Abstract

The Semi-Implicit Root solver (SIR) is an iterative method for globally convergent solution of systems of nonlinear equations. Since publication [5], SIR has proven robustness for a great variety of problems. We here present MATLAB and MAPLE codes for SIR, that can be easily implemented in any application where linear or nonlinear systems of equations need be solved efficiently. The codes employ recently developed efficient sparse matrix algorithms and improved numerical differentiation. SIR convergence is quasi-monotonous and approaches second order in the proximity of the real roots. Global convergence is usually superior to that of Newton’s method, being a special case of the method. Furthermore the algorithm cannot land on local minima, as may be the case for Newton’s method with linesearch.

Keywords: Newton method, Jacobian, root solver, equation solver, MATLAB

1. Motivation and significance

Systems of algebraic equations generally need be solved computationally, whether it be with direct methods or iterative methods. The Semi-Implicit Root solver (SIR [4]), reported on here, was developed in order to improve on the global convergence characteristics of the widely used Newton method. Linesearch [6] is often combined with Newton’s method to improve convergence but, unlike SIR, it may lead to local extrema rather than to the roots of the equations. SIR development was initially inspired by semi-implicit PDE algorithms [1],[2],[3] and evolved as a robust equation solver for the time-spectral method GWRM [4] for systems of PDEs; it is however generally applicable to systems of equations. Recently the algorithm, and its coding in MATLAB and MAPLE, has been substantially improved with respect to efficiency. Updated software and matrix handling is now used. We believe it would be beneficial to present ready-to-use codes, being the motivation for this paper. First, a brief overview of SIR is provided in section 2. For a thorough explanation of the SIR algorithm the reader is advised to consult [5]. An example application is given in section 3. Pseudocode, describing the algorithm in detail, can be found in section 4. In Appendix, MATLAB and MAPLE codes are provided.

2. Software description

The roots of the single equation \( f(x) = 0 \), with \( f(x) \equiv x - \varphi(x) \), are found by SIR after a reformulation in iterative form as

\[
x^{i+1} + \beta x^{i+1} = \beta x^i + \varphi(x^i),
\]

(1)

where \( \beta \) is a real and arbitrary parameter. Eq. 1 will have the same roots as the original equation. We cast it into the form

\[
x^{i+1} = \alpha (x^i - \varphi(x^i)) + \varphi(x^i),
\]

(2)
where $\alpha = \beta/(1 + \beta)$ is a parameter for optimizing global and local convergence. Introducing $\Phi(x; \alpha) \equiv \alpha(x - \varphi(x)) + \varphi(x)$ it may be shown that convergence requires $|\partial \Phi / \partial x| < 1$ to hold for the iterates $x^i$ in a neighbourhood of the root [5]. Newton’s method (for a single equation also known as Newton-Raphson’s method) assumes $|\partial \Phi / \partial x| = 0$ everywhere, thus potentially achieving maximized, second order convergence near the root. Newtons method is, however, not globally convergent because of the breakdown of the linear approximation for initial iterates $x^0$ positioned too distant from the root. This problem is remedied by SIR through enforcing monotonic convergence. The trick is to choose appropriate values of $|\partial \Phi / \partial x| \equiv R$ at each iteration $i$.

Thus SIR iterates the equation

$$x^{i+1} = \alpha^i(x^i - \varphi(x^i)) + \varphi(x^i), \quad (3)$$

using

$$\alpha^i = \frac{R^i - \varphi'(x^i)}{1 - \varphi'(x^i)}, \quad (4)$$

where $\varphi'(x) = d\varphi/dx$. Typically $R^i$ is initially given a value in the interval [0.5, 0.99] whereafter SIR automatically reduces it towards zero to achieve second order convergence in the vicinity of the root. Since monotonic convergence is guaranteed, SIR will consecutively find all real roots $x$ to the equation.

Generalizing to systems of equations, SIR now solves

$$x^{i+1} = A^i(x^i - \varphi(x^i)) + \varphi(x^i), \quad (5)$$

where

$$A = I + (R - I)J^{-1},$$

$$(R)_{mn} = \delta_{mn}R_m.$$ 

Here the Jacobian matrix $J$ has components $J_{mn} = \partial(x_m - \varphi_m(x))/\partial x_n$, $\delta_{mn}$ is the Kronecker delta and $I$ is the identity matrix. It is usually most economic to obtain $A$ by solving the linear matrix relation

$$\Phi' = (A - I)J + I,$$

using $\partial \Phi_m / \partial x_n = R_{mn}$ (diagonal matrix) and sparse matrix methods.

