A Java Expression Evaluator for Nonlinear Programming

J. Matias *Member, IAENG*, A. Correia *Member, IAENG*,
C. Serodio *Member, IAENG*, C. Teixeira, P. Mestre *Member, IAENG*

***Abstract*—Finding the optimal value for a problem is usual in many areas of knowledge where in many cases it is needed to solve Nonlinear Optimization Problems. For some of those problems it is not possible to determine the expression for its objective function and/or its constraints, they are the result of experimental procedures, might be non-smooth, among other reasons. To solve such problems it was implemented an API contained methods to solve both constrained and unconstrained problems. This API was developed to be used either locally on the computer where the application is being executed or remotely on a server. To obtain the maximum flexibility both from the programmers' and users' points of view, problems can be defined as a Java class (because this API was developed in Java) or as a simple text input that is sent to the API. For this last one to be possible it was also implemented on the API an expression evaluator. One of the drawbacks of this expression evaluator is that it is slower than the Java native code. In this paper it is presented a solution that combines both options: the problem can be expressed at run-time as a string of chars that are converted to Java code, compiled and loaded dynamically. To wide the target audience of the API, this new expression evaluator is also compatible with the AMPL format.**

***Index Terms*—Nonlinear Programming, Java, API, AMPL, Dynamic code generation.**

I. Introduction

It is usual in many areas of knowledge to have to find optimal values (solutions) for Nonlinear Optimization Problems. Some examples of such problems include the automatic tuning of Location Estimation Algorithms internal parameters, to fit the characteristics of the mobile terminal being used in the location system [1], or, tuning of Fuzzy Logic inference engine internal parameters [2] during the training phase.

Since for most of these problems it is not possible to determine the expression for its objective function and/or its constraints, derivative based methods cannot be used. In these case Direct Search Methods are the most suitable, because they do not need the use of derivatives or approximations to them.

With the dual objective of being used in areas as the above mentioned to optimize engineering processes, and to serve as a tool in research of optimization methods, specially derivative free methods, it was developed by the authors a Java-based API (Application Programming Interface).

This API can be used to solve Nonlinear Programming Problems using the methods referred in section II, as presented in [3] and [4]. Since this API was developed in Java, originally it only allowed the development of applications using this programming technology. To cope with this, and to allow remote access to it, it was extended to support Web Services [5] and [6].

A web version, accessible using an Internet browser, was also developed and presented in [7]. This web application allowed users to execute the above mentioned methods, using a set of predefined problems (stored in a database of problems). To increase the flexibility of that application, it was also allowed to users to define their own problems. For this to be possible it was needed to build an expression evaluator that would interpret the expression introduced by the user.

Because expressions are interpreted, and not compiled in Java code, this interpreter is a bottleneck of the API. Another drawback is the fact that there are widely accepted formats to define problems that could be used to input data to the API. For example it is common to find libraries of test problems defined in AMPL (A Modeling Language for Mathematical Programming).

To overcome these limitations, it was then developed an AMPL parser which has as output Java classes that can be called from the application.

One of the key features of the implemented solution is that those Java classes are not generated by the user or by the programmer at application compilation time. These classes are dynamically generated, loaded and executed by the API at runtime. The developed solution also allows the input of simple expressions (as it was already available in the API) that are dynamically converted into Java classes.

II. Optimization Methods

In some optimization problems it is not possible to determine its objective function or it might be too complex to be determined. On other cases the objective function (or the problem constraints) might be non-smooth or their derivatives are not known. On the examples presented in the

J. Matias is with CM-UTAD - Centre for the Mathematics, University of Trás-os-Montes and Alto Douro, Vila Real, Portugal, email:j_matias@utad.pt

A. Correia is with ESTGF-IPP, School of Technology and Management of Felgueiras Polytechnic Institute of Porto, Portugal, aldinacorreia@eu.ipp.pt

C. Serodio is with CITAB/UTAD - Centre for the Research and Technology of Agro-Environment and Biological Sciences, Vila Real, Portugal, and Algoritmi Research Centre, Guimaraes, Portugal, email: cserodio@utad.pt

C. Teixeira is with UTAD-University of Trás-os-Montes e Alto Douro, Vila Real, Portugal, Portugal, email:christopheteixeira4@gmail.com

P. Mestre is with CITAB/UTAD - Centre for the Research and Technology of Agro-Environment and Biological Sciences, Vila Real, Portugal, and Algoritmi Research Centre, Guimaraes, Portugal, email: pmestre@utad.pt

ISBN: 978-988-19251-0-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

previous section, they are a set of discrete results from an experimental procedure. In all these cases it cannont be used derivative-based optimization methods, as presented in [8]. In such problems one possible solution to cope with this is to use direct search methods that do not use derivatives or approximations to them. For further details see [9], [10] and [11].

