RECURSIVE ALGORITHMS FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS

Jiří Jan

(Received July 10, 1987)

Summary: A way of generalizing onedimensional root-finding algorithms to the multidimensional case by means of recursion is shown and means to make the algorithm robust are discussed. In the second part, the algorithm is modified so as to exploit sparsity of large systems of equations for reducing the recursion depth and consequently decreasing the computational requirements of the method.

Keywords: recursion, nonlinear equations, root-finding algorithms.
AMS Classification: 65H10.

INTRODUCTION

It is generally believed that most reliable methods to solve onedimensional equations of the type $f(x) = \emptyset$, i.e. bisection and regula falsi, cannot be generalized to the multidimensional case (where $\boldsymbol{f}$ is a vector function of a vector argument $\boldsymbol{x}$), as the sign function on which the both methods rely is not defined for vectors. Nevertheless, a different approach to the multidimensional generalization of onedimensional algorithms has been pointed out, which consists in their recursive use [1 – 3]. At the time of its first descriptions, no programming languages allowing the recursive call of procedures were available so that the method had to be described by less sophisticated programming means that made it difficult both to understand and to program.

This contribution deals with the exploitation of the abilities of the PASCAL language to express the true recursion and to define special nonnumerical types of data, in such a way that the basic structure of the algorithms is simply described and easily understood. It will also be shown how a failure of the root finding algorithm can be prevented. The second part of the paper is devoted to exploiting the sparsity of many practical systems of equations (e.g., systems modelling electrical circuits) to speed up the process of solution substantially. The algorithms are presented in the usual form of procedures in order to make them selfcontained and selfexplaining as much as possible; practical implementations may, of course, differ in some details in order to simplify the complete program structure.

No attention is paid here to the questions of automatic compilation of equations on the basis of circuit description; this was the subject of part of the report [2]. Sufficient conditions (not necessary, but fulfilled in most electrical circuits) for the algorithm to converge are discussed also in [2], and a straightforward physical interpretation of the algorithm when used to analyze electrical circuits is included in [5]. The work [4] applies the recursive approach to the problem of solving systems of nonlinear differential equations.

1. RECURSIVE METHOD FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS

1.1. **Principle of the method**

Let us have a system of nonlinear algebraic or transcendental equations

$$f(x) = 0 \,, \tag{1}$$

i.e.

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \,. \end{aligned}$$

A root $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ of the system is to be found in the region Ω,

$$\Omega\colon \quad x_j \in \langle {}^0x_j, {}^1x_j \rangle \,, \quad j = 1, 2, \ldots, n \,.$$

Without losing generality we will suppose that

$$\forall j \in \langle 1, n \rangle \,, \quad {}^0x_j = x_{0j} - r \,, \quad {}^1x_j = x_{0j} + r$$

where $\boldsymbol{x}_0 = (x_{01}, x_{02}, \ldots, x_{0n})$ is an initial estimate of the root and r is a chosen constant (labeled "radius") so that Ω is a (hyper) cube with a side $2r$.

The recursive method can be described as follows:

There are n iteration levels for n unknowns. On the k-th level the vector $(x_1, x_2, \ldots, x_{k-1})$ is constant during each complete iteration on this level as it is given from the $(k - 1)$-st level. The root x_k of the equation

$$f_k(x_1, x_2, \ldots, x_n) = 0 \tag{2a}$$

is found by onedimensional iteration in the course of which the vector $(x_{k+1}, x_{k+2}, \ldots, x_n)$ which is a function of x_k is being determined repeatedly before each calculation of a value of the function f_k by means of the same algorithms used on the levels $(k + 1)$, $(k + 2)$, $\ldots$, n.

Comment. This means that the equation (2a) can be rewritten as

$$f_k(x_1, x_2, \ldots, x_{k-1}, x_k, x_{k+1}(x_k), x_{k+2}(x_{k+1}(x_k)), \ldots, x_n(x_{n-1}(x_{n-2}(\ldots)))) = 0 \tag{2b}$$

The dependence of $\boldsymbol{x}$-components is given implicitly by means of equations similar to (2b) on lower levels, e.g., x_{k+1} as a root of f_{k+1} on the level $(k+1)$.

