$ values for the design variables have been computed. From the numerical result it can be seen that the weight varies in the range $1051,645 \div 11765,85$ kg and the displacements remain inside the interval $0,00460 \div 0,043036\ m$.

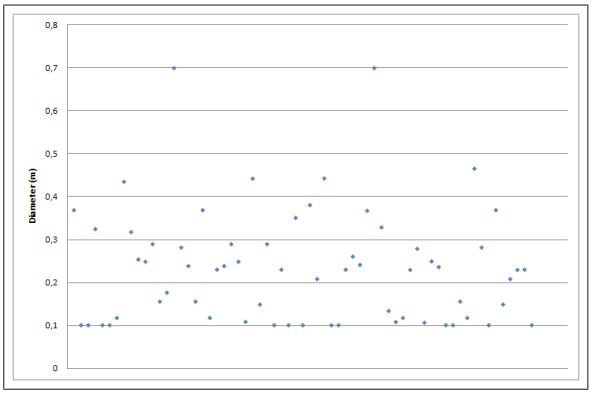


Figure 5.16: Distribution of bar diameters D between optimal solutions

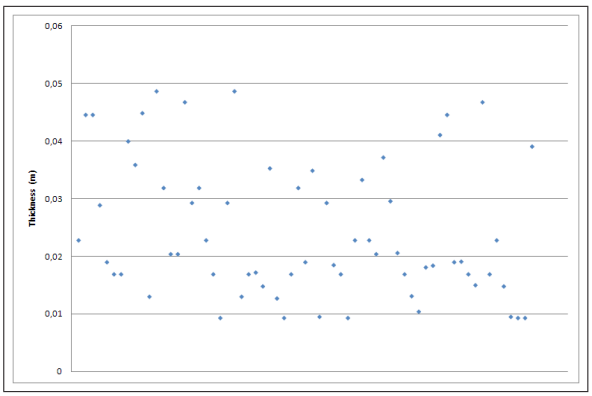


Figure 5.17: Distribution of bar thickness t between optimal solutions

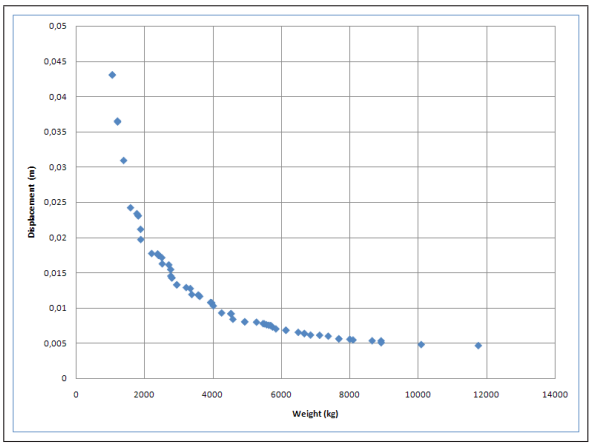


Figure 5.18: Weight-Displacement for optimal solutions

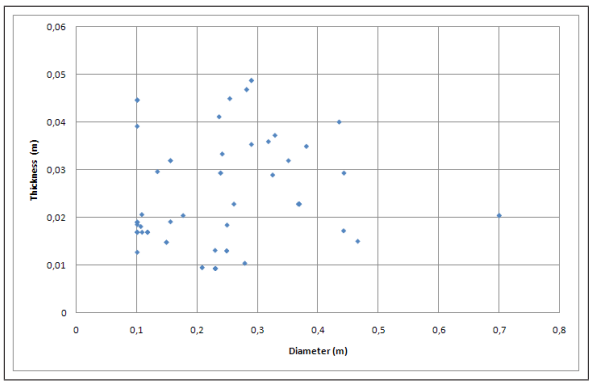


Figure 5.19: Optimal solutions in the design space

To see the points of the research space that have been investigated it is possible to look at Figure 5.20 where the total distribution of areas is visible and Figure 5.21 where each point represent a solution of the design space. In Figure 5.22 all the solution (of every generation) are represented in the

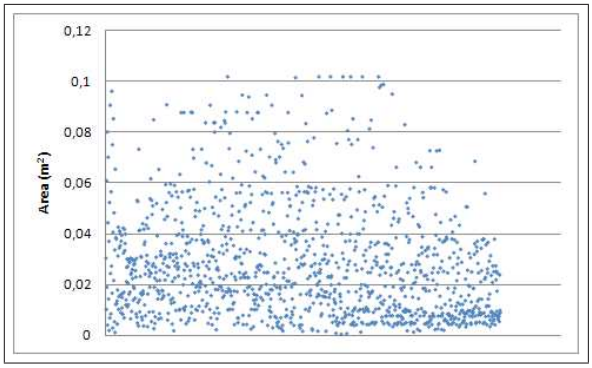


Figure 5.20: Distribution of bar area from the beginning to the end of the optimization

graph displacements–weight.

Finally, since the objective function includes both weight and displacements. The corresponding histories are shown in Figure 5.23 and Figure 5.24 where are represented the value assumed, respectively, by the weight and the displacements for the over one thousand of solutions that have been computed during the whole structural optimization process.

As told before, in those observations we have focused the attention on member 8, but similar considerations could be done for the other members of the structure.

