NLPToolbox: An Open-Source Nonlinear Programming Tool

Diego Lages
COPPE – Systems Engineering, Federal University of Rio de Janeiro
Rio de Janeiro / RJ
lages@cos.ufrj.br

Adilson Xavier
COPPE – Systems Engineering, Federal University of Rio de Janeiro
Rio de Janeiro / RJ
adilson@cos.ufrj.br

Nelson Maculan
COPPE – Systems Engineering, Federal University of Rio de Janeiro
Rio de Janeiro / RJ
maculan@cos.ufrj.br

ABSTRACT

Nonlinear programming problems (NLP) solvers require some level of flexibility. This flexibility must be supported on the method choice, on the parameters specification and on the problem modelling.

Few of the tools currently available can address this level of flexibility. This paper presents an open-source, complete and easy tool, named NLPToolbox, to achieve this purpose.

Given its open-source characteritics, it offers the opportunity to study nonlinear programming in an iterative way: by showing how the methods works and allowing all kinds of specifications: methods and parameters.

Altough being a work continually in progress, it is already usable. It is currently used in teaching nonlinear programming and solving some kinds of NLP problems, like clustering and Support Vector Machine classification. Its future lies on the optimization of the tool itself, improving the precision of the numeric algorithms and integrating new methods.

Categories and Subject Descriptors

G.1.6 [**NUMERICAL ANALYSIS**]: Optimization—*Constrained optimization, Convex programming, Global optimization, Gradient methods, Nonlinear programming, Unconstrained optimization*

General Terms

Optimization, Open-Source, Unconstrained Optimization: Quasi-Newton Methods, Conjugate Gradient, Constrained Optimization: Penalty Methods

1. INTRODUCTION

Many scientific challenges depend on solving nonlinear programming problems (NLP). Despite many tools (open-source, free or commercial) being available, usually they are designed as a "black-box", making it very hard to make parameter tunnings, test new approaches, create and validate new derived methods.

It is even harder to "connect" the solution of a NLP to another solution or data structure from another scientific problem. This lack of "connection", just like the problems listed above, could be solved using Software Engineering. The most common approaches to solve them are throught the creation of *frameworks* and components[9].

After searching for similar tools, libraries or components for NLP, a few were found. The ones found suffer from the problems listed, are incomplete, not open source or not well structured to be readily usable by third parties.

So, the NLPToolbox (Nonlinear Programming Toolbox) attempts to address all these concerns, being at the same time a tool for solving NLP and a component that can be embedded in others softwares. It can solve all kinds of NLP, constrained or not, using Newton, Quasi-Newton and Conjugate Gradient Methods. This tool is **open-source** (GPL), so anyone has full access to the source code. The GPL (GNU General Public Licence) warrants that any future changes must still be open-source.

It intends to be the most flexible one, so that it can be used as a component to other softwares (commercial or not) or scientific experiments, with well-defined interfaces. Moreover, it has also a graphic interface (GUI) so that NLP problems can be solved in a confortable way.

Beyond being a component and a tool, it intends to help the learning of nonlinear optimization. By the open-source, readable source code, the most common optimization methods can be learned by seeing how they are implemented, used and can still be changed. It is possible to show the iteration sequence step by step.

It is available, together with all source code, documentation and tests, at `http://www.cos.ufrj.br/~lages/nlptoolbox/`.

2. ARCHITECTURE

The tool itself is a component for solving NLP. Its architecture aims at the following objetives:

- **Platform Independence** - The NLPToolbox must be platform independent, allowing its use in any operating system or processor (Intel, Sparc, etc.);
- **Component-Based** - The NLPToolbox must be component-based, so that it can be easily used by other programs or components;
- **Complete** - The NLPToolbox must be complete, implementing most common nonlinear optimization methods: Newton, Quasi-Newton and Conjugate Gradient;

SIMUTools 2009, Rome, Italy

- **Fast Problem Implementation** - Usually, after problem design, too much time is lost in implementatiion of the solution into the tools, libraries, calculations (the Hessian, for instance), parameters tunning, testing, etc. In many instances, it takes more time than running the large-scale problem itself. One of the objetives of the NLPToolbox is to focus on the problem modelling, instead of implementation issues;
- **Numeric Precision Flexibility** - Most of optimization problems suffer from lack of numeric precision. There are many ways to deal with this problem, usually, dependant on its characteristics. The NLPToolbox must address this issue somehow;
- **Graphic User Interface (GUI)** - The NLPToolbox must have a GUI for solving problems easily. Current tools and libraries do need that code must be created to compute functions, gradients and Hessians, which can lead to many errors;
- **Parallelism** - Most of current tools and libraries **are not** parallel. As problems become bigger and parallel computers are becoming more common (for instance *dual or quad-core* computers), they should be solved concurrently.