It is obvious that if the equation (2) has been solved by this method, the $(n-k+1)$-dimensional system

$$
(3) \qquad \begin{aligned}
f_k(x_1, \ldots, x_n) &= 0\\
f_{k+1}(x_1, \ldots, x_n) &= 0\\
\vdots \qquad & \vdots\\
f_n(x_1, \ldots, x_n) &= 0
\end{aligned}
$$

has been solved as a whole.

If $k = 1$, the given system has been completely solved.

The corresponding algorithms can be seen in tab. I.

The procedure "rootn" looks for a root of an N-dimensional system, given by component functions which are all described in the body of the parameter function "vectorf", which has the basic form

```
function vectorf1 ({variable:} x: vector; {level:} k: integer);
begin case k of

    1: vectorf1 := "expression giving value of f1(x)";
    2: vectorf1 := "expression giving value of f2(x)";
    :
   N: vectorf1 := "expression giving value of fN(x)";

   end;
end.
```

The data type "vector" is defined as

```
type vector = array [1··N] of real;
```

The procedure "rootn" makes use of the initial estimate X_0 of the root and delivers the resulting root as the variable parameter X. In accordance with the above described method, the body of "rootn" consists of a single statement solving the first equation.

The core of the algorithm is the function "root" which is a quite ordinary one-dimensional root-finding routine except for the first row of its body where the initial limits of the root are calculated in terms of the level-corresponding component of X_0. The rest of the body is completely independent of the level. Its substantial feature is that it calculates the value f_k (on the k-th level) by means of the function „valuef" which is the only one that communicates with the vector $\boldsymbol{x}$, and before calculating the value f_k it calls the solution on the next lower level (if it exists). This most important statement is marked by an arrow.

The basic algorithm, as described, has two major disadvantages: It is unreliable with certain systems of equations, as the solution fails if on any level the function is not defined somewhere within the starting interval or has not different signs at the ends of the interval; it will be the subject of Chapter 1.2 how to cure it generally. The other difficulty is that the algorithm is of explosive complexity as the amount of calculations needed to find a root depends exponentially on the depth of the recursion n. The basic algorithm is thus suitable for systems with only a small number of equations (which is mostly not the case with circuit analysis). Chapter 2 presents a modification which utilizes the sparsity of practical systems of equations to decrease the recursion depth substantially.

Tab. I.

```
procedure rootn ({of} function vectorf: real; {with}
      N {dimensions}: integer; {in surroundings} radius: real;
      {around initial estimate} x∅: vector;
      {result:} var x: vector);
  function root ({level:} k: integer): real;
    var xb, xr, xl: real; s, sl, sr: boolean;
    function valuef (xk: real): real;
      begin x[k] := xk;
        if k< > N then x[k + 1] := root (k + 1);                ←
        valuef := vectorf (x, k);
      end; {valuef}
    begin {the following can be any algorithm solving the equation
            "valuef (x) = ∅"; to be concrete, bisection is shown:}
            xl := x∅ [k] − radius; xr := x∅ [k] + radius;
      {initializing bisection:} sl := valuef (xl) > ∅; sr := valuef (xr) > ∅;
            if sl = sr then halt;
      {bisection:} repeat xb := (xl + xr) * ∅·5;
                          s := valuef (xb) > ∅;
                          if sl = s then xl := xb
                                    else xr := xb
                   until abs (xr-xl) < = 1E − 7 * abs (xl);
      root := xb
    end; {root}
  begin
    x[1] := root (1)
  end; {rootn}
```

1.2. Causes of failure and the corresponding precautions

When deriving the basic algorithm in Chapter 1.1 we assumed that the function **f** is defined everywhere in Ω and also that there always exists a root in any equation of the type (2). This is not always fulfilled in real systems as even the component

Tab. II.

```
{label 1, 2, 3; and var fail: boolean; dx: real; have to be decleared}
function valuef (xk: real): real;
      label 4;
      begin x[k] := xk; fail := false;
            if k < > N then x [k + 1] := root (k + 1);
                  if fail then goto 4;
                  valuef := vectorf (x, k);
   4: end; {valuef}
                  ⋮
{more sophisticated "initializing bisection" follows:}
      dx := radius * 4.Ø; {search for a point where value f(x) is defined:}
      repeat dx := dx * Ø.5; x := xl + dx * Ø.5;
            repeat s := valuef (x) > Ø; if not fail then goto 1;
                  x := x + dx
            until x > xr
      until dx < Ø.Ø5 * radius;
      goto 3; {3: end of root, failure on the level k}
   1: {1-st point found; search for a point with opposite sign of valuef:}
      dx := dx * Ø.5; xl := x - dx; xlr := x; xrl := x; xr := x + dx;
      repeat x := (xl + xlr) * 0.5; sl := valuef (x) > Ø;
            if fail then xl := x else
              if sl = s then xlr := x
                 else begin xr := xlr; xl := x; goto 2 end;
            x := (xrl + xr) * Ø.5; sr := valuef (x) > Ø;
            if fail then xr := x else
              if sr = s then xrl := x
                 else begin xr := x; xl := xrl; sl := s; goto 2 end
      until (xlr - xl) < 0.05 * radius;
      fail := true; goto 3; {3: end root, failure on the level k}
   2: {bisection: (xr and xl have oppossite signs of function values)}
                  ⋮
```