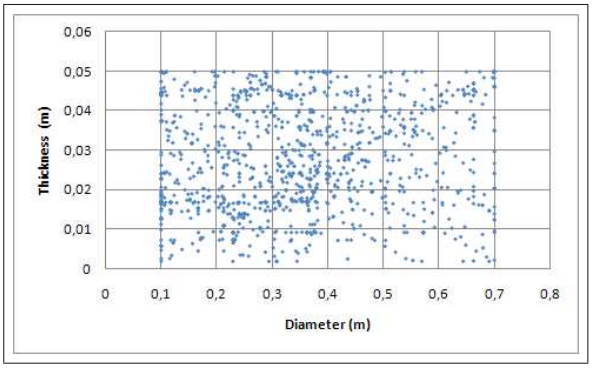


Figure 5.21: All the individuals in the design space

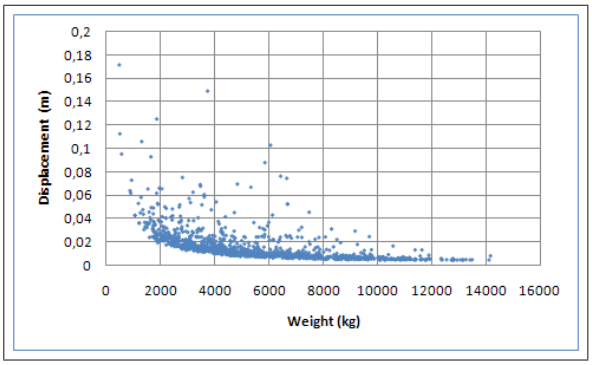


Figure 5.22: Performance of all the individuals

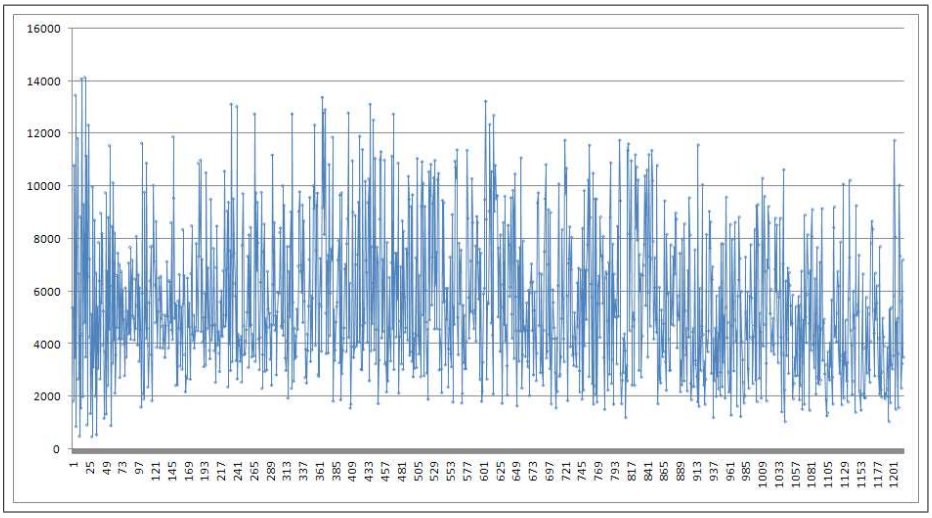


Figure 5.23: History of weight

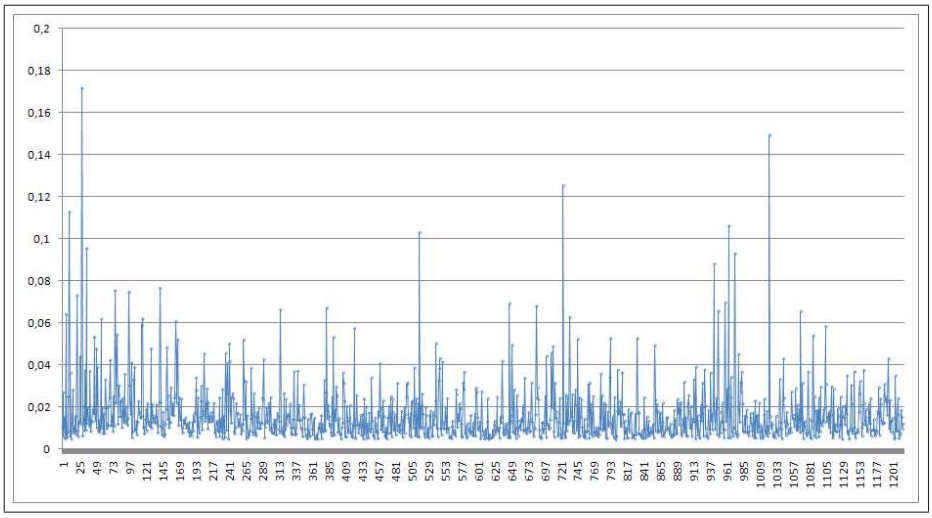


Figure 5.24: History of displacement

Since the chosen design variables represent the cross section areas, this applications belong to the class of *sizing* optimization.

5.6 Double hinged 2D truss

This example deals with the structure represented in Figure 5.25. The struc-

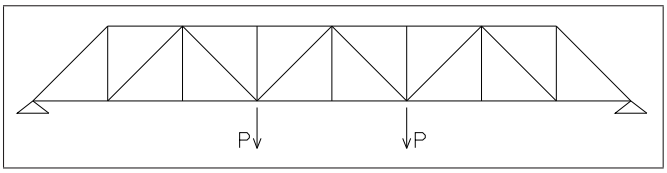


Figure 5.25: Structural scheme

ture has been discretized with 29 2D truss elements. The nodal numbering scheme can be seen in Figure 5.26. Since the structure is supposed to be

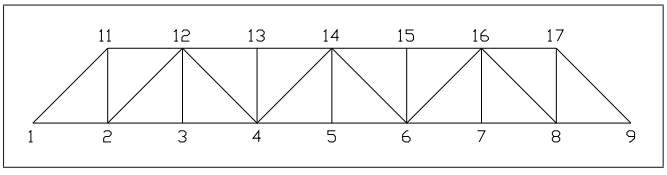


Figure 5.26: Nodal numbering scheme

symmetric and symmetrically loaded, the number of design variables has been reduced by using the section numbering scheme of Figure 5.27.

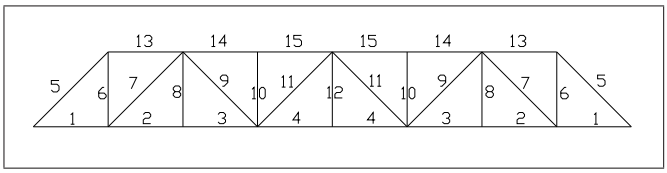


Figure 5.27: Sections' numbering scheme

5.6.1 Material and structural data

The data for the material are:

- Steel type Fe360 (S235)
- Modulus of elasticity $E = 206000\ MPa$
- Shear modulus $G = 80000\ MPa$
- Weight for unit of volume $\gamma = 78.50\ kN/m^3$
- Poisson's coefficient $\nu = 0.3$
- $\sigma_{\text{allowable}} = 160\ MPa\ (t < 40mm)$
- $\sigma_{\text{allowable}} = 140\ MPa\ (t \geq 40mm)$

The structural geometry is described by the nodal coordinates that are summed up in Table 5.3. The modulus of the external load P is equal to 100000 N and it is applied at nodes 4 and 6 (see Figure 5.25).

5.6.2 GA parameters

The parameters for the genetic algorithms are:

- Population size $s = 100$
- Maximum number of generations = 2000;
- Selection algorithm: roulette wheel;
- Mutation probability $p = 0.01$;
- Elitism activated (elitism size=1).

