OPEN SOURCE ENGINEERING

open ee ring

A SCILAB PROFESSIONAL PARTNER

powered by ENGIN SOFT

MULTIOBJECTIVE OPTIMIZATION? DO IT WITH SCILAB

Authors: *Silvia Poles*

Keywords. Multiobjective optimization; Scilab

Abstract: One of the Openeering team goal is to support optimization in companies daily activities. We never miss the opportunity to stress the importance of optimization and to explain how optimization can play a significant role in the design cycle.

In this tutorial we show how Scilab can be considered as a powerful multiobjective and multidisciplinary optimization software.

Contacts s.poles@openeering.com

1. Optimization? Do it with Scilab!

One of the Openeering team goal is to support optimization in companies daily activities. We never miss the opportunity to stress the importance of optimization and to explain how optimization can play a significant role in the design cycle. When we talk about optimization, we always refer to real-life applications as we know that our readers are interested in methods and software for solving industrial cases. Particularly, we refer to problems where multiple and nonlinear objectives are involved.

In this article we will introduce you Scilab[1], a numerical computing environment that should be considered as a powerful multiobjective and multidisciplinary optimization software. Scilab is a high-level matrix language with a syntax that is very similar to MATLAB®[2]. Exactly as MATLAB® does, Scilab allows to define mathematical models and to connect to existing libraries. As for MATLAB®, optimization is an important topic for Scilab. Scilab has the capabilities to solve both linear and nonlinear optimization problems, single and multiobjective, by means of a large collection of available algorithms.

Here we are presenting an overall idea of the optimization algorithms available in Scilab; the reader can find some code that can be typed and used in the Scilab console to verify the potential of this numerical computing environment for solving very common industrial optimization problems[3].

2. Linear and nonlinear optimization

As our readers already know, “optimize” means selecting the best available option from a wide range of possible choices. Doing this as a daily activity can be a complex task as, potentially, a huge number of choices should be tested when using a brute force approach.

The mathematical formulation of a general optimization problem can be stated as follows:

$$\min_{x \in S^n} (f_1(x), \cdots, f_k(x))$$
$$\text{subject to} \begin{cases} g_i(x) \geq 0 \\ g_l(x) = 0 \\ x \in S^n \end{cases}$$

(x1, ..., xn) are the variables, the free parameters which can vary in the domain S. Any time that *k>1*, we speak about multiobjective optimization.

This is a very general definition because it remains valid for continuous and discrete variables, for mixed integer problems, for linear and nonlinear functions, for linear and nonlinear constraints. It may exist several different tailored approaches for solving an optimization problem according to the number and type of variables, functions and constraints involved.

[1] Download Scilab for free at http://www.scilab.org/

[2] MATLAB is a registered trademark of The MathWorks, Inc

[3] Contact the author for the original version of the Scilab scripts

3. Graphical methods

Scilab is very helpful for solving daily optimization problems even simply by means of a graphical method.

For example, suppose that you would like to find out the minimum point of the Rosenbrock function. The contour plot can be a visual aid to identify the optimal area. Start up Scilab, copy the following Scilab script and you obtain the plot in **Figure 1**.

```
function f=rosenbrockC(x1, x2)
   x = [x1 x2];
   f = 100.0 *(x(2)-x(1)^2)^2 + (1-x(1))^2;
endfunction
xdata = linspace(-2,2,100);
ydata = linspace(-2,2,100);
contour( xdata , ydata , rosenbrockC , [1 10 100
1000])
```