functions f_k are sometimes defined only in a part of Ω (and the subregions for different k differ, having only a small intersection in a close vicinity of the root). But even more important, some of the equations of the type (2) often have no solution on the level k which in turn means that on the higher level the function "valuef $_{k+1}$" is not defined. If formulated as in Chap. 1.1, the algorithm is rather fragile because the nonexistence of a function value or of a root on any level immediately causes failure of the whole solution.

This problem can be solved in a simple way by using again the recursive approach, which means that the attention must be paid just to the onedimensional root-finding algorithm and the precautions are automatically distributed to all levels by the recursion. On a particular level k, two contingencies must be taken into account:

a) the function "valuef$_k$"(x_k) is not defined everywhere in the interval $\langle x_{0k} - r, x_{0k} + r\rangle$;

b) the function "valuef$_k$"(x_k) has no root in the interval for either of the following reasons:

i – the function is not defined in the interval at all,

ii – in the interval of definition of the function, there is no change of sign and consequently no root.

As the onedimensional root-finding algorithm needs two initial estimates of x_k with opposite signs of "valuef", its start must be preceded by an algorithm providing them. This can be done in two steps:

– first, to treat the contingency a) the interval must be searched through in order to find a point x_k^* where the function "valuef$_k$" is defined. If no such point is found then it means failure on the level k (contingency b$_i$).

– second, searching among the points already checked with undefined "valuef$_k$", lying in a neighbourhood of the point x_k^*, another point x_k^{**} must be determined for which

$$\operatorname{sign}(\text{value} f_k(x_k^{**})) \neq \operatorname{sign}(\text{value} f_k(x_k^*)).$$

If this search is unsuccessful (contingency b$_{ii}$), it again means level-k-failure, else the root-finding algorithm can start.

In Table II the changes are presented that have to be made in the original algorithm (Tab. I) to incorporate the above mentioned search. The flag "fail" is established which is reset at each entry to the function "valuef"; it may be set by a failure either on the lower level $k + 1$ or during the evaluation of f_k, in both cases inside the body of "valuef". The text of the lower part of Tab. II realizing the above mentioned two steps should replace the lines in the body of function "root" marked "initializing bisection". This part of program utilizes the information contained in the flag after each call of the function "valuef", and in turn sets the flag in the case of the level-k-failure.

2. MODIFICATION OF THE METHOD FOR SPARSE SYSTEMS

2.1. **Principle of minimization of the recursive depth**

As follows from the description of the basic algorithm, in the general case, when all functions f_k, $k = 1, \ldots, n$ depend on all variables x_k, $k = 1, \ldots, n$, the recursion depth is n. Nevertheless, most of practical systems of equations are sparse in the sense that each particular function f_k depends on only a small number of variables.

Without a substantial loss of generality we will suppose that the equations are reciprocally dependent, i.e. if f_k depends on x_j, then also f_j depends on x_k, and that the equations are so arranged that f_k depends on x_k for all k. (This is the case e.g. with the models of electric circuits with structurally reciprocal elements.) Even if there are some unidirectional dependences in the system, they can be regarded as bidirectional with zero transfer in the opposite direction. The structure of such a system can be visualized in the form of an incidence graph in which the vertices correspond to the individual equations and edges express the mutual dependences (see an example in Fig. 1).

