Universidade do Porto

FEUP Faculdade de Engenharia

Optimization

Professor Carlos Conceição António

Implementation of Classical Methods and Genetic Algorithms for Optimization

Doctoral Program in Mechanical Engineering

Igor Lopes

Porto, March 2015

Contents

1 Introduction **1**
- 1.1 Optimization . . . 1
- 1.2 Optimization techniques . . . 1
- 1.3 Constitutive parameter identification . . . 2
- 1.4 Work goals . . . 2
- 1.5 Document Layout . . . 3

2 Unidirectional Optimization **5**
- 2.1 Polynomial Approximation . . . 5
- 2.2 Golden Section Method . . . 7
  - 2.2.1 Finding initial bounds . . . 7
- 2.3 Conclusions . . . 7

3 Classical methods for n-dimensional optimization **11**
- 3.1 Powell's method . . . 11
- 3.2 Steepest descent method . . . 12
- 3.3 Conjugate direction methods . . . 12
- 3.4 Newton's method . . . 13
- 3.5 Quasi-Newton methods . . . 14
- 3.6 Unidirectional search in n-dimensional problems . . . 15
- 3.7 Convergence criteria . . . 15
- 3.8 Benchmark example . . . 16
- 3.9 Conclusions . . . 18

4 Genetic Algorithms **25**
- 4.1 Proposed algorithm . . . 25
- 4.2 Encoding variables . . . 25
- 4.3 Fitness evaluation . . . 26
- 4.4 Elitism . . . 27
- 4.5 Reproduction and mutation . . . 27
- 4.6 Stop criteria . . . 27
- 4.7 Application: 2 spring problem . . . 28
- 4.8 Conclusions . . . 29

5 6 spring system equilibrium **31**
- 5.1 Classical algorithms solutions . . . 32
- 5.2 Genetic algorithm solution . . . 33
- 5.3 Conclusions . . . 34

6 Hyperelastic Parameters Identification **37**
- 6.1 Hyperelasticity . . . 37
  - 6.1.1 Ogden's model . . . 37
- 6.2 Micro-mechanical based parameter identification . . . 37
  - 6.2.1 Classical methods solutions . . . 39
  - 6.2.2 Genetic algorithm solution . . . 41
- 6.3 Conclusions . . . 42

7 Conclusions **45**
- 7.1 Main conclusions . . . 45
- 7.2 Future Work . . . 45

A GPROPT source code **47**
A.1 Main routines 47
A.2 Math routines 49
A.3 Polynomial approximation routines 52
A.4 Golden section routines 53
A.5 Classical optimization methods routines 55
A.6 Genetic algorithm routines 59

B Finite Differences for Gradient and Hessian Matrix **63**

References **66**

1 Introduction

1.1 Optimization

Optimization may be defined as the process of finding the best (or optimal) solution for a given problem. This concept is intrinsic to the human quest for achieving better results while minimizing the effort. In spite of being a daily routine task, several strategies have been developed so that this process becomes systematic, which gave rise to *Optimization* as a research field. It is applicable to a broad range of activities, from management point of view, where maximum profit must be obtained minimizing production costs, to engineering activities, such as design a machine that fits all required parameters of power output with a minimum energy consumption. The increase of computational power has allowed to solve more complex optimization problems and the development of new optimization techniques.

The formulation of an optimization problem is usually defined as follows:

> Find the set of n unknown variables $\boldsymbol{x} \in \mathcal{R}^n$, that satisfies the imposed equality and inequality constraints ($\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0}$ and $\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}$), and minimizes the objective function $f(\boldsymbol{x})$.

Three important concepts arise here. Unknown variables, frequently named design variables, are the parameters that one can manipulate to change whatever system response to be optimized. Constraints applied to this variables limit the search space, and may be defined in terms of equality or inequality conditions. In the present work, only side constraints that define the maximum and minimum value for each variable are considered. Finally, the objective function quantifies the quality of a given solution. It is presented as a function to be minimized (cost or distance that a traveller must walk), however for maximizing some function $p(\boldsymbol{x})$, the objective function is simply defined as $f(\boldsymbol{x}) = -p(\boldsymbol{x})$.

1.2 Optimization techniques

Generally, optimization techniques may be divided into three groups: i) mathematical programming, ii) optimality criteria methods and iii) evolutionary algorithms. Mathematical programming methods, commonly known as classical optimization methods, make use of several numerical methods in order to minimize the objective function. Vanderplaats (1984) extensively treat this class of optimization methods. Some of these methods are present and used in the present work. The basic idea is to provide an initial guess for variables vector $\boldsymbol{x}_k$, and iteratively find next solutions with a search direction $\boldsymbol{s}$ and the optimal step length α in that direction:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \boldsymbol{s}. \tag{1.1}$$

Note that a minimum is reached when the gradient becomes null (necessary but not sufficient condition):

$$\nabla f(\boldsymbol{x}_{k+1}) = 0, \tag{1.2}$$

and the Hessian in the point is positive definite. This minimum is the global minimum if the objective function value is the lower that is found in the whole search space, which is difficult to prove. In fact these methods may not converge to the global minimum, since they tend to a minimum on the proximities of the initial guess solution. The strategy to partially overcome this limitation is to start from several initial guess points and compare the results.

Optimality criteria procedures are usually related to structural optimization. In this case an optimality criterion is defined, and the solution is deemed to be found when it is satisfied. This procedures are not treated in the present work, but more detail can be found in Venkayya (1989) and Kirsch (1993).

Finally, with evolutionary algorithms a different approach is considered, in which a population of possible solutions is analysed, instead of only one point per iteration (classical methods). Population evolves with procedures inspired in genetic, and some heuristic is introduced, since these procedures are probabilistic instead of deterministic. This increases the exploratory power, and allows these methods to reach the global minimum, which is an advantage over classical methods. Genetic algorithms, that are based on Darwin's natural selection theory, are the class of evolutionary algorithms studied in this work. Its fundamentals are presented by Chakraborty (2010). More details can be found in the books of Goldberg (1989) and Sakawa (2002), where the versatility of these methods is also shown, since their application ranges from optimization to artificial intelligence and machine learning. Sophistications have been proposed by several authors, such as the presented by Conceição António (2013), aiming to optimize composite structures, where the age of individuals becomes important, and a hierarchical genetic structure is introduced.

In fact, there is an increasing number of researchers that claim the nature is the best place to observe in order to develop adaptive strategies, which can be applied to optimization problems. Besides the already mentioned genetic algorithms, based on natural evolution, there is a class known as memetic algorithms, where population evolves taking into account cultural learning, i.e., based on the social behaviour observed in some animal populations, such as ants, bees or frogs (Serapião, 2009). The trigger for these methods was the book written by Dawkins (1989). In the very recent publication by Parpinelli and Lopes (2015), the subject of modelling an ecosystem, where several species interact, in a computational framework for optimization purposes is reviewed. Another approach, whose interest was recovered by Hopfield (1982), is to simulate the human brain function through neural networks. It also has broad applicability, including the solution of hard optimization problems (Lagoudakis, 1997).

1.3 Constitutive parameter identification

A possible application of optimization methods is constitutive parameter identification. In this case the constitutive parameters are the design variables, and the objective function to be minimized quantifies the difference between the computed constitutive response and the observed response, that can be obtained experimentally, for example.

Stahlschmidt (2010) performs elasto-plastic von Mises parameter identification with both classical and genetic methods, and refers several works where optimization methods are used to identify elasto-plastic, visco-plastic or even damage constitutive models parameters. Iacono (2007) proposes a more sophisticated method that combines a zero-order method (KNN: K-nearest-neighbours) with a technique that takes the optimization problem in a statistical way (Kalman-Filter), due to natural variation of experimental results. A general framework for development of constitutive models is given by Mahnken (2004), where parameter identification subject is well treated.

In section 6 of the present work a problem of hyperelastic parameter identification is presented, and results are obtained with some classical methods and genetic methods. In this case a heterogeneous material is considered, where the response is obtained by numerical homogenization, and Ogden's hyperelastic parameters are aimed to fit this response. A similar task was performed by Speirs (2007).

1.4 Work goals

The main goals of the present work consist in the study and implementation of some classical methods and a genetic algorithm for minimization of functions.

A program written in *Fortran 90* is developed, where all available methods are accessible to the user through the interface. It aims to be a general optimization program, where

more methods can be easily implemented. In what refers to user interface, before running the program one has to define the objective function in a routine/function called `EVALFUNC` (`EVALFUNC1` for the special case of an unidimensional problem). As the program is initiated the user just needs to define the number of variables, the method to be applied and its algorithmic parameters. In the end a result file is available for further analysis. The structure of the program is presented in Box 1.1. In the case of genetic algorithm, objective must be defined in `EVALFUNC` even for unidimensional problems. Source code is available in Appendix A.

Box 1.1 Structure of GPROPT - Generalized PROgram for OPTimization. Some routine names related to each step are indicated.

(i) Define number of design variables n [`MAIN`],

(ii) if $n = 1$, select method for minimizing function defined in `EVALFUNC1`. For polynomial approximation go to (iii), or golden section method go to (iv). If $n > 1$ go to (v),

(iii) choose the degree of the polynomial, give points data [`POLYINT`]. Polynomial coefficients and eventual existing extrema are output. Return to (i),

(iv) find bounds for initial interval, refine it according to the desired tolerance [`GOLDENINT`]. Minimum is output. Return to (i),

(v) choose a method for minimizing the function defined in `EVALFUNC`: one of the classical methods and go to (vi), or the genetic algorithm and go to (vii),

(vi) define side constraints, initial guess and other algorithmic parameters. Evolution of solution is output. Return to (i),

(vii) set side constraints and other algorithmic parameters [`GENETIC`]. Evolution of the fittest individuals is output. Return to (i).

Each of the implemented methods is briefly described throughout this document. Three distinct minimization problems are solved, and the results are discussed. One of those problems is related to constitutive parameters identification. Whenever it is found to be appropriate, suggestions for future work and possible improvements are presented.

1.5 Document Layout

After the brief introduction presented above, aiming to contextualize this work, the structure of this document is presented next.

In section 2, methods for minimization of functions of one variable are introduced, namely polynomial approximation and the golden section method. These methods are important since they are used to obtain the optimum step length in the framework of classical methods for n-dimensional problems. The latter are presented in section 3, where its implementation is also discussed, and a benchmark problem is solved. A genetic algorithm implementation is proposed in section 4, and the same problem is solved in order to assess its performance. A more complex problem, that is related to the equilibrium of a mechanical system, is solved in section 5 using both classical methods and the proposed genetic algorithm. In the beginning of section 6, hyperelastic constitutive models are introduced, in particular the Ogden's model. Then, a problem of constitutive parameter identification is solved, aiming to describe the behaviour of a heterogeneous hyperelastic material with an Ogden's law. Some of the implemented methods

are used, and the obtained results, as well as the performance of the several methods, are discussed.

Finally, main conclusions are summarized in Section 7, where some improvements to the present implementation, and other suggestions for future tasks, are presented.

As it was already referred, the source code of the developed program GPROPT is available in Appendix A. The numerical methods used to compute the gradient and the Hessian matrix of a general function are introduced in Appendix B.

2 Unidirectional Optimization

In this section two algorithms for searching the minimum of a function depending on only one variable are presented. In first place polynomial approximation is introduced, where the function is approximated from a small set of known points. Then, a method based on the iterative split of an initial interval is presented, which is known as the golden section method, due to the use of the golden ratio. These methods are employed in the search of the optimum step length, in a more general optimization problem (with n variables).

2.1 Polynomial Approximation

Consider a general 3$^{\text{rd}}$ degree polynomial:

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3. \tag{2.1}$$

In fact, with this expression one can consider 1$^{\text{st}}$, 2$^{\text{nd}}$ or 3$^{\text{rd}}$ degree approximations. Starting from the simplest case, a first degree approximation, all one need is two known points of the function under interest, or one point and its derivative, in order to determine the coefficients a_0 and a_1 ($a_2 = a_3 = 0$). However, in what refers to finding the minimum or maximum of a function, this approximation is quite useless, since a linear function does not have a maximum or a minimum. For the sake of completeness the polynomial coefficients are presented in (2.2) and (2.3), for two-points and one point approximation respectively:

$$\begin{cases} a_3 &= 0 \\ a_2 &= 0 \\ a_1 &= \dfrac{f_2 - f_1}{x_2 - x_1} \\ a_0 &= f_1 - a_1 x_1 \end{cases}, \tag{2.2}$$

$$\begin{cases} a_3 &= 0 \\ a_2 &= 0 \\ a_1 &= f_1' \\ a_0 &= f_1 - a_1 x_1 \end{cases}. \tag{2.3}$$

A second degree polynomial is suitable for approximate a maximum or minimum of a function. Given three points of the function $(x_i, f_i), i = 1, 3$, the resulting coefficients are given by:

$$\begin{cases} a_3 &= 0 \\ a_2 &= \dfrac{(f_3 - f_1)/(x_3 - x_1) - (f_2 - f_1)/(x_2 - x_1)}{x_3 - x_2} \\ a_1 &= \dfrac{f_2 - f_1}{x_2 - x_1} - a_2(x_1 + x_2) \\ a_0 &= f_1 - a_1 x_1 - a_2 x_1^2 \end{cases}. \tag{2.4}$$

Alternatively, two points and one derivative $(x_i, f_i), i = 1, 2$ and f_1' can be given. In this situation the coefficients are expressed by:

$$\begin{cases} a_3 &= 0 \\ a_2 &= \dfrac{(f_3 - f_1)/(x_3 - x_1) - f_1'}{x_3 - x_2} \\ a_1 &= f_1' - 2a_2 x_1 \\ a_0 &= f_1 - a_1 x_1 - a_2 x_1^2 \end{cases}. \tag{2.5}$$

For a cubic approximation the same thought is considered: it can be defined from a set of four points or from three known points and one derivative. In the first case the polynomial coefficients are:

$$\begin{cases} a_3 &= \dfrac{q_3 q_6 - q_4 q_5}{q_2 q_3 - q_1 q_4} \\ a_2 &= \dfrac{q_5 - a_3 q_1}{q_3} \\ a_1 &= \dfrac{f_2 - f_1}{x_2 - x_1} - a_3 \dfrac{x_2^3 - x_1^3}{x_2 - x_1} - a_2(x_1 + x_2) \\ a_0 &= f_1 - a_1 x_1 - a_2 x_1^2 - a_3 x_1^3 \end{cases}, \tag{2.6}$$

with:

$$\begin{cases} q_1 &= x_3^3(x_2 - x_1) - x_2^3(x_3 - x_1) + x_1^3(x_3 - x_2) \\ q_2 &= x_4^3(x_2 - x_1) - x_2^3(x_4 - x_1) + x_1^3(x_4 - x_2) \\ q_3 &= (x_3 - x_2)(x_2 - x_1)(x_3 - x_1) \\ q_4 &= (x_4 - x_2)(x_2 - x_1)(x_4 - x_1) \\ q_5 &= f_3(x_2 - x_1) - f_2(x_3 - x_1) + f_1(x_3 - x_2) \\ q_6 &= f_4(x_2 - x_1) - f_2(x_4 - x_1) + f_1(x_4 - x_2) \end{cases}. \tag{2.7}$$

In the latter, the coefficients are defined as follows:

$$\begin{cases} a_3 &= \dfrac{f_3 - f_1}{(x_3 - x_2)(x_3 - x_1)^2} - \dfrac{f_2 - f_1}{(x_3 - x_2)(x_2 - x_1)^2} + \dfrac{f_1'}{(x_2 - x_1)(x_3 - x_1)} \\ a_2 &= \dfrac{(f_2 - f_1)/(x_2 - x_1) - f_1'}{x_2 - x_1} - a_3(2x_1 + x_2) \\ a_1 &= f_1' - 2a_2 x_1 - 3a_3 x_1^2 \\ a_0 &= f_1 - a_1 x_1 - a_2 x_1^2 - a_3 x_1^3 \end{cases}. \tag{2.8}$$

The stationary point of a function is defined by $f'(x) = 0$. In the case of a quadratic function, this point is given in terms of the coefficients by (2.9), where a minimum is found if $a_2 > 0$, or if $a_2 < 0$ there is a maximum:

$$x^* = -\frac{a_1}{2a_2}. \tag{2.9}$$

For a cubic polynomial, two stationary points may exist, defined by:

$$x^* = \frac{-a_2 + \sqrt{b}}{3a_3} \vee x^* = \frac{-a_2 - \sqrt{b}}{3a_3}, \tag{2.10}$$

where:

$$b = a_2^2 - 3a_1 a_3. \tag{2.11}$$

If $b > 0$ there are two real stationary points, where the first corresponds to a minimum and the second to a maximum. On the other hand, for $b < 0$ it results in complex conjugate values, i.e., no real maximum or minimum exists. Finally, for $b = 0$ there is only one stationary point that is neither a minimum nor a maximum.

The implementation of polynomial approximation strategy is presented in Appendix A.3

2.2 Golden Section Method

A different approach is introduced with the golden section method. Given an initial interval of values of x, being the minimum (or maximum) in this interval, it is iteratively reduced such that its bounds tend to become closer to the value of the stationary point. It is relatively robust but the number of function evaluations is much larger than the required for polynomial approximation. At each iteration the interval is reduced according to the golden ratio:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618034. \tag{2.12}$$

Given the lower and upper bounds x_l and x_u, two new points are evaluated:

$$x_1 = x_u - \tau(x_u - x_l) \tag{2.13a}$$
$$x_2 = x_l + \tau(x_u - x_l), \tag{2.13b}$$

with $\tau = 1/\phi \approx 0.618034$.

Since the process for optimization usually implies the minimization of a function, this is the considered framework. If it is found that $f_1 < f_2$ then x_2 defines the new upper bound, x_1 is updated through (2.13a) and the new point x_2 corresponds to the previous x_1, thanks to the golden ratio. This particularity avoids further function evaluations. An equivalent sequence is applied if $f_1 > f_2$, where the lower bound is updated to x_1, the new x_1 is the previous x_2 and x_2 is given by (2.13b). A special case is found when $f_1 = f_2$, what results in cutting the two sides of the interval: new lower bound is x_1 upper bound is x_2, and new interior points are obtained with (2.13). The algorithm is summarized in Box 2.1.

The required number of iterations depends on the desired precision, that can be defined in a relative form as:

$$\varepsilon = \frac{(x_u - x_l)^{final}}{(x_u - x_l)^{initial}}. \tag{2.14}$$

Since in each iteration the interval amplitude is reduced to $\tau(x_u - x_l)$, thus the relative precision can be expressed as:

$$\varepsilon = \tau^N, \tag{2.15}$$

and the number of iteration is easily defined.

The routines related to the implementation of this method can be found in Appendix A.4.

2.2.1 Finding initial bounds

Consider a real function $f(x)$, with one existing minimum in its domain, but its position x_{min} is completely unknown. The definition of the initial boundaries is not easy in this context, which gives rise to the application of an automatic method for finding them.

Given an initial point x_0 and its function value f_0, lets assume that this value is the lower bound, and the upper bound is $x_u = x_0 + \Delta x$, where Δx depends on the specific problem. Comparing f_u and f_l, the bounds are updated until the minimum is caught inside the interval. The strategy is defined in Box 2.2.

2.3 Conclusions

Two distinct algorithms for finding the minimum of a function of one variable are presented in this section. In first place, all ingredients for polynomial approximations are introduced. The degree of the polynomial must be carefully chosen. A first degree approximation does not make sense in the search for stationary points. On the other hand, a third degree polynomial may introduce numerical problems if the given information leads to non-real extrema. In the

Box 2.1 Golden-Section algorithm for finding the minimum of a function.

Given the initial bounds and respective function values x_l, x_u, f_l and f_u, and the desired number of iterations N, :

(i) initialize counter $i = 0$, obtain x_1 and x_2 with (2.13), and evaluate f_1 and f_2;

(ii) do $i = i + 1$, if $i > N$ finish process;

(iii) if $f_1 < f_2$ then:

$$\begin{aligned} x_u &= x_2, \quad f_u = f_2, \\ x_2 &= x_1, \quad f_2 = f_1, \\ x_1 &= x_u - \tau(x_u - x_l), \quad \text{evaluate } f_1, \end{aligned}$$

and go to (ii);

(iv) if $f_1 > f_2$ then:

$$\begin{aligned} x_l &= x_1, \quad f_l = f_1, \\ x_1 &= x_2, \quad f_1 = f_2, \\ x_2 &= x_l + \tau(x_u - x_l), \quad \text{evaluate } f_2, \end{aligned}$$

and go to (ii);

(v) if $f_1 = f_2$ then:

$$\begin{aligned} x_l &= x_1, \quad f_l = f_1, \\ x_u &= x_2, \quad f_u = f_2, \\ x_1 &= x_u - \tau(x_u - x_l), \quad \text{evaluate } f_1, \\ x_2 &= x_l + \tau(x_u - x_l), \quad \text{evaluate } f_2, \end{aligned}$$

and go to (ii).

vicinity of an extrema, a function usually presents a behaviour close to a quadratic function, thus, a second degree polynomial approximation seems to be a wise choice. However, note that the obtained coefficients strongly depend on the given points, and found extrema point should be verified. The golden section method is more robust, but requires a higher number of function evaluations. This is not a problem if function value can be obtained through analytical expressions, since it is computationally inexpensive, but if a finite element problem must be solved for every function evaluation, computation time and effort increase.

For the examples presented in this work golden section method is used, as function evaluations do not require heavy computations.

However, as a suggestion, the combination of these two methods may result in a wise solution for general problems. The strategy would start by reducing the initial interval using only a few iterations, and in a second phase, as functions tend to be nearly quadratic near minima, use a quadratic approximation to find the minimum. This could reduce the number of function evaluations, without affecting significantly the robustness of the algorithm.

Box 2.2 Algorithm for finding the bounds that enclose the minimum of a function.

Given the initial guess x_0 and its function value f_0:

(i) assume $x_l = x_0$, $f_l = f_0$, $x_u = x_0 + \Delta x$ and evaluate f_u,

(ii) if $f_u > f_l$, the upper bound is found, but the lower is not, thus go to (v),

(iii) update upper bound:

$$x_1 = x_u, \quad f_1 = f_u,$$
$$x_u = x_1 + \phi(x_1 - x_l), \quad \text{evaluate } f_u,$$

(iv) if $f_u > f_1$ then the bounds are found, and the process is finished.
Else, update lower bound:

$$x_l = x_1, \quad f_l = f_1,$$

and go to (iii),

(v) initialize lower bound as $x_l = x_0 - \Delta x$, and evaluate f_l,

(vi) if $f_l > f_0$, lower bound is found, and the process is finished,

(vii) update lower bound:

$$x_1 = x_l, \quad f_1 = f_l,$$
$$x_l = x_1 - \phi(x_u - x_1), \quad \text{evaluate } f_l,$$

(viii) if $f_l > f_1$ then the bounds are found, and the process is finished.
Else, update upper bound:

$$x_u = x_1, \quad f_u = f_1,$$

and go to (vii).

3 Classical methods for n-dimensional optimization

Consider now the problem of the minimization of a function of n variables $f(\boldsymbol{x})$, where $\boldsymbol{x}$ is the vector of unknowns (design variables).

The general framework for classical optimization methods is presented. The procedure starts with an initial guess point $\boldsymbol{x}_0$, that is iteratively updated towards a minimum of the function. At each iteration, a search direction $\boldsymbol{s}$ is determined, which is the vector that gives the direction of the displacement of the previous point in the design space. The magnitude of the displacement should be the one that minimizes the function in that direction. It can be found introducing the step length α, such that the new iterative solution is given by:

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{s}_k. \tag{3.1}$$

The search direction can be determined in distinct ways, according to the chosen method. The optimum step length is found applying the concept of unidirectional optimization introduced in section 2, where α_k is the unknown.

These methods can be classified into zero-order methods if they only use function values, first-order methods if gradient information is considered, or second-order methods if the Hessian matrix is evaluated for the computation of search direction.

A detailed description of this class of methods is presented in Vanderplaats (1984). The implementation of the following methods can be found in Appendix A.5.

After introducing the implemented methods, a benchmark problem is solved to assess the correctness of the implementation.

3.1 Powell's method

Powell's method is a very popular zero-order method, due to its efficiency and reliability, that is based on the concept of conjugate directions. Two directions $\boldsymbol{s}^i$ and $\boldsymbol{s}^j$ are conjugated if the following statement is satisfied:

$$\left(\boldsymbol{s}^i\right)^T \boldsymbol{H} \boldsymbol{s}^j, \tag{3.2}$$

where $\boldsymbol{H}$ denotes the Hessian matrix (matrix of second derivatives).

With this method the Hessian matrix is not computed, but approximate conjugate directions are obtained. In first place, a search in n orthogonal directions ($\boldsymbol{s}^i = \boldsymbol{e}_i, i = 1, n$) is performed sequentially ($\boldsymbol{e}_i$ is the unit vector in direction i). The directions are stored in a matrix $\boldsymbol{H}$, with the respective vectors in columns, thus in the first iteration, this matrix assumes the identity form $\boldsymbol{H} = \boldsymbol{I}$. During the search in each direction, the optimum step length α_i is stored, and the respective column is updated as $\alpha_i \boldsymbol{s}^i$. In the end of the iteration, the matrix has the form:

$$\boldsymbol{H} = \begin{bmatrix} \alpha_1 \boldsymbol{s}^1 & \alpha_2 \boldsymbol{s}^2 & \alpha_3 \boldsymbol{s}^3 & \dots & \alpha_n \boldsymbol{s}^n \end{bmatrix}. \tag{3.3}$$

After this stage, a conjugate search direction approximation is computed as:

$$\boldsymbol{s}^{n+1} = \sum_{i=1}^{n} \alpha_i \boldsymbol{s}^i, \tag{3.4}$$

which is the sum of all columns of $\boldsymbol{H}$. The step search α_{n+1} must be found, and then $\boldsymbol{H}$ matrix is updated shifting each column one to the left, and introducing $\alpha_{n+1}\boldsymbol{s}^{n+1}$ in the last one. With the new system of conjugate directions the process is repeated in a new iteration. The algorithm is briefly presented in Box 3.1. Two problems may arise when using this method. In first place, if in some direction $\alpha_i = 0$, then the next search directions are not conjugate. Another issue is that due to the non-quadratic form of the function or to numerical

Box 3.1 Powell's Method algorithm for finding the minimum of a function.