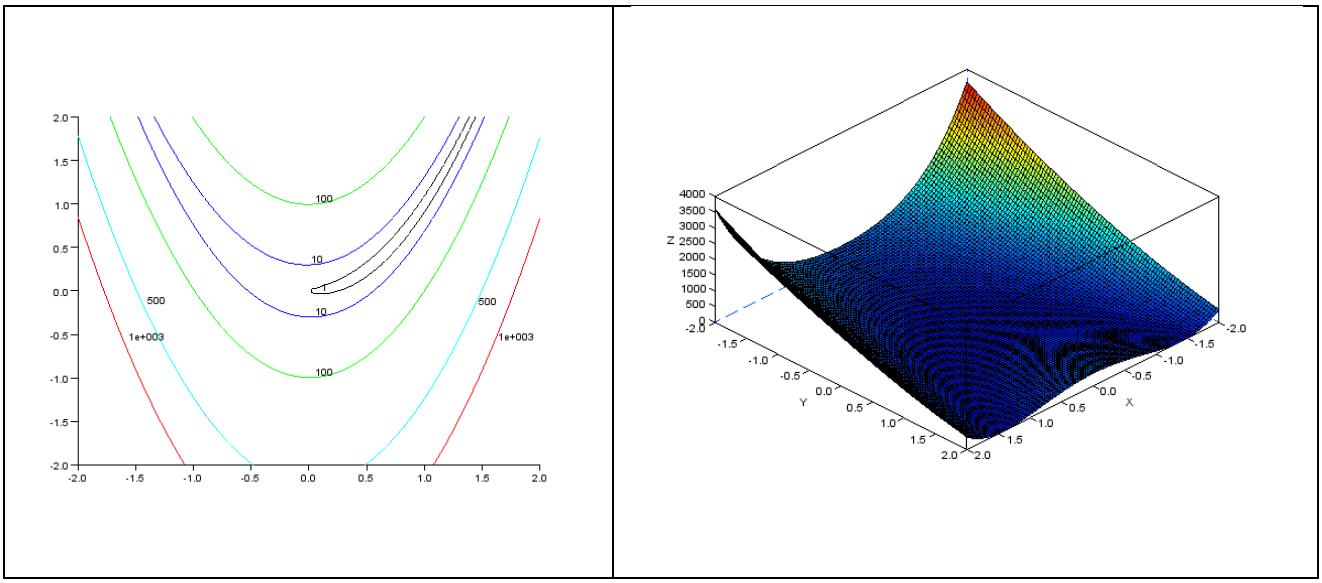

Figure 1: Contour plot (left) and 3d plot (right) of the Rosenbrock function. With this chart we can identify that the minimum is in the region of the black contour with label 1. These means that a good starting point for further investigations could be x=(0.5, ,0.5).

The contour plot can be the first step for finding an optimal solution. By the way, solving an optimization problem by means of graphical methods is only feasible when we have a limited number of input variables (2 or 3). In all other cases we need to proceed further and use numerical algorithms to find solutions.

4. Optimization algorithms

A large collection of different numerical methods is available for further investigations. There are tens of optimization algorithms in Scilab and each method can be used to solve a specific problem according to the number and smoothness of functions *f*, the number and type of variables *x*, the number and type of constraints *g*. Some methods can be more suitable for constrained optimization, others may be better for convex problems, others can be tailored for solving discrete problems. Specific methods can be useful for solving quadratic programming, nonlinear problems, nonlinear least squares, nonlinear equations, multi-objective optimization, and binary integer programming. Table 1 gives an overview of the optimization algorithms available in Scilab. Many other optimization methods are made available from the community every day as external modules using the ATOMS Portal,

http://atoms.scilab.org/.

For showing the potentiality of Scilab as an optimization tool, we can start from the most used optimization function: **optim**. This command provides a set of algorithms for nonlinear unconstrained and bound-constrained optimization problems.

Let's see what happens if we use the **optim** function for the previous problem:

```
function [f, g, ind]=rosenbrock(x, ind)
    f = 100*(x(2)-x(1)^2)^2+(1-x(1))^2;
    g(1) = -400.*(x(2)-x(1)^2)*x(1)-2.*(1.-x(1))
    g(2) = 200.*(x(2)-x(1)^2)
endfunction
x0 = [-1.2 1];
[f, x] = optim(rosenbrock, x0);
// Display results
mprintf("x = %s\n", strcat(string(x)," "));
mprintf("f = %e\n", f);
```

If we use x0=[-1.2 1] as initial point, the function converges easily to the optimal point x*=[1,1] with f=0.0.

