AN ALGORITHM FOR NONLINEAR LEAST SQUARES

M. Balda

Institute of Thermomechanics, Academy of Sciences of the Czech Republic, v. v. i.

Abstract

Optimization Toolbox of MATLAB represents very mighty apparatus for solution of wide set of optimization problems. Also basic MATLAB provides means for optimization purposes, e.g. backslash operator for solving set of linear equations or the function fminsearch for nonlinear problems. Should the set of equations be nonlinear, an application of fminsearch for finding the least squares solution would be inefficient. The paper describes a better algorithm for the given task.

1 Principles of Levenberg-Marquardt-Fletcher algorithm

Let us have a general overdetermined system of nonlinear algebraic equations

$$\boldsymbol{f}(\boldsymbol{x},\boldsymbol{c}) \doteq \boldsymbol{y} \qquad \Rightarrow \qquad \boldsymbol{f}(\boldsymbol{x},\boldsymbol{c}) - \boldsymbol{y} = \boldsymbol{r}\,. \tag{1}$$

Its solution, optimal in the least squares sense, is sought by minimizing $\|\boldsymbol{r}\|_2^2 = \boldsymbol{r}^T\boldsymbol{r}$. Necessary conditions for the optimum solution are zero values of partial derivatives of $\|\boldsymbol{r}\|_2^2$ due to unknown coefficients $\boldsymbol{c}$, i.e.

$$\frac{\partial\|\boldsymbol{r}\|_2^2}{\partial\boldsymbol{c}} = 2\,\frac{\partial\boldsymbol{r}^T}{\partial\boldsymbol{c}}\,\boldsymbol{r} = 2\,\boldsymbol{J}^T\boldsymbol{r} = 2\boldsymbol{v}\,. \tag{2}$$

Elements of $\boldsymbol{J}$, which is called Jacobian matrix, are $J_{ij} = \frac{\partial r_i}{\partial c_j}$. Vector $\boldsymbol{v}$ should equal zero vector in the point of optimal solution $\boldsymbol{c}^*$. It is sought after kth iteration in the form

$$\boldsymbol{c}^{(k+1)} = \boldsymbol{c}^{(k)} + \Delta\boldsymbol{c}^{(k)}\,. \tag{3}$$

Let residuals $\boldsymbol{r}(\boldsymbol{c})$ are smooth functions, then it holds:

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} + \frac{\partial\boldsymbol{r}^{(k)}}{\partial\boldsymbol{c}^{(k)}}\,\Delta\boldsymbol{c}^{(k)} + \cdots\,. \tag{4}$$

After some manipulations, the equation for the solution increment takes the form

$$\boldsymbol{A}^{(k)}\,\Delta\boldsymbol{c}^{(k)} - \boldsymbol{J}^{(k)^T}\boldsymbol{r}^{(k+1)} = -\boldsymbol{v}^{(k)}\,, \tag{5}$$

in which $\boldsymbol{A} = \boldsymbol{J}^T\boldsymbol{J}$. The solution would be found quite easily, should $\boldsymbol{r}^{(k+1)}$ be known. Unfortunately, this is not true. It was the reason why Levenberg substituted the second term in equation (**??**), $-\boldsymbol{J}^{(k)^T}\boldsymbol{r}^{(k+1)}$, by $\lambda\,\Delta\boldsymbol{c}^{(k)}$. The scalar λ serves for scaling purposes. For $\lambda = 0$, the method transforms into fast Newton method, which may diverge, while for $\lambda \to \infty$, the method approaches the stable steepest descent method.

Later, Marquardt changed λ into $\lambda^{(k)}$, so that the equation for $\Delta\boldsymbol{c}^{(k)}$ became

$$(\boldsymbol{A}^{(k)} + \lambda^{(k)}\,\boldsymbol{I})\,\Delta\boldsymbol{c}^{(k)} = -\boldsymbol{v}^{(k)}\,. \tag{6}$$

Values of $\lambda^{(k+1)}$ are varying in dependence on behavior of iteration process. For slow stable convergence of iterations, the new value $\lambda^{(k+1)} = \lambda^{(k)}/\nu$ is set, which accelerates the process. Should a sign of divergence be observed, the value is changed into $\lambda^{(k+1)} = \lambda^{(k)}\,\nu$. The usual value of ν is between 2 and 10.