When solving multidimensional equations, strict monotonous convergence cannot be guaranteed, since there is no known effective procedure (like root bracketing in the 1D case) to safely maintain the direction from a starting point towards a root. A procedure for ”quasi-monotonicity” is thus employed in SIR [5]. SIR is also safeguarded against certain pitfalls. An evident problem can be seen in Eq. (4); there is a risk that the $A$ matrix may become singular. The cure for this is subiteration, where $R_m$ values are modified towards unity. See [5] for further discussion of measures that enhance convergence.

Summarizing, at each iteration the SIR algorithm reduces $R_m$ values towards zero to approach second order convergence. If non-monotonous convergence becomes pronounced in any dimension, local subiteration by increasing $R_m$ values towards unity is used.

3. Illustrative example

SIR has been compared to Newton’s method with line-search (NL) for a large set of standard problems [5]. In several cases it features superior convergence characteristics, in particular when singularities appear in the computation of the $A$ matrix.

It is well known that the NL method may sometime land on local minima rather than on the roots of the equations. An example of this was provided in [5], where also the concept of ”convergence diagrams” were introduced. Since iterative solvers can be very sensitive to the starting points $x^0$, the diagram displays, using colour marking, the quality of convergence using a set of 61 by 61 uniformly distributed $x^0 \in [-5, 5]$. The two equations solved are $x_1 = \cos(x_2)$ and $x_2 = 3\cos(x_1)$.
The convergence diagrams clearly show that the global convergence properties of the SIR and NL schemes are complex even for this seemingly simple problem. It is seen that, in subiteration mode (SIR-s), convergence is rapid for nearly all starting points, whereas here standard SIR converges for about one third of the starting points. The NL method cannot match the convergence of SIR-s, due to excessive landing on non-zero local minima, and performs similarly as SIR.

Recently we have introduced a number of measures to improve SIR efficiency. An important example concerns the time-consuming computation of the A matrix, which acts as a "scaffold" when building the solution. The scaffold needs not be perfect, however. In nonlinear computations it may suffice to recompute A the first few iterations only (parameter mA; see MAPLE code), whereafter it accurately leads to convergence. The fact that the A is a constant for linear systems of equations is also useful.

The MATLAB implementation has been significantly improved by minimizing the number of symbolic operations performed.

4. MATLAB code for the example

Below follows a MATLAB coding of the example described above.

```matlab
% Find roots with SIR and fsolve
% Find roots (x1, x2)
% SIR phi(x1, x2) = x1 - f1(x1, x2) = cos(x2),
% phi2(x1, x2) = x2 - f2(x1, x2) = 3cos(x1).

disp('SIR solution')
disp(y2D)

disp('fsolve solution')
disp(y2D)

disp('2D case')
fun2D = @(x) [x(1) - cos(x(2)) .... x(2) - 3*cos(x(1))];

phi2D = @(x) [cos(x(2)), 3*cos(x(1))];
tic; y2D = fsolve(fun2D, x0); toc
disp('fsolve solution')
disp(y2D)

disp('2D case')
fun1D = @(x) [x - 2*cos(x)];

phi1D = @(x) [x - 2*cos(x)];
tic; y1D = fsolve(fun1D, x0); toc
disp('fsolve solution')
disp(y1D)

phi1D = @(x) [2*cos(x)];
tic; U1D = SIR(phi1D, x0, 0); toc
plot3(x(1), x(2), 0, ' Marker', 'o', ....
'MarkerEdgeColor', 'r', ....
'MarkerFaceColor', 'r', 'r')

phi2D = @(x) [cos(x(2)), 3*cos(x(1))];
tic; U2D = SIR(phi2D, x0, 1); toc

plot3(U2D(1), U2D(2), 0, ' Marker', 'o', ....
'MarkerEdgeColor', 'b', ....
'MarkerFaceColor', 'b')

disp('SIR solution')
disp(U2D)

disp('Surfplot')

[X, Y] = meshgrid(-5:0.65:5);
f1 = X - cos(Y); f2 = Y - 3*cos(X);
Z = (f1^2 + f2^2)^2/2;

s = surf(X, Y, Z, 'FaceAlpha', 0.45);
view([-37 60])

plot3(x0(1), x0(2), 0, ' Marker', 'o', ....
'MarkerEdgeColor', 'r', ....
'MarkerFaceColor', 'r', 'r')

plot3(U2D(1), U2D(2), (U2D(1)^2 - 3*cos(U2D(1))).^2/2, ....
'Marker', 'o', ....
'MarkerEdgeColor', 'b', ....
'MarkerFaceColor', 'b')

hold off
depsurf surfFig
```