Optimization problems that may appear can be of two natures: unconstrained optimization problems or constrained optimization problems. Unconstrained optimization problems have the form of (1):

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{1}$$

where:

- $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function;

Constrained optimization problems have the form of (2):

$$\begin{array}{ll} \min\limits_{x \in \mathbb{R}^n} & f(x) \\ s.t. & c_i(x) = 0, i \in \mathcal{E} \\ & c_i(x) \leq 0, i \in \mathcal{I} \end{array} \tag{2}$$

where:

- $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function as in (1);
- $c_i(x) = 0,\ i \in \mathcal{E}$, with $\mathcal{E} = \{1, 2, ..., t\}$, define the problem equality constraints;
- $c_i(x) \leq 0,\ i \in \mathcal{I}$, with $\mathcal{I} = \{t+1, t+2, ..., m\}$, represent the inequality constraints;
- $\Omega = \{x \in \mathbb{R}^n : c_i = 0,\ i \in \mathcal{E} \wedge c_i(x) \leq 0,\ i \in \mathcal{I}\}$ is the set of all feasible points, i.e., the feasible region.

Both types of problems, constrained and unconstrained, can be solve by the developed API (Fig. 1). To solve unconstrained problems the following Direct Search Methods have been added to it:

- A Coordinated Search algorithm, which can be analysed in detail in [8];
- Hooke and Jeves algorithm [11];
- An implementation of the algorithm of Audet et. al. as in [12], [13] and [14];
- The Nelder and Mead algorithm as in [15], or in [16] and [17];
- A Convergent Simplex algorithm can be analised in detail in analysed [8] and [18];

To solve constrained methods, were included in the API Penalty and Barrier methods. These methods transform problems of the form (2) into a sequence of problems in the form of (1), i.e., a sequence of unconstrained problems, which are then solved using the algorithms used to solve unconstrained problems. The new sequence of unconstrained problems that replaces the original problem, is defined by:

$$\Phi(x_k, r_k) : \min_{x_k \in \mathbb{R}^n} f(x_k) + r_k p(x) \tag{3}$$

where Φ is the new objective function, k is the iteration number , p is a function that penalises (penalty) or refuses (barrier) points that violates the constraints and r_k is a positive parameter.

In the API the following Penalty and Barrier methods were implemented:

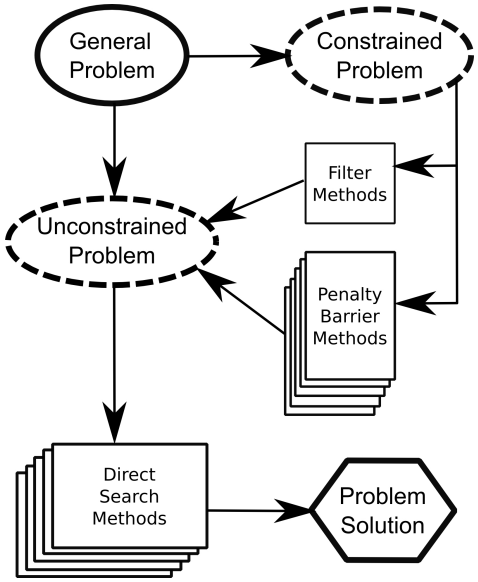

Figure 1. API Block Diagram

- A Nonstationary Penalty that can be analysed in [19];
- Adaptative Barrier as in [13], [14];
- Extreme Barrier as in [20] and [21];
- Classical Penalty as in [22], or in [23];
- ℓ_1 Penalty which can be analysed in [24], [25] and [26];

Also the Filters Method [27] was included in the API. Unlike the Penalty and Barrier methods, in this method the optimality and the feasibility are treated separately, considering that optimization problems are bi-objective programs. The goal is the minimization of both the objective function (optimality) and a continuous function (h) that aggregates the constraint function values (feasibility).