The previous example calculates both the value of the Rosenbrock function and its gradient, as the gradient is required by the optimization method. In many real case applications, the gradient can be too complicated to be computed or simply not available since the function is not known and available only as a black-box from an external function calculation. For this reason, Scilab has the possibility to compute the gradient using finite differences using the function **derivative** or the function **numdiff**.

For example the following code define a function f and compute the gradient on a specific point x.

```
function f=myfun(x)
  f=x(1)*x(1)+x(1)*x(2)
endfunction
x=[5 8]
g=numdiff(myfun,x)
```

These two functions (derivative and numdiff) can be used together with optim to minimize problem where gradient is too complicated to be programmed.

The **optim** function uses a quasi-Newton method based on **BFGS** formula that is an accurate algorithm for local optimization. On the same example, we can even apply a different optimization

approach such as the derivative-free Nelder-Mead **Simplex** [1] that is implemented in the function **fminsearch**. To do that we just have to substitute the line:

```
[f, x] = optim(rosenbrock, x0);
```

with

```
[x,f] = fminsearch(rosenbrock, x0);
```

This Nelder-Mead Simplex algorithm, starting from the same initial point, converges very close to the optimal point and precisely to x*=[1.000022 1.0000422] with f=8.177661e-010. This shows that the second approach is less accurate than the previous one: this is the price to pay in order to have a more robust approach that is less influenced by noisy functions and local optima.

Figure 2 shows the convergence of the Nelder-Mead Simplex method on the Rosenbrock function.

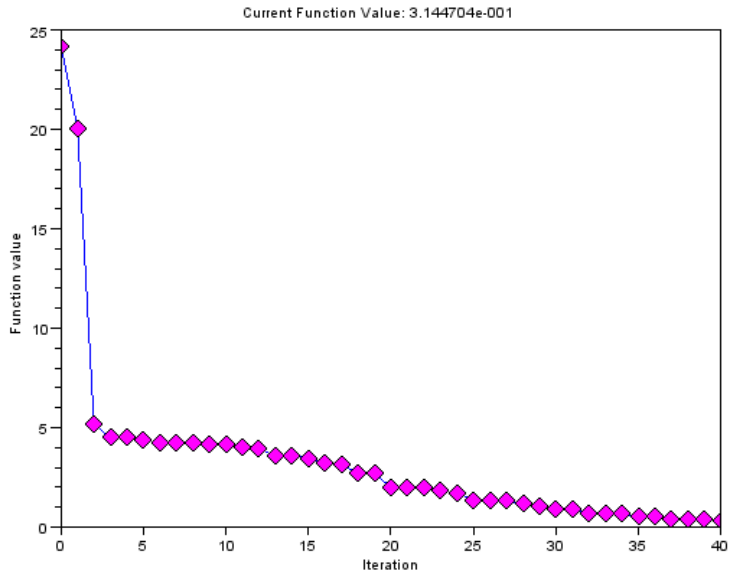

Figure 2: Convergence of the Nelder-Mead Simplex algorithms (fminsearch function) on the Rosenbrock example.

Objective	Bounds	Equality	Inequalities	Problem size	Gradient needed	Solver
Linear	y	l	l	m	-	linpro
Quadratic	y	l	l	m	-	quapro
						qld
				l	-	qpsolve
Nonlinear	y			l	y	optim
				s	n	neldermead
						optim_ga
				s	n	fminsearch
						optim_sa
Nonlinear Least Squares				l	optional	lsqrsolve
						leastsq
Min-Max	y			m	y	optim/nd
Multi-Obj.	y			s	n	optim_moga
			l*	l		semidef
Semi-Def.		l*	l*	l	n	lmisolve

Table 1: This table (originally published in [2]) gives an overview of the optimization algorithms available in Scilab and the type of optimization problems which can be solved. For the constraints columns, the letter "l" means "linear". For the problem size s,m,l indicate small, medium and large respectively that means less than ten, tens or hundreds of variables.

It is important to say that, in the previous example, the function is given by means of a Scilab script but this was only done for simplicity. It is always possible to evaluate the function *f* as an external function such as a C, Fortran or Java program or external commercial solver.

5. Parameter identification using measured data

In this short paragraph we show a specific optimization problem that is very common in engineering. We demonstrate how fast and easy can be making parameters identification for nonlinear systems, based on input/output data.

Suppose for example that we have a certain number of measurements in the matrix **X** and the value of the output in the vector Y. Suppose that we know the function describing the model (**FF**) apart from a set of parameters **p** and we would like to find out the value of those parameters. It is sufficient to write few lines of Scilab code to solve the problem:

```
//model with parameters p
function y = FF(x, p)
    y = p(1)*(x-p(2))+p(3)*x.*x;
endfunction

Z = [Y;X];
//The criterion for evaluating the error
function e = G(p,z)
   y = z(1),x=z(2);
   e = y-FF(x,p);
endfunction

//Solve the problem giving an initial guess for p
```

```
p0=[1;2;3]
[p,err]=datafit(G,Z,p0);
```

This method is very fast and efficient, it can find parameters for a high number of input/output data. Moreover it can take into consideration parameters bounds and weights for points.

6. Evolutionary Algorithms: Genetic and Multiobjective

Genetic algorithms [3] are search methods based on the mechanics of natural evolution and selection. These methods are widely used for solving highly non-linear real problem because of their ability of being robust against noisy and local optima. Genetic algorithms are largely used in real-world problems as well as in a number of engineering applications that were hard to solve with "classical" methods.

Using the genetic algorithms in Scilab is very simple: in a few lines it is possible to set the required parameters such as the number of generations, the population size, the probability of cross-over and mutation. Figure 3 shows a single objective genetic algorithm **optim_ga** on the Rosenbrock function. Twenty initial random points (in yellow) evolve through 50 generations towards the optimal point. The final generation is plotted in red.

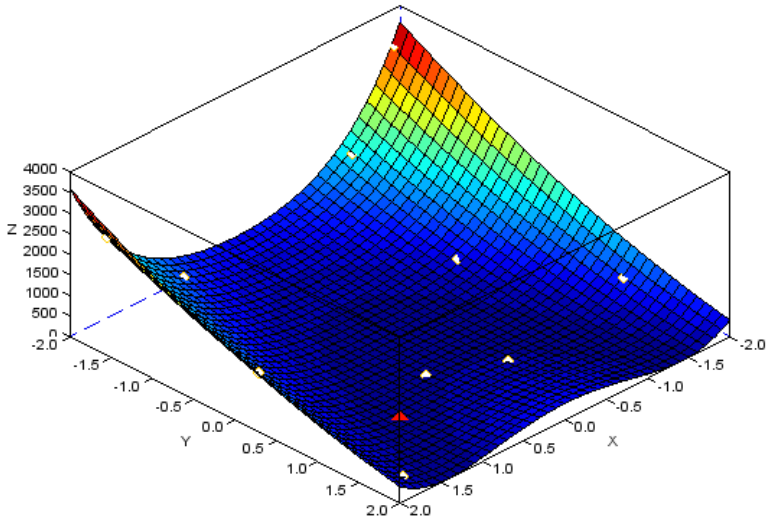

Figure 3: Optimization of the Rosenbrock function by means of a Genetic Algorithm. Initial random population is in yellow, final population is in red and converges to the real optimal solution.

7. Multiobjective

Scilab is not only for single objective problems. It can easily threat multiobjective optimization problems. Just to list one of the available methods, Scilab users can take advantage of the NSGA-II. NSGA-II is the second version of the famous "Non-dominated Sorting Genetic Algorithm" based on the work of Prof. Kalyanmoy Deb [4]. NSGA-II is a fast and elitist multiobjective evolutionary algorithm.

Figure 4 shows a multiobjective optimization run with NSGA-II using the test problem ZDT1. The test problem states:

```
function f=zdt1(x)
f1_x1 = x(:,1);
g_x2  = 1 + 9 * ((x(:,2)-x(:,1)).^2);
h     = 1 - sqrt(f1_x1 ./ g_x2);

f(:,1) = f1_x1;
f(:,2) = g_x2 .* h;
endfunction
```

With the ZDT1 we want to minimize both f1 and f2: this means that we are in front of a multiobjective problem. With these problems, the notion of optimal solutions changes. A multiobjective optimization does not produce a unique solution but a set of solutions. These solutions are named **non-dominated** or **Pareto** solutions, the set of solutions can be called **Pareto frontier**.

Figure 4 shows the solutions given by the Scilab's NSGA-II optimization algorithm for solving the ZDT1 problem. Red points on the top are the initial random populations, black points on the bottom the final Pareto population. The solid line represents the actual Pareto frontier that, in this specific example, is a continuous convex solution and is known. In this example, the concept of Pareto dominance is clear. Red points on the top are dominated by black points on the bottom because red points are worst than black points with respect to both objectives f1 and f2. On the contrary, all black points on the bottom figure are not dominating each others, we may say in this case that all the black points represent the set of efficient solutions.

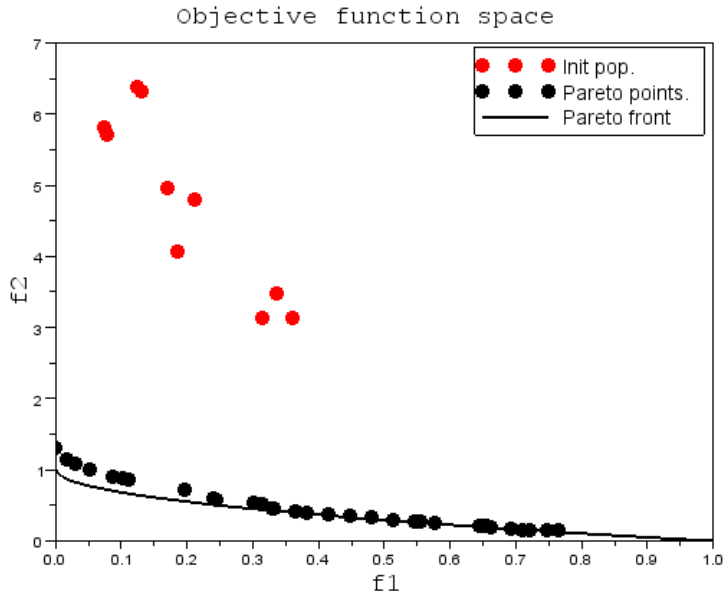

Figure 4: ZDT1 problem solved with the Scilab's NSGA-II optimization algorithm. Red points on the top represents the initial populations, black points on the bottom the final Pareto population. The solid line represents the Pareto frontier that, in this specific example, is a continuous convex solution.

To understand how Scilab recognizes the importance of multiobjective optimization, we can even notice that it has an internal function named **pareto_filter** that is able to filter non-dominated solutions on large set of data.

```
X_in=rand(1000,2);
F_in=zdt1(X_in);
[F_out,X_out,Ind_out] = pareto_filter(F_in,X_in)
drawlater;
plot(F_in(:,1),F_in(:,2),'.r')
plot(F_out(:,1),F_out(:,2),'.b')
drawnow;
```

The previous code generates 1,000 random input values, evaluates the zdt1 function and computes the non-dominated solutions. The last four lines of the code generate the following chart (Fig. 5) with all the points in red and the Pareto points in blue.

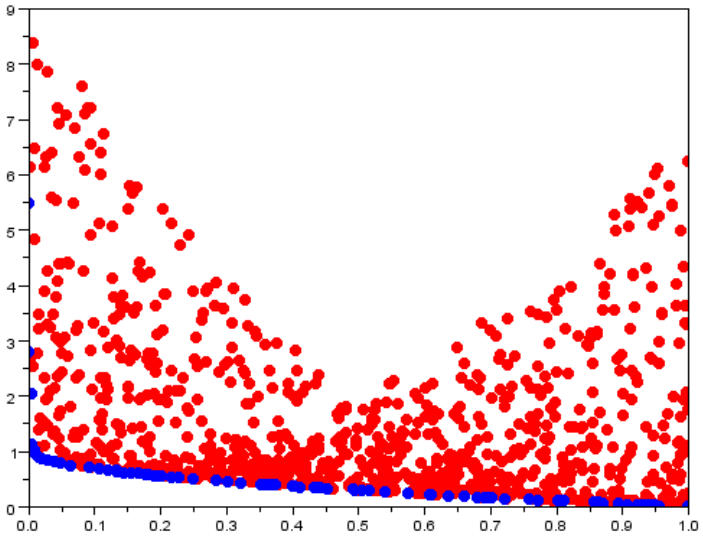

Figure 5: zdt1 function evaluate on 1,000 random points. Blue points are non-dominated Pareto solutions. The code for selecting the Pareto solution is reported in the text. The main function to be used is "pareto_filter".

8. Solving the cutting stock problem: reducing the waste

The cutting stock problem is a very common optimization problem in industries and it is economically significant. It consists on finding the optimal way of cutting a semi-processed product into different sizes in order to satisfy a set of customers' demands by using the material in the most efficient way. This type of problem arises very often in industries and can involve a variety of different goals such as minimizing the costs, minimizing the number of cutting, minimizing the waste of material and consequently costs to dumb it, and so on. Whatever the target is, it is always true that small improvements in cutting layout can result in remarkable savings of material and considerable reduction in production costs.

In this section we will show how to solve a one-dimensional (1D) cutting stock problem with Scilab.

Solving a 1D cutting stock problem is less complex than solving a 2D cutting stock problem (e.g. cutting rectangles from a sheet), nevertheless it represents an interesting and common problem. The 1D problem can arise, for example, in the construction industries where steel bars are needed in specified quantities and lengths and are cut from existing long bars with standard lengths. It is well-known that cutting losses are maybe the most significant cause of waste.

Suppose now that you are working for a company producing pipes that have usually a fixed length waiting to be cut. These tubes are to be cut into different lengths to meet customers' requests. How can we cut the tubes in order to minimize the total waste?

The mathematical formulation for the 1D cutting stock problems can be:

$$\min_x \sum_{i=1}^{n} c_i x_i$$

$$subject\ to \sum_{i=1}^{n} a_{ij} x_i = q_j \quad with\ j = 1, \dots, m$$

Where, *i* is the index of the patterns, *j* the index of the lengths, x_i are the number of cutting patterns *i* (decision variables) and c_i are the costs of the pattern *i*. $A=(a_{ij})$ the matrix of all possible patterns and q_j the customers' requests. We may say that the value a_{ij} indicates the number of pieces of length *j* within one pipe cut in the pattern *i*. The goal of this model is to minimize the objective function which consists of the total costs of the cutting phase. If c_i is equal to 1 for all the patterns, the goal corresponds to the total number of pipes required to accomplish the requirements.

Let's make a practical example and solve it with Scilab. Suppose that we have 3 possible sizes, 55mm, 26mm, and 24mm in which we can cut the original pipe of 100 mm. The possible patterns are:

1. One cut of type one and one of type two and zero of type three [1 1 0]
2. One cut of type one and one of type three [1 0 1]
3. Two cut of type two and two of type three [0 2 2]

These patterns define the matrix A. Then we have the costs that are 4, 3 and 1 for the pattern 1, 2 and 3 respectively. The total request from the customers are: 150 pieces of length 55mm, 200 with length equal to 26mm and 300 pieces with length 24mm.

For solving this problem in Scilab we can use this script.

```
//pattern
aij=[ 1 1 0;
      1 0 1;
      0 2 2];
//costs
ci=[4; 3; 1];
//request
qj=[150; 200; 300];

xopt = karmarkar(aij',qj,ci)
```

Running this script with Scilab you obtain xopt=[25, 125, 87.5], this means that to satisfy the requests reducing the total number of pipes we have to cut 25 times the pattern (1), 125 time with pattern (2) and 87.5 times the pattern (3).

We show here a simple case with only three different requests and three different patterns. The problem can be much more complicated, with many more options, many different dimensions, costs and requests. It may include the maximum number of cuts on a single piece, it may require a bit of effort in generating the list of feasible pattern (i.e. the matrix A). All these difficulties can be coded with Scilab and the logic behind the approach remains the same.

The previous script uses an interior point method [5] to solve this linear problem. The result gives and output that is not an integer solution, hence we need to approximate because we cannot cut 87.5 pipes with the third pattern. This approximated solution can be improved with another different optimization algorithm, for example evaluating the nearest integer solutions or using a more robust genetic algorithm. But even if we stop with the first step and we round the solution we have a good reduction of waste.

It is important to note that if we solve a problem with a different request, asking to obtain "at least" q_j, the constraints should be written as

$$subject\ to \sum_{i=1}^{n} a_{ij}x_i \geq q_j \quad with\ j = 1, \dots, m$$

and the Scilab script that takes into account inequality constraints results as follows:

```
//pattern
aij=[ 1 1 0;
      1 0 1;
      0 2 2];
//costs
ci=[4; 3; 1];
//request
qj=[150; 200; 300];

xopt = karmarkar([],[],ci,[],[],[],[],[],-aij',-qj,[0,0,0]',[])
```

The solution in this case becomes xopt=[0,150,100] with a total cost of 550. It is amusing to note that, with the problem formulated in this manner, we are able to reduce the total cost that was previously equal to 562.5, even increasing the total number of cuts[4].

9. Conclusions

As the solution of the cutting stock problem demonstrates, Scilab is not an educational tool but a product for solving real industrial problem. The cutting stock problem is a common issue in industries, and a good solution can result in remarkable savings. By the way, in this article we presented only a small subset of possibilities that a Scilab user can have for solving real-world problems. For further information on Scilab optimization algorithms and methodologies see [6-9].

For sake of simplicity, this paper shows only very trivial functions that have been used for the purpose of making a short and general tutorial. Obviously these simple functions can be substituted by more complex and time consuming ones such as FEM solvers or external simulation codes.

MATLAB users have probably recognized the similarities between the commercial software and Scilab. We hope, all other readers have not been scared from the fact that problems and methods should be written down as scripts. Once the logic is clear, writing down scripts can result in an agile and exciting activity.

[4] Thanks to Andreas Vlahinos for reporting this solution.

10. References

[1] Nelder, John A.; R. Mead (1965). "A simplex method for function minimization". Computer Journal 7: 308–313

[2] Scilab Optimization Datasheet, http://wiki.scilab.org/Scilab%20Optimization%20Datasheet

[3] David E. Goldberg. Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley, 1989

[4] Deb, K., Pratap. A, Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Transaction on Evolutionary Computation, 6(2), 181-197

[5] Narendra Karmarkar (1984). "A New Polynomial Time Algorithm for Linear Programming", Combinatorica, Vol 4, nr. 4, p. 373–395

[6] Baudin, Couvert, Steer. "Optimization In Scilab", Digiteo, INRIA

[7] Baudin M. "Nelder mead user's manual", Digiteo

[8] Baudin M., Steer S. "Optimization with Scilab, present and future", in Proceedings of 2009 International Workshop On Open-Source Software For Scientific Computation (Ossc-2009)

[9] Baudin M. "Introduction to optimization in Scilab", Consortium Scilab - Digiteo