3. FEATURES

The NLPToolbox implemented the architecture listed above, throught the following implementation:

- **Platform Independence** - The NLPToolbox was created using the Java language, that supports the majority of currents operating systems and processors, without the need of a separate program for each platform;
- **Component-Based** - The tool itself is a component, with well-defined interfaces.
- **Complete** - The following line-search methods are implemented:
  - Armijo [19];
  - Newton-Raphson [19];
  - Wolfe [19];
  - Secant [19].

  The following unconstrained nonlinear otimization methods:
  - Newton [19];
  - Conjugate Gradient: Hestenes-Stiefel [15];
  - Conjugate Gradient: Polak-Ribiere [20][21];
  - Conjugate Gradient: Daniel [6];
  - Conjugate Gradient: Fletcher-Reeves [12];
  - Conjugate Gradient: Conjugate Descendent [11];
  - Conjugate Gradient: Liu-Storey [22];
  - Conjugate Gradient: Dai-Yuan [5];
  - Conjugate Gradient: Hager-Zhang [14];
  - Quasi-Newton: BFGS [10][13][24];
  - Quasi-Newton: DFP [8];
  - Quasi-Newton: Broyden [7].

  The following constrained nonlinear otimization methods:
  - Penalty Method: inverse function penalty [19];
  - Penalty Method: logarithmic function penalty [19].
- **Fast Problem Implementation** - The problem specification can be **entered** in natural form on the GUI. The NLPToolbox automatically calculates the exact values of the objective function, the gradient and the Hessian, by using automatic differentiation procedures. This feature minimizes commons errors, as seen on figure 1.

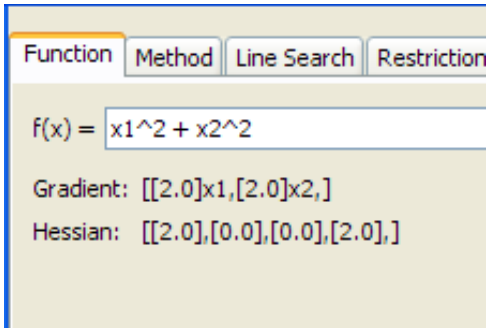

Figure 1: Method for inputting the function.

- **Numeric Precision Flexibility** - Instead of using just floating-point operations, the NLPToolbox supports five different numeric types, as seen on figure 2:
  - **Complex** - Complex Numbers;
  - **Float64** - Standard floating-point. But, some routines were created to minimize numeric precision errors;
  - **FloatingPoint** - Precision-garanteed floating-point.
  - **Rational** - Rational numbers
  - **Real** - An different approach of precision-garanteed floating-point.

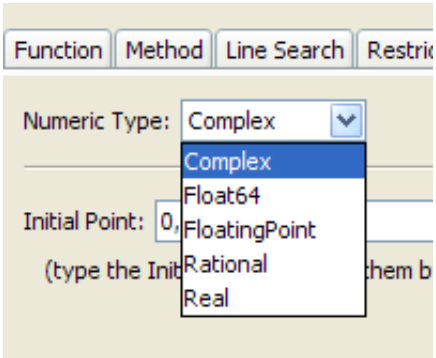

Figure 2: Configuring the numeric precision.

- **Graphical User Interface (GUI)** - The GUI offers the possibility of entering all the problem specification and configuring most of the parameters used in the methods, as seen on figure 3

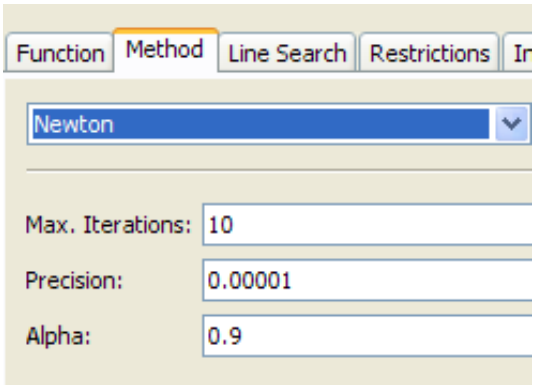

Figure 3: Selecting the minimization method and parameters.