Since it is not reasonable to have as a solution an infeasible point, priority must then be given to h. It must then be such that:

$$h(x) \geq 0 \text{ with } h(x) = 0 \text{ if and only if } x \text{ is feasible.}$$

We can then define h as:

$$h(x) = \|C_+(x)\|, \tag{4}$$

where $\|.\|$ is the norm of a vector and $C_+(x)$ is the vector of the $t + m$ values of the constraints in x, i.e, $c_i(x)$ for $i = 1, 2, ..., t + m$:

$$C_+(x) = \begin{cases} c_i(x) & \text{if} \quad c_i(x) > 0 \\ 0 & \text{if} \quad c_i(x) \leq 0 \end{cases}$$

III. Proposed Method

As above mentioned the API allows users to solve problems written in Java (hard-coded in the application) or defined at run time as a text string that is interpreted by the API.

In the first case we have a higher performance, because the problem is written in Java and therefore natively executed by the API. The drawback of this option is that the problem must exist at compilation time, or, the application must be prepared to dynamically load external class files and then execute them. This means that the final user of the software must

ISBN: 978-988-19251-0-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

know the insides of the API and know how to program in Java. While some API users are programmers, there are some users that only want to solve a problem using an application.

On the second case the problem is interpreted online, this gives a higher flexibility to the API and application developed using it. Problems and constraints are written as a string, for example `cos(x0)+ln(x2)`. The major drawback of this option is the execution speed.

In this paper it is presented and tested a solution that is a combination of both options. It is given to the user the ability to define at run time the problems, without the need to know how to code it in Java, with the performance achieved with the problems coded using Java. To make the API compatible with standard representations used to describe constrained and unconstrained problems, it was also added to the API the ability to read problems from a AMPL file.

This file can be located in at a local drive of the computer where the application is being executed or in a URL, using for example HTTP (Hypertext Transfer Protocol).

In Fig. 2 is depicted the diagram block of the presented solution. As above mentioned the input can be either a string expression of an AMPL file, that are converted to an intermediate format (that only exists in memory in the form of Java objects). Any other file formats can later on be added to the API and the result of parsing those files will always be this intermediate format.

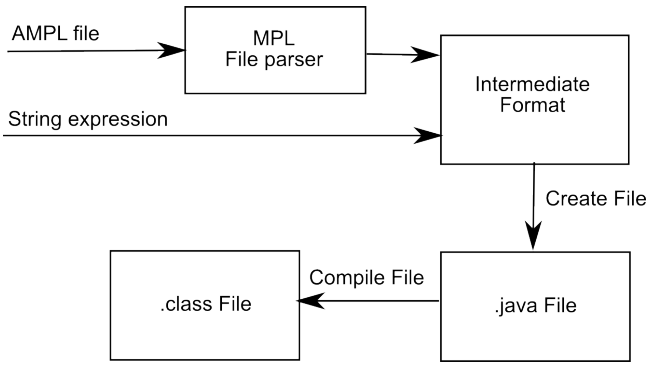

Figure 2. Diagram Block of the proposed solution

In Fig. 3 it is presented an example of an unconstrained problem defined in AMPL, in this case problem S201 from the Schittkowski collection [28]. The corresponding intermediate format for this problem, where all the mathematical expression are converted into a "Java friendly" format is: `4.0*Math.pow((x[(int)(1-1)]-5.0),2.0) + Math.pow((x[(int)(2-1)]-6.0), 2.0)` Some of the operations that must be made to the problem expression, and the constraints when they exist, are:

- Convert trigonometric expressions such `sin`, `cos` to the correspondent Java equivalent: `Math.sin` and `Math.cos`;
- Convert power expressions e.g. `2^3` to `Math.pow(2,3)`;
- Locate logarithms;
- Convert summations;
- Convert variable indexes into array indexes (variables are sent in an array with the dimension of the problem).

After this conversion, the Java file can be generated. An example of a Java file generated for problem S201 is presented on Fig. 4.

```
param N := 2;
var x{1..N};
minimize f:
4*(x[1]-5)^2+(x[2]-6)^2;
data;
var x:=
1 8
2 9;
solve;
display x;
```

Figure 3. Problem S201 defined in AMPL.

```
import algorithms.GeneralProblem;

public class s201 implements GeneralProblem
{

 @Override
 public int dim() {
  return (2);
 }

 @Override
 public double evaluatef(double[] x) {
  return (4.0*Math.pow((x[(int)(1-1)]-5.0),
    2.0)+Math.pow((x[(int)(2-1)]-6.0), 2.0)
    );
 }

}
}
```

Figure 4. Java code generated automatically for problem S201.