---

prof. Ing. Miroslav Balda, DrSc.
Veleslavínova 11, 301 00 Plzeň, phone: +420 377 236 415, e-mail: balda@cdm.it.cas.cz

Fletcher improved Marquardt strategy of λ adaptation significantly. He substituted the unity matrix $\boldsymbol{I}$ by a diagonal matrix $\boldsymbol{D}$ of scales in the formula (**??**). More to it, he introduced new quotient R which expresses how forecasted sum of squares agrees with the real one in the current iteration step. If R falls between preset limits $(R_{\mathrm{lo}}, R_{\mathrm{hi}})$, parameters of iteration do not change, otherwise changes of λ and ν follow. Value of λ is halved if $R > R_{\mathrm{hi}}$. Provided λ becomes lower than a critical value λ_c, it is cleared which causes the next iteration proceeds like in the Newton method. If $R < R_{\mathrm{lo}}$, parameter ν is set so that it holds $2 \leq \nu \leq 10$, and if λ were zero, a modification of λ_c and λ follows.

2 Function LMFsolve

Fletcher built his algorithm in FORTRAN and presented it in the report [**?**] more then 35 years ago. It was rewritten verbatim in the Central Research Institute ŠKODA and then used for many least square issues in dynamics, processing of experimental data, identification, etc. A code of the function has been later recasted into MATLAB with slight modifications at the end of eighties of the past century. The complete reconstruction of the function named LMFsolve appeared in the recent time (see [**?**]).

```
function [xf, S, cnt] = LMFsolve(varargin)
% Solve a Set of Overdetermined Nonlinear Equations in Least-Squares Sense.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% A solution is obtained by a Fletcher version of the Levenberg-Maquardt
% algoritm for minimization of a sum of squares of equation residuals.
%
% [Xf, Ssq, CNT] = LMFsolve(FUN,Xo,Options)
% FUN     is a function handle or a function M-file name that evaluates
%         m-vector of equation residuals,
% Xo      is n-vector of initial guesses of solution,
% Options is an optional set of Name/Value pairs of control parameters
%         of the algorithm. It may be also preset by calling:
%         Options = LMFsolve('default'), or by a set of Name/Value pairs:
%         Options = LMFsolve('Name',Value, ... ), or updating the Options
%                   set by calling
%         Options = LMFsolve(Options,'Name',Value, ...).
%
%    Name   Values {default}         Description
% 'Display'     integer     Display iteration information
%                            {0}  no display
%                             k   display initial and every k-th iteration;
% 'Jacobian'    handle      Jacobian matrix function handle; {@finjac}
% 'FunTol'      {1e-4}      norm(FUN(x),1) stopping tolerance;
% 'XTol'        {1e-7}      norm(x-xold,1) stopping tolerance;
% 'MaxIter'     {100}       Maximum number of iterations;
% 'ScaleD'                  Scale control:
%               value        D = eye(m)*value;
%               vector       D = diag(vector);
%                {[]}        D(k,k) = JJ(k,k) for JJ(k,k)>0, or
%                                   = 1 otherwise,
%                                     where JJ = J.'*J
% Not defined fields of the Options structure are filled by default values.
%
% Output Arguments:
% Xf        final solution approximation
% Ssq       sum of squares of residuals
% Cnt       >0          count of iterations
%           -MaxIter,   did not converge in MaxIter iterations

% Example:
% The general Rosenbrock's function has the form
%    f(x) = 100(x(1)-x(2)^2)^2 + (1-x(1))^2
% Optimum solution gives f(x)=0 for x(1)=x(2)=1. Function f(x) can be
% expressed in the form
%    f(x) = f1(x)^2 =f2(x)^2,
% where f1(x) = 10(x(1)-x(2)^2) and f2(x) = 1-x(1).
% Values of the functions f1(x) and f2(x) can be used as residuals.
% LMFsolve finds the solution of this problem in 5 iterations. The more