- **Parallelism** - the NLPToolbox is naturally parallelizable. All matrix and vector operations are automatically parallelized for large problem instances. The default is, for instance, to parallelize a multiplication of a dense matrix with more than 32 columns.[1]

4. PERFORMANCE

There is a belief that Java programs are slower than ones programmed in C or Fortran. The reason for this belief is that a Java program is interpreted and not executed natively, which is a wrong assumption. Modern JVMs (Java Virtual Machines) translate the program, at execution time, into the native architecture, and then, runs it. Many others language optimization techniques have been (and it is still being) incorporated into Java. The resulting performance gains make most Java programs runs as fast as equivalent counterparts written in C, Fortran or other natively compiled languages. In some instances, the Java program runs faster that its native counterpart. For a study comparing Java performance for numeric processing, see [4].

5. EXAMPLE

The NLPToolbox can be used in a very easy way, inputting the objective function and the parameters in the GUI. An alternative approach in shown below, supported by the NLPToolbox component. This procedure can be embedded into other software.

This example will show how to minimize the "banana function" (Rosenbrock[17]):

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1)^2$$

A `class` must be created to hold this function:

```
ExpressionFunction ef =
   new ExpressionFunction(
      "(1-x1)^2+100*(x2-x1)^2");
ef.setNumberfactory(new RationalNumberFactory());
```

[1]Some of the linear algebra routines were provided by the JScience[2] library (version 4.3). The expression parser was based on JEPLite[1], heavily adapted to the NLPToolbox. Both of the library sources codes are bundled with the source tree of the tool.

A `NumberFactory` is needed to choose the proper numeric precision. To use another function, just change the appropriate string in the code above. The component will identify automatically new variables $x_1, x_2, ..., x_n$[2]. Neither the gradient of the function nor its Hessian has to be entered.

Using the initial point $[\frac{-3}{2}, 1]$[3]:

```
Rational ix1 = Rational.valueOf(-3, 2)
Rational ix2 = Rational.valueOf(1, 1);

Vector<Rational> iv =
   DenseVector.valueOf(ix1,ix2);
```

Now, create the minimization method `class`, with the initial point listed above, 10 iterations at most, using the quadratic norm, using an alpha of $\frac{9}{10}$ (parameter for the *Newton* method) and with a precision of $\frac{1}{1000}$:

```
Newton<Rational> n = new Newton<Rational>();
n.setInitialPoint(iv);
n.setMaxiterations(10);
n.setNorm(new QuadraticNorm<Rational>());
n.setParameters(Rational.valueOf(9,10));
n.setPrecision(Rational.valueOf(1, 1000));
```

To run the minimization:

```
System.out.println(
   "min: " + n.minimize(ef).toString());
System.out.println(
   "iterations: " + n.getIterationscount());
System.out.println(
   "error: " + n.getErr());
```

Here is the full code set for the example:

```
import br.ufrj.cos.nlptoolbox.functions.
         ExpressionFunction;
import br.ufrj.cos.nlptoolbox.methods.Newton;
import br.ufrj.cos.nlptoolbox.numberfactories.
         RationalNumberFactory;
import org.jscience.mathematics.number.Rational;
import org.jscience.mathematics.vector.DenseVector;
import org.jscience.mathematics.vector.Vector;

/**
 *
 * @author Diego
 */
public class MainNewton {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        ExpressionFunction ef =
           new ExpressionFunction(
              "(1-x1)^2+100*(x2-x1)^2");
```

[2]The n must be identified: infinite functions or undefined variable length problems are not supported

[3]The *Vector* `class` type can represent a vector or point in space. It is a collection of numbers.

```
        ef.setNumberfactory(new RationalNumberFactory());
        Rational ix1 = Rational.valueOf(-2, 1);
        Rational ix2 = Rational.valueOf(1, 1);

        Vector<Rational> iv =
           DenseVector.valueOf(ix1,ix2);

        Newton<Rational> n = new Newton<Rational>();
        n.setInitialPoint(iv);
        n.setMaxinterations(10);
        n.setNorm(new QuadraticNorm<Rational>());
        n.setParameters(Rational.valueOf(9,10));
        n.setPrecision(Rational.valueOf(1, 1000));

        System.out.println(
           "minimização: " + n.minimize(ef).toString());
        System.out.println(
           "iterações: " + n.getIterationscount());
        System.out.println(
           "erro: " + n.getErr());

    }

}
```