Based on this information a `.java` file is created on the temporary directory of the system. This file is the compiled into a `.class` file that can be used by the application.

All the operations related with loading the AMPL file, generating the `.java` file and loading the `.class` file are completely transparent for the programmer. The API contains a class to represent problems, named `Problem`, that can be used as follows: `Problem pr = new Problem ("http://name.of.the.host/problem.file").` In this case variable `pr` will be an instance of a `Problem` that has been created, compiled and loaded on the fly. It can now be used as it was already possible in the API, for example using the implemented Audet et al. algorithm to solve an unconstrained problem: `Audet audet = new Audet(pr); audet.run().`

IV. Tests and Numerical Results

To assess the feasibility of the proposed method, several comparison tests were made between the proposed method, the previously implemented expression evaluator and problems written natively using Java. All tests were made using a Intel(R) Core (TM) i5-2400 CPU @ 3.10 GHz, with 4Gbyte of RAM, running Debian Linux 64-bit and Kernel 2.6.32.

ISBN: 978-988-19251-0-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Table I
ORIGINAL PARSER (AVERAGE TIME IN MS)

#	1	10	100	1 000	10 000	100 000
0	0.05	0	1.65	1.30	11.30	112.80
1	0	0.05	0.25	2.50	24.60	245.40
2	0	0	0.45	3.95	39.20	391.20
3	0.05	0	0.55	5.60	55.60	555.35
4	0	0.10	0.70	7.20	72.20	720.55
5	0	0.05	0.90	8.95	88.80	885.40

Table II
AMPL PARSER (AVERAGE TIME IN MS)

#	1	10	100	1 000	10 000	100 000
0	0	0	0.05	0.15	0.90	7.15
1	0	0	0.05	0.25	1.20	11.15
2	0	0	0.05	0.35	1.75	17.60
3	0	0	0.10	0.55	4.55	44.80
4	0	0	0.15	0.90	7.10	71.70
5	0	0	0.15	1.20	9.95	98.75

It was used the 64-bit Java platform compile and run the applications (version 1.6.0_26).

A. Expression evaluation

This first set of tests aims the benchmarking of the three options for expression evaluation.

Because the objective is to test those methods, in these tests were used six expressions that even though they are of simple calculus, their expression is rather complex. The used expression is of the form of Eq. (5):

$$\varphi(x_1, x_2) = \sum_{k=0}^{x_1} \cos\left(\frac{\sqrt{k+1}}{\ln(k+2)} \times x_2^k\right) \qquad (5)$$

Those tests were made considering $x_2 = 0.95$ and $x_1 \in \{0, 1, 2, 3, 4, 5\}$, resulting on several test expressions.

These expressions were evaluated using the three methods. Each test consisted in measuring the time required to compute 1, 10, 100, 1 000, 10 000 and 100 000 expression evaluations. Results from these tests are presented on Tables I, II and III, where are presented the results obtained using the original expression evaluator, the AMPL parser and native Java, respectively.

To be noticed that average times are presented in these tables, in miliseconds, and they were calculated using the Java internal time, which as a resolution of $1ms$. As a consequence the minimum time that can be measured is $1ms$. This explains the existence of some 0 value execution times, meaning that the process was faster than $1ms$.

As it was expected, evaluating the expressions using native Java is much faster than using the expression interpreter (original parser). These results can be better observed in Table IV were is presented a comparison of the relative performance of the three methods for 100.000 evaluations of the expression.

Another conclusion that can be drawn is that the performance of the expression evaluator using dynamic code generation and compilation has a performance similar to that of native Java.

Table III
NATIVE JAVA (AVERAGE TIME IN MS)

#	1	10	100	1 000	10 000	100 000
0	0	0	0	0.10	0.80	7.10
1	0	0.05	0	0.30	1.15	11.05
2	0	0	0.05	0.35	1.60	16.45
3	0	0	0.10	0.70	4.40	43.75
4	0	0.05	0.10	1.10	7.10	70.45
5	0	0	0.15	1.45	9.80	97.55

Table V
SOME SCHITTKOWSKI PROBLEMS (AVERAGE TIME IN MS)

	S201	S205	S206	S207	S208
AMPL (parser)	5.35	12.10	8.05	6.95	5.95
JAVA (native)	5.15	5.80	7.45	6.60	5.25
Dif.	0.20	6.30	0.60	0.35	0.70
Dif. (%)	3.89	108.62	8.05	5.30	13.33

B. Testing using Optimization Problems

This second set of tests consisted in testing the two best methods (based on the above presented results) using the version of Audet Algorithm implemented in the API, to solve a set of unconstrained problems from the Schittkowski collection. These problems were: S201, S205, S206, S207 and S208.