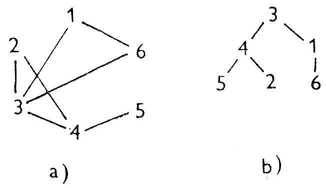

Fig. 1. Example of a system of equations:

$$
\begin{aligned}
&f_1(x_1, x_3, x_6) = 0\\
&f_2(x_2, x_3, x_4) = 0\\
&f_3(x_1, x_2, x_3, x_4, x_6) = 0\\
&f_4(x_2, x_3, x_4, x_5) = 0\\
&f_5(x_4, x_5) = 0\\
&f_6(x_1, x_3, x_6) = 0
\end{aligned}
$$

a) Incidence graph
b) Controlling tree

As the order of equations in the system is irrelevant, any of them, say k-th, can be chosen to be solved on the highest level of recursion and thus to be superior to all the others. The variable x_k is then constant during each calculation of "valuef$_k$" (x_k), that is, during solving the system consisting of all equations except the k-th. This inferior system can be described by the graph in which the vertex k and all its edges are deleted. It is possible that then the reduced graph is formed by several subgraphs which are not interconnected, which means that they are relatively (under the condition $x_k = \text{const}$) independent and may be solved separately (systems [2, 4, 5] and [1, 6] in Fig. 1 after removing the vertex 3). The same process can then be applied recursively in the subsystems.

The vertices removed from the graph according to the previous paragraph are compiled into another graph, called a controlling tree, which reflects the consecutive dividing of the original graph into hierarchicaly organized subgraphs. The tree is later used to control the multidimensional root-finding algorithm.

This approach enables us to reduce the recursion depth substantially by substituting a parallel arrangement of solutions for the nested one wherever possible (in Fig. 1, there are only 3 levels instead of 6). As the amount of calculations that are made parallelly is only linearly dependent on the number of parallel systems while in the nested arrangement it depends exponentially on the recursion depth, the exploitation of sparsity prevents the above mentioned explosive character of the algorithm's complexity and makes it practically possible to analyze even complicated systems.

2.2. **Algorithm for deriving a suboptimum controlling tree**

To find the absolutely optimal controlling tree would require the evaluation of all its possible arrangements which is practically not feasible. Therefore, only a suboptimum solution is possible, based on heuristic rules for the selection of the superior equation from any given (sub)system, that is, for selecting the "optimum" vertex that should be removed, possibly causing decomposition of the remaining vertices into separated subgraphs on the next or some further inferior level. The rules that have proved to provide the most reasonable results are as follows:

A) Choose the vertex that has the maximum of edges (as cancelling them simplifies the rest of the graph to the maximum extent).

B) Choose the vertex which being removed causes decomposition into the maximum number of subgraphs.

C) Choose the vertex which being removed causes decomposition with the possibly smallest variance in the number of vertices of the individual subgraphs.

Although the rule B seems to be most straightforward, it turned out that the best way is to use the rules in the order A, B, C (using the latter rule only in case the former gives an ambiguous result).

The corresponding algorithm, shown in Tab. III, reflects the recursive nature of the problem. To clarify its principle, first of all the used data structures have to be described. The vertices (nodes) are marked by integers in the range $1 \cdot\cdot N$, the incidence graph which means the input data for the algorithm is described by an i-indexed array of sets of vertices connected to the individual vertices i. The controlling tree as the output is mapped into a structure of dynamic variables linked by pointers, the corresponding data type definitions being as follows:

```
type node = 1··N; vertset = set of node;
     pointer = ↑ treecomponent;
     treecomponent = record
          vertex: node; graph: vertset;
          tonext, tosub: pointer
               end;
```

The crucial part of the algorithm is the procedure "compstep" which on each level communicates with an appropriate "treecomponent" to which the input parameter "totreecomponent" points. It selects the superior vertex out of the vertices contained in the set "totreecomponent↑.graph" (that has to be assigned before "compstep" starts) and realizes the decomposition of the "graph" into subgraphs that are then assigned to the newly established graph – parts in the inferior "treecomponents" beloging to the level k + 1. These are arranged in a stack to the top of which the pointer "tosub" points, while inside the stack the components are connected by means of pointers "tonext". In the body of "compstep" the recursive call of the same procedure on the lower level (accented in Tab. III by an arrow) ensures that the whole analysis is expressed in a very simple way. The complete tree as the result of the procedure applied to the incidence graph of Fig. 1 is shown in Fig. 2.