Figure 1: Surfplot of |f|^2 = (f_1^2 + f_2^2)/2, where
f_1 = x_1 - cos(x_2) and f_2 = x_2 - 3cos(x_1). Red point is initial choice x^0 = [-2, -2] and Blue point is the solution root (-0.684, 2.32) with an accuracy ε ∼ 10^{-11}.
5. Pseudocode

The semi-implicit real root solver, given by Eq. 5, has been coded in MATLAB and in MAPLE. A pseudo-code is provided below. Let us first, however, discuss the basic steps as the computation proceeds. After parameter and procedure definitions and initialization with estimate \(x^0\), there are three main steps of the iteration loop.

Firstly, for each iteration step new positions \(x^{i+1} = \Phi(x^i, A^i)\) are computed; Eq. (5).

Secondly, an optional validity check of the iteration step is carried out if convergence is slow, i.e. if at least one member of the set \(|x_n^i - x_n^{i-1}| - |x_n^{i-1} - x_n^{i-2}|, n = 1, ..., N\) is greater than zero. Here it is controlled that convergence is quasi-monotonous and that the \(A\) matrix is not singular or near-singular.

The requirement \((x_n^i - x_n^{i+1})(x_n^{i+1} - \Phi_n^{i+1}) > M_c\) guarantees quasi-monotonicity. The index \(k\) sweeps through \(K\) equidistantly placed points in the interval \([x_n^i, x_n^{i+1}]\). Large \(K\) values result in costly function evaluations; we have found however that the value \(K = 1\) is appropriate for most problems, including those of Table 1 in 5. The value \(M_c = 0\) would correspond to strictly monotonous convergence, which usually is unattainable. Instead by choosing a small, negative value for \(M_c\), quasi-monotonicity is allowed.

Furthermore, if the \(A\) matrix with components \(a_{mn}\) is near-singular, fatally large steps could result. We here require that all \(|a_{mn}| \leq a_{max}\), where \(a_{max} = 2\) is a good default value.

As a final third step, the slope parameter \(R^i\) is reduced towards zero to facilitate convergence approaching second order; thus we have \(R^{i+1} = \kappa R^i\), where standard values are \(R^0_n = 0.95\) and \(\kappa = 0.5\).

The iterations proceed until the desired accuracy is achieved or until the assigned number of iterations is exceeded. It should be remarked that the use of subiterations by calling SIR-s slows down the algorithm due to additional numerical operations and should only be used when required.

A formal pseudocode for SIR is given below and in 6. In Appendices A and B, MATLAB and MAPLE codes for SIR are provided.

6. Impact

SIR is routinely used as a robust solver of systems of nonlinear algebraic equations in GWRM applications 7. The GWRM is a time-spectral PDE solver that has been applied to linear and nonlinear PDEs. The latter include Burger’s equation, a nonlinear wave equation and chaotic equations relevant for numerical weather prediction.

The algorithm has extended global convergence as compared to Newton’s method and avoids landing on local minima, being problematic for Newton’s method using linesearch. Thus SIR has a potential of being widely used in a number of computational physics areas.

7. Conclusion

SIR is a recently developed solver for general systems of nonlinear algebraic equations. Global convergence is often superior to that of Newton’s method and Newton’s method with linesearch, as shown in an example. SIR is also simple to code; compact MATLAB and MAPLE codes are included as appendices.
References

[1] Gottlieb, D., Orszag, A., 1977. Numerical Analysis of Spectral Methods: Theory and Applications. SIAM, Philadelphia.

[2] Harned, D., Kerner, W., 1985. Semi-implicit method for three-dimensional compressible magnetohydrodynamic simulation. Journal of Computational Physics 60, 62–75.

[3] Harned, D., Schnack, D. D., 1986. Semi-implicit method for long time scale magnetohydrodynamic computations in three dimensions. Journal of Computational Physics 65, 57–70.