Given the initial guess $\boldsymbol{x}^*$, its dimension n and the maximum number of iterations m, initialize $\boldsymbol{H} = \boldsymbol{I}$ and $k = 0$:

(i) update iteration counter $k = k + 1$, finish process if $k > m$ or initialize $i = 0$ if this is not verified,

(ii) do $i = i + 1$, if $i > N$ go to (vi),

(iii) get search direction $\boldsymbol{s}^i$ as the i-th column of $\boldsymbol{H}$,

(iv) minimize the function in direction $\boldsymbol{s}^i$, finding step length α_i,

(v) update the respective column of $\boldsymbol{H}$ with $\alpha_i \boldsymbol{s}^i$, and $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_i \boldsymbol{s}^i$. Go to (ii),

(vi) compute new conjugate direction $\boldsymbol{s}^{n+1} = \sum_{i=1}^{n} \alpha_i \boldsymbol{s}^i$, and optimize objective function in this direction, updating solution $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_{n+1} \boldsymbol{s}^{n+1}$,

(vii) if convergence criteria is satisfied, finish process with solution $\boldsymbol{x}^*$. Else, shift each column of $\boldsymbol{H}$ one to the left, update the last one with $\alpha_{n+1} \boldsymbol{s}^{n+1}$ and go to (i).

imprecisions the conjugate direction vectors may become parallel. The simplest solution is to restart the procedure with $\boldsymbol{H} = \boldsymbol{I}$.

With this method, each iteration requires $n+1$ unidirectional optimizations. It also has the particularity that if the function is quadratic, then it will be optimized in n or fewer iterations. In the case of higher-order functions, convergence will be slower.

3.2 Steepest descent method

This method is based on the fact that in a point $\boldsymbol{x}^*$ of the function, the gradient $\nabla f(\boldsymbol{x}^*)$ (vector with first derivatives) gives the direction on which the function value increases with higher rate. It is straightforward that the direction opposite to the gradient is the one that returns the maximum decrease of its value. This explains the designation of the present method. The direction search in each iteration is simply given by:

$$\boldsymbol{s}^i = -\nabla f(\boldsymbol{x}_i). \tag{3.5}$$

Since it involves the determination of the gradient, this is a first-order method. Compared with the Powell's method, the steepest descent only requires one unidirectional search per iteration, but the determination of the gradient vector must be implemented. If it can be defined analytically, this is not a problem, however, if it requires the use of finite differences method (Appendix B), the number of function evaluations increases dramatically. The algorithm is presented in Box 3.2.

Despite its popularity and simplicity, it is the method with worst performance in the framework of first-order methods, since it does not use information of previous iterations. It is however a good choice for initializing other methods such as the conjugate direction method.

3.3 Conjugate direction methods

This first order method can be seen as a sophistication of the steepest descent method, introducing a modification that remarkably increases the performance. In this case, the search direction is given by:

$$\boldsymbol{s}^i = -\nabla f(\boldsymbol{x}_i) + \beta_i \boldsymbol{s}^{i-1}, \tag{3.6}$$

Box 3.2 Steepest Descent algorithm for finding the minimum of a function.

Given the initial guess $\boldsymbol{x}^*$, its dimension n and the maximum number of iterations m, initialize $i = 0$:

(i) update iteration counter $i = i + 1$, finish process if $k > m$,

(ii) compute the gradient at the present solution $\nabla f(\boldsymbol{x}^*)$, and define search direction as $\boldsymbol{s}^i = -\nabla f(\boldsymbol{x}^*)$,

(iii) perform unidirectional optimization, finding α_i and updating solution $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_i \boldsymbol{s}^i$,

(iv) if convergence criteria is satisfied, finish process with solution $\boldsymbol{x}^*$. Else, go to (i).

where, comparing with the previous method, some information of the previous iteration is introduced. The scalar β_i may be defined according to distinct authors suggestions. In the present work the definitions of Fletcher-Reeves (3.7) and Polak-Ribiere (3.8) are implemented:

$$\beta_i^{FR} = \frac{|\nabla f(\boldsymbol{x}_i)|^2}{|\nabla f(\boldsymbol{x}_{i-1})|^2}, \tag{3.7}$$

$$\beta_i^{PR} = \frac{|\nabla f(\boldsymbol{x}_i)|^2 - \nabla f(\boldsymbol{x}_i) \cdot \nabla f(\boldsymbol{x}_{i-1})}{|\nabla f(\boldsymbol{x}_{i-1})|^2}. \tag{3.8}$$

Note that the importance of the previous direction increases with bigger β_i values. In the first iteration, once there is no previous information, the concept of steepest descent is applied.

The process must be restarted when the obtained direction does not lead to an improvement of the solution, i.e., if:

$$\frac{df(\alpha_i)}{d\alpha_i} = \nabla f(\boldsymbol{x}_i) \cdot \boldsymbol{s}^i \geq 0, \tag{3.9}$$

then the search direction is reset to $\boldsymbol{s}^i = -\nabla f(\boldsymbol{x}_i)$. The algorithm is presented in Box 3.3.

Once this is a conjugate direction based method, in the case of a quadratic function the process minimizes the function in n or fewer iterations.

3.4 Newton's method

This second-order method requires the computation of the Hessian matrix, besides the gradient. The Hessian matrix is the matrix that contains second-order derivatives, defined as:

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}. \tag{3.10}$$

These derivatives may be obtained analytically or with the finite difference method (see Appendix B).

Starting with the second-order Taylor approximation:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^i) + \nabla f(\boldsymbol{x}^i) \cdot \delta\boldsymbol{x} + \frac{1}{2}\delta\boldsymbol{x} \cdot \boldsymbol{H}(\boldsymbol{x}^i)\delta\boldsymbol{x}, \tag{3.11}$$

the stationary condition leads to:

$$\nabla f(\boldsymbol{x}) \approx \nabla f(\boldsymbol{x}^i) + \boldsymbol{H}(\boldsymbol{x}^i)\delta\boldsymbol{x} = 0, \tag{3.12}$$

thus the iterative process is defined by:

$$\boldsymbol{x}^{i+1} = \boldsymbol{x}^i - \boldsymbol{H}^{-1}(\boldsymbol{x}^i)\nabla f(\boldsymbol{x}^i). \tag{3.13}$$

Box 3.3 Conjugate direction algorithm for finding the minimum of a function.

Given the initial guess $\boldsymbol{x}^*$, its dimension n and the maximum number of iterations m, initialize $i = 0$ and:

(i) compute the gradient at the initial guess $\nabla f(\boldsymbol{x}^*)$ and define search direction as

$$\boldsymbol{s}^1 = -\nabla f(\boldsymbol{x}^*),$$

(ii) update iteration counter $i = i + 1$, finish process if $k > m$,

(iii) perform unidirectional optimization, finding α_i and update solution $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_i \boldsymbol{s}^i$,

(iv) if convergence criteria is satisfied, finish process with solution $\boldsymbol{x}^*$,

(v) compute the gradient at the present solution $\nabla f(\boldsymbol{x}^*)$,

(vi) compute β_{i+1} with (3.7) or (3.8) and define search direction as $\boldsymbol{s}^{i+1} = -\nabla f(\boldsymbol{x}^*) + \beta_{i+1}\boldsymbol{s}^i$,

(vii) if $\nabla f(\boldsymbol{x}^*) \cdot \boldsymbol{s}^{i+1} \geq 0$ then $\boldsymbol{s}^{i+1} = -\nabla f(\boldsymbol{x}^*)$,

(viii) go to (ii).

In the situation of a quadratic function, then the minimum is found in just one iteration. Note that in the vicinity of a minimum a function usually has a quadratic form, therefore, with this technique and a relatively good initial guess, the solution will converge rapidly to the minimum, even if the function is not quadratic.

The search direction is expressed by:

$$\boldsymbol{s}^i = -\boldsymbol{H}^{-1}(\boldsymbol{x}^i)\nabla f(\boldsymbol{x}^i). \tag{3.14}$$

Unidirectional optimization may be used to find the step length α_i, but $\alpha_i = 1$ is a good solution especially when the solution is close to the minimum.

Some problems may appear if in some point the Hessian matrix becomes nearly singular, which happens if the objective function is approximately linear in one or more variables. In practice this would lead to an unstable search direction, i.e., a vector with huge components. This problem is shortcuted with the imposition of side constraints , that limit the value of the step length (see section 3.6). Newton's method algorithm is summarized in Box 3.4.

3.5 Quasi-Newton methods

Quasi-Newton methods are first order methods that aim to follow the procedure of Newton method, but do not compute the Hessian matrix directly. They try to approximate it instead. The search direction is given by:

$$\boldsymbol{s}^i = -\boldsymbol{M}\nabla f(\boldsymbol{x}^i), \tag{3.15}$$

where matrix $\boldsymbol{M}$ is an approximation to the inverse of the Hessian matrix. The convergence rate approaches to the attained with the Newton's method, thus these methods are a good alternative for problems where the Hessian matrix is difficult or too expensive to obtain. In the first iteration the identity matrix is used ($\boldsymbol{M}^0 = \boldsymbol{I}$), which means that the direction of

Box 3.4 Newton's method algorithm for finding the minimum of a function.

Given the initial guess $\boldsymbol{x}^*$, its dimension n and the maximum number of iterations m, initialize $i = 0$ and:

(i) compute the gradient $\nabla f(\boldsymbol{x}^*)$ and Hessian matrix $\boldsymbol{H}(\boldsymbol{x}^*)$ at the initial guess ,

(ii) update iteration counter $i = i + 1$, finish process if $k > m$,

(iii) compute search direction: $\boldsymbol{s}^i = -\boldsymbol{H}^{-1}(\boldsymbol{x}^i)\nabla f(\boldsymbol{x}^i)$,

(iv) perform unidirectional optimization, finding α_i and update solution $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_i \boldsymbol{s}^i$,

(v) if convergence criteria is satisfied, finish process with solution $\boldsymbol{x}^*$,

(vi) compute the gradient $\nabla f(\boldsymbol{x}^*)$ and Hessian matrix $\boldsymbol{H}(\boldsymbol{x}^*)$ at the present solution,

(vii) go to (ii).

steepest descent is considered. Then it is updated according to the following formulae:

$$\boldsymbol{M}^{i+1} = \boldsymbol{M}^i + \boldsymbol{D}^i \tag{3.16a}$$

$$\boldsymbol{D}^i = \frac{\sigma + \theta\tau}{\sigma^2}\boldsymbol{p}\boldsymbol{p}^T + \frac{\theta - 1}{\tau}\boldsymbol{M}^i\boldsymbol{y}\left(\boldsymbol{M}^i\boldsymbol{y}\right)^T - \frac{\theta}{\sigma}\left[\boldsymbol{M}^i\boldsymbol{y}\boldsymbol{p}^T + \boldsymbol{p}(\boldsymbol{M}^i\boldsymbol{y})^T\right] \tag{3.16b}$$

$$\boldsymbol{p} = \boldsymbol{x}^i - \boldsymbol{x}^{i-1} \tag{3.16c}$$

$$\boldsymbol{y} = \nabla f(\boldsymbol{x}^i) - \nabla f(\boldsymbol{x}^{i-1}) \tag{3.16d}$$

$$\sigma = \boldsymbol{p} \cdot \boldsymbol{y} \tag{3.16e}$$

$$\tau = \boldsymbol{y}^T\boldsymbol{M}^i\boldsymbol{y}. \tag{3.16f}$$

The scalar θ may assume distinct values, defining the family of variable metric methods. For $\theta = 0$ the method is called DFP (Davidon-Fletcher-Powell) method, whereas with $\theta = 1$ it is known as BFGS (Broydon-Fletcher-Goldfarb-Shanno) method.

Note that in this case information of previous iterations is introduced by a matrix, in contrast to the conjugate direction method that uses a scalar parameter, which results in better performances.

The algorithm developed to implement this method is presented in Box 3.5.

3.6 Unidirectional search in n-dimensional problems

The procedure of unidirectional search in problems with n design variables is an extension of the methods introduced in section 2. In the present work, the golden section method is the chosen to find the step length α_i that minimizes $f(\boldsymbol{x}_{i-1} + \alpha_i \boldsymbol{s}^i)$. Polynomial approximations strongly depend on the chosen points, despite the advantage that only few function evaluations are required. However, the problems analysed here do not involve expensive function evaluations. The algorithm is summarized in Boxes 3.6 and 3.7.

3.7 Convergence criteria

It is necessary to define and implement convergence criteria so that the process finishes when a sufficiently good solution is found or when a determined computational effort is reached. The latter is simply materialized setting a maximum number of iterations. The task of defining when a good solution is found is not so direct. Two approaches may be considered: a

Box 3.5 Quasi-Newton methods algorithm for finding the minimum of a function.

Given the initial guess $\boldsymbol{x}^*$, its dimension n, the value of θ and the maximum number of iterations m, initialize $i = 0$ and:

(i) compute the gradient $\nabla f(\boldsymbol{x}^*)$ at the initial guess and initialize $\boldsymbol{M} = \boldsymbol{I}$,

(ii) update iteration counter $i = i + 1$, finish process if $k > m$,

(iii) compute search direction $\boldsymbol{s}^i = -\boldsymbol{M}^i \nabla f(\boldsymbol{x}^i)$,

(iv) perform unidirectional optimization, finding α_i and update solution $\boldsymbol{x}^* = \boldsymbol{x}^* + \alpha_i \boldsymbol{s}^i$,

(v) if convergence criteria is satisfied, finish process with solution $\boldsymbol{x}^*$,

(vi) compute the gradient $\nabla f(\boldsymbol{x}^*)$ and update $\boldsymbol{M}$ with (3.16),

(vii) go to (ii).

new iteration does not improve the solution, or the gradient is nearly a null vector, which approximates the necessary condition for finding a stationary point (see (1.2)).

In what refers to zero-order methods, since the gradient is not evaluated, the convergence criteria must rely on the evolution of the objective function. In fact, this is the approach implemented in the present work, for all classical methods. A convergence tolerance ε_C is defined, and the process is finished when:

$$\Delta f^* \leq \varepsilon_C, \tag{3.19}$$

where Δf^* denotes the relative variation of the objective function:

$$\Delta f^* = \begin{cases} \dfrac{|f_i - f_{i-1}|}{|f_{i-1}|}, & \text{if } |f_{i-1}| > 1 \times 10^{-6} \\ |f_i - f_{i-1}|, & \text{else.} \end{cases} \tag{3.20}$$

3.8 Benchmark example

For the sake of verification of implemented optimization methods, in this section the problem of the equilibrium of a system with two springs is solved. It is presented in the beginning of Chapter 3 of Vanderplaats (1984), and it is revisited here. In Figure 3.1 this mechanical problem is schematically represented. Given the properties of the several components, and the applied forces, the equilibrium position must be found.

This problem can be seen as an optimization problem since the equilibrium position is the one that minimizes the potential energy, being the position defined by x_1 and x_2. The potential energy function is expressed by:

$$f = \frac{k_1}{2}\left[\sqrt{x_1^2 + (l_1 - x_2)^2} - l_1\right]^2 + \frac{k_2}{2}\left[\sqrt{x_1^2 + (l_2 - x_2)^2} - l_2\right]^2 - P_1 x_1 - P_2 x_2. \tag{3.21}$$

With the available data it can be easily plotted in the space of unknown variables (x_1, x_2), which is done in Figure 3.2. The equilibrium position is at $x_1 = 8.631$ cm, $x_2 = 4.533$ cm, where the potential energy value is -41.81 Ncm.

Side constraints are imposed to both variables so that $-12 \leq x_1, x2 \leq 12$. In the present implementation this is embedded in the algorithm for finding bounds of step length α_i, limiting it to a maximum and minimum value (see Box 3.7). The refinement of these bounds is

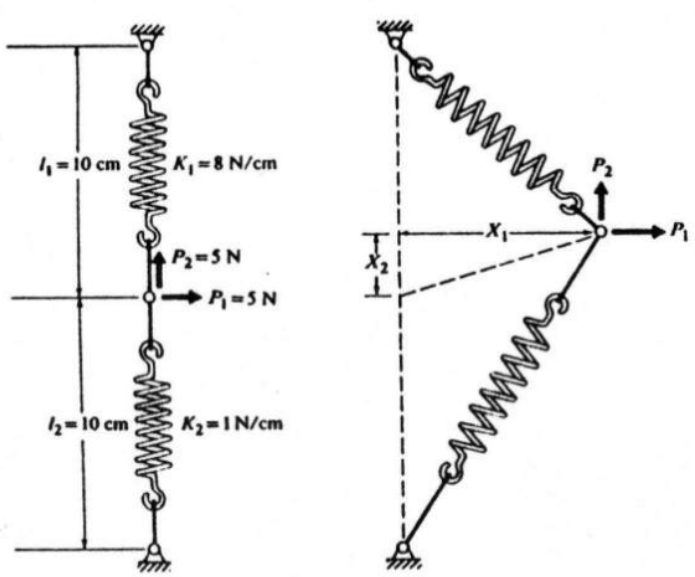


Figure 3.1: Representation of the 2 spring problem (Vanderplaats, 1984).

performed with the golden section method, until their amplitude reaches less than 1% of the initial counterpart. The initial guess for starting the search is $x_1 = -4$ and $x_2 = 4$ cm, and a maximum of 50 iterations is allowed. The tolerance for convergence is $\varepsilon_C = 1 \times 10^{-6}$.

The evolution of the solution, obtained with the several methods, is shown in Tables 3.1 to 3.7. For easier comparison, in Figure 3.3 the evolution of objective function is represented plotted.

Note that Powell's method converges in 5 iterations, but it corresponds to 15 unidirectional searches.

Table 3.1: Powell's method results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-6.3801	4.0000	32.5694	-2.3801
	-6.3801	2.5885	27.6108	-1.4115
	-5.8081	2.9276	25.9860	-0.2403
2	-5.8081	2.2114	24.5101	0.5074
	-5.2584	2.5374	23.1023	-0.2310
	-1.3897	-0.2086	8.3641	7.0369
3	9.0725	5.9962	-40.1204	19.0300
	9.3413	5.8055	-40.5629	0.4888
	8.9642	5.5941	-40.8947	-0.0351
4	9.1655	5.4512	-41.1461	0.7491
	8.8724	5.2870	-41.3342	-0.0273
	8.6500	4.5428	-41.8077	2.4228
5	8.6324	4.5329	-41.8082	0.0600
	8.6321	4.5320	-41.8082	0.0030
	8.6320	4.5319	-41.8082	0.0073

With the steepest descent method, the objective function decreases relatively fast in the initial iterations, but this convergence rate decreases as it approximates to the minimum.

In the case of the Fletcher-Reeves conjugate direction method, the original implementation

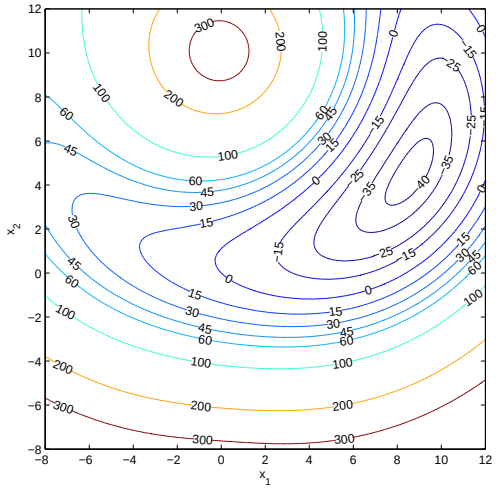


Figure 3.2: Potential energy function [Ncm] in terms of the position variables for the 2 spring problem.

led to a very weak convergence rate. Thus, a simple modification is proposed. The new search direction is now computed as:

$$\boldsymbol{s}^i = -\nabla f(\boldsymbol{x}_i) + \beta_i \alpha_i \boldsymbol{s}^{i-1}. \tag{3.22}$$

A comparison of the objective function evolution with both versions of the algorithm is plotted in Figure 3.4. In practice this modification creates a search direction that is closer to the steepest descent direction. However, it was later verified that for different initial points, the original version may perform much better, whereas with the proposed modification the convergence is always very similar to the steepest descent one. This means that the original Fletcher-Reeves algorithm is not so robust, and the proposed modification is not worthy since an improvement of the convergence is not achieved comparing to the steepest descent.

The Polak-Ribiere conjugate direction algorithm has proven to be more robust and efficient. In fact its convergence history is very close to the quasi-Newton methods (DFP and BFGS).

The method that has shown best performance is the Newton's method, as expected. It has reached the minimum in only 7 iterations. It is remarkable that as the solution becomes closer to the minimum, the step length tend to 1, which confirms that near the minimum a quasi-quadratic behaviour is observed.

3.9 Conclusions

Several classical optimization methods are presented in this section, and their performance is evaluated through the solution of a simple minimization problem. It became evident that, generally, the more sophistication is introduced, the better performance is attained. In all cases the expected result is achieved with good accuracy.

The best performance was observed with the Newton's method, where second order derivatives information is used. The zero order Powell's method needs a higher number of unidirectional searches, but does not require derivatives evaluation. In what refers to first order methods, the steepest descent method does not show good convergence rate when compared to other of the same class. Even with Powell's method the number of unidirectional searches is lower.

Conjugate direction methods have shown an improvement, but the Fletcher-Reeves method performance appears to be dependent on the initial point, whereas the Polak-Ribiere proved

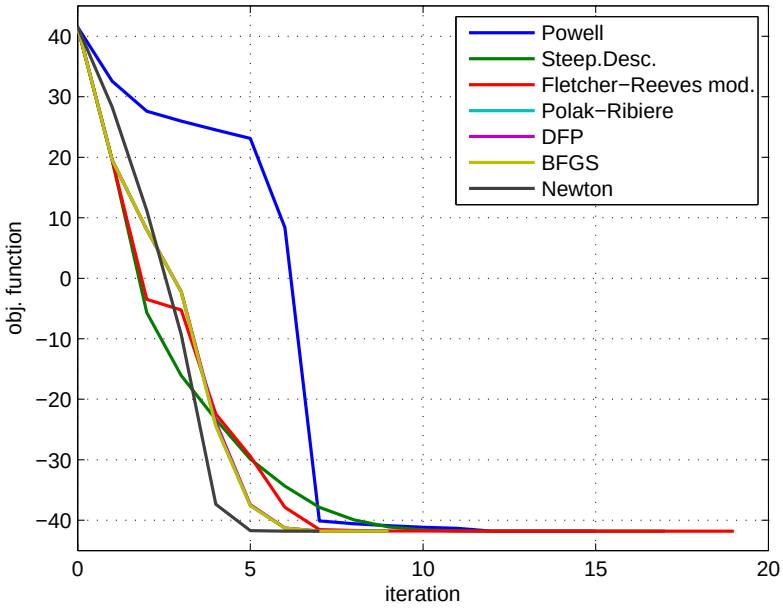


Figure 3.3: A comparison between the evolution of objective function value obtained with classical methods. For Powell's method it is plotted against number of unidirectional searches.

to be robust and with a convergence almost coincident with quasi-Newton methods, for this specific example. A modification is proposed to the Fletcher-Reeves search direction, which results in a vector very close to the steepest descent direction. Therefore, this modification does not actually improve the process. The quasi-Newton DFP and BFGS methods are very efficient, converging in 9 iterations (only more 2 than the Newton's method). Differences between DFP and BFGS results are almost negligible.

Table 3.2: Steepest descent method results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-4.7433	1.8211	19.5463	0.1214
2	2.1423	-0.5393	-5.6909	1.7972
3	2.7308	1.1683	-16.0899	0.1523
4	4.5846	0.5252	-23.3481	0.2883
5	5.1643	2.1962	-29.8897	0.2249
6	6.3487	1.7867	-34.3579	0.1803
7	6.8810	3.3279	-37.8574	0.3480
8	7.6069	3.0776	-39.9436	0.1426
9	7.9774	4.1384	-41.1090	0.5047
10	8.3110	4.0215	-41.5939	0.1290
11	8.4577	4.4393	-41.7522	0.6035
12	8.5556	4.4049	-41.7953	0.1253
13	8.5945	4.5135	-41.8055	0.6465
14	8.6164	4.5056	-41.8077	0.1243
15	8.6245	4.5282	-41.8081	0.6537
16	8.6288	4.5267	-41.8082	0.1206
17	8.6318	4.5322	-41.8082	0.8753

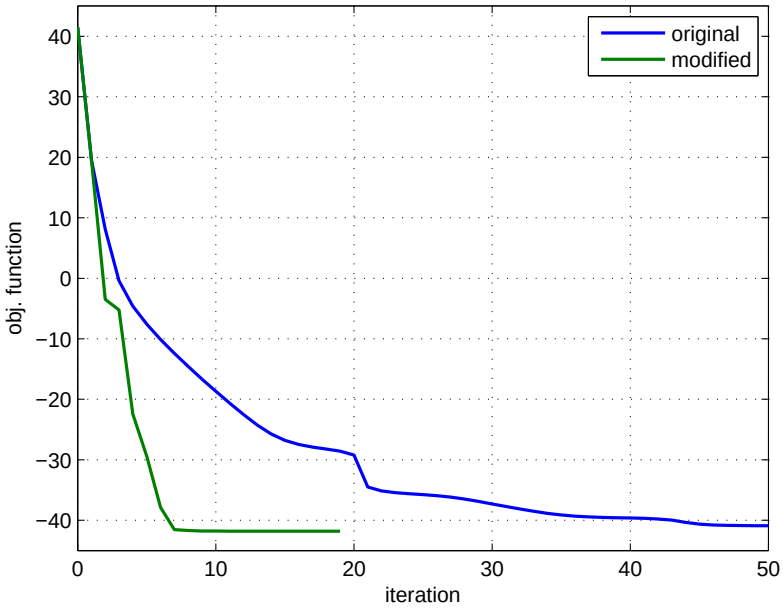


Figure 3.4: A comparison between the evolution of objective function value with original and modified Fletcher-Reeves method.

Box 3.6 Golden-Section algorithm for finding the minimum of a function of n variables.