The algorithm evaluates the self weight of the structure automatically and takes into account the instability of compress members according to $CNR-UNI$ 10011 national code.

node ID	x m	y m	z m
1	0,0	0,0	0,0
2	2,0	0,0	0,0
3	4,0	0,0	0,0
4	6,0	0,0	0,0
5	8,0	0,0	0,0
6	10,0	0,0	0,0
7	12,0	0,0	0,0
8	14,0	0,0	0,0
9	16,0	0,0	0,0
11	2,0	0,0	2,0
12	4,0	0,0	2,0
13	6,0	0,0	2,0
14	8,0	0,0	2,0
15	10,0	0,0	2,0
16	12,0	0,0	2,0
17	14,0	0,0	2,0

Table 5.3: Nodal coordinates

sizing optimization

The cross sections are supposed to be hollow circular sections. They are defined by two parameters: the diameter d_i and the thickness t_i with $i = 1, 2, \ldots, 15$. The thickness is assumed to be constant and equal to 0.005 m, while the diameter (in meters) must satisfy the constraint $0.1 \leq d_i \leq 0.2$. The numerical simulations gives the *optimum* result of Table 5.4 (solution id=1817) corresponding to a weight of 8547.09 N. The weight reduction (compared to the first random generation) is about 20.6%. The whole optimization process can be seen in Figure 5.28. It is shown that most of weight

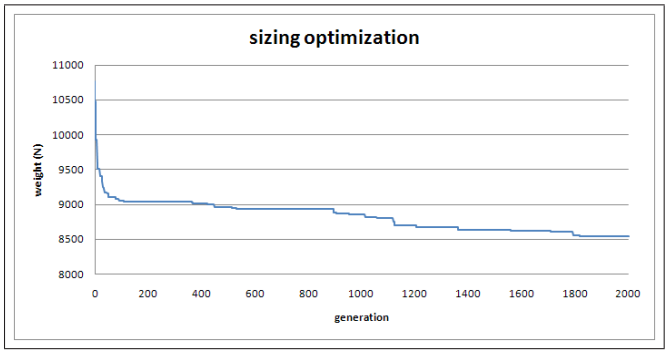


Figure 5.28: Sizing optimization: history (complete)

reduction is performed during the first 100 iterations: a more detailed view can be seen in Figure 5.29. Weight at iteration 100 equals to 9048.9N cor-

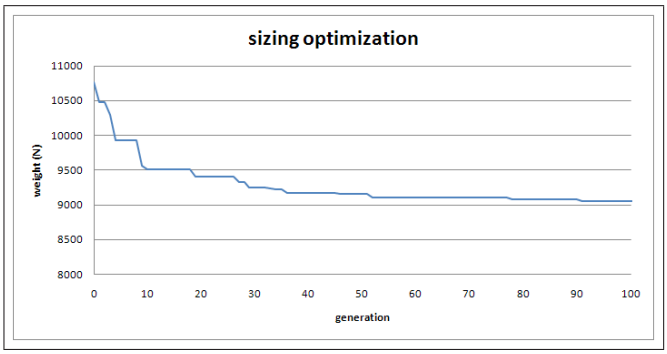


Figure 5.29: Sizing optimization: history (first 100 generations)

responding to a 15.9% of the initial weight. The final solution is presented in in Figure 5.30.

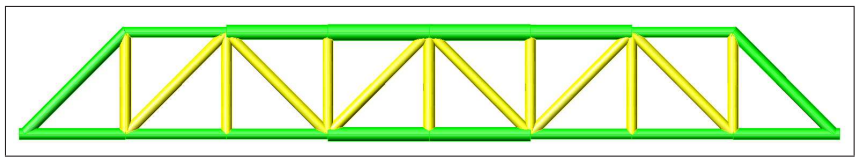


Figure 5.30: Sizing optimization of double hinged truss

topology optimization

In this topology optimization the structural nodes are allowed to move vertically and horizontally from their original position (see Table 5.3) by a quantity $delta = \pm 0.8$ m. All the members share the same cross section which is fixed (hollow circular section with diameter $d = 0.1143$ m and the thickness $t = 0.005$ m) and remains constant. The initial geometry is shown in Figure 5.31. The numerical procedure gives the *optimum* result of

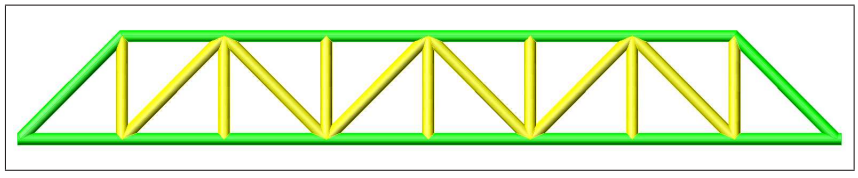


Figure 5.31: Topology optimization: initial solution

section ID	diameter m	thickness m
1	0.111	0.005
2	0.102	0.005
3	0.102	0.005
4	0.137	0.005
5	0.111	0.005
6	0.109	0.005
7	0.102	0.005
8	0.104	0.005
9	0.102	0.005
10	0.103	0.005
11	0.100	0.005
12	0.114	0.005
13	0.102	0.005
14	0.143	0.005
15	0.159	0.005

Table 5.4: Sizing optimization: best solution (id=1817), weight=8547.09 N

node ID	x m	y m	z m
1	0.000	0.000	0.000
2	2.216	0.000	0.242
3	3.898	0.000	0.479
4	5.875	0.000	0.460
5	8.000	0.000	0.549
6	10.125	0.000	0.460
7	12.102	0.000	0.479
8	13.784	0.000	0.242
9	16.000	0.000	0.000
11	2.132	0.000	1.262
12	4.008	0.000	1.720
13	5.422	0.000	2.526
14	8.000	0.000	2.444
15	10.578	0.000	2.526
16	11.992	0.000	1.720
17	13.868	0.000	1.262

Table 5.5: Topology optimization: best solution (id=1339), weight=10121.3 N

Table 5.5 (solution id=1339) corresponding to a weight of 10121.3 N. The weight reduction (compared to the first randomly created generation) equals to 6.6%. The final solution is shown in Figure 5.32. A collection of solutions

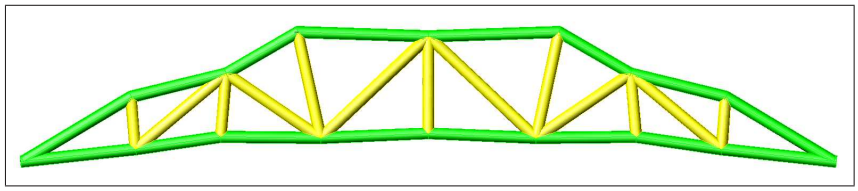


Figure 5.32: Topology optimization: final solution

found at different generations can be seen in Figure **??**. Finally Figures 5.34

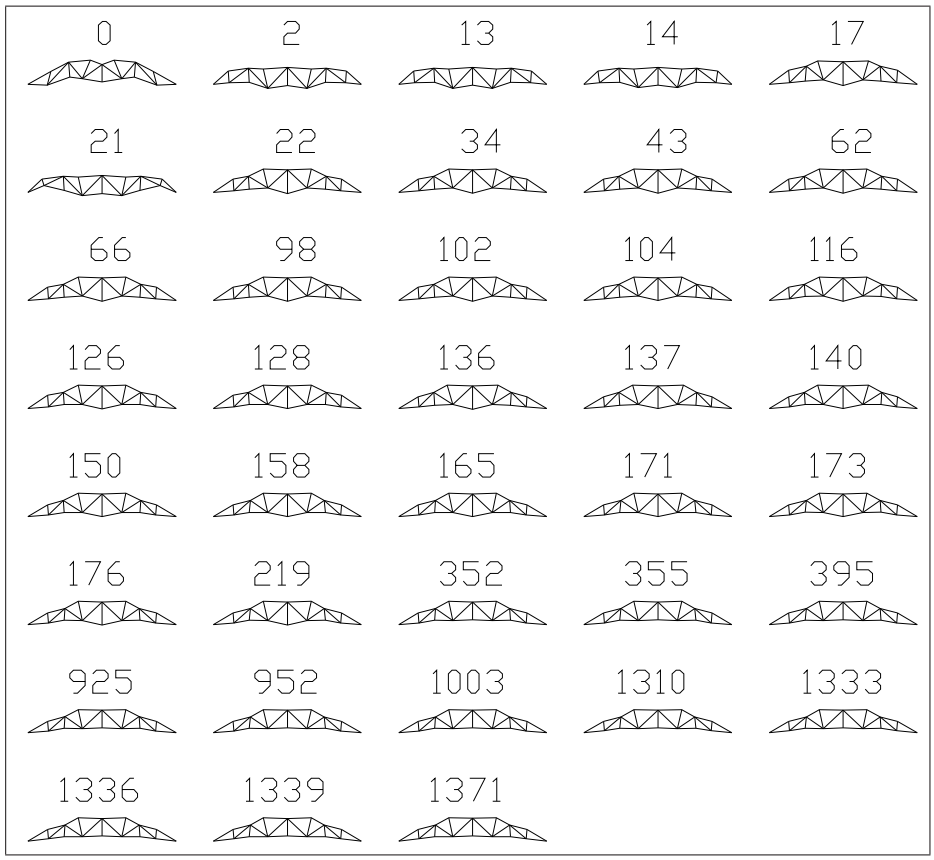


Figure 5.33: Topology optimization: solutions at different generations

and 5.35 show the history of weight during the optimization process.

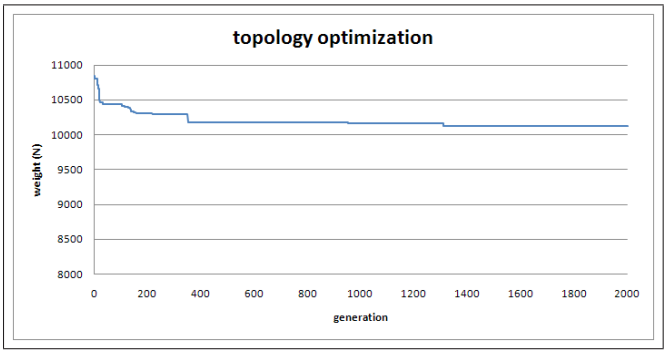


Figure 5.34: Topology optimization: history of weight (complete)

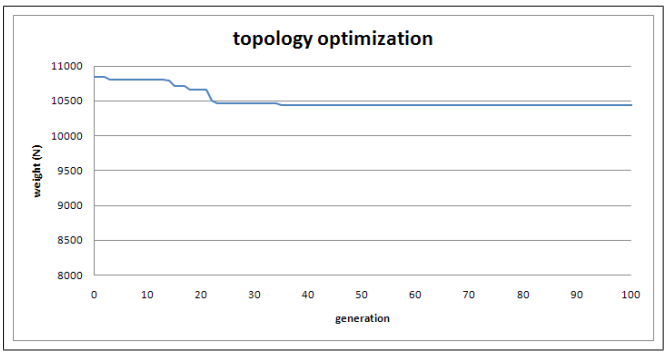


Figure 5.35: Topology optimization: history (first 100 generations)

combined optimization

This application is a combination of the two previous optimizations. Here the cross sections are supposed to be hollow circular sections. They are defined by two parameters: the diameter d_i and the thickness t_i with $i = 1, 2, \ldots, 15$. The thickness is assumed to be constant and equal to 0.005 m, while the diameter (in meters) must satisfy the constraint $0.1 \leq d_i \leq 0.2$. At the same time the structural nodes are allowed to move vertically and horizontally from their original position (see Table 5.3) by a quantity $delta = \pm 0.8$ m.

The numerical simulations gives the *optimum* result of Tables 5.6 and 5.7(solution id=1672) corresponding to a weight of 8728.59 N. It can be seen in Figure 5.36.

section ID	diameter m	thickness m
1	0.106	0.005
2	0.136	0.005
3	0.102	0.005
4	0.161	0.005
5	0.110	0.005
6	0.106	0.005
7	0.103	0.005
8	0.141	0.005
9	0.112	0.005
10	0.134	0.005
11	0.104	0.005
12	0.114	0.005
13	0.105	0.005
14	0.127	0.005
15	0.131	0.005

Table 5.6: Combined optimization: cross sections of best solution (id=1672)

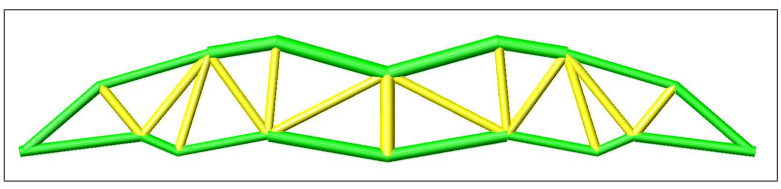


Figure 5.36: Combined optimization: final result

The weight reduction (compared to the first randomly created generation) is about 23.1% of the initial weight. The whole optimization process

node ID	x m	y m	z m
1	0.000	0.000	0.000
2	2.584	0.000	0.327
3	3.389	0.000	-0.014
4	5.363	0.000	0.354
5	8.000	0.000	-0.107
6	10.637	0.000	0.354
7	12.611	0.000	-0.014
8	13.416	0.000	0.327
9	16.000	0.000	0.000
11	1.670	0.000	1.429
12	4.065	0.000	2.134
13	5.557	0.000	2.376
14	8.000	0.000	1.692
15	10.443	0.000	2.376
16	11.935	0.000	2.134
17	14.330	0.000	1.429

Table 5.7: Combined optimization: nodal positions of best solution (id=1672)

can be seen in Figure 5.37. It is shown that most of weight reduction is

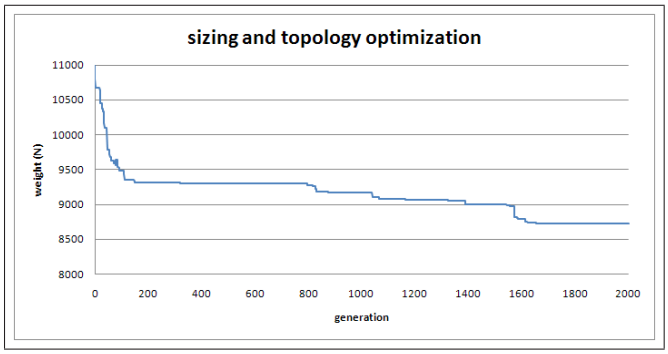


Figure 5.37: Combined optimization: history of weight (complete)

performed during the first 100 iterations. Weight at iteration 100 equals to 9487.2 N corresponding to a 16.4% of the initial weight. For this reason a more detailed view can be seen in Figure 5.38.