Results obtained for these problems are presented in Table V, where are presented the execution times for these five problems using both methods, and, the relative time difference between the Java native code and the dynamic code option. The presented results are the average execution time (in ms). To minimize the effect of outliers the algorithm was executed 30 times for each problem.

Analysing data it can be concluded that the proposed solution has a performance very similar to that obtained using Java native code (only 3.89% to 13.33% slower). There is an exception for problem S205 that needed the double of the time to be solved.

V. CONCLUSION AND FUTURE WORK

In this paper it was proposed a new method to evaluate expressions, to be used on a Java-based Nonlinear Optimization API. This new method combines the flexibility of a on the fly expression evaluator with the performance of hard-coded mathematical expressions inside the application code.

Java code is generated, compiled and loaded dynamically. This allows the implementation of applications that do not need to known in beforehand which problems must be optimized, without any loss of performance.

Also an AMPL parser was included in this expression evaluator, this allows users to use a standard file format to define their problem. Furthermore, this is one of the most used file formats to describe both constrained and unconstrained optimization problems. This allows users to use the API to solve problems that can be found in online test problems repositories. If these problems are located online, the user does not even need to download them to the hard drive, they can simply indicate to the API what is the URL

ISBN: 978-988-19251-0-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Table IV
COMPARATIVE 100 000 REPETIONS (SLOWER PERCENTAGE)

#	0	1	2	3	4	5
Native JAVA	7.10	11.05	16.45	43.75	70.45	97.55
%	100	100	100	100	100	100
AMPL Parser (%)	100.70	100.90	106.99	102.40	101.77	101.23
Original Parser (%)	1 588.73	2 220.81	2 378.12	1 269.37	1 022.78	907.64

of the remote file.

Two sets of tests were made to the proposed method to compare with the methods already implemented in the API and it can be concluded that it has a good performance, very similar to the performance of native Java code.

Although the good performance, it is still possible to do some more optimization on the Java code generated by the API. Because of its automated nature, there are some generalizations that may compromise the performance. As future work an extra step before compilation will be added to claen-up the code and try to make it even faster.

ACKNOWLEDGEMENT

This work is supported by the European Union Funds (FEDER/COMPETE - Operational Competitiveness Programme) and by national funds (FCT - Portuguese Foundation for Science and Technology) under the projects PEst-OE/MAT/UI4080/2011 and FCOMP-01-0124-FEDER-022696.

REFERENCES

[1] P. Mestre, L. Reigoto, L. Coutinho, A. Correia, and J. Matias, "RSS and LEA Adaptation for Indoor Location using Fingerprinting," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2012, WCE 2012, 4-6 July, 2012, London, U.K.*, 2012, pp. 1334–1339.

[2] P. Mestre, L. Coutinho, L. Reigoto, J. Matias, A. Correia, P. Couto, and C. Serodio, "Indoor location using fingerprinting and fuzzy logic," in *Advances in Intelligent and Soft Computing*, no. 107. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2011, pp. 363–374.

[3] J. Matias, A. Correia, P. Mestre, C. Fraga, and C. Serôdio, "Web-based application programming interface to solve nonlinear optimization problems," in *Lecture Notes in Engineering and Computer Science - World Congress on Engineering 2010*, vol. 3. London, UK: IAENG, (2010), pp. 1961–1966.

[4] A. Correia, J. Matias, P. Mestre, and C. Serôdio, "Direct-search penalty/barrier methods," in *Lecture Notes in Engineering and Computer Science - World Congress on Engineering 2010*, vol. 3. London, UK: IAENG, (2010), pp. 1729–1734.

[5] P. Mestre, J. Matias, A. Correia, and C. Serodio, "Direct Search Optimization Application Programming Interface with Remote Access," *IAENG International Journal of Applied Mathematics*, vol. 40, no. 4, pp. 251–261, Nov 2010.

[6] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko, "Comparison of performance of Web services, WS-Security, RMI, and RMI-SSL," *Journal of Systems and Software*, vol. 79, no. 5, pp. 689–700, May 2006, quality Software.