[4] Scheffel, J., 2011. Time-spectral solution of initial-value problems. In: Partial Differential Equations: Theory, Analysis and Applications. Nova Science Publishers, Inc., pp. 1–49.

[5] Scheffel, J., Håkansson, C., 2009. Solution of systems of nonlinear equations, a semi-implicit approach. Applied Numerical Mathematics.

[6] W. H. Press, S. A. Teukolsky, W. T. V., Flannery, B. P., 2007. Numerical Recipes; The Art of Scientific Computing 3rd Edition. Cambridge University Press.
Algorithm 1 Pseudo code: Semi-implicit root solver SIR

procedure SIR
input: A vector function $\varphi : \mathbb{R}^N \rightarrow \mathbb{R}^N$ and initial estimate $x^0 \in \mathbb{R}^N$.
output: A vector $x^\ast$, being a root of the matrix equation $x = \varphi(x)$.

parameters: $N$ - number of equations to be solved, $tol$ - solution accuracy, $I_{max}$ - max number of iterations, $I_{S}$ - max number of subiterations, $K$ - number of monotonicity check points, $\alpha_c$ - maximum allowed magnitude of $A$ matrix elements, $M_c$ - parameter for monotonicity check, $d\Phi dx_0$ - initial values of $\partial \Phi / \partial x_i$, $R_{fac}$ - factor controlling $\partial \Phi / \partial x_i$ at each iteration.

It is cheaper to solve $R = (A - I)J + I$ for $A$ rather than computing $J^{-1}$ as below.

$$conv = |x_i^i - x_i^{i-1}| - |x_i^{i-1} - x_i^{i-2}|$$

$$R := d\Phi dx_0, \ I := \text{Identity matrix}$$

$$J = \{\partial(x_m - \varphi_m(x)) / \partial x_n\}$$

for $i = 1$ to $I_{max}$ do

$x = x^0$

$A = I + (R - I)J^{-1}$

$x^1 = A(x - \varphi) + \varphi$

if sub and $\max(conv) > 0$ then

for $j = 1$ to $I_{S}$ do

$S_1$ = subiterated sequence for monotonicity test (see text)

$S_2$ = sequence of $|A_{nj}|$ for all dimensions $n$

if $\max(S_2) < \alpha_c$ and $\min(S_1) \geq M_c$ then return false

$R_n = (3R_n + 1)/4$

$x = x^0$

$A = I + (R - I)J^{-1}$

$x^1 = A(x - \varphi) + \varphi$

$R = R_{fac}R$

$\varepsilon = \sum_{n=1}^{N} |x_n^1 - x_n^0| / N$

$x^0 = x^i$

if $\varepsilon < tol$ then return false

$x^* = x$
Appendix A. Matlab code

function U = SIR(phi,x0,sub)

% Parameters

if sub==0
% Default values
Rfac=0.5; % Reduction of R at each iteration
p1 = 0.95; % Initial values of R
else
% Values when subiterating
Rfac=0.8; % Reduction of R at each iteration
p1 = 0.999; % Initial values of R
end

N=numel(x0); % Number of equations
tol=le-8; % Solution accuracy
imax=100; % Maximum number of iterations
Js=1000; % Maximum number of subiterations
alpha = 2; % Critical magnitude for monotonicity check
S1_min=5e-2; % Parameter for monotonicity check

% Initialization

x = x0(1:end); % Turn x0 into column vector
xold = zeros(size(x0)); % Store the old x0
R = dPdx*x0; % R0
J = @x(jacobian(phi,x,N));