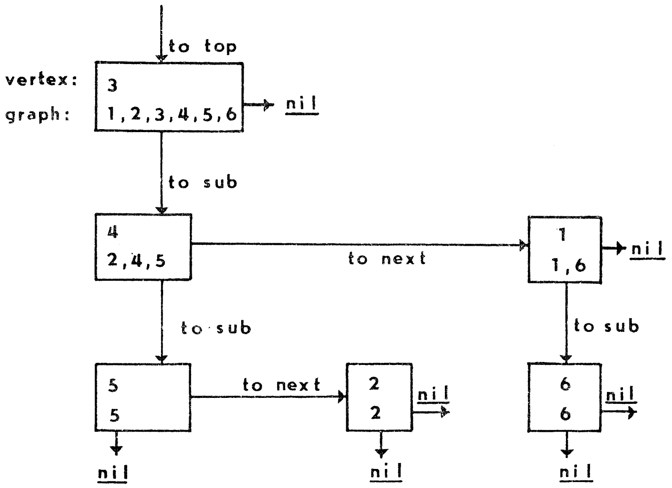

Fig. 2. Representation of the controlling tree as a PASCAL pointer structure.

2.3. Tree controlled recursive root-finding algorithm

The tree controlled algorithm must obey the same rule as the simpler one, described in Chap. 1.1, that is, before a function value on a certain level k can be calculated, all the inferior equations must be solved. While in the former case it meant just to call recursively the root-finding algorithm on the next level, now all the immediately subordinate equations need to be solved, which means to make the recursive call in a cycle until all the vertices immediately inferior to the k-level vertex have been treated. Inside the cycle, further nested recursive calls are done in the frame of lower

Tab. IIIa.

```
procedure compiletree ({incidence matrix:} incmat: array [node] of vertset;
  {pointer to root of tree:} var totop: pointer; N {vertices}: integer);
function vertsin (graph: vertset): integer;
  {determines the number of vertices in the graph}
procedure findmaxverts (var maxverts, {of} graph: vertset);
  {finds the set of vertices with maximum of links}
procedure formsubgraph (var {first} subgraph, {of} graph: vertset);
  {finds the first connected subgraph of the graph and moves its vertices from graph
    to subgraph}
procedure divide...
procedure compstep...
    {both described in detail in tab. IIIb}
begin new (totop);
  with totop ↑ do {initialization of root-treecomponent}
    begin graph := [ ]; tonext := nil;
          for i := 1 to N do graph := graph + incmat [i];
    end;
  compstep (totop)
end; {of compiletree – the tree is established}
```

Tab. IIIb.

```
procedure divide (inputgraph: vertset; {removing} k: node;
  {pointer to 1-st subgraph:} var tosub: pointer; var criterion: integer);
  var count, verts, max: node; this: pointer;
  begin {pushes individual subgraphs of inputgraph into a stack}
    inputgraph := inputgraph – [k]; tosub := nil; count := 0; max := 0;
    while inputgraph < > [ ] do
      begin new (this);
        formsubgraph (this ↑. graph, inputgraph);
        verts := vertsin (this ↑. graph);
        if verts > max then max := verts;
        count := count + 1;
        this ↑. tonext := tosub; tosub := this
      end;
      criterion := 1000 * count + N – max;
  end; {divide}
```

```
procedure compstep (totreecomponent: pointer);
  var max, p: integer; i, max: node; mxv: vertset; heap: pointer;
  begin while totreecomponent ⟨ ⟩ nil do
    begin with totreecomponent ↑ do
      begin {selection of the optimal vertex:}
        findmaxverts (mxv, graph); {rule A}
        max := 0; for i := 1 to N do {rule B and C}
          if i in mxv then
          begin mark (heap); divide (graph, i, tosub, p); release (heap);
            if p > max then begin max := p; i max := i end
          end; {i}
        vertex := i max; {optimal vertex found}
        {establishing lower level of tree:}
        divide (graph, i max, tosub, p);
        compstep (tosub); {compilation on the inferior level}          ←
      end; {with}
      totreecomponent := totreecomponent ↑ . tonext
    end; {of while}
  end; {compstep}
```

level root-finding procedures till the leaves of the tree are reached; then the control returns to higher levels.

The corresponding procedure "rootnsparse" is only slightly modified in comparison with the previously described "rootn"; the changes can be seen from Tab. IV. As there is one more input data structure for the procedure "rootnsparse" – the controlling tree, a new value-parameter "totop" appears which is pointing to the highest level (root-) component of the tree. Instead of having directly the level *k* as the parameter of the function "root", rather the pointer to it is used, and consequently at the beginning of the body of "root" the appropriate vertex number must be assigned to *k*. Also the statement in the body of "rootnsparse" must be arranged accordingly, i.e. with the pointer parameter and the pointer chosen index, respectively.