[7] P. Mestre, A. Correia, J. Matias, C. Teixeira, J. Almeida, and C. Serôdio, "A Web-based Tool to Evaluate the Iterative Processes of Direct Search Methods," in *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering 2011, WCE 2011, 6-8 July, 2011, London, U.K.*, 2011, pp. 1961–1966.

[8] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to Derivative-Free Optimization*. Philadelphia, USA: MPS-SIAM Series on Optimization, SIAM, (2009).

[9] R. Lewis, V. Torczon, and M. Trosset, "Direct search methods: Then and now," *J. Comput. Appl. Math.*, no. 124, pp. 191–207, (2000).

[10] T. Kolda, R. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM Review*, vol. 45, pp. 385–482, (2003).

[11] R. Hooke and T. Jeeves, "Direct search solution of numerical and statistical problems," *Journal of the Association for Computing Machinery*, vol. 8, no. 2, pp. 212–229, (1961).

[12] C. Audet and J. Dennis, "A pattern search filter method for nonlinear programming without derivatives," *SIAM Journal on Optimization*, vol. 5, no. 14, pp. 980–1010, (2004).

[13] C. Audet and J. E. D. Jr., "Mesh adaptive direct search algorithms for constrained optimization," *SIAM Journal on Optimization*, no. 17, pp. 188–217, (2006).

[14] ——, "A mads algorithm with a progressive barrier for derivative-free nonlinear programming," Les Cahiers du GERAD, École Polytechnique de Montréal, Tech. Rep. G-2007-37, (2007).

[15] J. Dennis and D. Woods, "Optimization on microcomputers: The nelder-mead simplex algorithm," *New Computing Environments: Microcomputers in Large-Scale Computing,*, vol. Wouk, A., ed., pp. 116–122, (1987).

[16] C. Kelley, *Iterative Methods for Optimization*. Philadelphia, USA: Number 18 in Frontiers in Applied Mathematics, SIAM, (1999).

[17] J. Lagarias, J. Reeds, M. Wright, and P. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 112–147, (1998).

[18] P. Tseng, "Fortified-descent simplicial search method: A general approach," *SIAM Journal on Optimization*, vol. 10, no. 1, pp. 269–288, (2000).

[19] F. Y. Wang and D. Liu, *Advances in Computational Intelligence: Theory And Applications (Series in Intelligent Control and Intelligent Automation)*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., (2006).

[20] X. Hu and R. Eberhart, "Solving constrained nonlinear optimization problems with particle swarm optimization," in *Proceedings of the Sixth World Multiconference on Systemics, Cybernetics and Informatics 2002 (SCI 2002)*, (2002), pp. 203–206.

[21] S. Zhang, "Constrained optimization by ϵ constrained hybrid algorithm of particle swarm optimization and genetic algoritnm," in *Proceedings of AI 2005: Advances in Artificial Intelligence*. Springer: Civil-Comp press, (2005), pp. 389–400.

[22] J. Matias, "Técnicas de penalidade e barreira baseadas em métodos de pesquisa directa e a ferramenta pnl-pesdir," Ph.D. dissertation, UTAD, Vila Real, (2003).

[23] A. Pereira, "Caracterização da função de penalidade exponencial num método de redução para programação semi-infinita," Ph.D. dissertation, Universidade do Minho, Braga, (2006).

[24] R. H. Byrd, J. Nocedal, and R. A. Waltz, "Steering exact penalty methods for nonlinear programming," *Optimization Methods & Software*, vol. 23, no. 2, pp. 197–213, (2008).

[25] N. I. M. Gould, D. Orban, and P. L. Toint, "An interior-point l_1-penalty method for nonlinear optimization," Rutherford Appleton Laboratory Chilton, Tech. Rep., (2003).

[26] T. Pietrzykowski, "An exact potential method for constrained maxima," *SIAM Journal on Numerical Analysis*, vol. 6(2), pp. 299–304, (1969).

[27] R. Fletcher, S. Leyffer, and P. L. Toint, "On the global convergence of an slp-filter algorithm," Dundee University, Dept. of Mathematics, Tech. Rep. NA/183, (1998).

[28] K. Schittkowski, *More Test Examples for Nonlinear Programming Codes*. Springer-Verlag, Berlin: Economics and Mathematical Systems, 1987.

ISBN: 978-988-19251-0-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)