% Iteration Loop

for n = 1:imax
  % I: Compute new alpha
  x = x0; % Initial guess
  phi0 = (phi(x))'; % Evaluate initial guess
  J0 = J(x); % Evaluate center diff if no Jacobian
  % Test for singularity
  condJ = recond(full(J0));
  if condJ <= eps
    error('Jacobian singular in SIR');
  end
  % Compute R-I
  R0 = R1proc(R,N);
  % Compute alpha
  alpha = alphaproc(R0,J0,N);
  x1 = Phi proc(x0,phi0,alpha);
  % Only subiterate if step length is increasing
  if sub && max(abs(double(x1-x0)) - abs(double(x0-xold))) > 0
    mon = x0-x1;
    % V: Check validity of x1
    for j=1:Js
      % Evaluate phi(x1)
      x = x1;
      % Initial guess
      phi1 = (phi(x))';
      % Evaluate initial guess
      S1 = mon.* (x1-Phi proc(x1,phi, alpha));
      S2 = max(abs(alpha),[],2);
      % Perform all comparisons at once,
      % store in boolean vector ifR
      % A particular row of ifR will be true if subiteration is
      % required in that dimension.
      ifR = (S2 >= alpha | S1 < S1_min) ;
      % Break if no subiterations are needed
      if max(ifR) == 0
        break;
      end
    end
    % Increase R towards I
    for i=1:N
      % Increase i:th dimension
      if ifR(i) == 1
        R(i) = (3*R(i)+1)*0.25;
      end
    end
    % Update RI
    R10 = RI1proc(R,N);
    % Recompute alpha
    alpha = alphaproc(R10,J0,N);
    x1 = Phi proc(x0,phi0,alpha);
  end
  % Subiterations end here:
end

% Accuracy test and update
eps_acc = mean(abs(x1-x0));
xold = x0; % Backup x0 so that it can be used to compare step lengths.
x0 = x1;
if eps_acc < tol
  break
end

% Decrease R
Appendix B. Maple code

Appendix B.1. SIR

1 SIR:=proc(N,phi,X,x,J,ITmax, Rfac, dPdx, sub, tol, mA)
2 global Phi, alpha, Rl, R, E, f, nsave, U, eps:
3 local Js,a,c,S1min, Phi_proc, Phi_one_proc, J_proc, i, j, k, m, n, alpha_proc, xvold, x0, x1, mon, conv, x0_save, x, Phi_proc,
4 S1, S2, ifR, xv:

5 # Root solver SIR #
6 # This Maple code finds a solution to the system of nonlinear equations f = x –
7 phi(x) = 0 from an initial guess
8 solution vector X. A semi-implicit
9 approach is used, being a
10 generalization of Newton’s method with
11 improved global convergence
12 characteristics. Newton’s method is
13 obtained by setting the parameter dPdx
14 = 0. For details, see J. Scheffel and C .
15 Hakansson, Appl. Numer. Math. 59 (2009) 2430.