Most of the changes due to the control by the tree are concentrated in the body of the function "valuef". First, the control goes down the tree using the pointer field "tosub" of the current "treecomponent", and then a cycle for all lower-level parallel tree-components is organized using their "tonext" pointer fields. The cycle can be interrupted prematurely as it has no sense to continue solving parallel systems when the solution of one of them has failed; the failure must then be treated on the current level as described in Chap. 1.2.

Tab. IV.

```
procedure rootnsparse (·· {the same parameters as for rootn} ··,
  {pointer to the top of controlling tree:} totop: pointer);
  function root ({pointer to the appropriate component of tree:}
                 tok: pointer): real;
    label 1, 2, 3;
    var xb, xr, xl, xlr, xrl, dx: real;
      s, sl, sr: boolean; k: node;
    function valuef (xk: real): real;
      label 4; var ptosub: pointer;
      begin x[k] := xk; fail := false;
        ptosub := tok ↑ . tosub;
        while ptosub < > nil do
          begin x [ptosub ↑ . vertex] := root (ptosub);        ←
            if fail then goto 4;
            ptosub := ptosub ↑ . tonext
          end;
        valuef := vectorf (x, k);
    4: end; {valuef}
    begin k := tok ↑ . vertex;
        ⋮ {the same as before}
    end; {root}
  begin
    x [totop ↑ . vertex] := root (totop)
  end; {rootnsparse}
```

CONCLUSION

The above described algorithm in its robust version proved to converge and to find a root under very general (and sometimes unfavourable) conditions. In comparison with conventional methods (Newton-Raphson iteration, iteration by components) it has some advantages: no need for calculating and inverting Jacobi's matrix, and for special means against divergence. It is relatively slow but when utilizing sparsity it is reasonably applicable even to larger systems. It seems to be a good tool in complicated cases when there is little a priori information on root location.

References

[1] *J. Jan:* Recursive method of numerical analysis of inertialess nonlinear circuits (in Czech). Library of research and scientific writings, Technical University Brno, B-57, 1975.

[2] *J. Jan, J. Holčík, J. Kozumplík:* Recursive method and general purpose program RANC to analyze nonlinear circuits (in Czech). Research report, project no. III-3-1/1, Technical University Brno, 1975.

[3] *J. Jan, O. Gotfrýd, J. Holčík, J. Kozumplík:* Analysis of nonlinear circuits by means of the generalized recursive method. Proc. of the II-nd Int. Conference on Electronic Circuits, Prague 1976.

[4] *P. Hladký:* Use of the recursive method in analysis of transients in nonlinear circuits (in Czech). Thesis, Dept. of Computers, Technical University of Brno, 1976.

[5] *J. Jan, O. Gotfrýd, J. Holčík, J. Kozumplík:* Recursive analysis of nonlinear circuits (in Czech). Slaboproudý obzor 39, 1978, no. 1.

[6] *J. Jan:* Recursive algorithms to solve systems of nonlinear equations. Proc. of the 7-th European Conference on Circuit Theory and Design, Prague 1985.

Souhrn

REKURZIVNÍ ALGORITMY PRO ŘEŠENÍ SOUSTAV NELINEÁRNÍCH ROVNIC

JIŘÍ JAN

Je ukázán způsob zobecnění jednorozměrných algoritmů pro nalezení kořenů na mnohorozměrný případ pomocí rekurze, a jsou diskutovány prostředky, umožňující dosáhnout robustnosti. V druhé části je algoritmus modifikován tak, aby využíval řídkosti velkých soustav rovnic k redukci hloubky rekurze a tím ke snížení rozsahu výpočtu.

Резюме

РЕКУРСИВНЫЕ АЛГОРИТМЫ ДЛЯ РЕШЕНИЯ СИСТЕМ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

JIŘÍ JAN

Указан способ обобщения на многомерный случай при помощи рекурсии одномерных алгоритмов для определения корней и рассмотрены средства для достижения робастности. Во второй части работы приведен алгоритм, использующий разреженность больших систем уравнений для уменьшения глубины рекурсии и тем самым для уменьшения объема вычислений.

Author's address: Doc. Ing. *Jiří Jan*, CSc., Katedra lékařské elektroniky VUT, Purkyňova 95 B, 612 00 Brno.