% complicated problem sounds:
% Find the least squares solution of the Rosenbrock valey inside a circle
```

```
% of the unit diameter centered at the origin. It is necessary to build
% third function, which is zero inside the circle and increasing outside it.
% This property has, say, the next function:
%    f3(x) = sqrt(x(1)^2 + x(2)^2) - r, where r is a radius of the circle.
% Its implementation using anonymous functions has the form
%    R  = @(x) sqrt(x'*x)-.5;    %   A distance from the radius r=0.5
%    ros= @(x) [10*(x(2)-x(1)^2); 1-x(1); (R(x)>0)*R(x)*1000];
%    [x,ssq,cnt]=LMFsolve(ros,[-1.2,1],'Display',1,'MaxIter',50)
% Solution: x = [0.4556; 0.2059],  |x| = 0.5000
% sum of squares: ssq = 0.2966,
% number of iterations: cnt = 18.
% Note:
% Users with older MATLAB versions, which have no anonymous functions
% implemented, have to call LMFsolve with named function for residuals.
% For above example it is
%
%   [x,ssq,cnt]=LMFsolve('rosen',[-1.2,1]);
%
% where the function rosen.m is of the form
%
%   function r = rosen(x)
%%  Rosenbrock valey with a constraint
%   R = sqrt(x(1)^2+x(2)^2)-.5;
%%  Residuals:
%   r = [ 10*(x(2)-x(1)^2)  %   first part
%         1-x(1)            %   second part
%         (R>0)*R*1000.     %   penalty
%       ];

% Reference:
% Fletcher, R., (1971): A Modified Marquardt Subroutine for Nonlinear Least
% Squares. Rpt. AERE-R 6799, Harwell

% M. Balda,
% Institute of Thermomechanics,
% Academy of Sciences of The Czech Republic,
% balda AT cdm DOT cas DOT cz
% 2007-07-02
% 2007-10-08    formal changes, improved description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   OPTIONS
    %%%%%%%
%               Default Options
if nargin==1 && strcmpi('default',varargin(1))
   xf.Display  = 0;         %   no print of iterations
   xf.Jacobian = @finjac;   %   finite difference Jacobian approximation
   xf.MaxIter  = 100;       %   maximum number of iterations allowed
   xf.ScaleD   = [];        %   automatic scaling by D = diag(diag(J'*J))
   xf.FunTol   = 1e-7;      %   tolerace for final function value
   xf.XTol     = 1e-4;      %   tolerance on difference of x-solutions
   return

%               Updating Options
elseif isstruct(varargin{1}) % Options=LMFsolve(Options,'Name','Value',...)
    if ~isfield(varargin{1},'Jacobian')
        error('Options Structure not Correct for LMFsolve.')
    end
    xf=varargin{1};          %   Options
    for i=2:2:nargin-1
        name=varargin{i};     %   option to be updated
        if ~ischar(name)
            error('Parameter Names Must be Strings.')
        end
        name=lower(name(isletter(name)));
        value=varargin{i+1};  %   value of the option
        if strncmp(name,'d',1), xf.Display  = value;
        elseif strncmp(name,'f',1), xf.FunTol   = value(1);
        elseif strncmp(name,'x',1), xf.XTol     = value(1);
        elseif strncmp(name,'j',1), xf.Jacobian = value;
        elseif strncmp(name,'m',1), xf.MaxIter  = value(1);
        elseif strncmp(name,'s',1), xf.ScaleD   = value;
        else   disp(['Unknown Parameter Name --> ' name])
        end
    end
    return
```

```
%               Pairs of Options
elseif ischar(varargin{1})  % check for Options=LMFSOLVE('Name',Value,...)
   Pnames=char('display','funtol','xtol','jacobian','maxiter','scaled');
   if strncmpi(varargin{1},Pnames,length(varargin{1}))
      xf=LMFsolve('default');  % get default values
      xf=LMFsolve(xf,varargin{:});
      return
   end
end

%   LMFSOLVE(FUN,Xo,Options)
    %%%%%%%%%%%%%%%%%%%%%%%%

FUN=varargin{1};            %   function handle
if ~(isvarname(FUN) || isa(FUN,'function_handle'))
   error('FUN Must be a Function Handle or M-file Name.')
end

xc=varargin{2};             %   Xo

if nargin>2                 %   OPTIONS
    if isstruct(varargin{3})
        options=varargin{3};
    else
        if ~exist('options','var')
            options = LMFsolve('default');
        end
        for i=3:2:size(varargin,2)-1
            options=LMFsolve(options, varargin{i},varargin{i+1});
        end
    end
else
    if ~exist('options','var')
        options = LMFsolve('default');
    end
end

x=xc(:);
lx=length(x);

r=feval(FUN,x);             % Residuals at starting point
%~~~~~~~~~~~~~~
S=r'*r;
epsx=options.XTol(:);
epsf=options.FunTol(:);
if length(epsx)<lx, epsx=epsx*ones(lx,1); end
J=options.Jacobian(FUN,r,x,epsx);
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
A=J.'*J;                    % System matrix
v=J.'*r;

D = options.ScaleD;
if isempty(D)
    D=diag(diag(A));        % automatic scaling
    for i=1:lx
        if D(i,i)==0, D(i,i)=1; end
    end
else
    if numel(D)>1
        D=diag(sqrt(abs(D(1:lx)))); % vector of individual scaling
    else
        D=sqrt(abs(D))*eye(lx);     % scalar of unique scaling
    end
end

Rlo=0.25; Rhi=0.75;
l=1;      lc=.75;     is=0;
cnt=0;
ipr=options.Display;
printit(-1);                %   Table header
d=options.XTol;             %   vector for the first cycle
maxit = options.MaxIter;    %   maximum permitted number of iterations

while cnt<maxit && ...      %   MAIN ITERATION CYCLE
    any(abs(d)>=epsx) && ...    %%%%%%%%%%%%%%%%%%%%%%
```

```
    any(abs(r)>=epsf)
    d=(A+l*D)\v;            % negative solution increment
    xd=x-d;
    rd=feval(FUN,xd);
%   ~~~~~~~~~~~~~~~~
    Sd=rd.'*rd;
    dS=d.'*(2*v-A*d);       % predicted reduction
    R=(S-Sd)/dS;

    if R>Rhi
        l=l/2;
        if l<lc, l=0; end
    elseif R<Rlo
        nu=(Sd-S)/(d.'*v)+2;
        if nu<2
            nu=2;
        elseif nu>10
            nu=10;
        end
        if l==0
            lc=1/max(abs(diag(inv(A))));
            l=lc;
            nu=nu/2;
        end
        l=nu*l;
    end
    cnt=cnt+1;
    if ipr>0 && (rem(cnt,ipr)==0 || cnt==1)
        printit(cnt,[S,l,R,x(:).'])
        printit(0,[lc,d(:).'])
    end
    if Sd>S && is==0
        is=1;
        St=S; xt=x; rt=r; Jt=J; At=A; vt=v;
    end
    if Sd<S || is>0
        S=Sd; x=xd; r=rd;
        J=options.Jacobian(FUN,r,x,epsx);
%       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        A=J.'*J;   v=J.'*r;
    else
        S=St; x=xt; r=rt; J=Jt; A=At; v=vt;
    end
    if Sd<S, is=0; end
end
xf = x;                         %   finat solution
if cnt==maxit, cnt=-cnt; end    %   maxit reached
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%   PRINTIT     Printing of intermediate results
%   %%%%%%%
function printit(cnt,mx)
%        ~~~~~~~~~~~~~~
%  cnt =  -1  print out the header
%           0  print out second row of results
%          >0  print out first row of results

if ipr>0
   if cnt<0                 %   table header
      disp('')
      disp(char('*'*ones(1,100)))
      disp(['  cnt   SUM(r^2)        l            R' blanks(21) 'x(i) ...'])
      disp([blanks(24) 'lc' blanks(32) 'dx(i) ...'])
      disp(char('*'*ones(1,100)))
      disp('')
   else                     %   iteration output
      if cnt>0 || rem(cnt,ipr)==0
          f='%12.4e ';
          form=[f f f f '\n' blanks(49)];
          if cnt>0
              fprintf(['%4.0f ' f f f ' x = '],[cnt,mx(1:3)])
              fprintf(form,mx(4:length(mx)))
          else
              fprintf([blanks(18) f blanks(13) 'dx = '], mx(1))
              fprintf(form,mx(2:length(mx)))
          end
```

```
          fprintf('\n')
       end
    end
end
end

%   FINJAC       numerical approximation to Jacobi matrix
%   %%%%%%
function J = finjac(FUN,r,x,epsx)
%~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
lx=length(x);
J=zeros(length(r),lx);
for k=1:lx
    dx=.25*epsx(k);
    xd=x;
    xd(k)=xd(k)+dx;
    rd=feval(FUN,xd);
%   ~~~~~~~~~~~~~~~~
    J(:,k)=((rd-r)/dx);
end
end
end
```

The main part of the function pursues three tasks. At first, it initiates optional parameters to default or user defined values. At second, initial conditions for the iteration process are determined as sum of squares of residuals corresponding to an initial guess of solution, and Jacobian matrix at that point. The last is an iteration loop executing the Levenberg-Marquardt algorithm in Fletcher's modification.

There are two subfunctions included in *LMFsolve.m*, i.e. *Printit* and *finjac*. The subfunction *Printit* displays intermediate results in each *ipr*th iteration. A non-negative value of *ipr* is set by a user as one of options. The output contains values of an iteration order k, sum of squares of residuals, λ and λ_c, quotient R, solution $c^{(k)}$ in selected iterations, and increments $\Delta c^{(k)}$.

A user has a possibility to control performance of the function by means of parameters, values of which may be set either as default or by calling *LMFsolve*. Default values of the parameters are set automatically if *LMFsolve* is called in the simplest form

```
[x,ssq,cnt] = LMFsolve(FUN,x0)
```

The names and default values of the parameters are as follows:

`'Display'`	`0`	do not display any intermediate results
`'Jacobian'`	`@finjac`	use subfunction *finjac* for evaluation of $\boldsymbol{J}$
`'MaxIter'`	`100`	maximum permitted number of iterations
`'Xtol'`	`1e-4`	tolerance for increments in each unknown
`'Ftol'`	`1e-7`	tolerance for sum of squares
`'ScaleD'`	`[]`	user's control of diagonal matrix of scales $\boldsymbol{D}$

3 Examples

The classical example for testing optimization algorithms is the Rosenbrock's function sometimes called "banana valley":

$$f(\boldsymbol{x}) = 100\,(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{7}$$

It is obvious that the function $f(\boldsymbol{x})$ is a sum of squares of two terms, $10\,(x_2 - x_1^2)$ and $1 - x_1$. In consequence of it, any function for minimization of a sum of squares can be used for getting a solution. The function *LMFsolve* has been tested on three cases of the Rosenbrock's function, the standard one under eqn (**??**), the same constrained by a circle of a unit diameter at the origin, and the similar case with the radius equal $\sqrt{1.5}$. The later problem is also solved in the Optimization Toolbox for demonstrating quadratic programming.

1. Rosenbrock's function.
   The original function is transformed into the least squares problem with residuals

$$f_1(\boldsymbol{x}) = 10\,(x_2 - x_1^2) \tag{8}$$
$$f_2(\boldsymbol{x}) = 1 - x_1 \tag{9}$$

   The solution is $\boldsymbol{x} = [1;\, 1]$.

2. Rosenbrock's function with circular constraint of radius $r = 0.5$
   For solutions of constraint problems, a user should define the unfeasible domain in the form of an additional residuum. In this case, the residuum has the general form

$$f_3(\boldsymbol{x}) = w_2 \sqrt{x_1^2 + x_2^2} - r \qquad = d_1 \text{ or} \tag{10}$$
$$f_3(\boldsymbol{x}) = w_3\,(x_1^2 + x_2^2 - r^2) \qquad = d_2 \text{ outside the circular domain} \tag{11}$$
$$f_3(\boldsymbol{x}) = 0 \qquad \text{inside} \tag{12}$$

   The residuum d_2 has been chosen in the form corresponding the Optimization Toolbox. The solution is $\boldsymbol{x} = [0.4557;\, 0.2059]$.

3. Rosenbrock's function with circular constraint of radius $r = \sqrt{1.5}$
   Formulae for $f_3(\boldsymbol{x})$ remain the same as in Case 2.

   The solution is $\boldsymbol{x} = [0.9073;\, 0.8228]$.

Parameters and numbers of iterations of testing runs are gathered in the table **??**. Symbols ∞ denote that the iteration process has not converge.

`ScaleD`				0	1	[ ]
$\boldsymbol{D}$				$\boldsymbol{O}$	$\boldsymbol{I}$	diag(diag($\boldsymbol{A}$))
Case	radius r	weight w	$f_3(\boldsymbol{x})$			
1	-	-	-	2	10	5
2	0.5	100	d_1	13	∞	80
2	0.5	100	d_2	∞	∞	13
3	$\sqrt{1.5}$	10	d_1	10	∞	27
3	$\sqrt{1.5}$	10	d_2	∞	25	57

Table 1: A survey of 5 testing examples

The trace of the iteration process starting in the common point [-1.2; 1] is shown for the case 2 with the penalty d_1 and `ScaleD=0` in the figure **??**. The inner part of the circle is the feasible area.

4 Conclusions

The presented function *LMFsolve* is a transcription of the Fletcher's FORTRAN version of Levenberg-Marquardt algorithm for approximate solution of an overdetermined system of non-linear algebraic equations in the least squares sense in MATLAB. The algorithm has been a bit simplified, what brings problems in extreme situations. They have been documented in solution of constrained Rosenbrock's function, where instabilities occur rather often.

Two kinds of penalties as additional equations have been used. In both cases, these functions have zero values inside a circle, a feasible domain, and are raising with a distance from its border. While the penalty d_1 is growing linearly, d_2 raised quadratically. the method is rather sensitive on the values of the function weight w.