16 # Input parameters#
17 #
18 # N – total number of equations
19 # phi – functions in fixed point
20 # equations x = phi(x)
21 # X – initial guess for
22 # x – solution vector
23 # J – Jacobian of f(x) = x – phi(x)
24 # ITmax – maximum number of iterations
25 # Rfac – reduction of R at each iteration
26 # dPdx – dPhi/dx = R at first iteration
27 # sub – for subiteration set =1 (rec=0)
28 # tol – solution accuracy
29 # mA – number of iterations that A
30 matrix is recomputed (Nonlinear
31 equations: use ITmax value for default
32 or try numbers like 5 to increase
33 efficiency. Linear equations: since A
34 is constant, use 0 when calling SIR
35 more than once)
36
37 Js :=20:
38 K :=1:
39 a :=2.0:
40 S1_min :=−5.0e−2:
41 proc
42
43 #—Vectors and Matrices.——#
44 unassign (’x’, ’R’):
45 xv := Vector(N, symbol = x):
46 E := IdentityMatrix(N, storage=sparse):
47 RI := DiagonalMatrix(Vector(N, symbol=R)–
48 Vector(N,1)):
49
50 #—Procedures.——#
51 # Semi-implicit function Phi(x,a)
global Phi:
Phi:=map(eval, alpha.*xv-phi)+phi):
end:
42
43 # Evaluates single row of Phi(x;a)
Phi[p] := proc(proc:=proc(k,x)
local m:
46 global Phi:
Phi[0] := add(alpha[k]*m((x[m]-phi[m]),m=1..N)+
phi[k]):
end:
49
50 # Semi-implicit parameters alpha[i,j]
51 alpha[proc]:=proc()
52 global alpha;
53 alpha:=E+Transpose(LinearSolve(Transpose(J)
Transposed(R))):
end:
54
55 # Initialization.--#
56 x[0] := X:
57 # Initial guess for root x = x0
58 xold[0] := Array(1..N):
59 R[0] := dPdx[0](Array(1..N,1)):
60 eps := 1.0:
61 # Dummy value for accuracy test
62
63 # Iteration loop.--#
64 for n from 1 to Tmax whileeps > tol od:
65 # First: compute new alpha matrix and iterate Phi.--#
66 if n<mA then alpha[proc]():
fi:
67 Phi[proc]():
68
69 # Second: assign new x positions.--#
70 for i from 1 to N do
71 x[i] := Phi[i];
72 x1[i] := x[i];
73 mon[i] := x0[i]-x1[i]:
74 od:
75
76 # Third: for sub=1, subiterate and make sure that valid step is taken.
78 # Valid step: 1) convergence should be
79 # 2) near-singular values of alpha should be avoided.
80 # Invalid step remedy: respective R[i] are adjusted towards 1 in # up to Js sub-
81 # iterations.
82 if sub=1 then
83 if max(seq(abs(x1[i]-x0[i]))>0 then print('inside');
84 for m from 1 to Js do
85 x0 := x0; x0[i] := x0[i];
86 for k from 1 to K do xt[k] := x0[i] + (K-xt[k]); od;
87 for k from 1 to K do
88 x0[i] := xt[k];
89 Phi[0] := Phi[k];
90 Phi[i] :=Phi[k];
91 od;
92 x0 := x0; x0[i] := x0[i];
93 S1 := seq(mon[i]*((xt[k]-Phi[k]),k=1..K),
94 if max(S2) < a.c and min(S1) >= S1min
95 then ifR[i] := 0 else ifR[i] := 1: fi:
96 od:
97
98 if max(seq(ifR[j],j=1..N)) = 0 then break:
99 fi:
100 for i from 1 to N do
101 ifR[i] := 1 then R[i] := (3*R[i]+1.0)
102 /4.0: fi:
103 x[i] := x0[i]:
104 od:
105 if nc<ma then alpha[proc]():
106 Phi[proc]():
107 for i from 1 to N do
108 x[i] := Phi[i];
109 x1[i] := x[i]:
110 od:
111
112 # Fourth: second order convergence means letting R values tend to zero. Save also iteration data for each x component.--#
113 for i from 1 to N do
114 R[i] := Rfac*R[i];
115 xold[i]=x0[i];
116 U[i,n]:=x[i]; if n=1 then U[i,0]:=xold[i];
117 fi:
118 od:
119
120 # End of iteration loop
121 #---Fifth: save the number of iterations and the new X vector (=x). Set x free.--#
122 nsave:=n;
123 for i from 1 to N do X[i] := x[i]:
124 od:
125 unassign('x');
126
127 # Appendix B.2. Compute Jacobian

128 # Create Jacobian J
129 # by differentiating f(x)=x-phi(x).
130 #)
131 #)
132 #)
133 #)
134 #)
135 #)
136 #)
137 #)
138 #)
139 #)
140 #)
141 #)
142 #)
143 #)
144 #)
145 #)
146 #)
147 #)
148 #)
149 #)
150 #)
151 #)
152 #)
153 #)
154 #)
155 #)
156 #)
157 #)
158 #)
159 #)
160 #)
161 #)
162 #)
163 #)
164 #)
165 #)
166 #)
167 #)
168 #)
169 #)
170 #)
171 #)
172 #)
173 #)
174 #)
175 #)
176 #)
177 #)
178 #)
179 #)
180 #)
181 #)
182 #)
183 #)
184 #)
185 #)
186 #)
187 #)
188 #)
189 #)
190 #)
191 #)
192 #)
193 #)
194 #)
195 #)
196 #)
197 #)
198 #)
199 #)
# J flag=2 - faster (recommended): uses sparse matrix algorithms when appropriate:

if J flag=2 then
  J := Vector Calculus [Jacobian] (seq(x[i] - phi [i], i=1..N), seq(x[j], j=1..N)):
fi:

# J flag=3 - really fast for large, sparse matrices (N>1500 or so). Algorithm below is for band matrix where parameters m1 and m2 stand for the number of subdiagonals and superdiagonals respectively:

if J flag=3 then
  J := Matrix (1..N, 1..N):
  for k from 0 to m1 do
    for i from m1+k to N do
      f[i] := x[i] - phi[i];
      J[i, i-m1+k] := diff(f[i], x[i-m1+k]);
    od:
  od:
  for k from 0 to m2-1 do
    for j from m2+k to N do
      f[j-m2+k] := x[j-m2+k] - phi[j-m2+k];
      J[j-m2+k, j] := diff(f[j-m2+k], x[j]);
    od:
  od:
fi:
end: