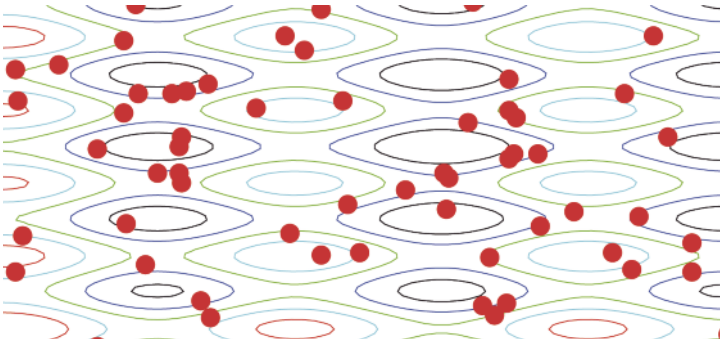
Scilab Datasheet

Optimization in Scilab

Scilab provides a high-level matrix language and allows to define complex mathematical models and to easily connect to existing libraries. That is why optimization is an important and practical topic in Scilab, which provides tools to solve linear and nonlinear optimization problems by a large collection of tools.



Overview of the industrial-grade solvers available in Scilab and the type of optimization problems which can be solved by Scilab.

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`

For the constraint columns, the letter "l" means linear, the letter "n" means nonlinear and "l*" means linear constraints in spectral sense.
For the problem size column, the letters "s", "m" and "l" respectively mean small, medium and large.

Focus on nonlinear optimization

The **optim** function solves optimization problems with nonlinear objectives, with or without bound constraints on the unknowns. The quasi-Newton method **optim/"qn"** uses a Broyden-Fletcher-Goldfarb-Shanno formula to update the approximate Hessian matrix. The quasi-Newton method has a $O(n^2)$ memory requirement. The limited memory BFGS algorithm **optim/"gc"** is efficient for large size problems due to its memory requirement in O(n). Finally, the **optim/"nd"** algorithm is a bundle method which may be used to solve unconstrained, non-differentiable problems. For all these solvers, a function that computes the gradient g must be provided. That gradient can be computed using finite differences based on an optimal step with the **derivative** function, for example.

The **fminsearch** function is based on the simplex algorithm of Nelder and Mead (not to be confused with Dantzig's simplex for linear optimization). This unconstrained algorithm does not require the gradient of the cost function. It is efficient for small problems, i.e. up to 10 parameters and its memory requirement is only $O(n^2)$. It generally requires only 1 or 2 function evaluations per iteration. This algorithm is known to be able to manage "noisy" functions, i.e. situations where the cost function is the sum of a general nonlinear function and a low magnitude noise function. The **neldermead** component provides three simplex-based algorithms which allow to solve unconstrained and nonlinearly constrained optimization problems. It provides an object oriented access to the options. The **fminsearch** function is, in fact, a specialized use of the **neldermead** component.

The flagship of Scilab is certainly the **optim** function, which provides a set of 5 algorithms for nonlinear unconstrained (and bound constrained) optimization problems.

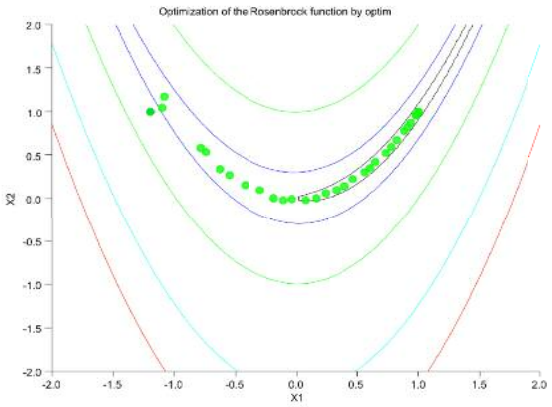


Figure 1: Optimization of the Rosenbrock function by the **optim** function.

The **optim** function is an unconstrained (or bound constrained) nonlinear optimization solver. The calling sequence is:

```
fopt = optim(costf,x0)
fopt = optim(costf,"b",lb,ub,x0)
fopt = optim(costf,"b",lb,ub,x0,algo)
[fopt,xopt] = optim(...)
[fopt,xopt,gopt] = optim(...)
```

where

- f is the objective function,
- x0 is the initial guess,
- lb is the lower bound,
- ub is the upper bound,
- algo is the algorithm,
- fopt is the minimum function value,
- xopt is the optimal point,
- gopt is the optimal gradient.

The **optim** function allows to use 3 different algorithms:

- algo = "qn": Quasi-Newton (the default solver) based on BFGS formula,
- algo = "gc": Limited Memory BFGS algorithm for large-scale optimization,
- algo = "nd": Bundle method for non-differentiable problems (e.g. min-max).

Features:

- Provides efficient optimization solvers based on robust algorithms.
- Objective function f in Scilab macros or external (dynamic link).
- Extra parameters can be passed to the objective function (with a list or array).
- Robust implementation:
  - Quasi-Newton based on the update of the Cholesky factors,
  - Line-Search of **optim/"qn"** based on a safeguarded cubic interpolation designed by Lemaréchal.

In the following script, we compute the unconstrained optimum of the Rosenbrock function:

```
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 1.0];
[ fopt , xopt ] = optim ( rosenbrock , x0 )
```

The previous script produces the following output:

```
-->[ fopt , xopt ] = optim ( rosenbrock , x0 )
 xopt  =
    1.    1.
 fopt  =
    0.
```

Computing the derivatives by finite differences

Scilab provides the **derivative** function which computes approximate gradients based on finite differences. The calling sequence is:

```
g = derivative(f,x)
g = derivative(f,x,h)
g = derivative(f,x,h,order)
[g,H] = derivative(...)
```

where

- f is the objective function,
- x is the point where to evaluate the gradient,
- h is the step,
- order is the order of the finite difference formula,
- g is the gradient (or Jacobian matrix),
- H is the Hessian matrix.

Features:

- Uses optimal step (manages limited precision of floating point numbers),
- Can handle any type of objective function: macro or external program or library,
- Provides order 1, 2 or 4 formulas.

In the following script, we compute the optimum of the Rosenbrock problem with finite differences:

```
function f=rosenbrock(x)
   f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2
endfunction
function [f, g, ind]=rosenbrockCost2(x, ind)
   f = rosenbrockF(x)
   g = derivative(rosenbrockF, x', order=4)
endfunction
```

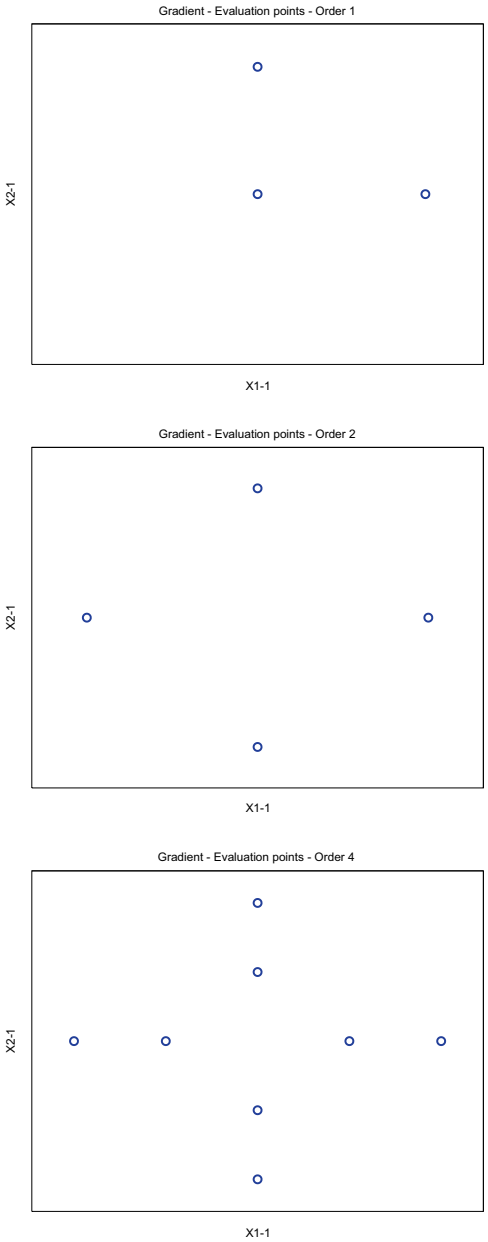


Figure 2: Pattern of the order 1, 2 and 4 finite differences formulas in the **derivative** function.

Linear and Quadratic Optimization

The Quapro module defines linear quadratic programming solvers. The matrices defining the cost and constraints must be full, but the quadratic term matrix is not required to be full rank.

Features:

- **linpro**: Linear programming solver,
- **quapro**: Linear quadratic programming solver,
- **mps2linpro**: Convert lp problem given in MPS format to linpro format.

The Quapro module is available through ATOMS:
http://atoms.scilab.org/toolboxes/quapro
To install the Quapro module:

```
atomsInstall("quapro");
```

and then re-start Scilab.

The **linpro** function can solve linear programs in general form:

```
Minimize c'*x
A*x   <= b
Aeq*x  = beq
lb <= x <= ub
```

The following example is extracted from "Operations Research: applications and algorithms", Wayne L. Winstons, Section 5.2, "The Computer and Sensitivity Analysis", in the "Degeneracy and Sensitivity Analysis" subsection. We consider the problem:

```
Min -6*x1 - 4*x2 - 3*x3 - 2*x4
such that:
2*x1 + 3*x2 + x3 + 2* x4 <= 400
  x1 +   x2 + 2*x3 +     x4 <= 150
2*x1 +   x2 +   x3 + 0.5*x4 <= 200
3*x1 +   x2 +             x4 <= 250;
x >= 0;
```

The following script allows to solve the problem:

```
c = [-6 -4 -3 -2]';
A = [
     2 3 1 2
     1 1 2 1
     2 1 1 0.5
     3 1 0 1
     ];
b = [400 150 200 250]';
ci=[0 0 0 0]';
cs=[%inf %inf %inf %inf]';
[xopt,lagr,fopt]=linpro(c,A,b,ci,cs)
```

This produces:

```
xopt = [50,100,2.842D-14,0]
```

Nonlinear Least Squares

Scilab provides 2 solvers for nonlinear least squares:

- **lsqrsolve**: Solves nonlinear least squares problems, Levenberg-Marquardt algorithm,
- **leastsq**: Solves nonlinear least squares problems (built over optim).

Features:

- Can handle any type of objective function, macro or external program or library,
- The gradient is optional.

In the following example, we are searching for the parameters of a system of ordinary differential equations which best fit experimental data. The context is a chemical reaction for processing waters with phenolic compounds.

We use the **lsqrsolve** function in a practical case:

```
function dy = myModel(t,y,a,b)
 // The right-hand side of the
 // Ordinary Differential Equation.
 dy(1) = -a*y(2)+y(1)+t^2+6*t+b
 dy(2) = b*y(1)-a*y(2)+4*t+(a+b)*(1-t^2)
endfunction

function f = myDifferences(x,t,yexp)
 // Returns the difference between the
 // simulated differential
 // equation and the experimental data.
 a = x(1)
 b = x(2)
 y0 = yexp(1,:)
 t0 = 0
 y_calc=ode(y0',t0,t,list(myModel,a,b))
 diffmat = y_calc' - yexp
 // Maxe a column vector
 f = diffmat(:)
endfunction

function f = myAdapter(x,m,t,yexp)
 // Adapts the header for lsqrsolve
 f = myDifferences(x,t,yexp)
endfunction

// 1. Experimental data
t = [0 1 2 3 4 5 6]';
yexp(:,1) = [-1 2 11 26 47 74 107]';
yexp(:,2) = [ 1 3 09 19 33 51 73]';

// 2. Optimize
x0 = [0.1;0.4];
y0 = myDifferences(x0,t,yexp);
m = size(y0,"*");
objfun = list(myAdapter,t,yexp);
[xopt,diffopt]=lsqrsolve(x0,objfun,m)
```

The previous script produces the following output:

```
-->[xopt,diffopt]=lsqrsolve(x0,objfun,m)
lsqrsolve: relative error between two consecutive
iterates is at most xtol.
```

```
diffopt  =
   0.
 - 2.195D-08
 - 4.880D-08
 - 8.743D-09
   2.539D-08
   7.487D-09
 - 2.196D-08
   0.
   6.534D-08
 - 6.167D-08
 - 7.148D-08
 - 6.018D-09
   2.043D-08
   1.671D-08
xopt  =
   2.
   3.
```

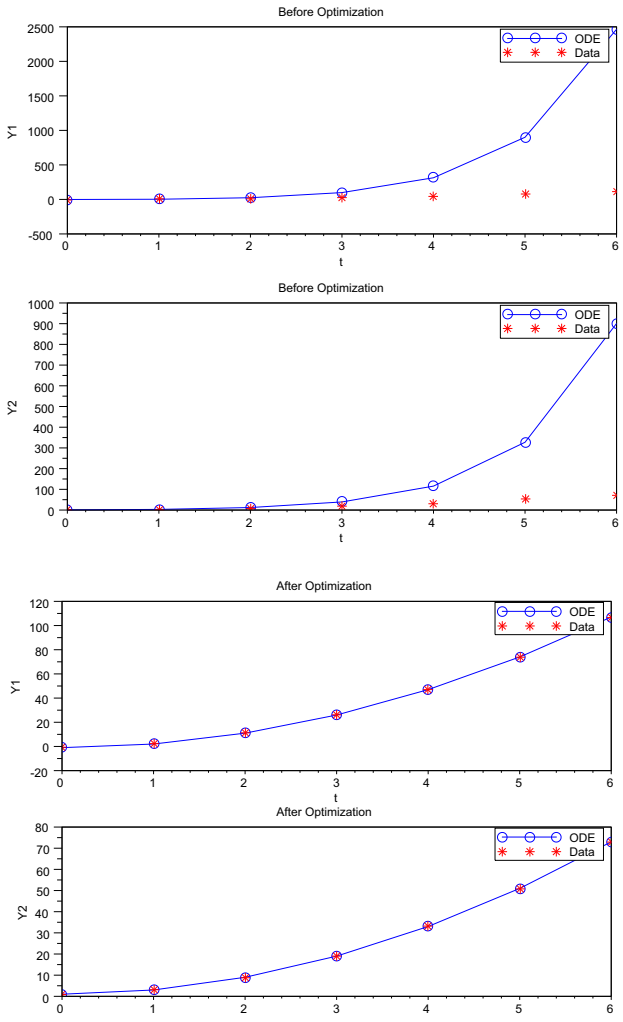


Figure 3: Searching for the best parameters fitting experiments data associated with a set of 2 Ordinary Differential Equations.

Genetic Algorithms

The genetic algorithms module in Scilab provides the following functions:

- **optim_ga**: A flexible genetic algorithm,
- **optim_moga**: A multi-objective genetic algorithm,
- **optim_nsga**: A multi-objective Niched Sharing Genetic Algorithm,
- **optim_nsga2**: A multi-objective Niched Sharing Genetic Algorithm version 2.

The following example is the minimization of the Rastrigin function defined by $y = x_1^2 + x_2^2 - \cos(12\,x_1) - \cos(18\,x_2)$. This function has several local minima, but only one global minimum, that is $(x_1^*, x_2^*) = (0, 0)$ associated with the value $f(x_1^*, x_2^*) = -2$. We use a binary encoding of the input variables, which performs better in this case:

```
// 1. Define the Rastrigin function.
function y = rastriginV(x1,x2)
 // Vectorized function for contouring
 y = x1.^2 + x2.^2-cos(12*x1)-cos(18*x2)
endfunction

function y = rastrigin(x)
 // Non-vectorized function for optimization
 y = rastriginV(x(1),x(2))
endfunction

function y = rastriginBinary(x)
 BinLen = 8
 lb = [-1;-1];
 ub = [1;1];
 tmp = convert_to_float(x,BinLen,ub,lb)
 y = rastrigin (tmp)
endfunction

// 2. Compute the optimum.
PopSize = 100;
Proba_cross = 0.7;
Proba_mut = 0.1;
NbGen = 10;
Log = %T;
gaprms=init_param();
gaprms=add_param(gaprms,"minbound",[-1;-1]);
gaprms=add_param(gaprms,"maxbound",[1;1]);
gaprms=add_param(gaprms,"dimension",2);
gaprms=add_param(gaprms,"binary_length",8);
gaprms=add_param(gaprms,"crossover_func",..
 crossover_ga_binary);
```

```
gaprms=add_param(gaprms,"mutation_func",..
 mutation_ga_binary);
gaprms=add_param(gaprms,"codage_func",..
 coding_ga_binary);
gaprms=add_param(gaprms,"multi_cross",%T);
[xpopopt,fpopopt,xpop0,fpop0] = ..
 optim_ga(rastriginBinary,PopSize,..
 NbGen,Proba_mut,Proba_cross,Log,gaprms);
```