Given the present solution $\boldsymbol{x}^*$, its dimension n and function value f_0, search direction $\boldsymbol{s}^i$ and number of iterations N:

(i) find bounds α_l and α_u (algorithm in Box 3.7) and respective function values f_l and f_u,

(ii) compute α_1 and α_2 as:

$$\alpha_1 = \alpha_u - \tau(\alpha_u - \alpha_l)$$
$$\alpha_2 = \alpha_l + \tau(\alpha_u - \alpha_l),$$

and compute respective function values:

$$f_1 = f(\boldsymbol{x}^* + \alpha_1 \boldsymbol{s}^i) \tag{3.17}$$
$$f_2 = f(\boldsymbol{x}^* + \alpha_2 \boldsymbol{s}^i), \tag{3.18}$$

(iii) do $i = i + 1$, if $i > N$ go to (vii);

(iv) if $f_1 < f_2$ then:

$$\begin{aligned} &\alpha_u = \alpha_2, \quad f_u = f_2, \\ &\alpha_2 = \alpha_1, \quad f_2 = f_1, \\ &\alpha_1 = \alpha_u - \tau(\alpha_u - \alpha_l), \quad \text{evaluate } f_1, \end{aligned}$$

and go to (ii);

(v) if $f_1 > f_2$ then:

$$\begin{aligned} &\alpha_l = \alpha_1, \quad f_l = f_1, \\ &\alpha_1 = \alpha_2, \quad f_1 = f_2, \\ &\alpha_2 = \alpha_l + \tau(\alpha_u - \alpha_l), \quad \text{evaluate } f_2, \end{aligned}$$

and go to (ii);

(vi) if $f_1 = f_2$ then:

$$\begin{aligned} &\alpha_l = \alpha_1, \quad f_l = f_1, \\ &\alpha_u = \alpha_2, \quad f_u = f_2, \\ &\alpha_1 = \alpha_u - \tau(\alpha_u - \alpha_l), \quad \text{evaluate } f_1, \\ &\alpha_2 = \alpha_l + \tau(\alpha_u - \alpha_l), \quad \text{evaluate } f_2, \end{aligned}$$

and go to (ii),

(vii) find $f_{min} = \min(f_l, f_1, f_2, f_u)$, and set respective step length as α_i.

Box 3.7 Algorithm for finding the bounds that enclose the minimum of a function of n variables.

Given the present solution $\boldsymbol{x}^*$, its dimension n, function value f_0 and search direction $\boldsymbol{s}^i$, determine limits α_{max} and α_{min} for step length according to the side constraints, and:

(i) assume $\alpha_l = 0$, $f_l = f_0$, $\alpha_u = 0.01\alpha_{max}$ and evaluate f_u,

(ii) if $f_u > f_0$, the upper bound is found, but the lower is not, thus go to (v),

(iii) update upper bound, limiting it to α_{max}:

$$\begin{aligned} &\alpha_1 = \alpha_u, \quad f_1 = f_u, \\ &\alpha_u = \alpha_1 + \phi(\alpha_1 - \alpha_l), \quad \text{evaluate } f_u, \end{aligned}$$

(iv) if $f_u > f_1$ then the bounds are found, and the process is finished.
Else, update lower bound:

$$\alpha_l = \alpha_1, \quad f_l = f_1,$$

and go to (iii),

(v) initialize lower bound as $\alpha_l = -0.01\alpha_{min}$, and evaluate f_l,

(vi) if $f_l > f_0$, lower bound is found, and the process is finished,

(vii) update lower bound limiting it to α_{min}:

$$\begin{aligned} &\alpha_1 = \alpha_l, \quad f_1 = f_l, \\ &\alpha_l = \alpha_1 - \phi(\alpha_u - \alpha_1), \quad \text{evaluate } f_l, \end{aligned}$$

(viii) if $f_l > f_1$ then the bounds are found, and the process is finished.
Else, update upper bound:

$$\alpha_u = \alpha_1, \quad \alpha_u = \alpha_1,$$

and go to (vii).

Table 3.3: Modified Fletcher-Reeves conjugate direction method results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-4.7433	1.8211	19.5463	0.1214
2	1.7284	-0.5866	-3.5042	1.7043
3	3.0879	-0.8048	-5.2457	0.0226
4	3.8141	1.5116	-22.4395	0.1602
5	5.6371	1.0831	-29.4314	0.2696
6	7.1763	3.9768	-37.8622	0.4712
7	8.3128	3.9607	-41.5450	0.1427
8	8.3257	4.2399	-41.6726	0.3020
9	8.5201	4.2877	-41.7601	0.2118
10	8.5062	4.3932	-41.7845	0.2283
11	8.5817	4.4159	-41.7973	0.2358
12	8.5747	4.4686	-41.8033	0.2309
13	8.6087	4.4784	-41.8059	0.2284
14	8.6056	4.5025	-41.8072	0.2338
15	8.6210	4.5071	-41.8077	0.2243
16	8.6198	4.5183	-41.8080	0.2398
17	8.6269	4.5205	-41.8081	0.2185
18	8.6264	4.5260	-41.8082	0.2619
19	8.6295	4.5268	-41.8082	0.1901

Table 3.4: Polak-Ribiere conjugate direction method results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-4.7433	1.8211	19.5463	0.1214
2	-1.0389	-0.3952	8.0276	1.0424
3	2.9388	-0.9764	-2.2508	0.1489
4	7.3901	0.8869	-24.3852	0.1537
5	6.7506	2.6387	-37.5779	0.1306
6	8.4895	3.8702	-41.3038	0.6378
7	8.6002	4.5059	-41.8066	0.1987
8	8.6316	4.5299	-41.8082	0.2292
9	8.6320	4.5318	-41.8082	0.2028

Table 3.5: Newton's method results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-6.0639	3.3233	28.3064	0.7766
2	-2.2382	0.0416	11.1956	0.0180
3	3.7596	-0.6184	-9.3064	0.4988
4	8.4308	2.8182	-37.3518	1.3091
5	8.3754	4.2687	-41.7124	1.1251
6	8.6319	4.5276	-41.8082	0.9434
7	8.6321	4.5319	-41.8082	0.9887

Table 3.6: DFP method ($\theta = 0$) results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-4.7433	1.8211	19.5463	0.1214
2	-1.0447	-0.3930	8.0385	1.0880
3	2.9230	-0.9762	-2.2173	2.2929
4	7.3341	0.8194	-23.7955	6.5656
5	6.7102	2.6183	-37.4317	5.1198
6	8.4749	3.8416	-41.2689	2.3548
7	8.6042	4.5107	-41.8070	2.0501
8	8.6318	4.5307	-41.8082	0.6657
9	8.6321	4.5322	-41.8082	1.3766

Table 3.7: BFGS method ($\theta = 1$) results for 2 spring problem.

Iteration	x_1	x_2	f	α
0	-4.0000	4.0000	41.5096	
1	-4.7433	1.8211	19.5463	0.1214
2	-1.0447	-0.3930	8.0382	1.0405
3	2.9348	-0.9764	-2.2416	1.2828
4	7.3880	0.8838	-24.3557	0.9584
5	6.7501	2.6400	-37.5776	0.4874
6	8.4891	3.8708	-41.3055	1.1028
7	8.5978	4.5048	-41.8064	1.6353
8	8.6318	4.5304	-41.8082	0.6751
9	8.6321	4.5319	-41.8082	1.1118

4 Genetic Algorithms

Genetic algorithms are based on Darwin's natural selection theory. A population in which each individual represents a possible solution (candidate) evolves according to natural principles of the fittest survival, reproduction with genetic information exchange and mutation, towards a population that has better individuals than the initial one. After working on adaptive systems during the sixties, in early seventies these methods started to become real shaped, and as a result Holland (1975) published the first monograph on this topic. Since then, genetic algorithms have been increasingly developed and employed in several fields, also thanks to increase of computational power. This subject is treated with great detail by Goldberg (1989), Whitley and Sutton (2012) and for the specific case of fuzzy multiobjective optimization by Sakawa (2002).

Goldberg (1989) enumerates four differences between genetic algorithms and classical optimization methods:

1. they work over design variables coding (genotype), and not on their own values,
2. search is made in parallel with a population of solutions, instead of single point search,
3. derivatives are not used, they only use objective function information,
4. evolution is based on probabilistic rules, instead of deterministic procedures.

In the literature these methods are seen as providing robust search in complex spaces.

The basic idea is to represent solutions (fenotype) as chromosomes, where information is stored in genes (genotype). For each individual is computed a fitness value, that quantifies its ability to survive and reproduce. Genetic operators, based on natural evolution of species (selection, reproduction and mutation), are applied to this chromosomes in order to obtain new generations. As living beings adapt to better fit to their living environment, variables are expected to evolve towards almost optimal solutions.

4.1 Proposed algorithm

In the this section a genetic algorithm is proposed, that is suitable for use with any number of design variables. The main structure of the proposed algorithm is presented in Box 4.1

Deeper detail on each stage is introduced in next sections.

4.2 Encoding variables

The solutions, defined by a set of real numbers, are encoded in a binary form, which is probably the most popular encoding technique due to its simplicity and the fact that only two alleles exist: 0 or 1. The number of genes needed depend on the number of design variables and on its desired precision.

For a problem with n_v design variables, the total number of genes is given by:

$$n_g = \sum_{i=1}^{n_v} k_i, \tag{4.1}$$

where k_i is the number of genes (or bits, since it assumes 0 or 1) related to the $i-th$ variable, which is related to its desired precision. Given the maximum and minimum value for each variable, the precision p_i is expressed as:

$$p_i = \frac{x_i^{max} - x_i^{min}}{2^{k_i} - 1}. \tag{4.2}$$

Box 4.1 Genetic algorithm proposed for finding the minimum of an objective function of n variables.

Given the side constraints for each variable (x_i^{min} and x_i^{max}) and the respective precision, define the number of individuals in the population n_p, survival rate s, mutation probability p_m and maximum number of generations m_{gen}:

(i) determine number of bits k_i for each variable, and total number of bits n_b,

(ii) determine number of individuals that survive and pass to next generation n_s,

(iii) initialize generation counter i_g

(iv) represent the population in a binary array $n_p \times n_b$, initialize it randomly,

(v) obtain fitness of each individual and rank them in the population,

(vi) increment generation counter $i_g = i_g + 1$. If $i_g > m_{gen}$ finish process,

(vii) a new population is generated. The n_s fittest individuals of the previous generation automatically pass to the present one,

(viii) offspring that comes from the previous generation complete the current one. Mutation may occur in some genes of these individuals, with a probability p_m,

(ix) compute fitness of each individual and rank them in the population,

(x) if stop criteria is verified, finish process,

(xi) go to (vi).

The real value is obtained from binary code as:

$$x_i^{real} = x_i^{min} + x_i^{bin} \frac{x_i^{max} - x_i^{min}}{2^{k_i} - 1}, \tag{4.3}$$

where x_i^{bin} is the real value that corresponds to the binary defined in the chromosome.

4.3 Fitness evaluation

In the present work, optimization strategy is tackled in the sense of minimizing an objective function. This means that the lower value objective function assumes, the greater must be the fitness value for the corresponding individual. A strategy based on the idea of Goldberg (1989) is implemented. In first place objective function $g(\boldsymbol{x})$ is evaluated for all individuals in the current population, and the maximum value g_{max}, that is related to the worst individual, is identified. For each chromosome i the fitness is determined as:

$$f(\boldsymbol{x}_i) = g_{max} - g(\boldsymbol{x}_i) + 0.01(g_{max} - g_{min}). \tag{4.4}$$

In this expression, the last term corresponds to an artificial increase of the fitness of all individuals in 1% of the best individual value. This avoids that the worst individual has null fitness, giving him the chance to reproduce, even though it is reduced. Fitness values cannot be compared between distinct generations since it depends on the worst individual. In order to verify the improvement over generations, the objective function must be considered.

4.4 Elitism

Individuals of current generation are sorted by decreasing fitness, before a new population is generated. When it happens, the first step is to transfer the n_s first individuals of present generation to the next one. This number of survival individuals is determined by the defined rate of survival s.

$$n_s = s \times n_p \tag{4.5}$$

This allows the user to define the degree of elitism to introduce in the algorithm. However, the value of survival rate shall be reduced, so that only few individuals pass to the next generation, otherwise the diversity of population would be lost too fast. In some way, this is related to the concept of *generation gap* introduced by De Jong (1975). The remaining individuals for next generation are obtained with reproduction operations over the current population.

4.5 Reproduction and mutation

The offspring of previous generation is obtained with a one-point crossover technique applied to several pairs of *parents*.

In first place, each one of the parents is picked from old generation according to *roulette* selection algorithm, which is widely known and well described in the literature (Goldberg, 1989; Sakawa, 2002). This algorithm allows to choose the parents randomly, but with a probability related to the fitness of each individual.

Chosen a pair of old individuals, they will interchange genetic information. Considering the binary string that defines one chromosome, one cut position is chosen randomly, with uniform probability distribution. Parents genotype is divided into two parts, the first before the cutting point, and the second after it. Then each son gets the first part of one parent and the second part of the other, thus two new individuals are created with characteristics that come from their parents.

In order to avoid diversity loss, a random search strategy is used: the mutation operator. In the current approach it randomly changes alleles of genes of the generated offspring. The probability of mutation of each gene is uniform and denoted by p_m. The strategy consists on looping over all genes of offspring, and for each one generate a random number between 0 an 1. If this value is lower than p_m then the respective allele is changed (0 to 1 or 1 to 0). Usually, the probability for mutation is low (1% or less).

Fitness of current population chromosomes is evaluated, and stop criteria is verified before a new generation is created.

4.6 Stop criteria

The simplest fashion to stop the process is when a defined number of generations has been created and evaluated. However it may happen that good solutions are achieved, or the evolution simply stuck, before this criterion is verified.

Two alternative approaches are considered: (i) the fittest individuals do not change for a number of iterations or (ii) the diversity of the population becomes too reduced. The former is easily verified comparing the top of best individuals between generations. The latter can be implemented considering that the diversity in the population is too reduced when for every locus there is an allele that dominates existing in more than 90% of the individuals, as suggested by Reeves (2003).

In the current implementation, the process stops when one of the three presented conditions is verified.

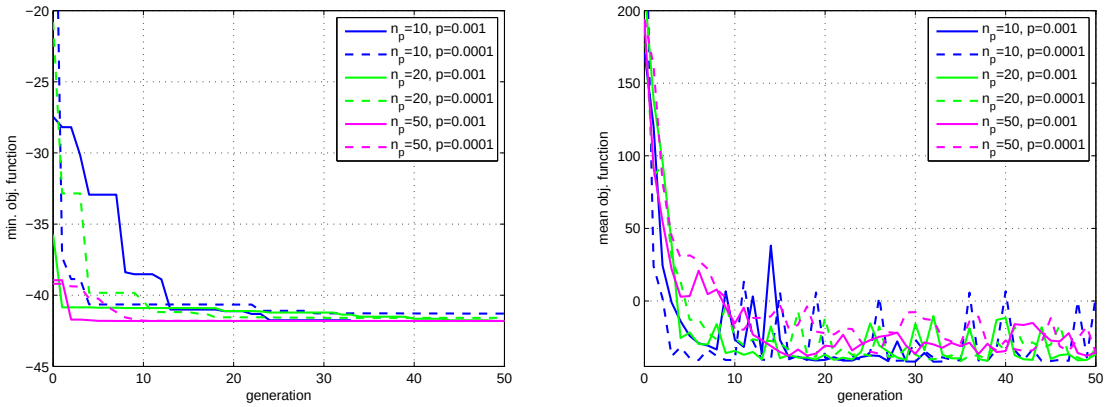
4.7 Application: 2 spring problem

In this section a simple application of the proposed genetic algorithm is considered, in order to verify its correct implementation and performance.

The problem presented in section 3.8 is solved considering different parameters. Side constraints are the same used with the classical methods: $-12 \leq x_1, x_2 \leq 12$.

At first this problem is solved considering distinct dimensions of population $n_p = (10, 20, 50)$ and precision for the solution $p_1 = p_2 = (0.001, 0.0001)$. The number of maximum generations is 50, the survival rate is chosen so that only one individual passes to next generation, and mutation probability is 1%. The procedure stops when maximum number of generation is achieved, diversity reduces too much, or when the top 3 does not change for 5 generations.

The evolution of the objective function of the best individual, and the mean of the overall population are plotted in Figure 4.1. The best result is obtained with $n_p = 50$ and $p = 0.001$, with $x_1 = 8.630$ and $x_2 = 4.528$ cm and $f = -41.8082$ Ncm.



(a) Minimum value on population.

(b) Mean value of population.

Figure 4.1: Evolution of the minimum and mean objective function values.

The influence of the size of the population and the chosen precision may be analysed observing Figure 4.1.

The precision directly affects the number of bits that defines the chromosome, and thus the number of possible solutions in the population. Considering the same number of individuals, it is more likely to obtain a good solution in the initialization with lower precisions. This is clear to happen for $n_p = 10$ or 20. However when the population increases, this difference becomes negligible since a good covering of the domain is achieved anyway. Moreover, note that better solutions are achieved with $p = 0.001$. This is not as expected, however it is explained by the fact that increasing the precision too much lead to a huge number of possible solutions, and the algorithm becomes more directed to exploring the domain than to improve the best individuals. It is concluded that the precision for the solution should be enough and no overestimated.

Analysing the effect of population size, for a reduced one, the genetic algorithm is not so well-conditioned, since the diversity of the population may decrease dramatically, even though the minimum is not attained. This happens with $n_p = 10$ for the coarser precision, that finished before the 50^{th} generation due to reduced diversity. In this case it finished with a very good solution, but usually this is not achieved. With a greater number of bits the loss of diversity may not be so problematic, but with a reduced population the exploring power is lost. The solution rapidly tends to a value that is seldom improved.

With a large population, the optimum is achieved rapidly, but at the expense of a large

number of function evaluations. Thus a compromise solution shall be considered. However, there is no formula for the ideal dimension of population. It depends on the the number of variables on the combination of side constraints and precision of each variable, and its nature is empirical.

Looking at Figure 4.1b, it can be said that the population is enhanced over the generations, since the value of the mean objective function decreases in a general way. The fluctuations on its evolution are due to mutated genes, or even some crossover operations, that result in poorly fitted individuals.

4.8 Conclusions

Genetic algorithms are based on the natural evolution of living beings species, that tend to adapt to their environment. The procedures that are implemented try to simulate mechanisms like survival of the fittest, reproduction and gene mutation.

The algorithmic parameters, like population size or precision of variables (number of genes), have strong influence on the performance of the method. However there is no closed form to determine the parameters that must be used. From the example of this section the main conclusion is that the precision shall not be too fine, since with a greater number of genes a bigger population is needed to explore more deeply the design space. Note that for greater number of individuals, more function evaluations must be performed.

5 6 spring system equilibrium

In this section one aims to apply the previously presented algorithms to a more complex example, and verify their behaviour. A problem that consists in finding the equilibrium of the mechanical system presented in Figure 5.1 is considered to perform this task. Like the problem in section 3.8, it can be treated as an optimization problem in which the objective function (potential energy) has to be minimized. The unknown variables are the position of each weight.

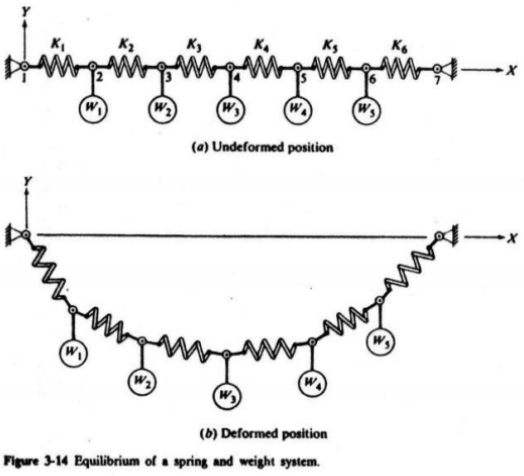


Figure 5.1: Representation of the 6 spring problem (Vanderplaats, 1984).

In the general case, being N the number of weights, the stiffness of each spring is expressed as:

$$k_i = 500 + 200\left(\frac{N}{3} - i\right)^2 \text{ N/m}, \quad i = 1, N+1, \tag{5.1}$$

and the weight is defined by:

$$w_j = 50j \text{ N}, \quad j = 1, N. \tag{5.2}$$

The potential energy function is given by the following expression:

$$f = \sum_{i=1}^{N+1} \frac{1}{2} k_i \cdot \Delta l_i^2 + \sum_{j=1}^{N} w_j y_j \text{ Nm}, \tag{5.3}$$

where the elongation Δl_i is determined by:

$$\Delta l_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} - l_i^0. \tag{5.4}$$

In this expression x_i and y_i define the position of i-th weight ($i = 1, N$). (x_0, y_0) and (x_{N+1}, y_{N+1}) denote the positions of the external connections (values in metre):

$$x_0 = 0 \tag{5.5}$$
$$y_0 = 0 \tag{5.6}$$
$$x_{N+1} = 60 \tag{5.7}$$
$$y_{N+1} = 0. \tag{5.8}$$

The initial length of the springs is $l_i^0 = 10$ m, $i = 1, N+1$.

The following side constraints are imposed:

$$\begin{aligned} 5 \leq & x_1 \leq 15 \\ 15 \leq & x_2 \leq 25 \\ 25 \leq & x_3 \leq 35 \\ 35 \leq & x_4 \leq 45 \\ 45 \leq & x_5 \leq 55 \\ -60 \leq & y_i \leq 10, \quad i = 1, 5. \end{aligned} \tag{5.9}$$

5.1 Classical algorithms solutions

The classical algorithms presented in section 3 are used to solve the present problem. The convergence tolerance is $\varepsilon_C = 10^{-6}$, and unidirectional search for step length is performed with the golden section method until the initial interval is reduced to 1% of its amplitude. A maximum of 50 iterations is allowed. Gradient and Hessian matrix are computed by the finite difference method (Appendix B).

The solutions obtained with these algorithms are presented in Table 5.1, and the evolution of the potential function is plotted in Figure 5.2.

For comparison, the best result found by Vanderplaats (1984) is shown in Table 5.2.

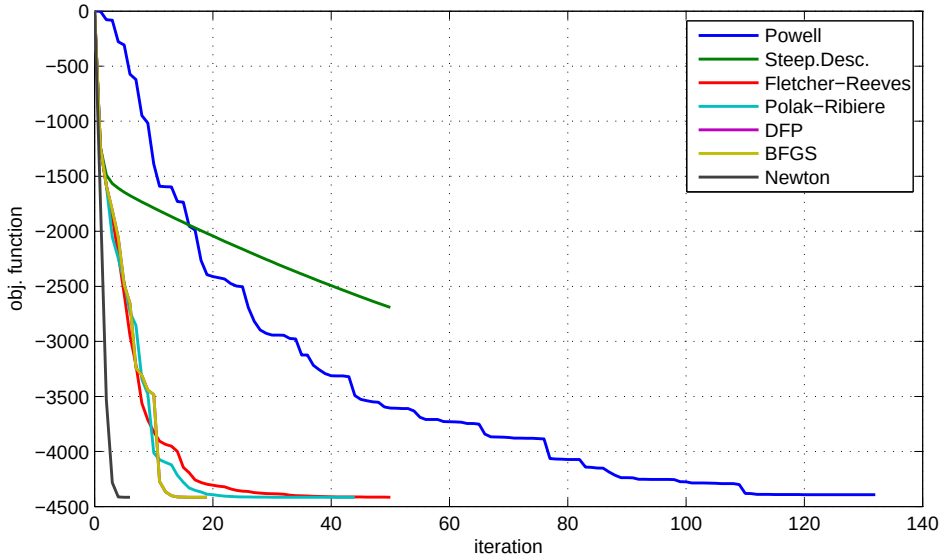


Figure 5.2: Evolution of objective function value observed with several classical methods. For Powell's method it is plotted against number of unidirectional searches.

In this case it is confirmed that the steepest descent algorithm results in very weak performance. While all the other methods are able to achieve a good solution in the prescribed number of iterations, this technique starts with a good convergence rate that is lost after a few iterations. Powell's method is closer to the expected solution but not as close as the remaining methods. The problem in this case is that the first variable (x_1) has stuck in its initial guess. This happened because in the first search in this direction $\alpha = 0$, and the search direction lost this component for later iterations. As referred in section 3.1, whenever this situation occurs, the base for conjugate directions should be reinitialized. This is not implemented in the present program, but is worthy to perform this task in the future. Anyway, even with this issue, the method proved its reliability.

Table 5.1: Solution obtained with classical algorithms for 6 spring problem. For Powell's method, the actual number of unidirectional searches is given.

	i	1	2	3	4	5	f	N_{iter}
Powell	x_i	10.0000	20.8798	31.5458	41.9651	51.6614	-4392.41	12 (132)
	y_i	-4.6799	-7.9711	-9.7758	-9.2609	-5.8744		
Steepest Descent	x_i	10.2139	20.4914	30.6773	40.7654	50.9103	-2692.32	50
	y_i	-1.1972	-2.3982	-3.5745	-4.5629	-4.3193		
Fletcher-Reeves	x_i	10.3911	21.1195	31.6962	42.1015	51.7680	-4415.36	50
	y_i	-4.2469	-7.8007	-9.8284	-9.4423	-6.0012		
Polak-Ribiere	x_i	10.3573	21.0910	31.6924	42.0943	51.7735	-4416.37	44
	y_i	-4.2849	-7.9069	-9.8656	-9.4033	-6.0148		
DFP	x_i	10.3550	21.0865	31.6867	42.0896	51.7712	-4416.38	19
	y_i	-4.2801	-7.8973	-9.8535	-9.3930	-6.0115		
BFGS	x_i	10.3550	21.0865	31.6868	42.0898	51.7713	-4416.38	19
	y_i	-4.2801	-7.8978	-9.8541	-9.3934	-6.0117		
Newton	x_i	10.3550	21.0866	31.6869	42.0898	51.7713	-4416.38	6
	y_i	-4.2801	-7.8974	-9.8536	-9.3934	-6.0118		

Table 5.2: Best result obtained by Vanderplaats (1984) for the 6 spring problem.

i	1	2	3	4	5	f
x_i	10.4	21.1	31.7	42.1	51.8	-4416
y_i	-4.28	-7.90	-9.86	-9.40	-6.01	

Conjugate direction methods show good performance, but not as good as quasi-Newton methods. In the initial iterations convergence is similar, but in the final stage the former take more iterations to converge. The difference between the DFP and BFGS methods is negligible in this case. Newton's method convergence rate is astonishing when comparing with the remaining methods. It has converged in only 6 iterations. The results are very close to the values obtained by Vanderplaats (1984).

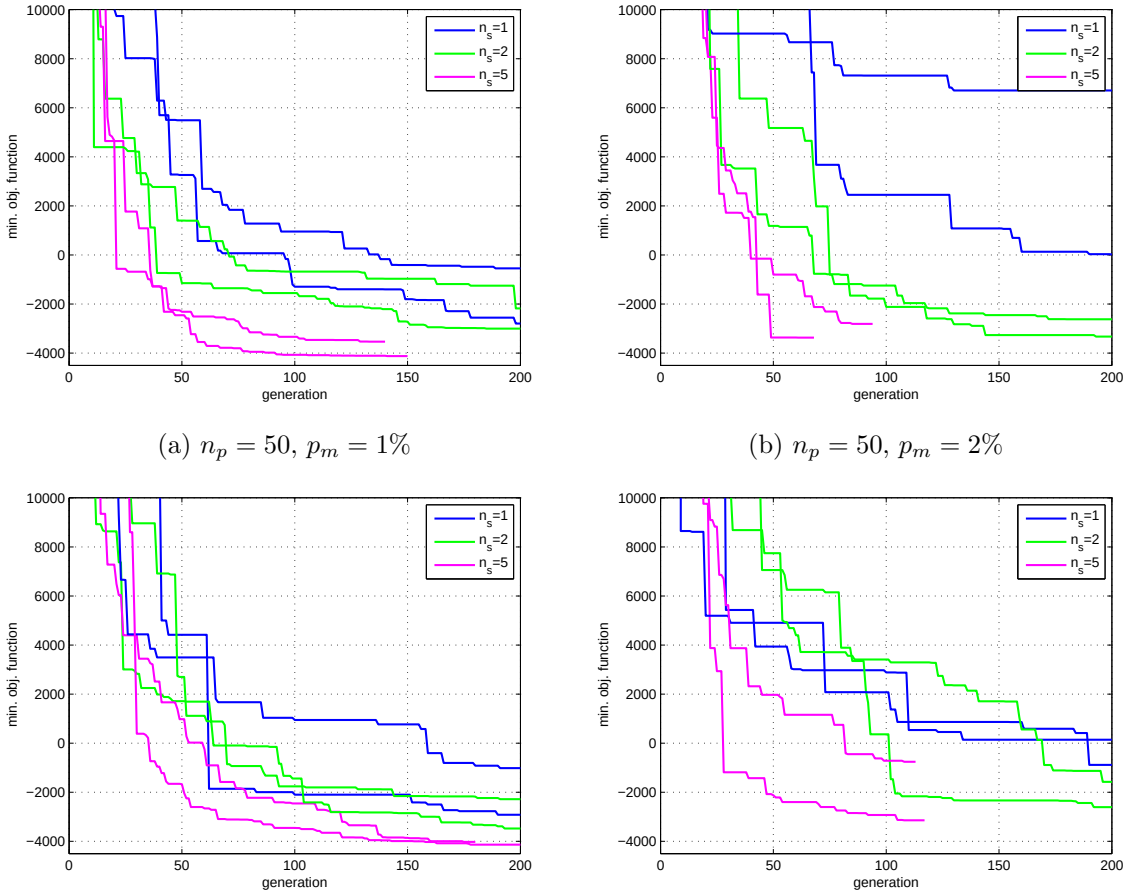
5.2 Genetic algorithm solution

The same problem is solved considering the genetic algorithm proposed in section 4. 12 distinct combinations of algorithmic parameters are analysed, and each set of parameters is used to obtain results in 2 times. The following parameters are varied: population size ($n_p = 50$ or 75), number of surviving individuals ($n_s = 1$, 2 or 5) and mutation probability ($p_m = 1\%$ or 2%).

The limits for each variable are given by (5.9), and the precision is set as $p = 0.001$ m for every coordinate. This results in a total of 155 bits per chromosome, with 14 bits for x_i and 17 for y_i. A maximum of 200 generations is allowed. The process is stopped when the diversity of the population reduces such that at every locus there is an allele dominance over 90%, or when the 5 fittest individuals remain the same after 5 generations.

The evolution of the potential function value for the best individual in the population is plotted in Figure 5.3. In Table 5.3 the final best solution is presented for every cases.

Observing Figure 5.3 it becomes quite evident that increasing the number of survival individuals leads to better performance of the algorithm, as curves representing the best chromosome tend to converge faster to lower values of the objective function. However note that the survival rate in the present problem is always lower than 10%, and it should not be much greater than this value, under the danger that the diversity of population is lost too soon.



(a) $n_p = 50$, $p_m = 1\%$

(b) $n_p = 50$, $p_m = 2\%$

(c) $n_p = 75$, $p_m = 1\%$

(d) $n_p = 75$, $p_m = 2\%$

Figure 5.3: Evolution of the minimum potential energy found in the population.

All the computations finished with the maximum number of generations, except those having the maximum number of survival considered here, that finished due to stability in the fittest individuals. Despite this fact, the best solutions are not as close to the expected values as the obtained with classical methods.

Another noticeable fact is that increasing the mutation probability from 1 to 2%, in general worst results are obtained. In fact, 1% is the upper bound for this value, since for greater probabilities the heritage of good characteristics may be destroyed by an excessive number of mutated genes.

When comparing the population size, in spite of not resulting in big differences in the final results, it is obvious that with a bigger population the exploratory power of the algorithm is increased, and as a consequence it becomes more robust as the probability of finding better individuals in each generation increases. This also explains why the convergence criteria is satisfied later for $n_p = 75$.

Finally, it must be noted that the heuristic component of this technique is clearly shown as with the same set of parameters, distinct results are obtained in different computations.

5.3 Conclusions

In the present section a problem related to the equilibrium of a mechanical system composed by springs and weights is solved with the previously presented optimization methods. Objective function coincides with the potential energy function, that must be minimized.

Classical methods are suitable to easily solve this problem. Except the steepest descent method, whose bad performance is confirmed, and Powell's method, due to the fact that its implementation does not obviate some numerical issues, all the implemented classical techniques achieve results very close to the expected value. Powell's method system of conjugate directions should be initialized when $\alpha = 0$. Newton's method convergence rate allows to obtain the result in only 6 iterations. DFP and BFGS methods resulted in similar evolution of solutions. Conjugate direction methods show a convergence rate close to the quasi-Newton methods, but in the final stage they take much more iterations to converge.

Genetic algorithm is also used to solve this problem, considering several sets of algorithmic parameters. The heuristic character of this technique is evident. The final results are not so close to the expected values, and they strongly depend on the set of parameters. A mutation probability above that is grater than 1% clearly leads to worst results and performance. With the presently considered values, the convergence becomes faster as the number of survival individuals increases. However this value should not be too large, as the population diversity may be lost. Obviously, with a greater number of individuals in population, the exploring power of the algorithm increases.

Although in the present problem the genetic algorithm does not seem to be worthy, this is due to the good behaviour of this objective function, that allows classical methods to easily find the minimum. Genetic algorithms are more suitable for hard optimization problems, where objective function usually have a difficult topology.

Table 5.3: Solutions for the 6 spring problem, obtained with genetic algorithm. Stop criteria: "gen". for maximum number of generations, "top" if top 5 does not change for 5 generations.

n_p	n_s	p_m	#	i	1	2	3	4	5	f	N_{gen}	Stop
50	1	1%	1	x_i	10.089	19.616	30.028	39.361	49.586	-545.31	200	gen.
				y_i	-4.586	-2.758	-3.809	-2.398	-1.513			
			2	x_i	11.258	21.797	32.490	42.717	50.179	-2794.75	200	gen.
				y_i	-4.677	-9.014	-11.949	-9.668	-2.705			
		2%	1	x_i	8.717	19.100	28.605	38.578	49.610	19.01	200	gen.
				y_i	-7.213	-8.300	-4.192	-2.258	-1.726			
			2	x_i	8.657	20.412	27.089	37.681	49.010	6702.66	200	gen.
				y_i	-7.350	-7.936	1.279	1.562	-0.038			
	2	1%	1	x_i	9.845	20.432	30.508	39.978	49.995	-3000.63	200	gen.
				y_i	-4.425	-7.064	-8.002	-4.876	-2.788			
			2	x_i	10.116	20.282	30.573	39.494	49.815	-2179.34	200	gen.
				y_i	-4.875	-7.492	-8.599	-3.777	-1.429			
		2%	1	x_i	10.271	20.384	29.764	40.323	50.318	-2618.11	200	gen.
				y_i	-2.923	-7.680	-9.124	-8.018	-3.828			
			2	x_i	10.263	19.859	30.420	41.258	50.940	-3324.22	200	gen.
				y_i	-2.871	-6.935	-8.243	-6.553	-4.156			
	5	1%	1	x_i	11.260	22.503	32.662	42.760	52.707	-3531.16	140	top
				y_i	-2.217	-4.575	-7.297	-7.462	-7.041			
			2	x_i	9.667	20.530	31.256	41.236	50.977	-4120.77	150	top
				y_i	-5.406	-8.404	-9.642	-7.951	-4.836			
		2%	1	x_i	8.625	19.478	29.675	40.683	50.743	-2807.62	94	top
				y_i	-6.428	-6.081	-7.447	-7.894	-4.933			
			2	x_i	10.465	21.132	32.814	42.899	53.197	-3370.44	68	top
				y_i	-4.043	-8.905	-9.785	-8.329	-7.671			
75	1	1%	1	x_i	11.491	23.071	33.107	43.277	53.313	-2917.29	200	gen.
				y_i	-0.790	-2.435	-6.928	-8.831	-7.310			
			2	x_i	9.212	19.632	30.848	41.198	51.161	-1017.08	200	gen.
				y_i	-1.946	-7.993	-12.282	-7.917	-5.667			
		2%	1	x_i	9.262	19.984	31.644	42.417	52.713	-884.81	200	gen.
				y_i	-0.955	-7.007	-6.337	-6.758	-7.662			
			2	x_i	10.095	18.454	28.347	39.025	49.904	141.35	200	gen.
				y_i	2.891	-2.733	-5.153	-5.529	-3.097			
	2	1%	1	x_i	8.637	20.647	31.495	41.744	51.182	-3475.87	200	gen.
				y_i	-7.578	-10.078	-11.257	-9.076	-5.080			
			2	x_i	9.610	20.749	31.759	41.120	50.790	-2279.44	200	gen.
				y_i	-4.355	-3.072	-2.514	-6.608	-4.168			
		2%	1	x_i	8.926	18.799	29.861	39.788	49.955	-1576.56	200	gen.
				y_i	-5.910	-6.954	-6.284	-7.531	-4.050			
			2	x_i	9.286	20.263	29.673	40.063	50.401	-2604.42	200	gen.
				y_i	-2.737	-5.054	-7.663	-6.363	-4.066			
	5	1%	1	x_i	10.194	20.440	31.261	41.529	51.488	-4022.49	180	top
				y_i	-3.619	-6.159	-7.492	-7.486	-5.667			
			2	x_i	10.320	20.517	30.894	41.242	51.254	-4136.65	200	gen
				y_i	-2.888	-6.633	-8.593	-8.097	-5.320			
		2%	1	x_i	10.914	22.833	33.133	41.694	51.620	-759.72	113	top
				y_i	-0.444	-1.036	1.254	-4.011	-5.705			
			2	x_i	9.804	21.703	33.050	43.659	53.924	-3143.47	117	top
				y_i	-5.394	-8.560	-9.318	-8.548	-7.878			

6 Hyperelastic Parameters Identification

6.1 Hyperelasticity

Finite hyperelasticity theory is able to derive hyperelastic constitutive models that are suitable to describe the behaviour of several materials such as rubbers, soft tissues and some polymeric materials, achieving good agreements to the real behaviour for considerable large strains.

A hyperelastic material (also referred as Green-elastic material in the literature) requires the existence of a free-energy function (per unit of volume) $\Psi = \Psi(\boldsymbol{F})$ such that the constitutive response, in terms of the first Piolla-Kirchhoff stress tensor, is given by:

$$\boldsymbol{P} = \boldsymbol{P}(\boldsymbol{F}) = \frac{\partial \Psi(\boldsymbol{F})}{\partial \boldsymbol{F}}. \tag{6.1}$$

6.1.1 Ogden's model

A hyperelastic material model is completely described by the free-energy function. Ogden (1972) has proposed a model whose function depends on the deformation gradient through the principal stretches (due to material objectivity and isotropy conditions). It is expressed by a series of N terms:

$$\hat{\Psi}(\lambda_1, \lambda_2, \lambda_3) = \sum_{p=1}^{N} \frac{\mu_p}{\alpha_p} \left(\lambda_1^{\alpha_p} + \lambda_2^{\alpha_p} + \lambda_3^{\alpha_p} - 3\right). \tag{6.2}$$

where μ_p and α_p are material constants.

Rubber-like materials as well as biological soft tissues are very compliant in shear, but strongly resist to volume changes, therefore they are modelled as incompressible materials. In fact, hyperelastic constitutive models are usually presented considering incompressibility.

The volume change can be quantified by means of the determinant of the deformation gradient. An incompressible deformation is defined by $\det \boldsymbol{F} = J = 1$. In terms of principal stretches this means that:

$$\lambda_1\lambda_2\lambda_3 = 1 \Leftrightarrow \lambda_3 = \frac{1}{\lambda_1\lambda_2}. \tag{6.3}$$

Considering plane stress state, (6.2) can be rewritten as:

$$\hat{\Psi}(\lambda_1, \lambda_2) = \sum_{p=1}^{N} \frac{\mu_p}{\alpha_p} \left(\lambda_1^{\alpha_p} + \lambda_2^{\alpha_p} + (\lambda_1\lambda_2)^{-\alpha_p} - 3\right). \tag{6.4}$$

Thus, in this particular case, first Piolla-Kirchhoff tensor components are given by:

$$\begin{aligned} P_{11} &= \sum_{p=1}^{N} \mu_p \left(\lambda_1^{\alpha_p - 1} - \lambda_1^{-\alpha_p - 1}\lambda_2^{-\alpha_p}\right) \\ P_{22} &= \sum_{p=1}^{N} \mu_p \left(\lambda_2^{\alpha_p - 1} - \lambda_1^{-\alpha_p}\lambda_2^{-\alpha_p - 1}\right). \end{aligned} \tag{6.5}$$

6.2 Micro-mechanical based parameter identification

Consider now a material with a heterogeneous microstructure, composed by a hyperelastic matrix and uniformly distributed inclusions. The global behaviour may be obtained by numerical homogenization. For more details on this technique one may read de Souza Neto and Feijóo (2006) or Reis (2014). The basic idea is to apply the macro-deformation to a Representative Volume Element (RVE) of the microstructure, solve the micro-equilibrium problem, and obtain the homogenized response as a volumetric average of the micro-stresses. In the present

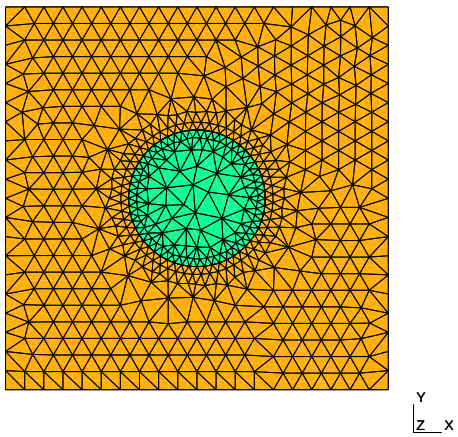


Figure 6.1: Representative Volume Element of the microstructure, and its finite element discretization. Dimensions: $1 \times 1 \times 0.1$ mm.

case, the RVE that represents the microstructure is shown in Figure 6.1, where the matrix is modelled by an incompressible Ogden's law with $N = 3$:

$$\begin{aligned} \alpha_1 &= 1.3 & \mu_1 &= 630 \text{ kPa} \\ \alpha_2 &= 5.0 & \mu_2 &= 1.2 \text{ kPa} \\ \alpha_3 &= -2.0 & \mu_3 &= -10 \text{ kPa}, \end{aligned} \tag{6.6}$$

and rigid inclusions, whose volume fraction is 10%, are also considered as an incompressible hyperelastic material, with $N = 1$, $\alpha_1 = 2$ and $\mu_1 = 6000$ kPa.

This RVE is submitted to uni-axial and bi-axial tension, considering plane stress state. These macroscopic loadings are defined by the respective deformation gradients, where incompressibility is imposed:

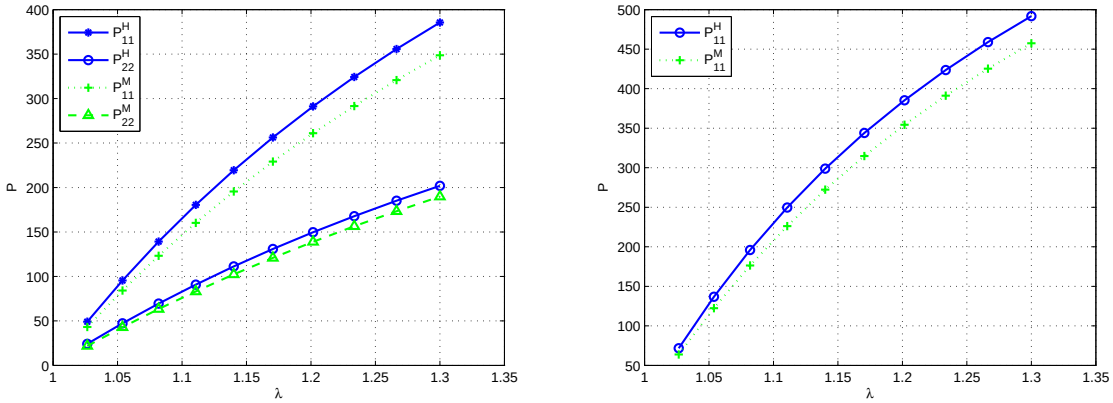
$$\boldsymbol{F}_{UA} = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \lambda^{-1} \end{bmatrix} \tag{6.7}$$

$$\boldsymbol{F}_{BA} = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda^{-2} \end{bmatrix}. \tag{6.8}$$

The global behaviour is obtained through numerical homogenization, where the micro-scale problem is solved with the finite element method. Quadratic triangular elements are used to discretize the RVE, as shown in Figure 6.1. Periodic boundary condition is imposed, since a uniform distribution of inclusions is considered. Loadings are applied in 10 increments, until $\lambda = 1.3$, and the homogenized first Piolla-Kirchhoff stress tensor is obtained.

Relevant components of the obtained homogenized stress tensor are plotted in Figure 6.2, against computed values considering matrix properties (equation (6.5)).

It is observed that the presence of a stiffer inclusion affects the response, increasing the stress values, as it is expected. Thus, to describe the global response using an Ogden's law, suitable parameters must be found. This task may be performed using optimization methods. Ogden et al. (2004) identified hyperelastic parameters from experimental data. Speirs (2007) has developed a similar work, where parameters are found from homogenized response. These



(a) uni-axial (b) equi-biaxial

Figure 6.2: Evolution of Piolla-Kirchhoff tensor components obtained with numerical homogenization (P^H), and values computed with matrix properties (P^M).

authors have used *Matlab* least squares curve fitting tool to obtain the parameters. Here it is intended to use some of the presented optimization methods (sections 3 and 4) to perform this task.

The objective function is defined so that the difference between the fitted solution (P^{UA}_{ii} and P^{EB}_{11}) and the homogenized response ($P^{H,UA}_{ii}$ and $P^{H,EB}_{11}$) is minimized in every increment k, for both uni-axial and equi-biaxial loadings:

$$f(\boldsymbol{x}) = \sum_{k=1}^{10} \left(\frac{P^{UA}_{11}(\boldsymbol{x}, \lambda_k)}{P^{H,UA}_{11}(\lambda_k)} - 1 \right)^2 + \left(\frac{P^{UA}_{22}(\boldsymbol{x}, \lambda_k)}{P^{H,UA}_{22}(\lambda_k)} - 1 \right)^2 + \left(\frac{P^{EB}_{11}(\boldsymbol{x}, \lambda_k)}{P^{H,EB}_{11}(\lambda_k)} - 1 \right)^2 . \tag{6.9}$$

The vector of unknown variables stores the hyperelastic parameters. It is assumed that the global response may be described by an Ogden's hyperelastic law with $N = 3$, thus:

$$\boldsymbol{x} = \{\alpha_1 \ \alpha_2 \ \alpha_3 \ \mu_1 \ \mu_2 \ \mu_3\}^T . \tag{6.10}$$

6.2.1 Classical methods solutions

Powell's method, Polak-Ribiere conjugate direction, BFGS and Newton's methods are chosen to solve the present parameter identification problem. Following side constraints are imposed:

$$\begin{aligned} -3 \leq & \alpha_1 \leq 3 \\ -2 \leq & \alpha_2 \leq 7 \\ -5 \leq & \alpha_3 \leq 2 \\ 300 \leq & \mu_1 \leq 900 \text{ kPa} \\ -2 \leq & \mu_2 \leq 5 \text{ kPa} \\ -15 \leq & \mu_3 \leq 15 \text{ kPa}, \end{aligned} \tag{6.11}$$

and matrix properties (6.6) are considered as initial guess. A maximum of 50 iterations is allowed, the tolerance for unidirectional search is 0.01, and convergence tolerance is $\varepsilon_C = 10^{-6}$. Final results are presented in Table 6.1.

With Powell's method the solution reached a limit ($\mu_3 = 15$ kPa). It happened that $\alpha_i = 0$ during the process, and the approximation to conjugate directions was lost. Polak-Ribiere and BFGS returned similar results, where little change in the parameters is observed. With

Table 6.1: Solutions obtained with classical algorithms for hyperelastic parameter identification. For Powell's method, the actual number of unidirectional searches is given. Initial guess is given by (6.6).

	i	1	2	3	$f(\times 10^{-2})$	N_{iter}
Powell	α_i	1.460	5.951	-0.0598	0.2625	5 (35)
	μ_i	636.359	1.952	15.000		
Polak-Ribiere	α_i	1.506	5.071	1.480	0.2210	14
	μ_i	629.944	1.310	-9.596		
BFGS	α_i	1.506	5.071	1.477	0.2210	9
	μ_i	629.943	1.310	-9.598		
Newton	α_i	1.604	3.484	-1.079	0.1763	10
	μ_i	588.682	-0.257	-0.370		

Newton's method the best result is achieved. Components of Piolla-Kirchhoff and absolute errors are plotted in Figure 6.3 to compare the results with the homogenized behaviour. Polak-Ribiere solution is not plotted once it is too close to BFGS one.

It is observed that although some difference in the parameters exist, the evolution of components is well approximated with all solutions, when comparing to the initial guess (see Figure 6.2). Analysing the absolute error, it becomes evident that generally Newton's method solution is the one that better fits the homogenized solution. Only the first component for higher stretch level in uni-axial loading becomes worst than the remaining solutions.

The difference in the solution obtained with distinct methods may indicate that the objective function has several relative minima. In order to assess this hypothesis, solutions are obtained considering a different initial guess:

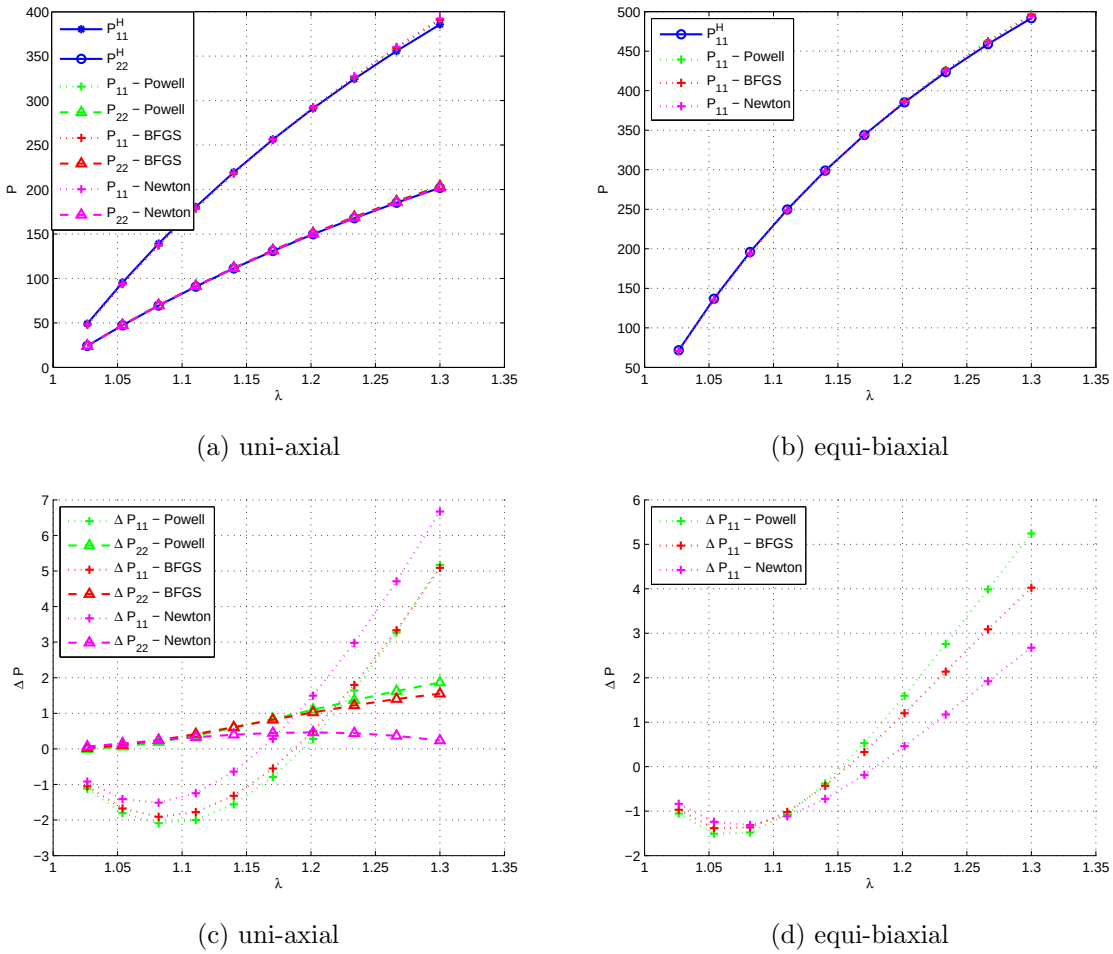
$$\begin{aligned} \alpha_1 &= 1.0 & \mu_1 &= 700 \text{ kPa} \\ \alpha_2 &= 2.0 & \mu_2 &= 2.0 \text{ kPa} \\ \alpha_3 &= -3.0 & \mu_3 &= -5.0 \text{ kPa}, \end{aligned} \tag{6.12}$$

that are presented in Table 6.2.

Table 6.2: Solutions obtained with classical algorithms for hyperelastic parameter identification. For Powell's method, the actual number of unidirectional searches is given. Initial guess is given by (6.12).

	i	1	2	3	$f(\times 10^{-2})$	N_{iter}
Powell	α_i	1.320	4.260	1.981	0.5039	6 (42)
	μ_i	705.682	5.000	-8.321		
Polak-Ribiere	α_i	1.338	2.111	1.274	0.5511	9
	μ_i	699.920	1.941	-3.709		
BFGS	α_i	1.348	7.000	0.722	0.4930	23
	μ_i	696.474	0.467	-7.588		
Newton	α_i	1.451	5.689	-2.085	0.1503	23
	μ_i	674.216	-0.120	15.000		

In fact it is clear that from a distinct initial point the solutions converge for values completely different from the ones presented in Table 6.1. The objective function compares a finite number of points, thus it is not extraordinary to understand that this leads to a difficult topology. It is known that for this kind of problems, classical methods strongly depend on the initial point, and tend to the closest minimum. In order to overcome this situation, genetic algorithms are used to try to achieve the global minimum.



(a) uni-axial

(b) equi-biaxial

(c) uni-axial

(d) equi-biaxial

Figure 6.3: Evolution of Piolla-Kirchhoff tensor components and absolute error between solutions from classical methods and homogenized response (initial guess given by (6.6)).

6.2.2 Genetic algorithm solution

In order to obtain solutions that may be closer to the global minimum, genetic algorithm optimization is employed. Side constraints are defined by (6.11). Two sets of algorithm parameters, defined in Table 6.3, are considered. The program runs twice for each set, and results are also presented in Table 6.4.

It is easily observed that these results are, in general, better than the ones obtained with classical algorithms. However, they are not conclusive in what refers to the global minimum, because they are quite different. These solutions are obtained at the expense of more function evaluations, which for this case is not prohibitive. In order to refine the solution, an hybrid strategy is now introduced: the solutions that come from genetic algorithms are used as initial guess for a classical method. Here the Newton's method is chosen, and the results are shown in Table 6.5.

Except for the case of set 1-2, this refinement does not introduce appreciable changes in the solution. A common fact is that in all cases the solution was taken to the boundaries imposed by side constraints. This may indicate that the limits should allow a broader range of values. However, the results obtained in the present strategy proved to be better than all previous results. The absolute errors are plotted in Figure 6.4.

In the case of problems where function evaluations are expensive, a hybrid strategy should be used. In first place a genetic algorithm is applied in order to explore the entire domain,

Table 6.3: Algorithm parameters used for fitting hyperelastic Ogden's law.

	n_p	n_s	p_m	precision	
set 1	50	5	1%	α_1	0.001
				α_2	0.001
				α_3	0.001
				μ_1	0.5
				μ_2	0.001
				μ_3	0.001
set 2	30	3	1%	α_1	0.01
				α_2	0.01
				α_3	0.01
				μ_1	0.5
				μ_2	0.01
				μ_3	0.01

Table 6.4: Solutions obtained with genetic algorithm for hyperelastic parameter identification.

	#	i	1	2	3	$f(\times 10^{-2})$	N_{gen}	Stop
set 1	1	α_i	1.628	6.973	-0.993	0.1303	180	top
		μ_i	596.5	-1.778	12.533			
	2	α_i	1.433	2.521	-3.319	0.1752	144	top
		μ_i	675.5	-1.750	5.574			
set 2	1	α_i	1.657	6.982	-0.299	0.1318	388	top
		μ_i	581.0	-2.000	11.520			
	2	α_i	1.405	5.038	-2.270	0.1562	251	top
		μ_i	690.0	1.688	14.048			

until a certain admissible computing time is spent. Then, the best individuals are used as initial guess for classical algorithms.

6.3 Conclusions

In the present section some implemented optimization algorithms (sections 3 and 4) are used to solve a problem of material parameters identification.

The global response of a heterogeneous medium composed by a soft matrix and rigid circular inclusions, both described by incompressible hyperelastic Ogden's laws, is obtained through numerical homogenization. A new Ogden's free-energy function is determined so that it fits to the homogenized response in the case of uni-axial and equi-biaxial loadings.

It was observed that the objective function has several minimum points, thus the classical methods solutions depend on the initial guess and rarely achieve global minimum. Moreover, distinct methods result in different solutions, which is not observed in the examples of previous sections. This indicates that the topology of the present objective function is complicated.

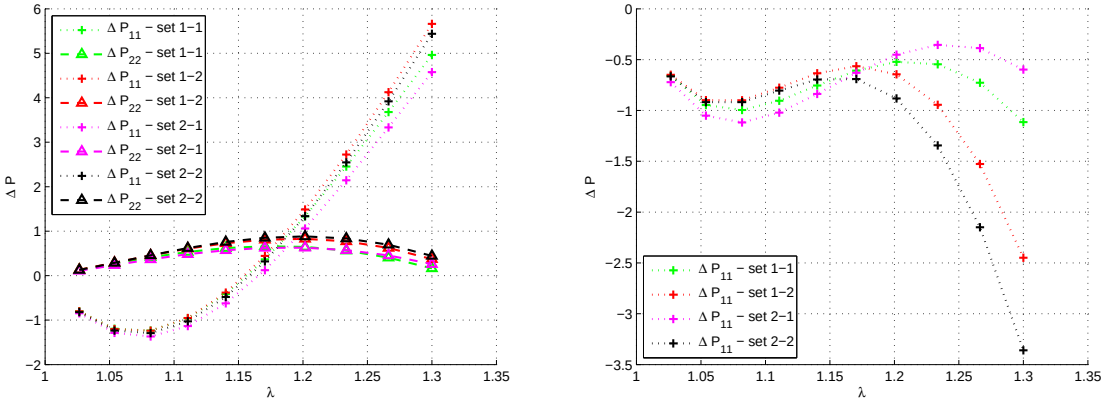
Genetic algorithm results returns solutions with lower values of objective function than classical methods, but more function evaluations are necessary. Although the present problem does not require heavy computations, in a general framework, a hybrid strategy should be used to obtain the best of both worlds.

Side constraints should be carefully chosen to avoid minima falling on the boundaries of the search space.

With this problem it is proven that for difficult optimization problems, where several minima may be found, genetic algorithms are able to achieve better results than classical

Table 6.5: Solutions obtained by refining genetic results (Table 6.4) with Newton's method.

	#	i	1	2	3	$f(\times 10^{-2})$	N_{iter}
set 1	1	α_i	1.628	7.000	-0.997	0.1300	2
		μ_i	596.560	-1.782	12.617		
	2	α_i	1.490	2.528	-2.327	0.1490	12
		μ_i	653.473	-2.000	9.695		
set 2	1	α_i	1.657	7.000	-0.299	0.1317	2
		μ_i	580.932	-1.992	11.508		
	2	α_i	1.409	5.262	-2.200	0.1552	2
		μ_i	689.442	1.387	15.000		



(a) uni-axial (b) equi-biaxial

Figure 6.4: Evolution of absolute error between solutions from Table 6.5 and homogenized response.

methods.

7 Conclusions

7.1 Main conclusions

The goal of the present work is to implement computational methods for optimization, and apply them to solve some problems. Classical methods and genetic algorithms, for both unidimensional or n-dimensional problems, are the main subject in the present document.

In first place, two methods for minimizing functions of one variable are introduced: polynomial approximations and the golden section method. The importance of these methods is mainly related to the search of the optimal step length. The golden section method is chosen to perform unidirectional search in n-dimensional problems, due to its robustness. In spite of requiring a large number of function evaluations, in the problems tackled here it is not computationally expensive.

In what refers to classical methods for the minimization of n variable functions, it is clear that the steepest descent method has a very poor convergence rate. On the other side, the Newton's method converge in very few iterations. The main drawback related to this method is that it requires the computation of the Hessian matrix, which increases the number of function evaluations per iteration. Quasi-Newton methods also have good convergence rate, and do not need the computation of second-order derivatives.

The performance verified with genetic algorithms is strongly dependent on the chosen algorithmic parameters. However, there are not closed-form formulae to define these parameters, that must be set in an empirical way, according to the peculiarities o each problem. Precision of variables should be defined as only the enough, in order to avoid chromosomes with too many genes. Increasing population size allows to better explore the design space, but more function evaluations must be performed. Mutation probability must be low, to introduce some randomness without destroying eventual good properties of the offspring. Increasing the number of survivals leads to faster convergence to the final solution, but if too much survivals exist, it may converge to a solution that is not close to the minimum. Therefore, a compromise solution must be taken, and several computations with distinct algorithmic parameters may be performed.

Finally, if the objective function shows a "good behaviour", i.e., if it is smooth, without discontinuities, and with only one minimum in the design space, then classical methods are the right choice, since they easily find the minimum value. On the contrary, for hard optimization problems, with complex objective functions, genetic algorithms are more likely to achieve the global minimum, whereas the former converge to a local minimum close to the initial guess. This fact becomes clear in the problem of parameter identification, where the objective function is built to compare the difference between a finite number of points, which causes it to assume a complex topology. Hybrid techniques are suitable for this type of problems, since in first place approximate solutions of the global minimum are obtained with genetic algorithms, and then they are refined through classical methods.

7.2 Future Work

Suggestions for subsequent tasks arising from the present work are presented.

The developed program may be considered as a draft for a more complete and reliable tool. Several improvements may be implemented. In first place, more techniques could be implemented, like memetic algorithms, since it was developed in a modular sense. Sophistication of the already implemented algorithms is not less important. Especially for Powell's method, reinitialization of the conjugate direction base should be done whenever a column of the matrix becomes null. Unidirectional search should also be optimized in a computational sense. Note that the golden section method requires a great number of function evaluations, therefore, the combination of this method with polynomial approximation is a wise solution.

In what refers to the proposed genetic algorithm, the main correction that should be implemented is to avoid the existence of distinct individuals with exactly the same genetic information. This situation is not favourable to the performance of this method. It also would be interesting to perform a deeper and systematized study on the influence of the algorithmic variables on the behaviour of genetic algorithms.

A GPROPT source code

A.1 Main routines

Main routine:

```
! GPROPT - Generalized PROgram for OPTimization
!
! The purpose of this program is to be general enough so
    that several
! optimization algorithms can be implemented
!
! Igor Lopes, February 2015
PROGRAM GPROPT
    IMPLICIT NONE
! Declaration
    INTEGER :: NALGO, NDIM
! Begin algorithm
01  WRITE(*,*)'Introduce␣number␣of␣design␣variables␣(0␣-␣
     QUIT):'
    READ(*,*) NDIM
    IF(NDIM.LT.0)THEN
        WRITE(*,*)'Number␣of␣design␣variables␣must␣be␣
            larger␣or␣equal␣to␣1'
        GOTO 01
    ELSEIF(NDIM.EQ.1)THEN
        WRITE(*,*)'Uni-dimensional␣problem'
        GOTO 11
    ELSEIF(NDIM.EQ.0)THEN
        GOTO 99
    ENDIF
10  WRITE(*,*)'Select␣Optimization␣algorithm:'
    WRITE(*,*)'1␣-␣Powell␣Method'
    WRITE(*,*)'2␣-␣Steepest␣Descent'
    WRITE(*,*)'3␣-␣Fletcher-Reeves␣Conjugate␣Direction'
    WRITE(*,*)'4␣-␣Polak-Ribiere␣Conjugate␣Direction'
    WRITE(*,*)'5␣-␣Quasi-Newton␣Methods␣(Variable␣Metrics
        )'
    WRITE(*,*)'6␣-␣Newton␣Method'
    WRITE(*,*)'7␣-␣Genetic␣Algorithm'
    WRITE(*,*)'0␣-␣QUIT'
    WRITE(*,*)'(Introduce␣corresponding␣number)'
    READ(*,*) NALGO

    SELECT CASE(NALGO)
    CASE (1)
        WRITE(*,*)'Powell␣s␣Method'
        CALL POWELL(NDIM)
    CASE(2)
        WRITE(*,*)'Steepest␣Descent'
        CALL STEEPDESC(NDIM)
    CASE(3)
        WRITE(*,*)'Fletcher-Reeves␣Conjugate␣Direction'
        CALL FLETCHERREEVES(NDIM)
    CASE(4)
        WRITE(*,*)'Polak-Ribiere␣Conjugate␣Direction'
        CALL POLAKRIBIERE(NDIM)
    CASE(5)
        WRITE(*,*)'Quasi-Newton␣Methods'
        CALL QUASINEWTON(NDIM)
    CASE(6)
        WRITE(*,*)'Newton␣Method'
        CALL NEWTON(NDIM)
    CASE(7)
        WRITE(*,*)'Genetic␣Algorithm'
        CALL GENETIC(NDIM)
    CASE DEFAULT
        WRITE(*,*)'Invalid␣option!'
        GOTO 10
    CASE(0)
        GOTO 99

    END SELECT
    GOTO 01
11  CONTINUE
    !1D
20  WRITE(*,*)'Select␣Optimization␣algorithm:'
    WRITE(*,*)'1␣-␣Polynomial␣approximation'
    WRITE(*,*)'2␣-␣Golden␣Section␣Method'
    WRITE(*,*)'3␣-␣Genetic␣Algorithm'
    WRITE(*,*)'0␣-␣QUIT'
    WRITE(*,*)'(Introduce␣corresponding␣number)'
    READ(*,*) NALGO

    SELECT CASE(NALGO)
    CASE (1)
        WRITE(*,*)'Polynomial␣approximation'
        CALL POLYINT
    CASE(2)
        WRITE(*,*)'Golden␣Section␣Method'
        CALL GOLDENINT
    CASE(3)
        WRITE(*,*)'Genetic␣Algorithm'
        CALL GENETIC(NDIM)
    CASE DEFAULT
        WRITE(*,*)'Invalid␣option!'
        GOTO 20
    CASE(0)
        GOTO 99
    END SELECT
    GOTO 01

99  CONTINUE
    WRITE(*,*)'CLOSING...'
END PROGRAM
```

Interface routine for polynomial approximation (1 variable):

```
! Interface routine for uni-dimensional optimization
    through
! polynomial approximation.
! Igor Lopes, February 2015
SUBROUTINE POLYINT
    IMPLICIT NONE

    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/

    REAL(8),DIMENSION(4) ::  X,F
    REAL(8) ::  XMIN, FMIN
    INTEGER ::  NPOLY
    CHARACTER STRING*256

    ! Innititalization
    X=R0; F=R0

101 FORMAT('Linear␣approximation:␣F=a0+a1*X')
102 FORMAT('Quadratic␣approximation:␣F=a0+a1*X+a2*X^2')
103 FORMAT('Cubic␣approximation:␣F=a0+a1*X+a2*X^2+a3*X^3'
    )
110 FORMAT("Insert␣derivative␣F'_1")
111 FORMAT('Insert␣point␣X_1,F_1')
112 FORMAT('Insert␣point␣X_2,F_2')
113 FORMAT('Insert␣point␣X_3,F_3')
114 FORMAT('Insert␣point␣X_4,F_4')
120 FORMAT("F'_1=",F10.5)
121 FORMAT('X_1=',F10.5,2X,'F_1=',F10.5)
122 FORMAT('X_2=',F10.5,2X,'F_2=',F10.5)
123 FORMAT('X_3=',F10.5,2X,'F_3=',F10.5)
124 FORMAT('X_4=',F10.5,2X,'F_4=',F10.5)
    ! Begin algorithm
10  WRITE(*,*)'Choose␣kind␣of␣polynomial␣approximation:'
    WRITE(*,*)'1␣-␣Linear␣1-point'
    WRITE(*,*)'2␣-␣Linear␣2-point'
    WRITE(*,*)'3␣-␣Quadratic␣2-point'
    WRITE(*,*)'4␣-␣Quadratic␣3-point'
    WRITE(*,*)'5␣-␣Cubic␣3-point'
    WRITE(*,*)'6␣-␣Cubic␣4-point'
    WRITE(*,*)'0␣-␣QUIT'
    READ(*,*) NPOLY

    SELECT CASE(NPOLY)
    CASE(0)
        WRITE(*,*)'CLOSING...'
        STOP
    CASE(1)
        STRING='linear1.res'
        CALL RESULTFILE(STRING)
        WRITE(11,101)
        WRITE(*,101)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,110);READ(*,*)F(2)
        WRITE(11,121)X(1),F(1)
        WRITE(11,120)F(2)
        CALL POLYLIN(1,X,F)
    CASE(2)
        STRING='linear2.res'
        CALL RESULTFILE(STRING)
```

```
        WRITE(11,101)
        WRITE(*,101)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,112);READ(*,*)X(2),F(2)
        WRITE(11,121)X(1),F(1)
        WRITE(11,122)X(2),F(2)
        CALL POLYLIN(2,X,F)
    CASE(3)
        STRING='quadratic2.res'
        CALL RESULTFILE(STRING)
        WRITE(11,102)
        WRITE(*,102)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,110);READ(*,*)F(3)
        WRITE(*,112);READ(*,*)X(2),F(2)
        WRITE(11,121)X(1),F(1)
        WRITE(11,120)F(3)
        WRITE(11,122)X(2),F(2)
        CALL POLYQUAD(2,X,F,XMIN,FMIN)
    CASE(4)
        STRING='quadratic3.res'
        CALL RESULTFILE(STRING)
        WRITE(11,102)
        WRITE(*,102)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,112);READ(*,*)X(2),F(2)
        WRITE(*,113);READ(*,*)X(3),F(3)
        WRITE(11,121)X(1),F(1)
        WRITE(11,122)X(2),F(2)
        WRITE(11,123)X(3),F(3)
        CALL POLYQUAD(3,X,F,XMIN,FMIN)
    CASE(5)
        STRING='cubic3.res'
        CALL RESULTFILE(STRING)
        WRITE(11,103)
        WRITE(*,103)
        WRITE(*,110);READ(*,*)F(4)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,112);READ(*,*)X(2),F(2)
        WRITE(*,113);READ(*,*)X(3),F(3)
        WRITE(11,121)X(1),F(1)
        WRITE(11,120)F(4)
        WRITE(11,122)X(2),F(2)
        WRITE(11,123)X(3),F(3)
        CALL POLYCUBIC(3,X,F,XMIN,FMIN)
    CASE(6)
        STRING='cubic4.res'
        CALL RESULTFILE(STRING)
        WRITE(11,103)
        WRITE(*,103)
        WRITE(*,111);READ(*,*)X(1),F(1)
        WRITE(*,112);READ(*,*)X(2),F(2)
        WRITE(*,113);READ(*,*)X(3),F(3)
        WRITE(*,114);READ(*,*)X(4),F(4)
        WRITE(11,121)X(1),F(1)
        WRITE(11,122)X(2),F(2)
        WRITE(11,123)X(3),F(3)
        WRITE(11,124)X(4),F(4)
        CALL POLYCUBIC(4,X,F,XMIN,FMIN)
    CASE DEFAULT
        WRITE(*,*)'Invalid option!'
        GOTO 10
    END SELECT
    CLOSE(UNIT=11)
END SUBROUTINE
```

Interface routine for unidimensional minimization through the golden section method:

```
! Interface routine for uni-dimensional optimization
     through
! the golden section method.
! Igor Lopes, February 2015
SUBROUTINE GOLDENINT
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    !Locals
    REAL(8),DIMENSION(2) ::  X0
    REAL(8),DIMENSION(4) ::  XF, F
    REAL(8) ::  TOL, GOLD, XI, XMIN, XMAX, FMIN,
         EVALFUNC1,FI
    INTEGER ::  I , NITER
    CHARACTER STRING*256,ANS*1

    ! Innititalization
    X0=R0; XF=R0

    GOLD=(R1+DSQRT(R5))/R2
    STRING='goldensection.res'
    !
    CALL RESULTFILE(STRING)
    ! Begin algorithm
    WRITE(*,*)'Automatic bounds search?(Y/N)'
    READ(*,*)ANS
    IF(ANS.EQ.'Y'.OR.ANS.EQ.'y')THEN
        WRITE(*,*)'Give a lower limit for X...'
        READ(*,*)XMIN
        WRITE(*,*)'and an upper limit...'
        READ(*,*)XMAX
        WRITE(*,*)'and an initial point for the search.'
        READ(*,*)XI
        FI=EVALFUNC1(XI)
        CALL FINDBOUNDS1D(XI,FI,X0(1),X0(2),XMAX,XMIN)
        WRITE(*,*)'Automatic bounds'
        WRITE(11,*)'Automatic bounds'
        WRITE(*,*)'X_l=',X0(1),'X_u=',X0(2)
        WRITE(11,*)'X_l=',X0(1),'X_u=',X0(2)
        GOTO 10
    ENDIF

    WRITE(*,*)'Insert lower and upper bounds:'
    WRITE(*,*)'X_lower='
    READ(*,*)X0(1)
    WRITE(*,*)'X_upper='
    READ(*,*)X0(2)
10  WRITE(*,*)'Define relative tolerance:'
    READ(*,*)TOL
    WRITE(11,*)'Tolerance:',TOL
    ! Number of iterations
    NITER=INT(LOG(TOL)/LOG(GOLD-R1))+1
    ! Determine minimum
    CALL GOLDENSECTION(X0,XF,F,NITER)
    XMIN=XF(1)
    FMIN=F(1)
    DO I=2,4
        IF(F(I).LT.FMIN)THEN
            FMIN=F(I)
            XMIN=XF(I)
        ENDIF
    ENDDO
    WRITE(*,*) 'Minimum: X_min=',XMIN,' with F_min=',FMIN
    WRITE(11,*) 'Minimum: X_min=',XMIN,' with F_min=',
         FMIN
    CLOSE(UNIT=11)
END SUBROUTINE
```

Routine that opens output file:

```
! This routine opens a file whose name is given
! in STRING variable
!
! Igor Lopes, February 2015
SUBROUTINE RESULTFILE(STRING)
    IMPLICIT NONE
    CHARACTER STRING*256

    OPEN(UNIT=11,FILE=STRING,STATUS="UNKNOWN")
END SUBROUTINE
```

Evaluate function of one variable:

```
! Evaluate function of one variable X
DOUBLE PRECISION FUNCTION EVALFUNC1(X)
    IMPLICIT NONE
    !
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R3 /3.0D0/
    !
    REAL(8) :: X
    !
    EVALFUNC1=R1-R3*X+EXP(R2*X)
    RETURN
END FUNCTION
```

Evaluate function of n variables:

```
! Evaluate function of NDIM variables,
! stored in vector X
DOUBLE PRECISION FUNCTION EVALFUNC(X,NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) RP5 /0.5D0/
    ! Arguments
    INTEGER     :: NDIM
    REAL(8),DIMENSION(NDIM) :: X
    ! Locals
    INTEGER :: I
    REAL(8) :: A1,A2,K1,K2,L1,L2,P1,P2
    ! Algorithm - 2 spring problem
    K1=8.0E0    ! Spring constant 1 (N/cm)
    K2=1.0E0    ! Spring constant 2 (N/cm)
    L1=10.0E0   ! Initial length 1 (cm)
    L2=10.0E0   ! Initial length 2 (cm)
    P1=5.0E0    ! Force 1 (N)
    P2=5.0E0    ! Force 2 (N)

    A1=DSQRT(X(1)*X(1)+(L1-X(2))**2)
    A2=DSQRT(X(1)*X(1)+(L2+X(2))**2)
    EVALFUNC=RP5*K1*(A1-L1)**2+RP5*K2*(A2-L2)**2-P1*X(1)-
        P2*X(2)
99  RETURN
END FUNCTION
```

A.2 Math routines

Function that computes real value of a binary string:

```
! Converts the binary string in BIN to its real value
! Igor Lopes, February 2015
DOUBLE PRECISION FUNCTION BIN2REAL(BIN,NBIT)
    IMPLICIT NONE
    REAL(8) R0 /0.0D0/
    REAL(8) R2 /2.0D0/
    ! Arguments
    INTEGER NBIT
    INTEGER,DIMENSION(NBIT) :: BIN
    ! Locals
    INTEGER :: IBIT
    ! Begin algorithm
    BIN2REAL=R0
    DO IBIT=1,NBIT
        BIN2REAL=BIN2REAL+BIN(IBIT)*R2**(NBIT-IBIT)
    ENDDO
    RETURN
END FUNCTION
```

Routine that computes the gradient of a function:

```
! Sub-routine that computes the gradient of the objective
     function.
! It can be computed through the finite difference method
    , or defined
! by its analytical expression.
! Igor Lopes, February 2015
!
! This specific case refers to the equilibrium of a 2
    spring system,
! presented on page 72 of "Numerical Optimization
    Techniques for
! Engineering Design", by Vanderplaats.
SUBROUTINE GRADIENT(GRAD,X,NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R3 /3.0D0/
    REAL(8) PERT /1.0D-4/
    ! Arguments
    INTEGER     :: NDIM
    REAL(8),DIMENSION(NDIM) :: X,GRAD
    ! Locals
    INTEGER :: I
    REAL(8) :: A1,A2,K1,K2,L1,L2,P1,P2,F1,F,EVALFUNC
    REAL(8),DIMENSION(NDIM) :: X1
    ! Algorithm
    K1=8.0E0    ! Spring constant 1 (N/cm)
    K2=1.0E0    ! Spring constant 2 (N/cm)
    L1=10.0E0   ! Initial length 1 (cm)
    L2=10.0E0   ! Initial length 2 (cm)
    P1=5.0E0    ! Force 1 (N)
    P2=5.0E0    ! Force 2 (N)
    !
    A1=DSQRT(X(1)*X(1)+(L1-X(2))**2)
    A2=DSQRT(X(1)*X(1)+(L2+X(2))**2)
    GRAD(1)=K1*X(1)*(A1-L1)/A1+K2*X(1)*(A2-L2)/A2-P1
    GRAD(2)=K1*(X(2)-L1)*(A1-L1)/A1+K2*(X(2)+L2)*(A2-L2)/
        A2-P2
    ! Alternative: finite differences method
    !F=EVALFUNC(X,NDIM)
    !DO I=1,NDIM
    !    X1=X
    !    X1(I)=X1(I)+PERT
    !    F1=EVALFUNC(X1,NDIM)
    !    GRAD(I)=(F1-F)/PERT
    !ENDDO
END SUBROUTINE
```

Routine that computes the Hessian matrix of a function:

```
! Sub-routine that computes the Hessian matrix of a
    function,
! through the finite difference method.
! Igor Lopes, February 2015
SUBROUTINE HESSIAN(H,X,NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R4 /4.0D0/
    REAL(8) PERT /1.0D-4/
    ! Arguments
    INTEGER     :: NDIM
    REAL(8),DIMENSION(NDIM) :: X
    REAL(8),DIMENSION(NDIM,NDIM) :: H
    ! Locals
    INTEGER :: I,J
    REAL(8) :: F0,F1,F2,F3,F4,EVALFUNC
    REAL(8),DIMENSION(NDIM) :: X1,X2,X3,X4
    ! Algorithm
    F0=EVALFUNC(X,NDIM)
    DO I=1,NDIM
        DO J=1,NDIM
            IF(I.EQ.J)THEN
                X1=X
                X2=X
                X1(I)=X(I)+PERT
                X2(I)=X(I)-PERT
                F1=EVALFUNC(X1,NDIM)
                F2=EVALFUNC(X2,NDIM)
                H(I,I)=(F1-R2*F0+F2)/(PERT*PERT)
            ELSE
                X1=X
                X2=X
                X3=X
                X4=X
                X1(I)=X(I)+PERT
                X1(J)=X(J)+PERT
                X2(I)=X(I)+PERT
                X2(J)=X(J)-PERT
```

```
                X3(I)=X(I)-PERT
                X3(J)=X(J)+PERT
                X4(I)=X(I)-PERT
                X4(J)=X(J)-PERT
                F1=EVALFUNC(X1,NDIM)
                F2=EVALFUNC(X2,NDIM)
                F3=EVALFUNC(X3,NDIM)
                F4=EVALFUNC(X4,NDIM)
                H(I,J)=(F1-F2-F3+F4)/(R4*PERT*PERT)
            ENDIF
        ENDDO
    ENDDO
END SUBROUTINE
```

Routine for matrix product:

```
! Sub-routine that computes the Hessian matrix of a
    function,
! through the finite difference method.
! Igor Lopes, February 2015
SUBROUTINE HESSIAN(H,X,NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R4 /4.0D0/
    REAL(8) PERT /1.0D-4/
    ! Arguments
    INTEGER     :: NDIM
    REAL(8),DIMENSION(NDIM) :: X
    REAL(8),DIMENSION(NDIM,NDIM) :: H
    ! Locals
    INTEGER :: I,J
    REAL(8) :: F0,F1,F2,F3,F4,EVALFUNC
    REAL(8),DIMENSION(NDIM) :: X1,X2,X3,X4
    ! Algorithm
    F0=EVALFUNC(X,NDIM)
    DO I=1,NDIM
        DO J=1,NDIM
            IF(I.EQ.J)THEN
                X1=X
                X2=X
                X1(I)=X(I)+PERT
                X2(I)=X(I)-PERT
                F1=EVALFUNC(X1,NDIM)
                F2=EVALFUNC(X2,NDIM)
                H(I,I)=(F1-R2*F0+F2)/(PERT*PERT)
            ELSE
                X1=X
                X2=X
                X3=X
                X4=X
                X1(I)=X(I)+PERT
                X1(J)=X(J)+PERT
                X2(I)=X(I)+PERT
                X2(J)=X(J)-PERT
                X3(I)=X(I)-PERT
                X3(J)=X(J)+PERT
                X4(I)=X(I)-PERT
                X4(J)=X(J)-PERT
                F1=EVALFUNC(X1,NDIM)
                F2=EVALFUNC(X2,NDIM)
                F3=EVALFUNC(X3,NDIM)
                F4=EVALFUNC(X4,NDIM)
                H(I,J)=(F1-F2-F3+F4)/(R4*PERT*PERT)
            ENDIF
        ENDDO
    ENDDO
END SUBROUTINE
```

Routine for transpose a matrix:

```
! Subroutine for determine matrix transpose AT=A^T
!
! Igor Lopes, February 2015
SUBROUTINE TRANSPOSE(A,AT,NROWA,NCOLA)
    IMPLICIT NONE
    ! Arguments
    INTEGER :: NROWA , NCOLA
    REAL(8), DIMENSION(NROWA,NCOLA)  :: A
    REAL(8), DIMENSION(NCOLA, NROWA) :: AT
    ! Locals
    INTEGER I, J
    ! Begin algorithm
    DO J=1,NCOLA
            DO I=1,NROWA
                    AT(J,I)=A(I,J)
            ENDDO
    ENDDO
END SUBROUTINE
```

The following routines are not implemented by the author of this work, but are presented here since they are used in this program. A scalar product function is presented now:

```
! BEGIN_DOUBLE_PRECISION_FUNCTION SCAPRD
! Scalar product of double precision vectors
!
! This function returns the scalar product between its
    two double
! precision vector arguments \smparm{U}.\smparm{V}.
!
! BEGIN_PARAMETERS
! DOUBLE_PRECISION U      >  Array of components of a
    double
! C                          precision vector.
! DOUBLE_PRECISION V      >  Array of components of a
    double
! C                          precision vector.
! INTEGER          N      >  Dimension of \smparm{U} and
    \smparm{V}.
! END_PARAMETERS
!
! E.de Souza Neto, May 1996: Initial coding
!
DOUBLE PRECISION FUNCTION SCAPRD (U,  V,  N)
!
      IMPLICIT NONE
      !DATA DECLARATION
      REAL(8) R0  /0.0D0/

      !SCALAR VARIABLES FROM ARGUMENTS
      INTEGER N

      !ARRAYS FROM ARGUMENTS
      REAL(8), DIMENSION(N):: U
      REAL(8), DIMENSION(N):: V

      !LOCAL SCALAR VARIABLES
      INTEGER I
      I=0
! SCALAR PRODUCT OF DOUBLE PRECISION VECTORS U AND V OF
    DIMENSION N
      SCAPRD=R0
      DO 10 I=1,N
        SCAPRD=SCAPRD+U(I)*V(I)
  10  CONTINUE
      RETURN
END
```

The following three routines are needed to compute the inverse of a matrix:

```
SUBROUTINE RMINVE (A , AI , NSIZE , ERROR)
IMPLICIT NONE
!PARAMETER DECLARATION
INTEGER, PARAMETER:: MSIZE=500
!DATA DECLARATION
REAL(8) R0     /0.0D0/
REAL(8) R1     /1.0D0/
CHARACTER*6 NAME
DATA NAME/'RMINVE'/
!SCALAR VARIABLES FROM ARGUMENTS
INTEGER NSIZE
LOGICAL ERROR
!ARRAYS FROM ARGUMENTS
REAL(8), DIMENSION(NSIZE,NSIZE):: A
REAL(8), DIMENSION(NSIZE,NSIZE):: AI
!LOCAL SCALAR VARIABLES
INTEGER ISIZE, JSIZE
REAL(8) DUMMY, DETA
!LOCAL ARRAYS
INTEGER, DIMENSION(NSIZE):: INDX
!INITIALIZE LOCAL VARIABLES
ISIZE=0  ; JSIZE=0
DUMMY=R0 ; DETA=R0
INDX=0
!*********************************************
! Evaluate inverse matrix without determinant
!*Arrays
! A      - A matrix
!=AI     - inversed A matrix
!*Variables
! NSIZE - Size of the matrix
!=ERROR - Error flag
!*********************************************
! Set up identity matrix
! ----------------------
ERROR=.FALSE.
DO 20 ISIZE=1,NSIZE
        DO 10 JSIZE=1,NSIZE
                AI(ISIZE,JSIZE)=R0
        10    CONTINUE
        AI(ISIZE,ISIZE)=R1
20 CONTINUE
! LU decompose the matrix
CALL LUDCMPX(A     ,INDX   ,DUMMY,NSIZE ,NSIZE ,ERROR )
DETA=DUMMY
! Find inverse by columns
DO 30 ISIZE=1,NSIZE
        CALL LUBKSBX(A      ,AI(1,ISIZE),INDX   ,NSIZE ,
             NSIZE )
! Obtain determinant
        DETA=DETA*A(ISIZE,ISIZE)
30 CONTINUE
999 CONTINUE
RETURN
END


SUBROUTINE LUDCMPX (A , INDX , D , N , NP , ERROR )
IMPLICIT NONE
!PARAMETER DECLARATION
INTEGER, PARAMETER:: NMAX=50
!DATA DECLARATION
REAL(8) R0     /0.0D0/
REAL(8) R1     /1.0D0/
REAL(8) TINY   /1.0D-19/
CHARACTER          NAME*6
DATA NAME/'LUDCMP'/
!SCALAR VARIABLES FROM ARGUMENTS
INTEGER N, NP
LOGICAL ERROR
REAL(8) D
!ARRAYS FROM ARGUMENTS
REAL(8), DIMENSION(NP,NP):: A
INTEGER, DIMENSION(N):: INDX
!LOCAL SCALAR VARIABLES
INTEGER I, J, K, IMAX
REAL(8) AAMAX, SUM, DUM
!LOCAL ARRAYS
REAL(8), DIMENSION(N):: VV
!INTIALIZE LOCAL SCALAR VARIABLES
I=0      ; J=0     ; K=0     ; IMAX=0
AAMAX=R0 ; SUM=R0 ; DUM=R0 ; VV=R0
!*********************************************
! Routine to do LU decomposition
! See NUMERICAL RECIPES p35.
!*ACRONYM
! LU_DeCOmPosition
!*DESCRIPTION
!*EXTERNAL
! Arrays
!=A       - LU decomposed matrix
!=INDX    - Permutation vector
! Variables
!=D       - Row interchange indicator
! N       - Size of the problem
! NP      - Physical size of A matrix ( NP >= N )
! ERROR   - Error flag
!*INTERNAL
! Arrays
! VV      - store the implicit scalling of each rows
!*********************************************
! Loop over each rows to get the implicit scalling
    factors
!
     ----------------------------------------------------------


DO 12 I=1,N
        AAMAX=R0
        DO 11 J=1,N
                IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J
                    ))
        11    CONTINUE
        IF(AAMAX.EQ.R0)THEN
                ERROR=.TRUE.
                GOTO 999
        ENDIF
        VV(I)=R1/AAMAX
12 CONTINUE
! Loop over columns of Crout's methods
! ------------------------------------
DO 19 J=1,N
        IF (J.GT.1) THEN
! Calculate upper triangle components Bij see equation
     (2.3.12), except i=j
                DO 14 I=1,J-1
                        SUM=A(I,J)
                        IF (I.GT.1)THEN
                                DO 13 K=1,I-1
                                        IF(ABS(A(I,K)).LT
                                             .TINY.AND.
                                             ABS(A(K,J))
                                             .LT.TINY)
                                             GOTO 13
! If A(I,K) and A(K,J) are very small, skip the following
      line
                                        SUM=SUM-A(I,K)*A(
                                             K,J)
                                13          CONTINUE
                                A(I,J)=SUM
                        ENDIF
                14      CONTINUE
        ENDIF
! Initialize for the search for largest pivot element
        AAMAX=R0
! Calculate diagonal components Bij see equation(2.3.12)
    ,i=j, and
! lower triangle components aij see equation 2.3.13, i=j
    +1,..N
        DO 16 I=J,N
                SUM=A(I,J)
                IF (J.GT.1)THEN
                        DO 15 K=1,J-1
                                IF(ABS(A(I,K)).LT.TINY.
                                     AND.ABS(A(K,J)).LT.
                                     TINY)GOTO 15
! If A(I,K) and A(K,J) are very small, skip the following
      line.
                                SUM=SUM-A(I,K)*A(K,J)
                        15       CONTINUE
                        A(I,J)=SUM
                ENDIF
! Find largest pivot element
                DUM=VV(I)*ABS(SUM)
                IF (DUM.GE.AAMAX) THEN
                        IMAX=I
                        AAMAX=DUM
                ENDIF
        16   CONTINUE
! Interchange rows
        IF(J.NE.IMAX)THEN
                DO 17 K=1,N
                        DUM=A(IMAX,K)
                        A(IMAX,K)=A(J,K)
                        A(J,K)=DUM
                17     CONTINUE
! Change sign of row channge indicator and interchange
     scale factor
                D=-D
                VV(IMAX)=VV(J)
        ENDIF
! record interchange row number
        INDX(J)=IMAX
! Divided by the pivot element and get final aij values,
     see equation 2.3.13,
! i=j+1,..N
        IF(J.NE.N)THEN
                IF(ABS(A(J,J)).LT.TINY)A(J,J)=TINY
                DUM=R1/A(J,J)
                DO 18 I=J+1,N
                        A(I,J)=A(I,J)*DUM
                18     CONTINUE
        ENDIF
! Go back for next column
19 CONTINUE
IF(ABS(A(N,N)).LT.TINY)A(N,N)=TINY
999 CONTINUE
```

```
RETURN
END

SUBROUTINE LUBKSBX (A , B , INDX ,  N , NP)
IMPLICIT NONE
!DATA DECLARATION
REAL(8) R0     /0.0D0/
CHARACTER          NAME*6
DATA NAME/'LUBKSB'/
!SCALAR VARIABLES FROM ARGUMENTS
INTEGER N, NP
!ARRAYS FROM ARGUMENTS
REAL(8), DIMENSION(NP,NP):: A
REAL(8), DIMENSION(N):: B
INTEGER, DIMENSION(N):: INDX
!LOCAL SCALAR VARIABLES
INTEGER II, I, LL, J
REAL(8) SUM
!INITIALIZE LOCAL SCALAR VARIABLES
II=0;    I=0;     LL=0;    J=0;
SUM=R0;
!*********************************************
! Routine to solve the set of N linear equations AX=B
! See NUMERICAL RECIPES p36.
!*ACRONYM
! LU_BacK_SuBustitutions
!*DESCRIPTION
!*EXTERNAL
! Arrays
! A       - LU decomposed matrix
!=B       - Right hand side matrix as input and stored
     solutions as output
! INDX    - Permutation vector
! Variables
! N       - Size of the problem
! NP      - Physical size of A matrix
!*********************************************
II=0
! Do forward substitution, equation (2.3.6)
DO 12 I=1,N
        LL=INDX(I)
        SUM=B(LL)
        B(LL)=B(I)
        IF (II.NE.0)THEN
                DO 11 J=II,I-1
                        SUM=SUM-A(I,J)*B(J)
                11      CONTINUE
        ELSEIF (SUM.NE.R0) THEN
                II=I
        ENDIF
        B(I)=SUM
12 CONTINUE
! Do the backsubstitution, equation (2.3.7)
DO 14 I=N,1,-1
        SUM=B(I)
        IF(I.LT.N)THEN
                DO 13 J=I+1,N
                        SUM=SUM-A(I,J)*B(J)
                13      CONTINUE
        ENDIF
! Store a component of the solution vector Xi
        B(I)=SUM/A(I,I)
14 CONTINUE
RETURN
END
```

A.3 Polynomial approximation routines

Linear approximation:

```
! Routine for linear approximation
! Igor Lopes, February 2015
SUBROUTINE POLYLIN(NPOINT,X,F)
    IMPLICIT NONE
    ! Arguments
    INTEGER NPOINT
    REAL(8) :: XMIN, FMIN
    REAL(8),DIMENSION(4) :: X , F
    ! Locals
    REAL(8) :: A0,A1
    ! Begin algorithm
    IF(NPOINT.EQ.1)THEN
        A1=F(2)
        A0=F(1)-F(2)*X(1)
    ELSEIF(NPOINT.EQ.2)THEN
        A1=(F(2)-F(1))/(X(2)-X(1))
        A0=F(1)-A1*X(1)
    ELSE
        WRITE(*,*)'Wrong number of input points in
             POLYLIN'
        GOTO 99
    ENDIF
    WRITE(*,*)'Approximation coefficients:'
    WRITE(*,*)'a0=',A0
    WRITE(*,*)'a1=',A1
    WRITE(11,*)'Approximation coefficients:'
    WRITE(11,*)'a0=',A0
    WRITE(11,*)'a1=',A1
99 CONTINUE
END SUBROUTINE
```

Quadratic approximation:

```
! Routine for quadratic approximation
! Igor Lopes, February 2015
SUBROUTINE POLYQUAD(NPOINT,X,F,XMIN,FMIN)
    IMPLICIT NONE
    REAL(8) R0 /0.0D0/
    REAL(8) R2 /2.0D0/
    ! Arguments
    INTEGER NPOINT
    REAL(8) :: XMIN, FMIN
    REAL(8),DIMENSION(4) :: X , F
    ! Locals
    REAL(8) :: A0,A1,A2
    ! Begin algorithm
    IF(NPOINT.EQ.2)THEN
        A2=((F(2)-F(1))/(X(2)-X(1))-F(3))/(X(2)-X(1))
        A1=F(3)-R2*A2*X(1)
        A0=F(1)-A1*X(1)-A2*X(1)*X(1)
    ELSEIF(NPOINT.EQ.3)THEN
        A2=((F(3)-F(1))/(X(3)-X(1))-(F(2)-F(1))/(X(2)-X
            (1)))/(X(3)-X(2))
        A1=(F(2)-F(1))/(X(2)-X(1))-A2*(X(1)+X(2))
        A0=F(1)-A1*X(1)-A2*X(1)*X(1)
    ELSE
        WRITE(*,*)'Wrong number of input points in
             POLYQUAD'
        GOTO 99
    ENDIF
    WRITE(*,*)'Approximation coefficients:'
    WRITE(*,*)'a0=',A0
    WRITE(*,*)'a1=',A1
    WRITE(*,*)'a2=',A2
    WRITE(11,*)'Approximation coefficients:'
    WRITE(11,*)'a0=',A0
    WRITE(11,*)'a1=',A1
    WRITE(11,*)'a2=',A2
    ! Minimum or Maximum
    IF(A2.LT.R0)THEN
        WRITE(11,*)'A maximum is found for'
        WRITE(*,*)'A maximum is found for'
    ELSEIF(A2.GT.R0)THEN
        WRITE(11,*)'A minimum is found for'
        WRITE(*,*)'A minimum is found for'
    ELSE
        WRITE(11,*)'Linear function. Neither minimum nor
             maximum.'
        GOTO 99
    ENDIF
    XMIN=-A1/(R2*A2)
    FMIN=A0+A1*XMIN+A2*XMIN*XMIN
    WRITE(11,*)'X=',XMIN,'F_approx=',FMIN
    WRITE(*,*)'X=',XMIN,'F_approx=',FMIN
99 CONTINUE
END SUBROUTINE
```

Cubic approximation:

```
! Routine for cubic approximation
! Igor Lopes, February 2015
SUBROUTINE POLYCUBIC(NPOINT,X,F,XMIN,FMIN)
    IMPLICIT NONE
    REAL(8) R0 /0.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R3 /3.0D0/
    REAL(8) SMALL /1.0D-8/
    ! Arguments
    INTEGER NPOINT
    REAL(8) :: XMIN, FMIN
    REAL(8),DIMENSION(4) :: X , F
    ! Locals
    REAL(8) :: A0,A1,A2,A3,Q1,Q2,Q3,Q4,Q5,Q6,XMAX,FMAX
    ! Begin algorithm
    IF(NPOINT.EQ.3)THEN
        Q1=(F(3)-F(1))/((X(3)-X(2))*(X(3)-X(1))**2)
        Q2=(F(2)-F(1))/((X(3)-X(2))*(X(2)-X(1))**2)
        Q3=F(4)/((X(3)-X(1))*(X(2)-X(1)))
        A3=Q1-Q2+Q3
        A2=((F(2)-F(1))/(X(2)-X(1))-F(4))/(X(2)-X(1))-A3
            *(R2*X(1)+X(2))
        A1=F(4)-R2*A2*X(1)-R3*A3*X(1)*X(1)
        A0=F(1)-A1*X(1)-A2*X(1)*X(1)-A3*X(1)*X(1)*X(1)
    ELSEIF(NPOINT.EQ.4)THEN
        Q1=X(3)**3*(X(2)-X(1))-X(2)**3*(X(3)-X(1))+X(1)
            **3*(X(3)-X(2))
        Q2=X(4)**3*(X(2)-X(1))-X(2)**3*(X(4)-X(1))+X(1)
            **3*(X(4)-X(2))
        Q3=(X(3)-X(2))*(X(2)-X(1))*(X(3)-X(1))
        Q4=(X(4)-X(2))*(X(2)-X(1))*(X(4)-X(1))
        Q5=F(3)*(X(2)-X(1))-F(2)*(X(3)-X(1))+F(1)*(X(3)-X
            (2))
        Q6=F(4)*(X(2)-X(1))-F(2)*(X(4)-X(1))+F(1)*(X(4)-X
            (2))
        A3=(Q3*Q6-Q4*Q5)/(Q2*Q3-Q1*Q4)
        A2=(Q5-A3*Q1)/Q3
        A1=(F(2)-F(1))/(X(2)-X(1))-A2*(X(1)+X(2))-A3*(X
            (2)**3-X(1)**3)/(X(2)-X(1))
        A0=F(1)-A1*X(1)-A2*X(1)*X(1)-A3*X(1)*X(1)*X(1)
    ELSE
        WRITE(*,*)'Wrong number of input points in
            POLYCUBIC'
        GOTO 99
    ENDIF
    WRITE(*,*)'Approximation coefficients:'
    WRITE(*,*)'a0=',A0
    WRITE(*,*)'a1=',A1
    WRITE(*,*)'a2=',A2
    WRITE(*,*)'a3=',A3
    WRITE(11,*)'Approximation coefficients:'
    WRITE(11,*)'a0=',A0
    WRITE(11,*)'a1=',A1
    WRITE(11,*)'a2=',A2
    WRITE(11,*)'a3=',A3
    ! Minimum or Maximum
    Q1=A2*A2-R3*A1*A3
    IF(Q1.LT.R0)THEN
        WRITE(*,*)'Unreal results'
        GOTO 99
    ELSEIF(Q1.LT.SMALL)THEN
        WRITE(*,*)'Neither minimum nor maximum is found'
        GOTO 99
    ENDIF
    XMIN=(-A2+DSQRT(Q1))/(R3*A3)
    FMIN=A0+A1*XMIN+A2*XMIN*XMIN+A3*XMIN**3
    XMAX=(-A2-DSQRT(Q1))/(R3*A3)
    FMAX=A0+A1*XMAX+A2*XMAX*XMAX+A3*XMAX**3
    WRITE(*,*)'A maximum is found for'
    WRITE(*,*)'X=',XMAX,'F_approx=',FMAX
    WRITE(*,*)'A minimum is found for'
    WRITE(*,*)'X=',XMIN,'F_approx=',FMIN
    WRITE(11,*)'A maximum is found for'
    WRITE(11,*)'X=',XMAX,'F_approx=',FMAX
    WRITE(11,*)'A minimum is found for'
    WRITE(11,*)'X=',XMIN,'F_approx=',FMIN
99 CONTINUE
END SUBROUTINE
```

A.4 Golden section routines

Golden section method for unidimensional minimization:

```
! Routine for uni-dimensional optimization through
! the golden section method.
! Igor Lopes, February 2015
SUBROUTINE GOLDENSECTION(XO,XF,F,NITER)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    ! Arguments
    REAL(8),DIMENSION(2) ::  XO
    REAL(8),DIMENSION(4) ::  XF, F
    INTEGER ::  NITER
    ! Locals
    REAL(8)      :: TOL, TAU, EVALFUNC1, GOLD
    INTEGER      :: ITER

    ! Initialize
    F=R0
    GOLD=(R1+DSQRT(R5))/R2
    ! Formats
105 FORMAT('
     -------------------------------------------------------
     ')
110 FORMAT('ITER.',6X,'X_l',13X,'X_1',13X,'X_2',13X,'X_u'
     ,/11X,'F_l',13X,'F_1',13X,'F_2',13X,'F_u',13X,'tol.
     ')
115 FORMAT(I3,4X,F12.6,4X,F12.6,4X,F12.6,4X,F12.6,/11X,
     G12.6,4X,G12.6,4X,G12.6,4X,G12.6,4X,F12.6)
    WRITE(11,105)
    WRITE(11,110)
    WRITE(11,105)
    TAU=R1/GOLD
    ! Begin algorithm
    XF(1)=XO(1)
    XF(4)=XO(2)
    F(1)=EVALFUNC1(XF(1))
    F(4)=EVALFUNC1(XF(4))
    XF(2)=XF(4)-TAU*(XF(4)-XF(1))
    XF(3)=XF(1)+TAU*(XF(4)-XF(1))
    F(2)=EVALFUNC1(XF(2))
    F(3)=EVALFUNC1(XF(3))
    TOL=(XF(4)-XF(1))/(XO(2)-XO(1))
    WRITE(11,115)0,XF(1),XF(2),XF(3),XF(4),F(1),F(2),F(3)
         ,F(4),TOL
    WRITE(11,105)
    DO ITER=1,NITER
        IF(F(2).GT.F(3))THEN
            XF(1)=XF(2)
            F(1)=F(2)
            XF(2)=XF(3)
            F(2)=F(3)
            XF(3)=XF(1)+TAU*(XF(4)-XF(1))
            F(3)=EVALFUNC1(XF(3))
        ELSEIF(F(2).LT.F(3))THEN
            XF(4)=XF(3)
            F(4)=F(3)
            XF(3)=XF(2)
            F(3)=F(2)
            XF(2)=XF(4)-TAU*(XF(4)-XF(1))
            F(2)=EVALFUNC1(XF(2))
        ELSE
            XF(1)=XF(2)
            F(1)=F(2)
            XF(4)=XF(3)
            F(4)=F(3)
            XF(2)=XF(4)-TAU*(XF(4)-XF(1))
            F(2)=EVALFUNC1(XF(2))
            XF(3)=XF(1)+TAU*(XF(4)-XF(1))
            F(3)=EVALFUNC1(XF(3))
        ENDIF
        TOL=(XF(4)-XF(1))/(XO(2)-XO(1))
        WRITE(11,115)ITER,XF(1),XF(2),XF(3),XF(4),F(1),F
            (2),F(3),F(4),TOL
        WRITE(11,105)
    ENDDO
END SUBROUTINE
```

Routine for finding initial bounds in unidimensional problems:

```
! Routine for finding initial bounds for the golden
! section method.
! Igor Lopes, February 2015
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! ARGUMENTS
!! XO,FO          >   Initial step and corresponding
     function
!! XL,XU          <   Lower and upper limits
!! XMAX,XMIN      >   Side constraints: limits for domain
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
SUBROUTINE FINDBOUNDS1D(XO,FO,XL,XU,XMAX,XMIN)
    IMPLICIT NONE
    ! Parameters
    REAL(8) GOLD /1.61803398875/
    REAL(8) R1 /1.0D0/
    INTEGER MITER /50/
    ! Arguments
    REAL(8)      :: XO,XL,XU,XMAX,XMIN,FO
    ! Locals
    REAL(8)      :: XN,X1,FU,FL,F1,EVALFUNC1
    INTEGER      :: I, NITER
    ! Begin algorithm
    XU=XO+R1
    FU=EVALFUNC1(XU)
    IF(FU.GE.FO)THEN
        ! Upper limit is found... but lower limit is not
        GOTO 10
    ENDIF
    XL=XO
    FL=FO
    DO
        X1=XU
        F1=FU
        XU=X1+(X1-XL)*GOLD
        IF(XU.GT.XMAX)THEN
            XU=XMAX
            GOTO 99
        ENDIF
        FU=EVALFUNC1(XU)
        IF(FU.GE.F1)GOTO 99
        XL=X1
        FL=F1
    ENDDO
    ! Find lower limit
10  XL=XO-R1
    FL=EVALFUNC1(XL)
    IF(FL.GE.FO)GOTO 99 !XL is found
    DO
        X1=XL
        F1=FL
        XL=X1-(XU-X1)*GOLD
        IF(XL.LT.XMIN)THEN
            XL=XMIN
            GOTO 99
        ENDIF
        FL=EVALFUNC1(XL)
        IF(FL.GE.F1)GOTO 99
        XU=X1
        FU=F1
    ENDDO
99  CONTINUE
END SUBROUTINE
```

Golden section method for n-dimensional minimization:

```
! Routine for n-dimensional optimization through
! the golden section method.
! Igor Lopes, February 2015
SUBROUTINE GOLDENSECTIONNDIM(XO,AO,AN,FN,NITER,SDIR,NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    ! Arguments
    INTEGER ::  NITER,NDIM
    REAL(8),DIMENSION(NDIM) ::  XO,SDIR
    REAL(8),DIMENSION(4) ::  AN, FN
    REAL(8),DIMENSION(2) ::  AO
    ! Locals
    INTEGER      :: ITER
    REAL(8)      :: TOL, TAU, EVALFUNC, GOLD
    REAL(8),DIMENSION(NDIM) ::  XN
    ! Initialize
    FN=R0
    GOLD=(R1+DSQRT(R5))/R2
    TAU=R1/GOLD
    ! Begin algorithm
    AN(1)=AO(1)
    AN(4)=AO(2)
    XN=XO+AN(1)*SDIR
    FN(1)=EVALFUNC(XN,NDIM)
    XN=XO+AN(4)*SDIR
    FN(4)=EVALFUNC(XN,NDIM)
    AN(2)=AN(4)-TAU*(AN(4)-AN(1))
    AN(3)=AN(1)+TAU*(AN(4)-AN(1))
    XN=XO+AN(2)*SDIR
    FN(2)=EVALFUNC(XN,NDIM)
    XN=XO+AN(3)*SDIR
    FN(3)=EVALFUNC(XN,NDIM)
    DO ITER=1,NITER
        IF(FN(2).GT.FN(3))THEN
            AN(1)=AN(2)
            FN(1)=FN(2)
            AN(2)=AN(3)
            FN(2)=FN(3)
            AN(3)=AN(1)+TAU*(AN(4)-AN(1))
            XN=XO+AN(3)*SDIR
            FN(3)=EVALFUNC(XN,NDIM)
        ELSEIF(FN(2).LT.FN(3))THEN
            AN(4)=AN(3)
            FN(4)=FN(3)
            AN(3)=AN(2)
            FN(3)=FN(2)
            AN(2)=AN(4)-TAU*(AN(4)-AN(1))
            XN=XO+AN(2)*SDIR
            FN(2)=EVALFUNC(XN,NDIM)
        ELSE
            AN(1)=AN(2)
            FN(1)=FN(2)
            AN(4)=AN(3)
            FN(4)=FN(3)
            AN(2)=AN(4)-TAU*(AN(4)-AN(1))
            XN=XO+AN(2)*SDIR
            FN(2)=EVALFUNC(XN,NDIM)
            AN(3)=AN(1)+TAU*(AN(4)-AN(1))
            XN=XO+AN(3)*SDIR
            FN(3)=EVALFUNC(XN,NDIM)
        ENDIF
    ENDDO
END SUBROUTINE
```

Routine for finding initial bounds in n-dimensional problems:

```
! Routine for determining step length bounds. It is used
     prior to the
! the golden section method for unidirectional search in
     n-dimensional
! problem.
! Igor Lopes, February 2015
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! ARGUMENTS
!! XO,FO          >   Initial point and corresponding
     function
!! ALPHL,ALPHU    <   Lower and upper limits
!! XMAX,XMIN      >   Side constraints: limits for domain
!! SDIR           >   Search direction vector
!! NDIM           >   Dimension of the problem (nr design
     variables)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
SUBROUTINE FINDBOUNDS(XO,FO,ALPHL,ALPHU,XMAX,XMIN,SDIR,
     NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    REAL(8) DELTA /1.0D-2/
    REAL(8) INF /1.0D-8/
    INTEGER MITER /50/
    ! Arguments
    INTEGER NDIM
    REAL(8),DIMENSION(NDIM) :: XO, XMAX, XMIN, SDIR
    REAL(8)      :: ALPHL, ALPHU, FO
    ! Locals
    REAL(8),DIMENSION(NDIM) :: XN
    REAL(8)      :: GOLD, ALPH1, AMAX, AMIN, F1, FU, FL,
         EVALFUNC, AUX
    INTEGER      :: I, NITER
    ! Initialize
    GOLD=(R1+DSQRT(R5))/R2
```

```
    AMAX=1.0E9
    AMIN=-1.0E9
    DO I=1,NDIM
        IF(DABS(SDIR(I)).LE.INF)GOTO 5
        IF(SDIR(I).GT.INF)THEN
            AUX=(XMAX(I)-XO(I))/SDIR(I)
            IF(AUX.LT.AMAX) AMAX=AUX
            AUX=(XMIN(I)-XO(I))/SDIR(I)
            IF(AUX.GT.AMIN) AMIN=AUX
        ELSE
            AUX=(XMIN(I)-XO(I))/SDIR(I)
            IF(AUX.LT.AMAX) AMAX=AUX
            AUX=(XMAX(I)-XO(I))/SDIR(I)
            IF(AUX.GT.AMIN) AMIN=AUX
        ENDIF
5       CONTINUE
    ENDDO
    IF(AMAX.LT.AMIN)THEN
        PAUSE
    ENDIF
    ! Begin algorithm
    ALPHU=DELTA*AMAX
    XN=XO+ALPHU*SDIR
    FU=EVALFUNC(XN,NDIM)
    IF(FU.GE.FO)THEN
        ! Upper limit is found... but lower limit is not
        GOTO 10
    ENDIF
    ALPHL=R0
    FL=FO
    DO
        ALPH1=ALPHU
        F1=FU
        ALPHU=ALPH1+(ALPH1-ALPHL)*GOLD
        IF(ALPHU.GT.AMAX) GOTO 15
        DO I=1,NDIM
            XN(I)=XO(I)+ALPHU*SDIR(I)
        ENDDO
        FU=EVALFUNC(XN,NDIM)
        IF(FU.GE.F1)GOTO 99
        ALPHL=ALPH1
        FL=F1
    ENDDO
    ! Find lower limit
10  ALPHL=DELTA*AMIN
    XN=XO+ALPHL*SDIR
    FL=EVALFUNC(XN,NDIM)
    IF(FL.GE.FO)GOTO 99 !XL is found
    DO
        ALPH1=ALPHL
        F1=FL
        ALPHL=ALPHL-(ALPHU-ALPH1)*GOLD
        IF(ALPHL.LT.AMIN)GOTO 20
        DO I=1,NDIM
            XN(I)=XO(I)+ALPHL*SDIR(I)
        ENDDO
        FL=EVALFUNC(XN,NDIM)
        IF(FL.GE.F1)GOTO 99
        ALPHU=ALPH1
        FU=F1
    ENDDO

15  ALPHU=AMAX
    DO I=1,NDIM
        XN(I)=XO(I)+ALPHU*SDIR(I)
    ENDDO
    FU=EVALFUNC(XN,NDIM)
    GOTO 99

20  ALPHL=AMIN
    DO I=1,NDIM
        XN(I)=XO(I)+ALPHL*SDIR(I)
    ENDDO
    FL=EVALFUNC(XN,NDIM)

99  CONTINUE
END SUBROUTINE
```

A.5 Classical optimization methods routines

Powell's method:

```
! Routine for n-dimensional optimization through
! the Powell's method.
! Igor Lopes, February 2015
SUBROUTINE POWELL(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) CONVTOL /1.0D-6/
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    REAL(8)                  :: GOLD, FO, TOL, EVALFUNC,
         AMIN, FMIN, RELDIF
    REAL(8),DIMENSION(NDIM) :: XO, XN, GRAD, SDIR, SDCJ,
         XMIN, XMAX
    REAL(8),DIMENSION(NDIM,NDIM) :: HMAT
    REAL(8),DIMENSION(2)     :: ALPHO
    REAL(8),DIMENSION(4)     :: ALPH, FN
    INTEGER                  :: I, IDIM, ITER, MITER,
         NITER
    CHARACTER                :: STRING*256
    ! Initialize
    STRING='powell.res'
    CALL RESULTFILE(STRING)
    HMAT=R0
    DO IDIM=1,NDIM
        HMAT(IDIM,IDIM)=R1
    ENDDO
    GOLD=(R1+DSQRT(R5))/R2
    ! Formats
105   FORMAT('----------------------------------------')
110   FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha'
     )
115   FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣Xmin␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
         ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    WRITE(11,*)'Maximum␣number␣of␣iterations=',MITER
    WRITE(11,*)'Tolerance␣for␣step␣search=',TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-1))+1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,105)
    WRITE(*,105)
    WRITE(11,110)(I,I=1,NDIM)
    WRITE(*,110)(I,I=1,NDIM)
    WRITE(11,105)
    WRITE(*,105)
    WRITE(11,115)0,XO,FO
    WRITE(*,115)0,XO,FO
    WRITE(11,105)
    WRITE(*,105)
    DO ITER=1,MITER
        SDCJ=R0
        DO IDIM=1,NDIM
            SDIR=HMAT(:,IDIM)
            CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,
                 XMIN,SDIR,NDIM)
            CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER
                 ,SDIR,NDIM)
            ! Determine minimum
            AMIN=ALPH(1)
            FMIN=FN(1)
            DO I=2,4
                IF(FN(I).LT.FMIN)THEN
                    FMIN=FN(I)
                    AMIN=ALPH(I)
                ENDIF
            ENDDO
            XN=XO+AMIN*SDIR
            XO=XN
            FO=FMIN
            HMAT(:,IDIM)=AMIN*SDIR
            SDCJ=SDCJ+AMIN*SDIR
            WRITE(11,115)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
            WRITE(*,115)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ENDDO
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
             ,SDCJ,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
             SDCJ,NDIM)
        ! Determine minimum
```

```
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDCJ
        WRITE(11,115)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,115)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(11,105)
        WRITE(*,105)
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        XO=XN
        FO=FMIN
        DO IDIM=1,NDIM-1
            HMAT(:,IDIM)=HMAT(:,IDIM+1)
        ENDDO
        HMAT(:,NDIM)=SDCJ
    ENDDO
99  WRITE(11,105)
    WRITE(*,105)
    CLOSE(UNIT=11)
END SUBROUTINE
```

Steepest descent method:

```
! Routine for n-dimensional optimization through
! the steepes descent method.
! Igor Lopes, February 2015
SUBROUTINE STEEPDESC(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) GOLD /1.61803398875D0/
    REAL(8) CONVTOL /1.0D-6/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    REAL(8)     :: FO,TOL,EVALFUNC,AMIN,FMIN,NORM,SCAPRD,
         RELDIF
    REAL(8),DIMENSION(NDIM) :: XO,XN,GRAD,SDIR,XMAX,XMIN
    REAL(8),DIMENSION(2)    :: ALPHO
    REAL(8),DIMENSION(4)    :: ALPH,FN
    INTEGER     :: I,ITER,MITER,NITER
    CHARACTER   :: STRING*256
    ! Initialize
    STRING='steepestdescent.res'
    CALL RESULTFILE(STRING)
    ! Formats
5   FORMAT('-----------------------------------------')
10  FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha')
15  FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣Xmin␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
        ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-1))+1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,10)(I,I=1,NDIM)
    WRITE(*,10)(I,I=1,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,15)0,XO,FO
    WRITE(*,15)0,XO,FO
    DO ITER=1,MITER
        CALL GRADIENT(GRAD,XO,NDIM)
        NORM=SCAPRD(GRAD,GRAD,NDIM)
        IF(NORM.LT.CONVTOL)GOTO 99
        SDIR=-GRAD
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
             ,SDIR,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
             SDIR,NDIM)
        ! Determine minimum
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDIR
        WRITE(11,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        XO=XN
        FO=FMIN
    ENDDO
99  WRITE(11,5)
    WRITE(*,5)
    CLOSE(UNIT=11)
END SUBROUTINE
```

Fletcher-Reeves conjugate direction method:

```
! Routine for n-dimensional optimization through
! the Fletcher-Reeves conjugate direction method.
! Igor Lopes, February 2015
SUBROUTINE FLETCHERREEVES(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) GOLD /1.61803398875D0/
    REAL(8) CONVTOL /1.0D-6/
    REAL(8) RO /0.0D0/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    REAL(8)     :: FO,TOL,EVALFUNC,AMIN,FMIN,SCAPRD,ANOR,
         BNOR,BETA,SLOPE,RELDIF
    REAL(8),DIMENSION(NDIM) :: XO,XN,GRAD,SDIR,XMIN,XMAX
    REAL(8),DIMENSION(2)    :: ALPHO
    REAL(8),DIMENSION(4)    :: ALPH,FN
    INTEGER     :: I,ITER,MITER,NITER
    CHARACTER   :: STRING*256
    ! Initialize
    STRING='fletcher_reeves.res'
    CALL RESULTFILE(STRING)
    ! Formats
5   FORMAT('-----------------------------------------')
10  FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha')
15  FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣Xmin␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
        ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-1))+1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,10)(I,I=1,NDIM)
```

```
    WRITE(*,10)(I,I=1,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,15)0,XO,FO
    WRITE(*,15)0,XO,FO
    !
    CALL GRADIENT(GRAD,XO,NDIM)
    ANOR=SCAPRD(GRAD,GRAD,NDIM)
    SDIR=-GRAD
    DO ITER=1,MITER
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
             ,SDIR,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
             SDIR,NDIM)
        ! Determine minimum
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDIR
        WRITE(11,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        XO=XN
        FO=FMIN
        CALL GRADIENT(GRAD,XN,NDIM)
        BNOR=SCAPRD(GRAD,GRAD,NDIM)
        BETA=BNOR/ANOR
        SDIR=BETA*SDIR-GRAD
        !SDIR=BETA*AMIN*SDIR-GRAD ! trial modification
        ANOR=BNOR
        SLOPE=SCAPRD(SDIR,GRAD,NDIM)
        IF(SLOPE.GE.R0)SDIR=-GRAD
    ENDDO
99  WRITE(11,5)
    WRITE(*,5)
    CLOSE(UNIT=11)
END SUBROUTINE
```

Polak-Ribiere conjugate direction method:

```
! Routine for n-dimensional optimization through
! the Polak-Ribiere conjugate direction method.
! Igor Lopes, February 2015
SUBROUTINE POLAKRIBIERE(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) GOLD /1.61803398875D0/
    REAL(8) CONVTOL /1.0D-6/
    REAL(8) R0 /0.0D0/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    REAL(8)      :: FO,TOL,EVALFUNC,AMIN,FMIN,SCAPRD,ANOR,
         BNOR,CNOR,BETA,SLOPE,RELDIF
    REAL(8),DIMENSION(NDIM) :: XO,XN,GRAD,SDIR,GRADO,XMIN
         ,XMAX
    REAL(8),DIMENSION(2)    :: ALPHO
    REAL(8),DIMENSION(4)    :: ALPH,FN
    INTEGER      :: I,ITER,MITER,NITER
    CHARACTER    :: STRING*256
    ! Initialize
    STRING='polak_ribiere.res'
    CALL RESULTFILE(STRING)
    ! Formats
5   FORMAT('----------------------------------------')
10  FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha')
15  FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣␣Xmin␣␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
         ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-1))+1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,10)(I,I=1,NDIM)
    WRITE(*,10)(I,I=1,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,15)0,XO,FO
    WRITE(*,15)0,XO,FO
    !
    CALL GRADIENT(GRADO,XO,NDIM)
    ANOR=SCAPRD(GRADO,GRADO,NDIM)
    SDIR=-GRADO
    DO ITER=1,MITER
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
             ,SDIR,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
             SDIR,NDIM)
        ! Determine minimum
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDIR
        WRITE(11,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        XO=XN
        FO=FMIN
        CALL GRADIENT(GRAD,XO,NDIM)
        BNOR=SCAPRD(GRAD,GRAD,NDIM)
        CNOR=SCAPRD(GRAD,GRADO,NDIM)
        BETA=(BNOR-CNOR)/ANOR
        SDIR=BETA*SDIR-GRAD
        ANOR=BNOR
        SLOPE=SCAPRD(SDIR,GRAD,NDIM)
        GRADO=GRAD
        IF(SLOPE.GE.R0)SDIR=-GRAD
    ENDDO
99  WRITE(11,5)
    WRITE(*,5)
    CLOSE(UNIT=11)
END SUBROUTINE
```

Newton's method:

```
! Routine for n-dimensional optimization through
! the Newton method.
! Igor Lopes, February 2015
SUBROUTINE NEWTON(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    REAL(8) CONVTOL /1.0D-6/
    REAL(8) INF /1.0D-8/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    INTEGER                     :: I, ITER, MITER, NITER
    REAL(8)                     :: FO, TOL, EVALFUNC,
         AMIN, FMIN, GOLD, RELDIF
    REAL(8),DIMENSION(NDIM)     :: XO, XN, GRAD, GRADO,
         SDIR, XMAX, XMIN
    REAL(8),DIMENSION(NDIM,NDIM):: HMAT, HINV
```

```
    REAL(8),DIMENSION(2)         :: ALPHO
    REAL(8),DIMENSION(4)         :: ALPH, FN
    CHARACTER    :: STRING*256
    LOGICAL      :: ERROR
    ! Initialize
    GOLD=(R1+DSQRT(R5))/R2
    STRING='newton.res'
    CALL RESULTFILE(STRING)
    ! Formats
5   FORMAT('----------------------------------------')
10  FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha')
15  FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣Xmin␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
        ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-1))+1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,10)(I,I=1,NDIM)
    WRITE(*,10)(I,I=1,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,15)0,XO,FO
    WRITE(*,15)0,XO,FO
    !
    CALL GRADIENT(GRAD,XO,NDIM)
    CALL HESSIAN(HMAT,XO,NDIM)
    DO ITER=1,MITER
        CALL RMINVE(HMAT,HINV,NDIM,ERROR)
        CALL MATPRD(HINV,GRAD,SDIR,NDIM,NDIM,1)
        SDIR=-SDIR
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
            ,SDIR,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
            SDIR,NDIM)
        ! Determine minimum
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDIR
        WRITE(11,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        ! Update HMAT
        GRADO=GRAD
        CALL GRADIENT(GRAD,XN,NDIM)
        CALL HESSIAN(HMAT,XN,NDIM)
        XO=XN
        FO=FMIN
    ENDDO
99  WRITE(11,5)
    WRITE(*,5)
    CLOSE(UNIT=11)
END SUBROUTINE
```

Quasi-Newton methods:

```
! Routine for n-dimensional optimization through
! quasi-Newton methods: DFP or BFGS.
! Igor Lopes, February 2015
SUBROUTINE QUASINEWTON(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) CONVTOL /1.0D-6/
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) R5 /5.0D0/
    ! Arguments
    INTEGER :: NDIM
    ! Locals
    REAL(8)          :: GOLD, FO, TOL, EVALFUNC, AMIN,
        FMIN, SCAPRD, ANOR, &
                            BNOR, BETA, THETA, SIGMA, TAU
                                , RELDIF
    REAL(8),DIMENSION(NDIM)      :: XO, XN, GRAD, GRADO,
        SDIR, XMAX, XMIN
    REAL(8),DIMENSION(NDIM,1)    :: DX, DGRAD, AVEC
    REAL(8),DIMENSION(1,NDIM)    :: DXT, DGRADT, AVECT
    REAL(8),DIMENSION(NDIM,NDIM) :: HMAT, DMAT, AMAT
    REAL(8),DIMENSION(2)         :: ALPHO
    REAL(8),DIMENSION(4)         :: ALPH, FN
    INTEGER          :: I, ITER, MITER, NITER
    CHARACTER        :: STRING*256
    ! Initialize
    GOLD=(R1+DSQRT(R5))/R2
    STRING='quasi_newton.res'
    CALL RESULTFILE(STRING)
    HMAT=R0
    DO I=1,NDIM
        HMAT(I,I)=R1
    ENDDO
    ! Formats
5   FORMAT('----------------------------------------')
10  FORMAT('ITER.',6X,<NDIM>('X_'I3,10X),'F',14X,'alpha')
15  FORMAT(I3,4X,<NDIM>(G12.6,3X),G12.6,3X,G12.6)
    ! Begin algorithm
20  WRITE(*,*)'Define␣THETA␣value:'
    WRITE(*,*)'0␣-␣DFP␣(minimum)'
    WRITE(*,*)'1␣-␣BFGS␣(maximum)'
    READ(*,*)THETA
    IF(THETA.LT.R0.OR.THETA.GT.R1)THEN
        WRITE(*,*)'THETA␣out␣of␣range.'
        GOTO 20
    ENDIF
    WRITE(*,*)'Define␣limits␣for␣each␣variable:'
    WRITE(11,*)'Variables␣limits:'
    WRITE(11,*)'i␣␣␣␣␣Xmin␣␣␣␣␣Xmax'
    DO I=1,NDIM
        WRITE(*,*)'Xmin_',I,'␣Xmax_',I
        READ(*,*)XMIN(I),XMAX(I)
        WRITE(11,*)I,XMIN(I),XMAX(I)
    ENDDO
    WRITE(*,*)'Give␣an␣initial␣guess␣for␣design␣variables
        ␣X'
    READ(*,*)(XO(I),I=1,NDIM)
    WRITE(*,*)'Define␣maximum␣number␣of␣iterations'
    READ(*,*)MITER
    WRITE(*,*)'Define␣tolerance␣for␣step␣search'
    READ(*,*)TOL
    WRITE(11,*)'THETA=',THETA
    WRITE(11,*)'Maximum␣number␣of␣iterations=',MITER
    WRITE(11,*)'Tolerance␣for␣step␣search=',TOL
    NITER=INT(LOG(TOL)/LOG(GOLD-R1))+R1
    !
    FO=EVALFUNC(XO,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,10)(I,I=1,NDIM)
    WRITE(*,10)(I,I=1,NDIM)
    WRITE(11,5)
    WRITE(*,5)
    WRITE(11,15)0,XO,FO
    WRITE(*,15)0,XO,FO
    !
    CALL GRADIENT(GRAD,XO,NDIM)
    DO ITER=1,MITER
        CALL MATPRD(HMAT,GRAD,SDIR,NDIM,NDIM,1)
        SDIR=-SDIR
        CALL FINDBOUNDS(XO,FO,ALPHO(1),ALPHO(2),XMAX,XMIN
            ,SDIR,NDIM)
        CALL GOLDENSECTIONNDIM(XO,ALPHO,ALPH,FN,NITER,
            SDIR,NDIM)
        ! Determine minimum
        AMIN=ALPH(1)
        FMIN=FN(1)
        DO I=2,4
            IF(FN(I).LT.FMIN)THEN
                FMIN=FN(I)
                AMIN=ALPH(I)
            ENDIF
        ENDDO
        XN=XO+AMIN*SDIR
        WRITE(11,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        WRITE(*,15)ITER,(XN(I),I=1,NDIM),FMIN,AMIN
        ! Check convergence
        RELDIF=DABS(FMIN-FO)
```

```
        IF(DABS(FO).GT.CONVTOL)RELDIF=RELDIF/DABS(FO)
        IF(RELDIF.LT.CONVTOL)THEN
            !FINISH PROCESS
            GOTO 99
        ENDIF
        ! Update HMAT
        GRADO=GRAD
        CALL GRADIENT(GRAD,XN,NDIM)
        DO I=1,NDIM
            DX(I,1)=XN(I)-XO(I)
            DGRAD(I,1)=GRAD(I)-GRADO(I)
        ENDDO
        SIGMA=SCAPRD(DX,DGRAD,NDIM)
        CALL TRANSPOSE(DX,DXT,NDIM,1)
        CALL TRANSPOSE(DGRAD,DGRADT,NDIM,1)
        CALL MATPRD(HMAT,DGRAD,AVEC,NDIM,NDIM,1)
        TAU=SCAPRD(DGRAD,AVEC,NDIM)
        CALL MATPRD(DX,DXT,AMAT,NDIM,1,NDIM)
            DMAT=AMAT*(SIGMA+TAU*THETA)/(SIGMA*SIGMA)
            CALL TRANSPOSE(AVEC,AVECT,NDIM,1)
            CALL MATPRD(AVEC,AVECT,AMAT,NDIM,1,NDIM)
            DMAT=DMAT+AMAT*(THETA-R1)/TAU
            CALL MATPRD(AVEC,DXT,AMAT,NDIM,1,NDIM)
            DMAT=DMAT-AMAT*THETA/SIGMA
            CALL MATPRD(DX,AVECT,AMAT,NDIM,1,NDIM)
            DMAT=DMAT-AMAT*THETA/SIGMA
            HMAT=HMAT+DMAT
            XO=XN
            FO=FMIN
        ENDDO
99      WRITE(11,5)
        WRITE(*,5)
        CLOSE(UNIT=11)
END SUBROUTINE
```

A.6 Genetic algorithm routines

Main routine:

```
! Main routine for genetic algorithms for optimization.
! Igor Lopes, February 2015
SUBROUTINE GENETIC(NDIM)
    IMPLICIT NONE
    ! Parameters
    REAL(8) RO /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    ! Argument
    INTEGER NDIM
    ! Locals
    INTEGER     :: IBIT, ICOUNT, IDIM, IPOP, IGEN, NPOP,
         NTBIT, MGEN,&
                   NGENT, NSONS, NSURV, NTOPOUT, NTOPT
    INTEGER,DIMENSION(NDIM) :: NBITS
    REAL(8),DIMENSION(NDIM) :: XMIN, XMAX, PRECIS
    REAL(8),ALLOCATABLE,DIMENSION(:)    :: VFITO, VFIT,
         OBJFUN
    INTEGER,ALLOCATABLE,DIMENSION(:,:)  :: POPO, POP
    REAL(8),ALLOCATABLE,DIMENSION(:,:)  :: XVEC
    CHARACTER   :: STRING*256
    REAL(8)     :: RANDN, SRATE, PMUT
    LOGICAL     :: THEEND
    !
100 FORMAT('Define␣limits␣for␣each␣variable:')
105 FORMAT('Xmin_',I3,',␣Xmax_',I3)
107 FORMAT('␣i␣',2X,'␣Xmin_i␣',2X,'␣Xmax_i␣',2X,'nr.bits'
     ,2X,'Precision')
108 FORMAT(I3,2X,F8.3,2X,F8.3,3X,I5,3X,F8.6)
110 FORMAT('Define␣number␣of␣individuals␣in␣the␣
     population:')
115 FORMAT('Define␣precision␣for␣each␣variable:')
117 FORMAT('nr.␣bits␣per␣chromossome:',I5,/'nr.␣
     chromossomes/individuals␣in␣the␣population:',I5,&
                  /'total␣nr.␣of␣bits␣in␣population:',
                       I5)
120 FORMAT('X_',I3,':')
125 FORMAT('Define␣maximum␣number␣of␣generations:')
127 FORMAT('Maximum␣number␣of␣generations:',I5)
130 FORMAT('Define␣survival␣rate␣(0-1):')
135 FORMAT('No␣offspring␣with␣this␣survival␣rate...')
137 FORMAT('Survival␣rate:',F8.5,/'nr.␣of␣surviving␣
     individuals:',I5,&
                  /'nr.␣of␣sons␣in␣new␣population:',I5)
140 FORMAT('Define␣mutation␣probability␣(0-1):')
143 FORMAT('Mutation␣probability:',F8.5,/'Expected␣nr.␣of
     ␣mutated␣bits:',I5)
145 FORMAT('Value␣is␣outside␣admissible␣range...')
150 FORMAT('Define␣nr.␣of␣TOP␣individuals␣for␣ouput:')
153 FORMAT('nr␣TOP␣cannot␣be␣greater␣than␣nr␣individuals
     ...')
155 FORMAT('Define␣nr␣of␣TOP␣individuals␣and␣nr␣of␣
     generations'/&
            'without␣changes␣in␣TOP␣for␣termination␣
                 criteria:')
157 FORMAT('Termination␣criteria:␣i)␣Maximum␣number␣of␣
     generations,'/&
           'ii)␣Reduction␣of␣diversity␣in␣population␣-␣
                every␣locus␣is␣has␣more␣than␣90%␣of␣the
                ␣same␣allele,'/&
           'iii)␣TOP␣',I3,'␣does␣not␣change␣for␣',I3,'␣
                generations.')
160 FORMAT('#GEN',2X,'TOP',6X,<NDIM>('X_'I3,10X),'obj.
     Func',3X,'Fitness')
163 FORMAT('----------------------------------------')
    ! Initialize
    CALL RANDOM_SEED
    STRING='geneticalgorithm.res'
    CALL RESULTFILE(STRING)
    ICOUNT=0
    POPO=0
    VFITO=RO
    THEEND=.FALSE.
    ! Algorithm
    WRITE(*,100)
    DO IDIM=1,NDIM
        WRITE(*,105)IDIM,IDIM
        READ(*,*)XMIN(IDIM),XMAX(IDIM)
    ENDDO
    WRITE(*,115)
    NTBIT=0
    WRITE(11,107)
    DO IDIM=1,NDIM
        WRITE(*,120)IDIM
        READ(*,*)PRECIS(IDIM)
        NBITS(IDIM)=CEILING(LOG(R1+(XMAX(IDIM)-XMIN(IDIM)
             )/PRECIS(IDIM))/LOG(R2))
        NTBIT=NTBIT+NBITS(IDIM)
        WRITE(11,108)IDIM,XMIN(IDIM),XMAX(IDIM),NBITS(
             IDIM),PRECIS(IDIM)
    ENDDO
    WRITE(*,110)
    READ(*,*)NPOP
    WRITE(11,117)NTBIT,NPOP,NTBIT*NPOP
    WRITE(*,125)
    READ(*,*)MGEN
    WRITE(11,127)MGEN
10  WRITE(*,130)
    READ(*,*)SRATE
    IF(SRATE.LT.RO.OR.SRATE.GT.R1)THEN
        WRITE(*,145)
        GOTO 10
    ENDIF
    NSURV=INT(SRATE*NPOP)
    NSONS=NPOP-NSURV
    IF(NSONS.EQ.0)THEN
        WRITE(*,135)
        GOTO 10
    ENDIF
    WRITE(11,137)SRATE,NSURV,NSONS
15  WRITE(*,140)
    READ(*,*)PMUT
    IF(PMUT.LT.RO.OR.PMUT.GT.R1)THEN
        WRITE(*,145)
        GOTO 15
    ENDIF
    WRITE(11,143)PMUT,INT(PMUT*NTBIT*NSONS)
20  WRITE(*,150)
    READ(*,*)NTOPOUT
    IF(NTOPOUT.GT.NPOP)THEN
        WRITE(*,153)
        GOTO 20
    ENDIF
25  WRITE(*,155)
    READ(*,*)NTOPT,NGENT
    IF(NTOPOUT.GT.NPOP)THEN
        WRITE(*,153)
        GOTO 25
    ENDIF
    WRITE(*,157)NTOPT,NGENT
    WRITE(11,157)NTOPT,NGENT
    WRITE(11,163);WRITE(*,163)
    WRITE(11,160)(IDIM,IDIM=1,NDIM);WRITE(*,160)(IDIM,
         IDIM=1,NDIM)
    WRITE(11,163);WRITE(*,163)
    !
    ALLOCATE(POPO(NPOP,NTBIT))
    ALLOCATE(POP(NPOP,NTBIT))
    ALLOCATE(VFIT(NPOP))
    ALLOCATE(VFITO(NPOP))
```

```
    ALLOCATE(OBJFUN(NPOP))
    ALLOCATE(XVEC(NDIM,NPOP))
    DO IBIT=1,NTBIT
        DO IPOP=1,NPOP
            CALL RANDOM_NUMBER(RANDN)
            POP(IPOP,IBIT)=NINT(RANDN)
        ENDDO
    ENDDO
    !
    CALL FITNESS(XMIN,XMAX,XVEC,POP,VFIT,OBJFUN,NBITS,
         NDIM,NPOP,NTBIT)
    CALL STATISTICS(0,XVEC,VFIT,OBJFUN,NDIM,NPOP,NTOPOUT)
    WRITE(11,163);WRITE(*,163)
    DO IGEN=1,MGEN
        POPO=POP
        VFITO=VFIT
        ! Survival of the fittest is implicit
        ! Reproduction
        CALL REPRODUCTION(POPO,POP,VFITO,VFIT,NSURV,NPOP,
             NTBIT)
        ! Mutation
        IF(PMUT.NE.R0)CALL MUTATION(POP,NPOP,PMUT,NSURV,
             NTBIT)
        WRITE(11,163);WRITE(*,163)
        !
        CALL FITNESS(XMIN,XMAX,XVEC,POP,VFIT,OBJFUN,NBITS
             ,NDIM,NPOP,NTBIT)
        !
        CALL STATISTICS(IGEN,XVEC,VFIT,OBJFUN,NDIM,NPOP,
             NTOPOUT)
        WRITE(11,163);WRITE(*,163)
        CALL TERMINATION(POPO,POP,NPOP,NTBIT,NTOPT,NGENT,
             ICOUNT,THEEND)
        IF(THEEND)GOTO 99
    ENDDO

99  CLOSE(UNIT=11)

END SUBROUTINE
```

Routine for determining fitness values and sort population:

```
! Routine that computes fitness of each individual,
! and sorts them by decreasing fitness.
! Igor Lopes, February 2015
SUBROUTINE FITNESS(XMIN,XMAX,XVEC,POP,VFIT,OBJF,NBITS,
     NDIM,NPOP,NTBIT)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    ! Argument
    INTEGER :: NDIM, NPOP, NTBIT
    REAL(8) :: FMAX, FMIN
    INTEGER,DIMENSION(NDIM) :: NBITS
    REAL(8),DIMENSION(NPOP) :: VFIT, OBJF
    REAL(8),DIMENSION(NDIM) :: XMIN, XMAX
    REAL(8),DIMENSION(NDIM,NPOP)    :: XVEC
    INTEGER,DIMENSION(NPOP,NTBIT)   :: POP
    ! Locals
    INTEGER     :: IBIT, IDIM, IPOP, IPOS, ISUB, NPOS
    REAL(8)     :: AUX, BIN2REAL, EVALFUNC
    REAL(8),DIMENSION(NPOP) :: VFITO, OBJFO
    INTEGER,DIMENSION(NPOP,NTBIT)   :: POPO
    REAL(8),DIMENSION(NDIM,NPOP)    :: XVECO
    ! Algorithm
    ! Decode from binary to real
    DO IPOP=1,NPOP
        IPOS=0
        DO IDIM=1,NDIM
            AUX=BIN2REAL(POP(IPOP,IPOS+1:IPOS+NBITS(IDIM)
                ),NBITS(IDIM))
            XVEC(IDIM,IPOP)=XMIN(IDIM)+AUX*(XMAX(IDIM)-
                XMIN(IDIM))/(R2**NBITS(IDIM)-R1)
            IPOS=IPOS+NBITS(IDIM)
        ENDDO
        OBJF(IPOP)=EVALFUNC(XVEC(:,IPOP),NDIM)
    ENDDO
    ! Minimization of function -> higher fitness for
        lower function value
    FMAX=MAXVAL(OBJF)
    FMIN=MINVAL(OBJF)
    DO IPOP=1,NPOP
        VFIT(IPOP)=FMAX-OBJF(IPOP)+0.01*(FMAX-FMIN)
        !artifitial increase of fitness in 1% of the best
            fitness
    ENDDO
    ! Sort by decreasing fitness
    POPO=POP
    VFITO=VFIT
    XVECO=XVEC
    OBJFO=OBJF
    POP=0
    VFIT=MINVAL(VFIT)
    POP(1,:)=POPO(1,:)
    VFIT(1)=VFITO(1)
    XVEC(:,1)=XVECO(:,1)
    OBJF(1)=OBJFO(1)
    DO IPOP=2,NPOP
        DO ISUB=1,IPOP-1
            IF(VFITO(IPOP).GE.VFIT(ISUB))THEN
                DO IPOS=IPOP,ISUB+1,-1
                    VFIT(IPOS)=VFIT(IPOS-1)
                    POP(IPOS,:)=POP(IPOS-1,:)
                    OBJF(IPOS)=OBJF(IPOS-1)
                    XVEC(:,IPOS)=XVEC(:,IPOS-1)
                ENDDO
                VFIT(ISUB)=VFITO(IPOP)
                POP(ISUB,:)=POPO(IPOP,:)
                OBJF(ISUB)=OBJFO(IPOP)
                XVEC(:,ISUB)=XVECO(:,IPOP)
                GOTO 10
            ENDIF
        ENDDO
        VFIT(IPOP)=VFITO(IPOP)
        POP(IPOP,:)=POPO(IPOP,:)
        OBJF(IPOP)=OBJFO(IPOP)
        XVEC(:,IPOP)=XVECO(:,IPOP)
10      CONTINUE
    ENDDO
END SUBROUTINE
```

Output routine:

```
! Output routine for genetic algorithms
! Igor Lopes, February 2015
SUBROUTINE STATISTICS(IGEN,XVEC,VFIT,OBJFUN,NDIM,NPOP,
     NTOP)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) P10 /0.10D0/
    REAL(8) P90 /0.90D0/
    ! Argument
    INTEGER :: IGEN, NDIM, NPOP, NTOP
    REAL(8),DIMENSION(NPOP) :: VFIT, OBJFUN
    REAL(8),DIMENSION(NDIM,NPOP)    :: XVEC
    ! Locals
    INTEGER     :: IBIT, IDIM, IPOP, IPOS, ISUB, NPOS
    REAL(8)     :: AUX
    ! Initialize
100 FORMAT(I4,2X,I3,2X,<NDIM>(F12.6,3X),F12.6,3X,G12.6)
105 FORMAT('Objective␣Function␣max.␣value:',G12.6)
110 FORMAT('Objective␣Function␣min.␣value:',G12.6)
115 FORMAT('Objective␣Function␣average:␣␣␣',G12.6)
    ! Algorithm
    DO IPOP=1,NTOP
        WRITE(11,100)IGEN,IPOP,(XVEC(IDIM,IPOP),IDIM=1,
            NDIM),OBJFUN(IPOP),VFIT(IPOP)
        WRITE(*,100)IGEN,IPOP,(XVEC(IDIM,IPOP),IDIM=1,
            NDIM),OBJFUN(IPOP),VFIT(IPOP)
    ENDDO
    WRITE(11,105)MAXVAL(OBJFUN);WRITE(*,105)MAXVAL(OBJFUN
        )
    WRITE(11,110)MINVAL(OBJFUN);WRITE(*,110)MINVAL(OBJFUN
        )
    AUX=R0
    DO IPOP=1,NPOP
        AUX=AUX+OBJFUN(IPOP)
    ENDDO
    AUX=AUX/NPOP
    WRITE(11,115)AUX;WRITE(*,115)AUX
99  CONTINUE
END SUBROUTINE
```

Routine for generating offspring:

```
! Routine for generating sons through crossover technique
      .
! Igor Lopes, February 2015
SUBROUTINE REPRODUCTION(POPO,POP,VFITO,VFIT,NSURV,NPOP,
     NTBIT)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R2 /2.0D0/
    ! Argument
    INTEGER :: NSURV, NPOP, NTBIT
    REAL(8),DIMENSION(NPOP) :: VFIT, VFITO
    INTEGER,DIMENSION(NPOP,NTBIT)   :: POPO, POP
    ! Locals
    INTEGER     :: IPOP, JPOP, IPOS, NPOS
    REAL(8),DIMENSION(NPOP) :: VAFIT
    REAL(8)     :: SUM, RANDN
    INTEGER,DIMENSION(2,NTBIT) :: PAR
    ! Initialize
    VAFIT=R0
    SUM=R0
    IPOS=NSURV
    ! Algorithm
    ! Parents
    VAFIT(1)=VFITO(1)
    SUM=VFITO(1)
    DO IPOP=2,NPOP
        VAFIT(IPOP)=VAFIT(IPOP-1)+VFITO(IPOP)
        SUM=SUM+VFITO(IPOP)
    ENDDO
    VAFIT=VAFIT/SUM
    DO WHILE(IPOS.LT.NPOP)
        ! 1st parent
        CALL RANDOM_NUMBER(RANDN)
        DO IPOP=1,NPOP
            IF(VAFIT(IPOP).GE.RANDN)THEN
                PAR(1,:)=POPO(IPOP,:)
                GOTO 10
            ENDIF
        ENDDO
        ! 2nd parent
10      CALL RANDOM_NUMBER(RANDN)
        DO JPOP=1,NPOP
            IF(VAFIT(JPOP).GE.RANDN)THEN
                IF(JPOP.EQ.IPOP)GOTO 10
                PAR(2,:)=POPO(JPOP,:)
                GOTO 15
            ENDIF
        ENDDO
        ! Position for crossover: NTBIT possible
             positions
15      CALL RANDOM_NUMBER(RANDN)
        RANDN=RANDN*(NTBIT-1)
        NPOS=CEILING(RANDN)
        ! Sons stored in new population vector
        ! 1st son
        IPOS=IPOS+1
        IF(IPOS.GT.NPOP)GOTO 99
        POP(IPOS,1:NPOS)=PAR(1,1:NPOS)
        POP(IPOS,NPOS+1:NTBIT)=PAR(2,NPOS+1:NTBIT)
        ! 2nd
        IPOS=IPOS+1
        IF(IPOS.GT.NPOP)GOTO 99
        POP(IPOS,1:NPOS)=PAR(2,1:NPOS)
        POP(IPOS,NPOS+1:NTBIT)=PAR(1,NPOS+1:NTBIT)
    ENDDO
99  CONTINUE
END SUBROUTINE
```

Routine for mutation:

```
! In this routine, some genes of offspring are mutated
! Igor Lopes, February 2015
SUBROUTINE MUTATION(POP,NPOP,PMUT,NSURV,NTBIT)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R0 /0.0D0/
    REAL(8) R2 /2.0D0/
    ! Argument
    INTEGER :: NPOP, NSURV, NTBIT
    INTEGER,DIMENSION(NPOP,NTBIT)   :: POP
    REAL(8) :: PMUT
    ! Locals
    INTEGER     :: IPOP, IBIT, ICOUNT
    REAL(8)     :: RANDN
    ! Initialize
    ICOUNT=0
    ! Algorithm
    DO IBIT=1,NTBIT
        DO IPOP=NSURV+1,NPOP
            CALL RANDOM_NUMBER(RANDN)
            IF(RANDN.LE.PMUT)THEN
                IF(POP(IPOP,IBIT).EQ.0)THEN
                    POP(IPOP,IBIT)=1
                ELSE
                    POP(IPOP,IBIT)=0
                ENDIF
                ICOUNT=ICOUNT+1
            ENDIF
        ENDDO
    ENDDO
    WRITE(11,*)'Nr.␣mutated␣bits=',ICOUNT
    WRITE(*,*)'Nr.␣mutated␣bits=',ICOUNT
END SUBROUTINE
```

Termination criteria verification:

```
! Routine that checks termination criteria
! Igor Lopes, February 2015
SUBROUTINE TERMINATION(POPO,POP,NPOP,NTBIT,NTOPT,NGENT,
     ICOUNT,THEEND)
    IMPLICIT NONE
    ! Parameters
    REAL(8) R1 /1.0D0/
    REAL(8) R2 /2.0D0/
    REAL(8) P10 /0.10D0/
    REAL(8) P90 /0.90D0/
    ! Argument
    INTEGER :: NPOP, NTBIT, NTOPT, NGENT, ICOUNT
    INTEGER,DIMENSION(NPOP,NTBIT)   :: POP,POPO
    LOGICAL :: THEEND
    ! Locals
    INTEGER     :: IBIT, IPOP
    REAL(8)     :: AUX, ISUM
    ! Initialize
100 FORMAT('TOP␣',I3,'␣has␣not␣changed␣in␣the␣last␣',I3,'
     ␣generations!')
105 FORMAT('Reduced␣diversity␣in␣population!')
    ! Algorithm
    ! Termination criteria
    ! N generations without changing top
    DO IBIT=1,NTBIT
        DO IPOP=1,NTOPT
            IF(POP(IPOP,IBIT).NE.POPO(IPOP,IBIT))THEN
                ICOUNT=0
                GOTO 10
            ENDIF
        ENDDO
    ENDDO
    ICOUNT=ICOUNT+1
    IF(ICOUNT.EQ.NGENT)THEN
        WRITE(11,100)NTOPT,NGENT
        WRITE(*,100)NTOPT,NGENT
        THEEND=.TRUE.
        GOTO 99
    ENDIF
10  CONTINUE
    ! Check diversity
    DO IBIT=1,NTBIT
        ISUM=0
        DO IPOP=1,NPOP
            ISUM=ISUM+POP(IPOP,IBIT)
        ENDDO
        AUX=ISUM/NPOP
        IF(AUX.GT.P10.AND.AUX.LT.P90)GOTO 99
    ENDDO
    WRITE(11,105)
    WRITE(*,105)
    THEEND=.TRUE.
    !
99  CONTINUE
END SUBROUTINE
```

B Finite Differences for Gradient and Hessian Matrix

For a function of n variables $f(\boldsymbol{x})$, the gradient is given by:

$$\nabla f(\boldsymbol{x^*})_i = \left.\frac{\partial f}{\partial x_i}\right|_{\boldsymbol{x}^*}. \tag{B.1}$$

Applying the finite differences method, it can be approximated by:

$$\nabla f(\boldsymbol{x^*}) \approx \frac{f(\boldsymbol{x}^*) - f(\boldsymbol{x}^* + \Delta x \boldsymbol{e}_i)}{\Delta x}\boldsymbol{e}_i, \tag{B.2}$$

where Δx is an imposed perturbation and $\boldsymbol{e}_i$ is the unit vector with direction i-th direction.

In what refers to the Hessian matrix, it is defined as:

$$H(\boldsymbol{x}^*)_{ij} = \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_{\boldsymbol{x}^*}. \tag{B.3}$$

In this situation two cases must be considered for the finite differences approximation. The diagonal elements of the matrix ($i = j$) are approximated by:

$$H(\boldsymbol{x}^*)_{ii} \approx \frac{f(\boldsymbol{x}^* + \Delta x \boldsymbol{e}_i) - 2f(\boldsymbol{x}^*) + f(\boldsymbol{x}^* - \Delta x \boldsymbol{e}_i)}{\Delta x^2}, \tag{B.4}$$

whereas the remaining ($i \neq j$) are obtained as:

$$H(\boldsymbol{x}^*)_{ij} \approx \frac{f_1 - f_2 - f_3 + f_4}{4\Delta x^2}, \tag{B.5}$$

with

$$f_1 = f(\boldsymbol{x}^* + \Delta x \boldsymbol{e}_i + \Delta x \boldsymbol{e}_j) \tag{B.6}$$
$$f_2 = f(\boldsymbol{x}^* + \Delta x \boldsymbol{e}_i - \Delta x \boldsymbol{e}_j) \tag{B.7}$$
$$f_3 = f(\boldsymbol{x}^* - \Delta x \boldsymbol{e}_i + \Delta x \boldsymbol{e}_j) \tag{B.8}$$
$$f_4 = f(\boldsymbol{x}^* - \Delta x \boldsymbol{e}_i - \Delta x \boldsymbol{e}_j). \tag{B.9}$$

The implementation of these strategies is straightforward. Δx must be a small value, but chosen carefully in order to avoid numerical problems. In the present work the perturbation is $\Delta x = 1 \times 10^{-4}$.

References

Chakraborty, R. (2010). Fundamentals of genetic algorithms: Ai course, lecture 39-40. www.myreaders.info/09-Genetic_Algorithms.pdf.

Conceição António, C. A. (2013). Local and global Pareto dominance applied to optimal design and material selection of composite structures. *Structural and Multidisciplinary Optimization 48*, 73–94.

Dawkins, R. (1989). *The Selfish Gene*. Oxford paperbacks. Oxford University Press.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph. D. thesis, Ann Arbor, MI, USA.

de Souza Neto, E. and R. Feijóo (2006). Variational foundations of multi-scale constitutive models of solid:: Small and large strain kinematical formulation. LNCC R&D Report 16/2006, LNCC.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Holland, J. (1975). *Adaptation in natural and artificial systems.* Ann Arbor: The University of Michigan Press.

Hopfield, J. (1982). Neural networks and physical systems with emergent collective computational capabilities. *Proceedings of the National Academy of Science 79*, 2254 – 2558.

Iacono, C. (2007). *Procedures for parameter estimates of computational models for localized failure.* Ph. D. thesis, The Netherlands.

Kirsch, U. (1993). *Structural optimization: fundamentals and applications.* Springer-Verlag.

Lagoudakis, M. G. (1997). Neural networks and optimization problems; a case study: The minimum cost spare allocation problem. Technical report, The Center for Advanced Computer Studies, University of Southwestern, Louisiana.

Mahnken, R. (2004). Chapter 19: Identification of material parameters for constitutive equations. In *Encyclopedia of Computational Mechanics.* John Wiley & Sons, Ltd.

Ogden, R. W. (1972). Large deformation isotropic elasticity - on the correlation of theory and experiment for incompressible rubber-like solids. *Proc. R. Soc. Lond. A 326*, 565–584.

Ogden, R. W., G. Saccomandi, and I. Sgura (2004). Fitting hyperelastic models to experimental data. *Computational Mechanics 34*, 484–502.

Parpinelli, R. S. and H. S. Lopes (2015). A computational ecosystem for optimization: review and perspectives for future research. *Memetic Computing 7*, 29–41.

Reeves, C. (2003). Chapter 3: Genetic algorithms. In F. Glover and G. A. Kochenberger (Eds.), *Handbook of Metaheuristics*, Volume 57 of *International Series in Operations Research & Management Science*, pp. 55–82. Springer US.

Reis, F. (2014). *Multi-Scale Modelling and Analysis of Heterogeneous Solids at Finite Strains.* Ph. D. thesis, Faculdade de Engenharia da Universidade do Porto.

Sakawa, M. (2002). *Genetic Algorithms and Fuzzy Multiobjective Optimization.* MA, USA: Kluwer Academic Publishers.

Serapião, A. B. S. (2009). Fundamentos de otimização por inteligência de enxames: uma visão geral. *Sba Controle & Automação 20*(3), 271 – 304.

Speirs, D. (2007). *Characterisation of Materials with Hyperelastic Microstructures Through Computational Homogenisation and Optimisation Methods*. Ph. D. thesis, Swansea University.

Stahlschmidt, J. (2010). Estudos de identificação de parâmetros elasto-plásticos utilizando métodos de otimização. Master's thesis, UDESC, Joinville, Brasil.

Vanderplaats, G. N. (1984). *Numerical optimization techniques for engineering design: with applications*. McGraw-Hill.

Venkayya, V. B. (1989). Optimality criteria: A basis for multidisciplinary design optimization. *Computational Mechanics 5*, 1–21.

Whitley, D. and A. Sutton (2012). 21 - genetic algorithms ? a survey of models and methods. In G. Rozenberg, T. Bäck, and J. Kok (Eds.), *Handbook of Natural Computing*, pp. 637–671. Springer Berlin Heidelberg.