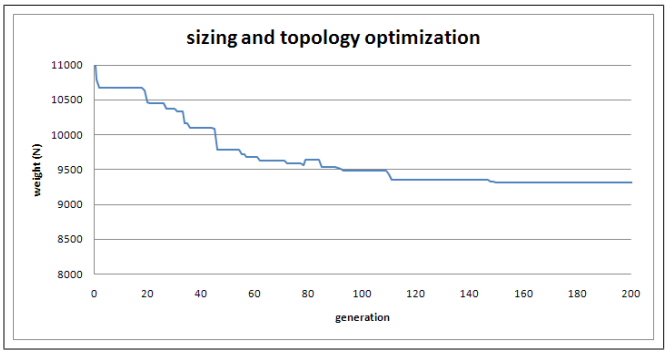


Figure 5.38: Combined optimization: history of weight (first 200 generations)

final comments

A comparison among sizing, topology and combined (sizing+topology) optimization can be seen in Figure 5.39. For this problem, after 2000 generations the sizing optimization seems to behave better than topology optimization, but in terms of relative weight reduction both of them are outperformed by the combined technique (see Table 5.8).

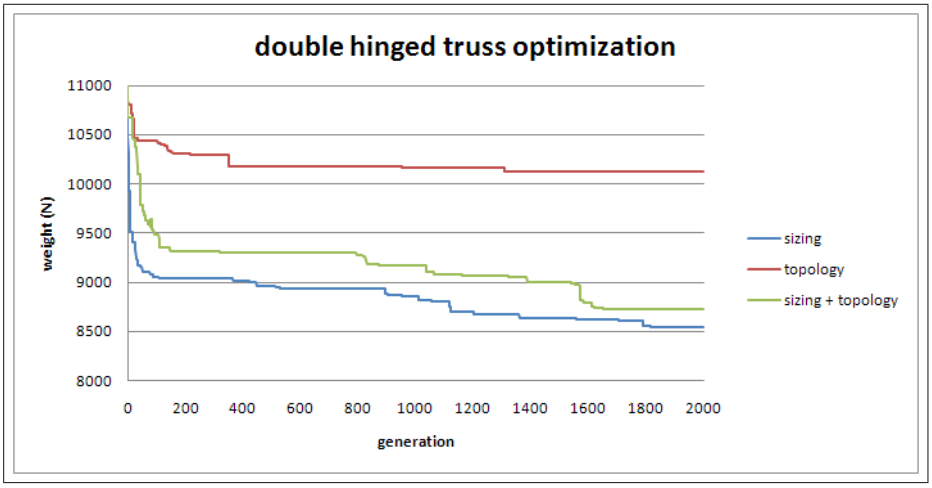


Figure 5.39: Comparison between different optimizations

5.7 3D truss structure

This model takes its inspiration from a real roof structure that spans about 56 meters. It is double hinged and loaded by one load condition which

optimization	initial weight (N)	final weight (N)	weight reduction
sizing	10760.6	8547.09	20.6%
topology	10841	10121.3	6.6%
sizing & topology	11347.3	8725.59	23.1%

Table 5.8: Comparisons between different optimizations for the double hinged 2D truss (see Figure 5.25)

includes:

- self weight (automatically computed by the optimization algorithm);
- roof (dead) loads: 0.5 KN/m^2;
- snow: 0.48 KN/m^2 (according to Italian law);

Roof loads and snow are applied to the top structural nodes according to their area of influence. An optimization procedure has been applied which is characterized by the following parameters:

- population size: 100;
- elitism size: 1;
- selection: roulette wheel;
- maximum number of iterations: 2000;
- mutation probability $p = 1\%$;

The procedure includes a finite element analysis and a post processing phase, in which both the member resistance[10] and instability are considered. The steel type is Fe510 (S355), its weight per unit of volume is 78.50 kN/m^2. The allowable normal stresses are $\sigma_{adm} = 240MPa$ for section thickness $t \leq 40$ mm and $\sigma_{adm,thick} = 210MPa$ for $t < 40$ mm; the modulus of elasticity E_s is 206000 MPa. The structure can be seen in Figure 5.40 and Figure 5.41.

A topology optimization has been performed. The cross sections are considered fixed and immutable, while the positions of the lower nodes can translate vertically. In Figure 5.42 their lower and higher admissible positions are marked respectively with green and red spheres. The difference between the highest and lower values is 3 meters. The top and restrained nodes are fixed for architectonic reasons.

Some result can be found in Figure 5.44 where the evolution towards the final form finding is shown. During the procedure a weight reduction of roughly 3% respect to initial (traditional) design has been achieved. The weight's history is shown in Figure 5.43.

[10]The allowable stress method is employed.

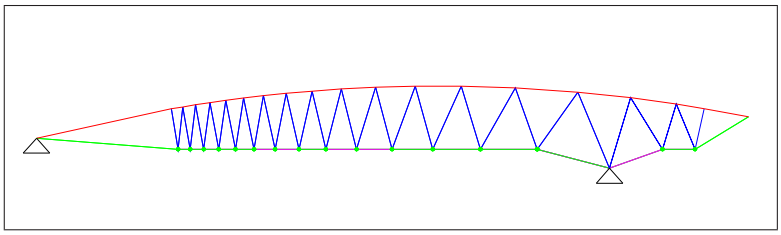


Figure 5.40: Structural scheme

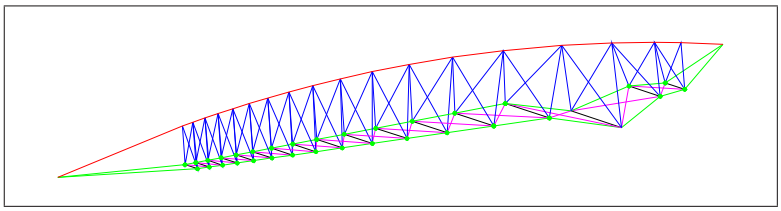


Figure 5.41: 3D view

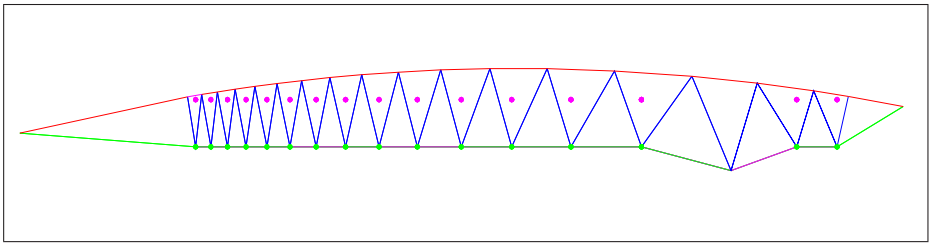


Figure 5.42: The green and red spheres represents, respectively, the higher and lower bound for vertical translation of bottom nodes