6. VERIFICATION AND VALIDATION

In order to check whether it implements correctly the algorithms proposed, automated tests were created. These tests were implemented using JUnit[3], and focused on the three main aspects of the tool:

- **Line Search** - Some simple functions were used to check if different line search methods worked perfectly;
- **Minimization Algorithm** - Tests were made to check if the algorithms were correctly implemented. Some of the functions proposed in [17] plus some simple functions were used;
- **Numeric precision** - Tests were made to check if different numeric precisions leads to the same aproximate results.

Many minimization methods did converge and some methods did not get close enough to the global optimum, with the default initial point and parameters. The numeric precision did also some influence on the convergence to the results, depending on the method. A summary of these results, using some of the functions from its automated tests, can be seen on the appendix, on tables 1, 2 and 3.

7. CONCLUSION

NLPToolbox is an evolving optimization tool, based on a modular design that allows for easy implementation of new features. Currently, it has the following limitations:

- **Large-Scale problems support** - It is necessary to implemente some features in order to solve really large problems. Some of these can be found on [18], [16] and [23];
- **Floating-Point Processing** - It is possible to improve the floating-point precision routines using ones usually available in numerical libraries, such as Harwell library;
- **LAPACK Integration** - NLPToolbox can be easily integrated with the LAPACK library for numeric processing. However, some of these routines are not parallel;
- **Constrained Optimization** - The constrained optimization methods available are still limited. Other methods are being implemented;
- **Other unconstrained minimization methods** - Other methods of unconstrained optimization will be implemented;
- **Expression parse optimization** - The symbolic function parser can be optimized for native evaluation.

8. ACKNOWLEDGMENTS

We thank Geraldo Veiga from COPPE/UFRJ for the help in the text and CAPES/CNPq for financial support.

9. REFERENCES

[1] JEPLite, http://jeplite.sourceforge.net/ (last visit in 01/18/2009).
[2] JScience, http://jscience.org/ (last visit in 01/18/2009).
[3] JUnit, http://www.junit.org (last visit in 01/18/2009).
[4] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbe, F. Huet, and G. L. Taboada. Current state of java for hpc. August 2008.
[5] Y. H. Dai and Y. Yuan. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM J. Optim.*, (10):177–182, 1999.
[6] J. W. Daniel. The conjugate gradient method for linear and nonlinear operator equations. *SIAM J. Numer. Anal.*, (4):10–26, 1964.
[7] W. C. Davidon. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, (19):577–593, October 1965.
[8] W. C. Davidon. Variable metric method for minimization. *SIAM Journal on Optimization*, (1):1–17, 1991.
[9] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley Professional, Boston, Massachusetts, 1998.
[10] R. Fletcher. A new approach to variable metric algorithms. *Computer Journal*, (13):317–322, 1970.
[11] R. Fletcher. *Practical Methods of Optimization vol. 1: Unconstrained Optimization.* John Wiley and Sons, New York, 1987.
[12] R. Fletcher and C. Reeves. Function minimization by conjugate gradients. *Comput. J.*, (7):149–154, 1964.
[13] D. Goldfarb. A family of variable metric updates derived by variational means. *Mathematics of Computation*, (24):23–26, 1970.
[14] W. W. Hager and H. Zhang. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM J. Optim.*, 2003.
[15] M. R. Hestenes and E. L. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards*, (49):409–436, 1952.

[16] D. C. Liu and J. Nocedal. On the Limited Memory Method for Large Scale Optimization. In *Mathematical Programming*, 1989.

[17] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Trans. Math. Softw.*, 7(1):17–41, 1981.

[18] J. Nocedal. Updating Quasi-Newton Matrices with Limited Storage. In *Mathematics of Computation*, 1980.

[19] J. Nocedal and S. J. Wright. *Numerical optimization*. Springer Verlag, New York, NY, 1999.

[20] E. Polak and G. Ribière. Note sur la convergence de directions conjugées. *Rev. Francaise Informat Recherche Opertionelle*, (3e Année 16):35–43, 1969.

[21] B. T. Polyak. The conjugate gradient method in extreme problems. *USSR Comp. Math. Math. Phys.*, (9):94–112, 1969.

[22] B. T. Polyak. Efficient generalized conjugate gradient algorithms, part 1: Theory. *J. Optim. Theory Appl.*, (69):129–137, 1991.

[23] P. L. R. H. Byrd and J. Nocedal. A Limited Memory Algorithm for Bound Constrained Optimization. In *SIAM Journal on Scientific and Statistical Computing*, 1995.

[24] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, (24):647–656, 1970.

APPENDIX

A. NLPTOOLBOX RUNNING TIMES

In the following tables are listed the running times for some of the automated tests functions. All the tests were performed in a Intel Pentium IV 2.16 Ghz, with 2 Gb of RAM, running Windows XP and Java 1.6.0_11b03. In the Newton method, an α step is set to 0.9, by default. Notice that, in the tests, the function parser is used, having some effect on the overall performance.

The Freudenstein and Roth[17] function is defined by:

$$f(x,y) = (-13+x+((5-y)y-2)y)^2+(-29+x+((y+1)y-14)y)^2$$

The "Big Rosenbrock" function is the Rosenbrock[17] function with 20 variables:

$$f(x) = \sum_{i=1}^{19}[(1-x_i)^2 + 100(x_{i+1}-x_i)^2]$$

The maximum number of line search iterations was 20, the maximum number of iterations was 10000 and the initial points were:

- $x_1^2 + x_2^2$ - $(-2, 1)$;
- **Rosenbrock** - $(-2, 1)$;
- **Freudenstein and Roth** - $(0.5, -2)$;
- **Big Rosenbrock** - $(-2, 1, ..., 1)$.

[4]More than 5 minutes.
[5]NaN.
[6]Minimum with the required precision not found.

	Iterations	Seconds
$x_1^2 + x_2^2$		
Complex	5	0.047
Float64	5	0
FloatingPoint	5	0
Rational	5	0.046
Real	5	0.016
Rosenbrock		
Complex	5	0.047
Float64	5	0
FloatingPoint	5	0
Rational	*[4]	*[4]
Real	5	0.016
Freudenstein and Roth		
Complex	6	0.047
Float64	6	0
FloatingPoint	6	0.015
Rational	*[4]	*[4]
Real	6	0.016
Big Rosenbrock		
Complex	5	0.125
Float64	5	0.031
FloatingPoint	*[4]	*[4]
Rational	5	109.343
Real	*[5]	*[5]

Table 1: Sample results using the Newton method.

	Iterations	Seconds
$x_1^2 + x_2^2$		
Complex	6	0.016
Float64	6	0
FloatingPoint	6	0.016
Rational	*[5]	*[5]
Real	*[5]	*[5]
Rosenbrock		
Complex[6]	11	0.031
Float64	182	0.079
FloatingPoint[6]	3526	13.062
Rational	*[4]	*[4]
Real	*[5]	*[5]
Freudenstein and Roth		
Complex	349	0.234
Float64	134	0.157
FloatingPoint[6]	91	0.781
Rational	*[4]	*[4]
Real	*[5]	*[5]
Big Rosenbrock		
Complex	2356	94.750
Float64	5023	199.172
FloatingPoint	*[4]	*[4]
Rational	*[4]	*[4]
Real	*[5]	*[5]

Table 2: Sample results using the Fletcher-Reeves method.

	Iterations	Seconds
$x_1^2 + x_2^2$		
Complex	5	0.016
Float64	5	0
FloatingPoint	5	0
Rational	5	0.046
Real	1	0.016
Rosenbrock		
Complex	55	0.094
Float64	82	0.109
FloatingPoint	20	0.109
Rational	$*^4$	$*^4$
Real	$*^5$	$*^5$
Freudenstein and Roth		
Complex	12	0.032
Float64	10	0
FloatingPoint	13	0.031
Rational	$*^4$	$*^4$
Real	$*^5$	$*^5$
Big Rosenbrock		
Complex	542	26.312
Float64	558	26.516
FloatingPoint	$*^4$	$*^4$
Rational	$*^4$	$*^4$
Real	$*^5$	$*^5$

Table 3: Sample results using the BFGS method.