The previous script produces the following output:

```
-->[pop_opt,fobj_pop_opt,pop_init,fobj_pop_init] =
optim_ga(rastriginBinary, ..
-->   PopSize, NbGen, Proba_mut, Proba_cross, Log,
ga_params);
Initialization of the population
Iter. 1 - min/max value = -1.942974/0.085538
Iter. 2 - min/max value = -1.942974/-0.492852
Iter. 3 - min/max value = -1.942974/-0.753347
Iter. 4 - min/max value = -1.942974/-0.841115
Iter. 5 - min/max value = -1.942974/-0.985001
Iter. 6 - min/max value = -1.942974/-1.094454
Iter. 7 - min/max value = -1.942974/-1.170877
Iter. 8 - min/max value = -1.987407/-1.255388
Iter. 9 - min/max value = -1.987407/-1.333186
Iter. 10 - min/max value = -1.987407/-1.450980
```

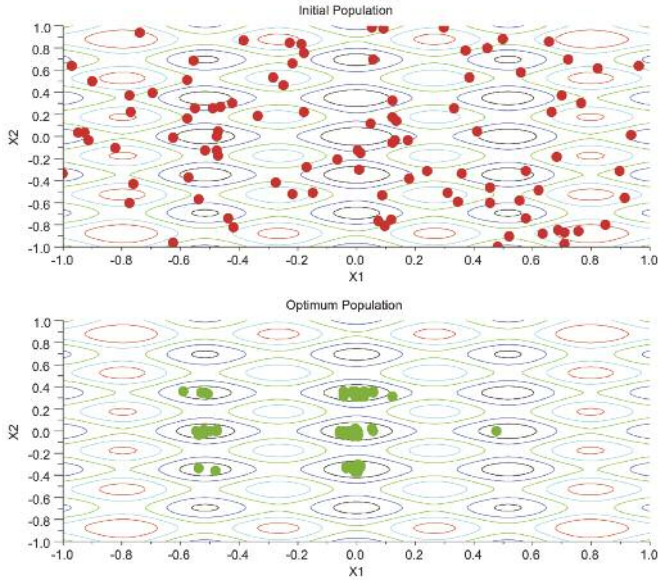


Figure 4: Optimization of the Rastrigin function by the `optim_ga` function.

Open-Source libraries

- **`optim`** is based on Modulopt, a collection of optimization solvers.
  http://www-rocq.inria.fr/~gilbert/modulopt/
- **`lsqrsolve`** is based on Minpack, a Fortran 77 code for solving nonlinear equations and nonlinear least squares problems.
  http://www.netlib.org/minpack/
- Quapro: Eduardo Casas Renteria, Cecilia Pola Mendez (Universidad De Cantabria), improved by Serge Steer (INRIA) and maintained by Allan Cornet, Michaël Baudin (Consortium Scilab - Digiteo).
- **`qld`**: Designed by M.J.D. Powell (1983) and modied by K. Schittkowski, with minor modications by A. Tits and J.L. Zhou.
- **`qp_solve`**, **`qpsolve`**: The Goldfarb-Idnani algorithm and the Quadprog package developed by Berwin A. Turlach.
- **`linpro`**, **`quapro`**: The algorithm developed by Eduardo Casas Renteria and Cecilia Pola Mendez.

- Thanks to Marcio Barbalho for providing the material of the Non Linear Least Squares example.

- All scripts are available on Scilab wiki at: http://wiki.scilab.org/Scilab%20Optimization%20Datasheet