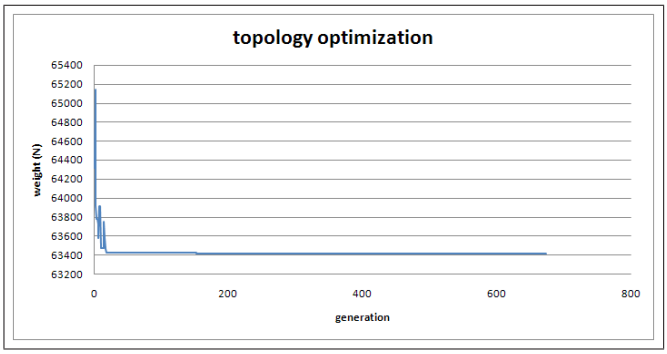


Figure 5.43: History of weight

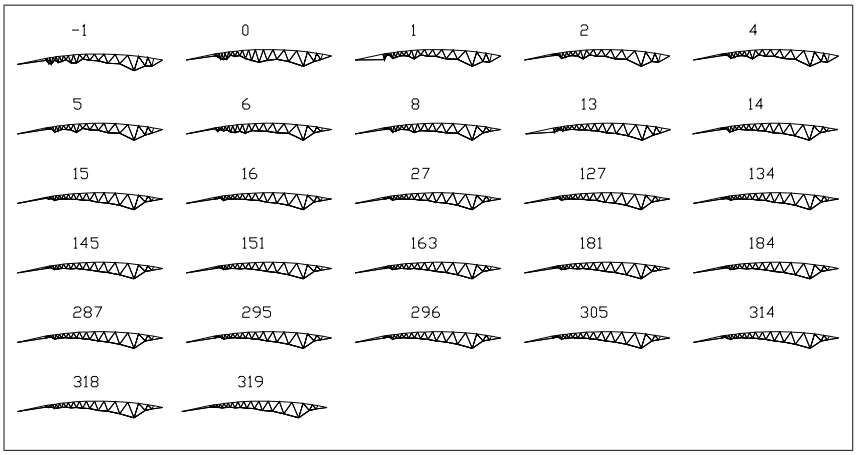


Figure 5.44: Solutions at different generations

5.8 Conclusions

Structural optimization aims to achieve the "best" design for a structure and it has been shown in the previous chapters that, especially in the most recent approaches, its roots extend to different areas of engineering, mathematics, science and technology. For this reason, from a certain point of view, it could also be considered as a multidisciplinary topic.

In the field of structural optimization, there are various methods that can be successfully used. Most of them can be grouped in two categories. The first category, gradient based methods, makes use of the computation of derivatives to search for the optimum. So most of them implicitly assume that the problem is convex, that a minimum solution can be found, and that the solution exists. But in structural mechanics this could be an issue since, often, the problem may be discontinuous and disjointed.

For this reason, other methods which are independent of the gradients of the functions used are necessary, such as heuristic methods. Most of them have been developed taking inspiration from observation of nature and are based on relatively simple rules and common sense.

It seems clear, today, that the techniques based on global search (or *soft*) are much more convenient in contrast to those based on local search (*hard*). In fact the latter are substantially gradient-based techniques that focus their attention on a single solution that try to improve by moving around its neighborhood with the so-called *hill climbing* technique. Some of them, such as the *penalty function*, the *augmented Lagrangian* and the *conjugate gradient* methods aim to find a local optimum, searching in a direction depending from local gradient. Others apply the first and second order conditions to find a local minimum, by solving a system of nonlinear functions. They usually try to find a solution in the neighborhood of the starting point. If the problem has more than one local optimum the result of the optimization procedure depend strongly from the choice of the starting point. Besides that, the convergency is not assured. Moreover when the objective function or the constraints presents a form of discontinuity (of any kind), the gradient's computation become difficult and instable.

On the contrary, the global search techniques create a large set of potential solutions that are, from time to time, selected and modified.

Genetic algorithms can be seen as a really valuable alternative to the more traditional hill climbing algorithms for many reasons. First of all, from the numerical point of view, there are only computation of function in place of gradients. This allows a better numeric efficiency and avoids the typical problems of gradient-based methods: difficulties in computations and numerical instabilities when there are some sort of discontinuity. Moreover the algorithm begins with a population of individuals randomly located in the whole research space, and so the problem of dependency from the starting point is strongly reduced. Therefore they have a larger amount of probabil-

ity of finding the global optimum avoiding to get drifted and captured by local optimum values as frequently happen to *hard* methods.

Finally one of the most important advantages of genetic algorithms is the feature of being particularly suited to be implemented with techniques of parallel and distributed computing, since each individual belonging to a given generation is independent from the others and, therefore, can be processed in parallel.

For this reason, the optimization program that has been written in C++, has been designed for parallel computing from the beginning, being the optimization with genetic algorithms an ideal candidate for parallelization (also taking into account the high computational cost of the optimization problems). An alternative way to encode floating point data in structural optimization problems has been presented as well.

At the end of the present work a numerical application of the proposed approach is discussed.

For many years optimization has remained to be of more academic interest maybe because of its mathematical complexities [74]. Respect to the *Finite element analysis (FEA)* that, in recent years has become a widely used tool for engineers of many disciplines, structural optimization has undoubtedly achieved far less popularity. The aim is to show that nowadays the field of structural optimization algorithms can be considered mature and that now it can become accessible to the engineers and scientists. This is mainly due to two push factors. First the availability of high performance computing power at low cost, and secondly new algorithms that can finally handle a really large number of design variables and constraints. This work has tried to support this idea.

5.8.1 Structural optimization for practising engineer

In the opinion of the author of the present thesis there is the belief that a the structural optimization can be a useful tool for structural design. A rational use of computer technologies doesn't harm or limit the creativity which must be always present int the design activity. Designing is a really complex work, it contains many aspects which come from knowledge, technique and art. It must satisfy, at the same time, requirements that are often in contrast each other and that only rarely can be expressed in mathematical form. The ability of keeping all those different needs in equilibrium is its essence. It is mainly a product of intuition and creativity and it doesn't must be influenced by the presence or absence of analysis facilities such as the finite element analysis or the proposed optimization method.

Instead these tools should be seen as instruments that can help and support the designer to take his decisions, that can give information and suggest directions for development, but leaving the designers free to follow or ignore those according to his own sensibility.

In the case of optimization neither the objective function nor the constraints can include all the needs a designer must take into account (such as aesthetic sense and all the things that can't be expressed mathematically), so in practice is not necessary to find the global optimum in rigorous sense. In many occasion the designer wants to see a direction, to feel a sensation and so also suboptimal optimization can be of interest. For this reason it is not necessary to adopt a very strict convergence criterion, and useful information can be found from the individuals without the best fitness as well.

5.9 Summary

An approach based on genetic algorithm and parallel computing has been presented and discussed for structural optimizations. Some details on its implementation are given and explained. Numerical simulations demonstrate the applicability of the proposed approach for the optimization of truss structures. The results show that parallel computing seems to be a valuable feature for optimization of large structure, since the genetic procedure requires a lot of finite element analysis. Sharing that computations to a variable number of computers can effectively improve the time required to optimize.

The adopted architecture allow to mix computers with different specifications, age and power. The optimization procedure is entirely managed by the master PC and a variable number of clients can be added or removed without difficulty.

The genetic algorithm seem to be a valuable technique for optimization of structure, since doesn't suffer many of the limitations of the gradient based approach. But, on its own, introduce new difficulties in practical application. Large computational resource are required. This requirements grows very fast as the structures become more and more complex. This first issue has been addressed with the proposal of an approach based on parallel computing.

5.10 Future work

This is only a starting point. Much more work is possible to do. What has been presented here is just a sort of framework. It is open to upgrades and extensions.

Future work can include developments on the algorithms and on parallel implementation. In particular the approach could be applied to large size structures, with a very high number of variables. Since the calibration of the parameters is a really important and difficult part of the optimization procedure, the influence of the parameters could be evaluated by applying

the methodology on classical test taken from literature. Some test could be done to see if the system scales well. Eventually an integration to a database could also be taken into account and so on.

List of Figures

List of Tables

Contents

Bibliography

[1] Zadeh L. A. Fuzzy sets. *Information and Control*, 8(3):338–353, June 1965.

[2] Hojjat Adeli and Kamal C. Sarma. *Cost optimization of structures: fuzzy logic, genetic algorithms, and parallel computing*. John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, 2006.

[3] Rao C. R. Al Jadaan Omar, Rajamani Lakishmi. Improved selection operator for ga. *Journal of Theoretical and Applied Information Technology*, 4(4):269–277, 2008.

[4] Hassani B. *Homogenization and topological structural optimization*. PhD thesis, DDepartment of Civil Engineering, University of Wales, Swansea, 1996.

[5] Rosen J. B. The gradient projection method for non linear programming. part ii: Nonlinear constraints. *The society for industrial and applied mechanics journal*, 9(4):514–532, 1961.

[6] E.M.L. Beale. *Introduction to optimization*. John Wiley & Sons, 1988.

[7] Martin P. Bendsøe. *Optimization of structural topology, shape and material*. Springer, Heidelberg, 1995.

[8] Martin P. Bendsøe and Ole Sigmund. *Topology Optimization: Theory, Methods and Applications*. Springer Verlag, Berlin Heidelberg New York, 2004.

[9] Rodrigues H. C. Bendsøe M. P., Rasmussen J. Topology and boundary optimization as an integrated tool for computer aided design. In *Proceedings of the International Conference on Engineering Optimization in Design Processes*, pages 27–34, Germany, Sep 3-4 1990. Karlsruhe Nuclear Research Center.

[10] Blaise Barney (blaiseb@llnl.gov). Introduction to parallel computing. Livermore Computing.

[11] Leonard Conrad Breebaart. *Rule-based Compilation of data-parallel programs*. PhD thesis, Technische Universiteit Delft, 2003.

[12] Zienkiewicz O. C. and Taylor R. L. *The Finite Element Method for Solid and Structural Mechanics*. Elsevier, Amsterdam, 6th edition, 2005.

[13] Goldfarb D. A family of variable metric methods derived by variational means. *Math. Comput.*, 24:23–26, 1970.

[14] Powell M. J. D. A fast algorithm for nonlinearly constrained optimization calculations. In *Proceedings of the 1977 Dundee conference on numerical analysis*. Springer-Verlag, 1978. Lecture notes in mathematics, pp 144-157.

[15] Whitley D. Practical guidelines for evolutionary algorithms.

[16] Whitley D. he genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In Schaffer J.D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116 – 121, San Mateo, 1989. Morgan Kaufmann.

[17] Whitley D. Genetic algorithms and evolutionary computing. In *Van Nostrand's Scientific Encyclopedia*. John Wiley & Sons, 2002.

[18] Whitley D., Mathias K., Rana S., and Dzubera J. Building better test function. In Eshelman L., editor, *Proceedings of the sixth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1995.

[19] Whitley D. and Rana S. Representation, search and genetic algorithms. In *Proceedings of the 14th national conference on artificial intelligence*. AAAI Press / MIT Press, 1997.

[20] Shanno D.F. Conditioning of quasi newton methods for function minimization. *Math. Comput.*, 24:647–656, 1970.

[21] Goldberg D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York, 1989.

[22] Broyden C. G. The convergence of a class of double rank minimization algorithm 2. the new algorithm. *J. Inst. Math. Appl.*, 6:222–231, 1970.

[23] Krzesinski G. Shape optimization and identification of 2-d elastic structure by the bem. In *Proceedings of the International Conference on Engineering Optimization in Design Processes*, pages 51–58, Germany, Sep 3-4 1990. Karlsruhe Nuclear Research Center.

[24] Marsaglia G. Random numbers for c: End, at last? posted to sci.stat.math., 1999.

[25] Vanderplaats G.N. Conmin - a fortran program for constrained function minimization. TM X-62282, NASA, 1973.

[26] Thierens D. Goldberg D.E., Deb K. Toward a better understanding of mixing in genetic algorithms. *journal of the society of instrument and control engineers*, 32(1):10–16, 1993.

[27] Holland J. H. *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, Ann Arbor, 1975.

[28] Press W. H., Teukolsky S. A., Vetterling W. T., and Flannery B. P. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 2002.

[29] Press W. H., Teukolsky S. A., Vetterling W. T., and Flannery B. P. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.