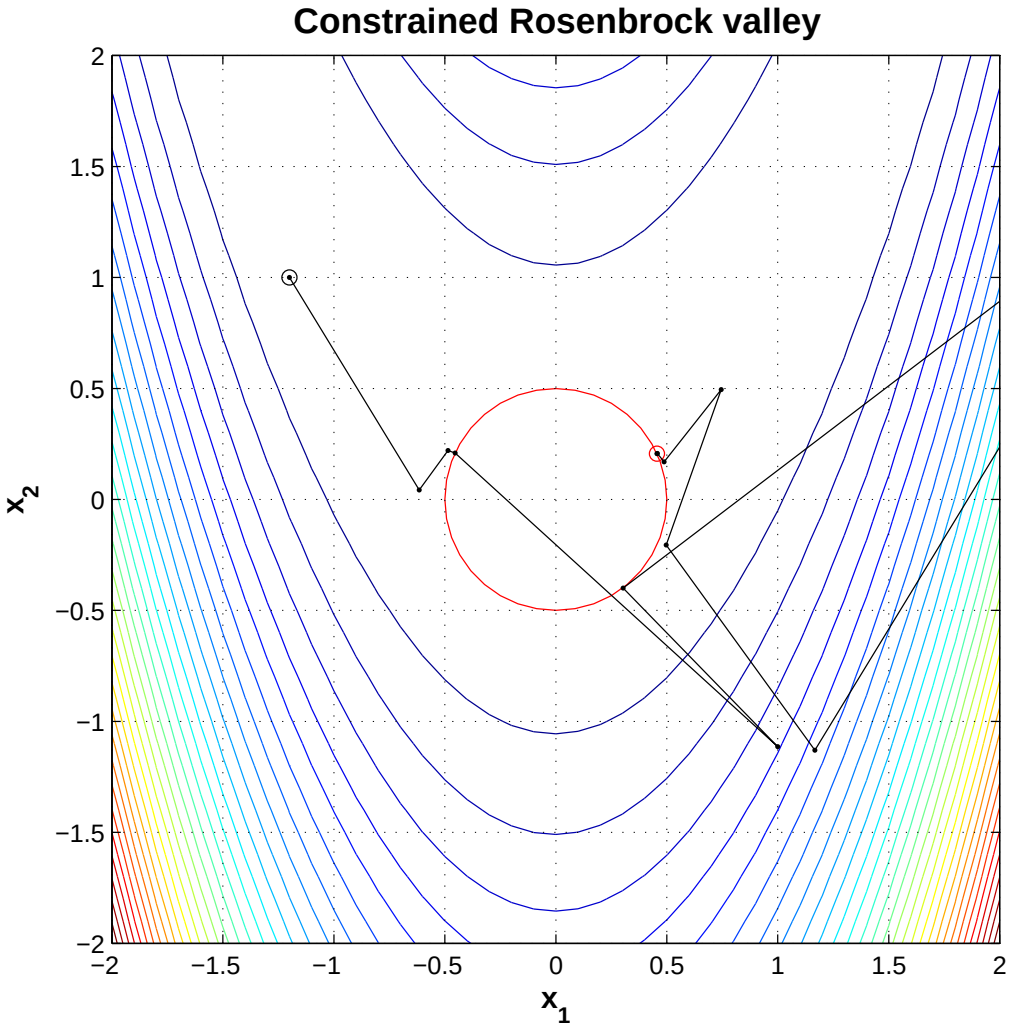

Figure 1: The trace of the iteration process

The function behaves rather good for standard least squares problems occurring in multivariable curve-fitting. This is a frequent task in experimental data processing. One of those issues has been a regression of large sets of fatigue lives of parts caused by multiaxial dynamic loading. The nonlinear regression function possessed six unknown parameters, which were obtained in 18 iterations. Nevertheless, the function *LMFsolve* needs modifications which will improve its reliability. Updated versions of *LMFsolve* will appear in the MathWorks File Exchange at [**?**].

Acknowledgments.

The work has been supported by the grant project 101/05/0199 of the Czech Science Foundation and the project AVOZ 20760514 of the Academy of Sciences of the Czech Republic.

Reference

[1] R. Fletcher (1971): A Modified Marquardt Subroutine for Nonlinear Least Squares. Rpt. AERE-R 6799, Harwell

[2] M. Balda (2007): LMFsolve.m: Levenberg-Marquardt-Fletcher algorithm for nonlinear least squares problems. MathWorks, File Exchange, ID 16063
http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=16063