[30] Koza J., Keane M., Streeter M., Mydlowec W., Yu J., and Lanza G. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[31] Sloan J. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI (Nutshell Handbooks)*. O'Reilly Media, Inc., November 2004.

[32] Baker J.E. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[33] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, 3rd edition, 1997.

[34] Karl Kopper. *The Linux enterprise cluster*. No Starch Press, San Francisco, 2005.

[35] Davis L., editor. *Handbook of genetic algorithms*. Van Nostrand Reinhold, New York, 1991.

[36] Meek J. L. *Computer Methods in Structural Analysis*. Routledge, New York, NY, 10001, 1991.

[37] Schrage L. A more portable fortran random number generator. *ACM Transactions on mathematical software*, 5(2):132–138, 1979.

[38] Scott Robert Ladd. *Genetic algorithms in C++*. M&T Books, New York, 1996.

[39] Doig A. G. Land A. H. An automatic method for solving discrete programming problems. *econometrica*, 28:497–520, 1960.

[40] Eshelman L.J. and Schaffer J.D. Preventing premature convergence in genetic algorithms by preventing incest. In Belew R. and Booker L.B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 115–122, San Mateo, 1991. Morgan Kaufmann.

[41] Hutter M. Fitness uniform selection to preserve genetic diversity. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 783–788, Manno (Lugano), CH, January 2001. IEEE.

[42] Lozano M., Herrera F., and Cano J.R. Replacement strategies to maintain useful diversity in steady-state genetic algorithms. In *Advances in Soft Computing*, volume 32, pages 85–96. Springer, Berlin / Heidelberg, 2005.

[43] Mitchell A. G. M. The limits of economy of material in frame-structures. *Philosophical Magazine*, 8:589–597, 1904.

[44] E. Maute K., Ramm. General shape optimisation. an integrated model for topology and shape optimisation. In *First World Congress of Structural and Multidisciplinary Optimization*, pages 299–306, Germany, May 28 1995.

[45] Goldberg D.E. Miller B.L. Genetic algorithms, tournament selection and the effect of noise. Technical Report 95006, Univerisity of Illinois at Urbana-Champaign, Department of General Engineering, Urbana, IL, July 1995.

[46] Reddy J. N. *An introduction to the finite element method*. Mc-Graw Hill, New York, 2nd edition, 1993.

[47] Kartam Nabil, Ian Flood, and Garrett James H. Jr. *Artificial neural networks for civil engineers: Fundamentals and Applications*. ASCE, New York, 1997.

[48] Jorge Nocedal and Stephen J.Wright. *Numerical optimizaion*. Springer, 233 Spring Street, New York, NY 10013, USA, 1999.

[49] Banichuk N.V. *Introduction to optimization of structures*. Springer-Verlag, New York, 1990.

[50] Bratley P., Fox L.B., and Schrage L.E. *A guide to simulation (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[51] LEcuyer P. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–751, 1988.

[52] LEcuyer P. Uniform random number generators: A review. In Andradttir S., editor, *Proceedings of the 1997 Winter Simulation Conference*, Piscataway, NJ, 1997. IEEE.

[53] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31(10):1192–1201, 1988.

[54] Rozvany G. I. N. Prager W. Optimization of structural geometry. In Cesarj L. Bednarek A. R., editor, *Dynamical Systems*, pages 265–293. Academic Press, New York, 1977.

[55] Adelman H. M. Pritchard J. I. Differential equation based method for accurate approximation in optimization. In *AIAA ASME ASCE AHS ASC 31st structures, structural dynamics and material conference*, pages 414–424, Long Beach, CA, April 2-4 1990.

[56] Osvaldo M. Querin. *Evolutionary structural optimisation: stress based formulation and implementation*. PhD thesis, Department of Aeronautical Engineering, University of Sydney, Australia, 1997.

[57] Fletcher R. A new approach to variable metric algorithms. *Computer J.*, 13(3):317–322, 1970.

[58] Grandhi R. Structural optimization with frequency constraints - a review. *AIAA J*, 31:2296–303, 1993.

[59] Huges T. J. R. *The finite element method: linear static and dynamic finite element analysis*. Dover, Mineola, New York, 2000.

[60] G.I.N Rozvany. *Structural design via optimality criteria*. Kluwer academic publishers, Dordrecht, 1989.

[61] Kirsh U. Rozvany G. I. N., Bendse M.P. Layout optimization of structures. *Appl. Mech. Rev.*, 48(2):41–119, 1995.

[62] T.L. Saaty. *Optimization in integers and related extremal problems*. McGraw-Hill, New York, 1970.

[63] Yerry M. A. Schephard M. S. Automatic finite element modeling for use with three-dimensional shape optimization. *The Optimimum Shape*, pages 113–135, 1986.

[64] Farsi B. Schmit L. A. Some approximation concepts for structural synthesis. *AIAA Journal*, 12(5):692–699, 1974.

[65] ShiZhen and ZhouYang C.T. Comparison of steady state and elitist selection genetic algorithms. In *Proceedings of the International Conference on Intelligent Mechatronics and Automation*, pages 495 – 499, Aug. 26-31 2004.

[66] Artem Sokolov, Darrell Whitley, and Andre' Motta Salles Barreto. A note on the variance of rank-based selection strategies for genetic algorithms and genetic programming. *Genetic Programming and Evolvable Machines*, 8(3):221–237, 2007.

[67] Raphael T.Haftka and Zafer Gürdal, editors. *Elements of structural optimization*, Dordrecht Boston London, 1993. Kluwer Academic Publishers.

[68] Raphael T.Haftka and Manohar P.Kamat, editors. *Elements of structural optimization*, The Haugue Boston London, 1985. Martinus Nijhoff Publishers, a member of the Kluwer Academic Publishers.

[69] E. Van Keulen F., Hinton. Topology design of plate and shell structures using the hard kill method. *Advances in structural engineering optimization*, pages 137–149, 1996.

[70] G.N. Vanderplaats. *Numerical optimizaion tecniques for engineering*. McGraw-Hill, New York, 1984.

[71] Stadler W. Natural structural shapes of shallow arches. *J. Appl. Mech.*, 44:291–298, 1977.

[72] Stadler W. Natural structural shapes (the static case). *Q. J. Mech. Appl. Math.*, 31:169–217, 1978.

[73] Hemp W.S. *Optimum structures*. Oxford University Press, 1973.

[74] Y.M. Xie and G.P. Steven. *Evolutionary structural optimization*. Springer Verlag, London, limited edition, 1997.

[75] Andreae P. Xie H., Zhang M. Another investigation on tournament selection: modelling and visualisation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1468–1475, New York, NY, USA, 2007. ACM.

[76] Botkin M.E. Yang R.J. A modular approach for three-dimensional shape optimization of structures. *AIAA Journal*, 25(3):492–497, 1987.